SOFTWARE ENGINEERING

UNIT-I

INTRODUCTION

Computers were very slow in the initial years and lacked sophistication. If similar improvements could have occurred to aircrafts, now personal mini-airplanes should have become available, costing as much as a bicycle, and flying at over 1000 times the speed of the supersonic jets. The more powerful a computer is, the more sophisticated programs can it run. Therefore, with every increase in the raw computing capabilities of computers, software engineers have been called upon to solve increasingly larger and complex problems, and that too in cost-effective and efficient ways. Software engineers have coped up with this challenge by innovating and building upon their past programming experiences.

Software engineering is a systematic and cost-effective technique for software development. These techniques help develop software using an engineering approach. The innovations and past experiences towards writing good quality programs cost- effectively, have contributed to the emergence of the software engineering discipline.

For example: If someone wants to travel from Punjab to Delhi. There are two approaches one can follow to achieve the same result:

1. The normal approach is just to go out and catch the bus/train that is available.
2. A systematic approach is constructed as Firstly check on Google Maps about distance and after analyzing the timing of trains and buses online and after that, match user's preference suppose user have some work till 4:00 PM and trains slot are: 1:00 PM, 6:00 PM then the user will choose 6:00 PM time slot and reach Delhi.

From the above situation, one can easily analyze that Creating a systematic Approach is more optimal and time and cost-effective as compared to a normal approach. This will same occur while designing Software. So in Software Engineering working on an Engineering or a Systematic approach is more beneficial.

What is software engineering?

A popular definition of software engineering is: "A systematic collection of good program development practices and techniques".Good program development techniques have resulted from research innovations as well as from the lessons learnt by programmers through years of programming experiences. An alternative definition of software engineering is :"An engineering approach to develop software".

Suppose you have asked a petty contractor to build a small house for you. Petty contractors are not really experts in house building.They normally carryout minor repair works and at most undertake very small building works such as the construction of boundary walls. For example,he might not know the optimal proportion in which cement and sand should be mixed to realize sufficient strength for supporting the roof.In such situations, he would have to fallback upon his intuitions. Of course,the house constructed by him may not look as good as one constructed by a professional,may lack proper planning,and display several defects and imperfections. It may even cost more and take longer to build. The failure might come in several forms —the building might collapse during the construction stage itself due to his ignorance of the basic theories concerning the strengths of materials; the construction might get unduly delayed, and quantities of raw materials required,the times at which these are required,etc.In short,to be successful in constructing a building of large magnitude, one needs a good understanding of various civil and architectural engineering techniques such as analysis, estimation, prototyping, planning,designing, and testing.

For sufficiently small-sized problems,one might proceed according to one's intuition and succeed;though the solution may have several imperfections, cost more, take longer to complete,etc.

As is usual in all engineering disciplines,in software engineering several conflicting goals are encountered while solvinga problem.In such situations, several alternate solutions are first proposed.An appropriate solution is chosen out of the candidate solutions based on various trade-offs that need to be made on account of issues of cost,maintainability, and usability.Therefore,while arriving at the final solution,several iterations and are possible.

Abstraction Versus Decomposition

Two important principles that are deployed by software engineering to overcome the problems arising due to human cognitive limitations are—abstraction and decomposition

Abstraction:

In software engineering and computer science, abstraction is a technique for arranging complexity of computer systems. Abstraction means displaying only essential information and hiding the details. It works by establishing a level of simplicity on which a person interacts with the system, suppressing the more complex details below the current level. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Consider a real life example of a man driving a car.

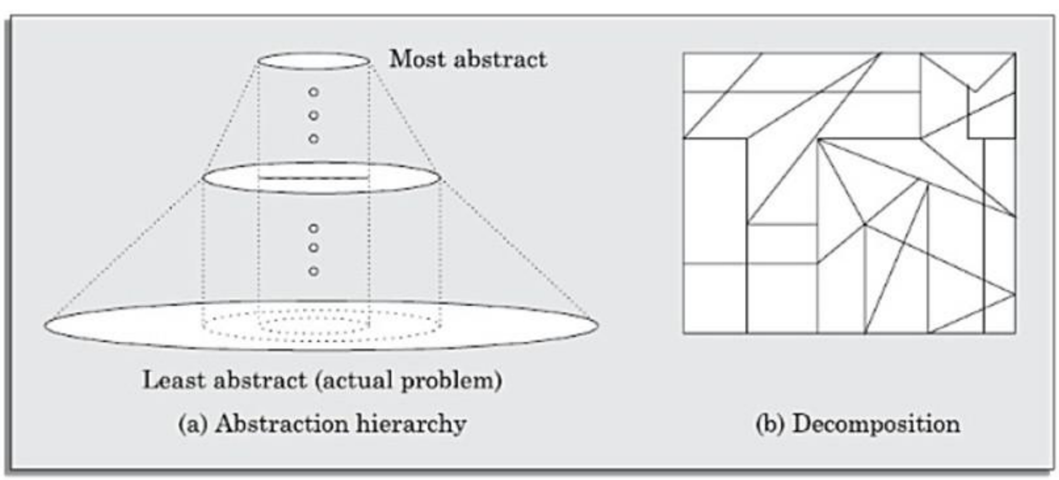


Fig: Abstraction and Decomposition

Decomposition:

It is another important principle that is available in the repertoire of a software engineer to handle problem complexity. This principle is profusely made use by several software engineering techniques to contain the exponential growth of the perceived problem complexity. The decomposition principle is popularly known as the divide and conquer principle.The decomposition principle advocates decomposing the problem into many small independent parts. The small parts are then taken up one by one and solved separately. The idea is that each small part would be easy to grasp and understand and can be easily solved. The full problem is solved when all the parts are solved.

Evolution of software design techniques

1. Early Computer Programming: Early commercial computers were very slow and too elementary as compared to today's standards. Even simple processing tasks took considerable computation time on those computers. No wonder that programs at that time were very small in size and lacked sophistication. Those programs were usually written in assembly languages.

Program lengths were typically limited to about a few hundreds of lines of monolithic assembly code. Every programmer developed his own individualistic style of writing programs according to his intuition and used this style ad hoc while writing different programs.

2. High-level Language Programming: Computers became faster with the introduction of semiconductor technology in the early 1960s. Faster semiconductor transistors replaced the prevalent vacuum tube-based circuits in a computer. With the availability of more powerful computers, it became possible to solve larger and more complex problems.

At this time, high-level languages such as FORTRAN, ALGOL, and COBOL were introduced. This considerably reduced the effort required to develop software and helped programmers to write larger programs.Writing each high-level programming construct in effect enables the programmer to write several machine instructions. However, the programmers were still using the exploratory style of software development.

3. Control Flow-based Design: A program's control flow structure indicates the sequence in which the program's instructions are executed. In order to help develop programs having good control flow structures, the flowcharting technique was developed. Even today, the flowcharting technique is being used to represent and design algorithms.

4. Data Structure-oriented Design: Computers became even more powerful with the advent of Integrated Circuits (ICs) in the early 1970s. These could now be used to solve more complex problems. Software developers were tasked to develop larger and more complicated software. which often required writing in excess of several tens of thousands of lines of source code.

It is much more important to pay attention to the design of the important data structures of the program than to the design of its control structure. Design techniques based on this principle are called Data Structure-oriented Design.

Example: Jackson's Structured Programming (JSP) technique developed by Michael Jackson (1975). In JSP methodology, a program's data structure is first designed using the notations for sequence, selection, and iteration. The JSP methodology provides an interesting technique to derive the program structure from its data structure representation.

5. Data Flow-oriented Design: As computers became still faster and more powerful with the introduction of very large scale integrated (VLSI) Circuits and some new architectural concepts, more complex and sophisticated software were needed to solve further challenging problems. Therefore, software developers looked out for more effective techniques for designing software and Data Flow-Oriented Design techniques were proposed.

The functions are also called as processes and the data items that are exchanged between the different functions are represented in a diagram is known as a Data Flow Diagram (DFD).

6. Object-oriented Design: Object-oriented design technique is an intuitively appealing approach, where the natural objects (such as employees, etc.) relevant to a problem are first identified and then the relationships among the objects such as composition, reference, and inheritance are determined. Each object essentially acts as a data hiding is also known as data abstraction.

Object-oriented techniques have gained wide spread acceptance because of their simplicity, the scope for code and design reuse, promise of lower development time, lower development cost, more robust code, and easier maintenance.

let us first examine what are the current challenges in designing software. First,program sizes are further increasing as compared to what was being developed a decade back. Second, many of the present day software are required to work in a client-server environment through a web browser-based access(called web-based software).

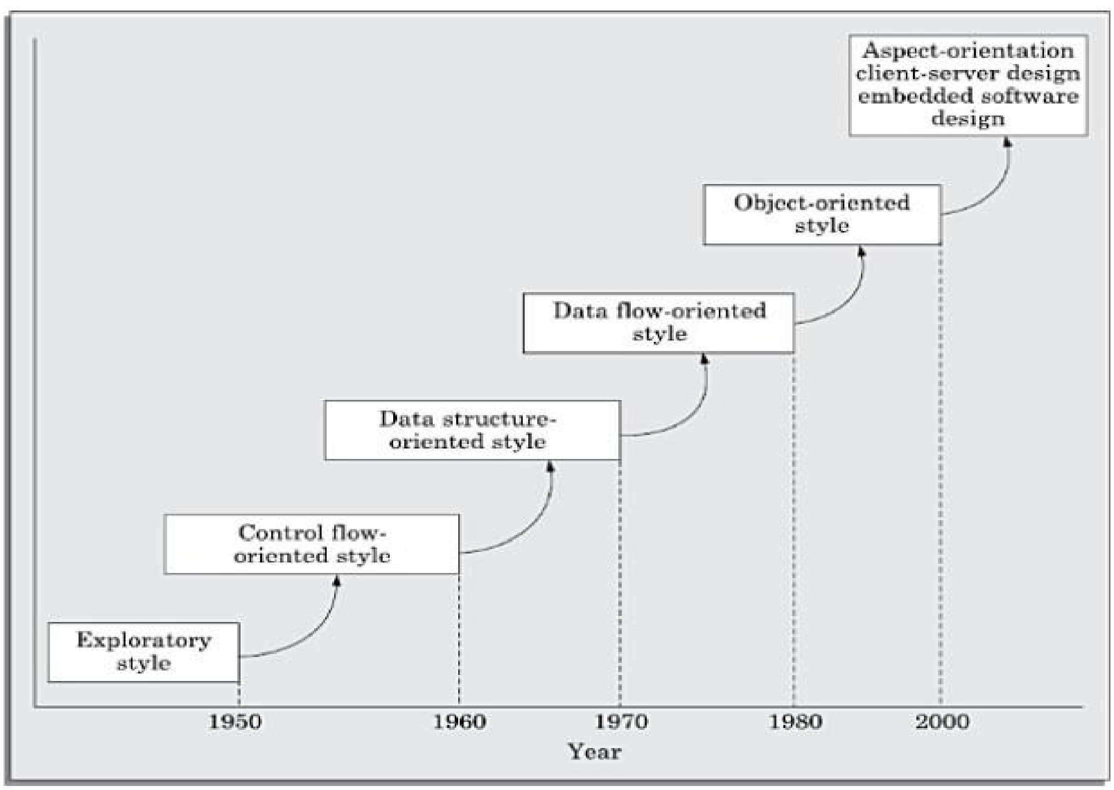


Figure 1.12: Evolution of software design techniques.

The improvements to the software design methodologies over the last five decades have indeed been remarkable. These new techniques include life cycle models,specification techniques,project management techniques,testing techniques,debugging techniques,quality assurance techniques,software measurement techniques,computer aided software engineering(CASE)tools,etc

Software life cycle: It is well known that all living organisms undergo a life cycle. For example when a seed is planted,It germinates, grows into a full tree,and finally dies.Based on this concept of a biological life cycle,the term software life cycle has been defined to imply the different stages (or phases)over which a software evolves from an initial customer request for it,to a fully developed software,and finally to a stage where it is no longer useful to any user, and then it is discarded.

Software Development life cycle

software life cycle model (also termed process model) is a pictorial and diagrammatic representation of the software life cycle. A life cycle model represents all the methods required to make a software product transit through its life cycle stages. SDLC is a systematic process for building software that ensures the quality and correctness of the software built. SDLC process aims to produce high-quality software that meets customer expectations. The system development should be complete in the pre-defined time frame and cost. SDLC consists of a detailed plan which explains how to plan, build, and maintain specific software. Every phase of the SDLC life Cycle has its own process and deliverables that feed into the next phase. SDLC stands for Software Development Life Cycle and is also referred to as the Application Development life-cycle.

- It offers a basis for project planning, scheduling, and estimating
- Provides a framework for a standard set of activities and deliverables
- It is a mechanism for project tracking and control
- Increases visibility of project planning to all involved stakeholders of the development process
- Increased and enhance development speed
- Improved client relations
- Helps you to decrease project risk and project management plan overhead.

Requirement analysis--→ Feasibility study--→ Design--→ Coding--→ Testing--→ Installation/Deployment--→ Maintenance

- Phase 1: Requirement collection and analysis
- Phase 2: Feasibility study
- Phase 3: Design
- Phase 4: Coding
- Phase 5: Testing
- Phase 6: Installation/Deployment
- Phase 7: Maintenance

Phase 1: Requirement collection and analysis:

The requirement is the first stage in the SDLC process. It is conducted by the senior team members with inputs from all the stakeholders and domain experts in the industry. Planning for the quality assurance requirements and recognization of the risks involved is also done at this stage.

This stage gives a clearer picture of the scope of the entire project and the anticipated issues, opportunities, and directives which triggered the project.

Requirements Gathering stage need teams to get detailed and precise requirements. This helps companies to finalize the necessary timeline to finish the work of that system.

Phase 2: Feasibility study:

Once the requirement analysis phase is completed the next sdlc step is to define and document software needs. This process conducted with the help of 'Software Requirement Specification' document also known as 'SRS' document. It includes everything which should be designed and developed during the project life cycle.

There are mainly five types of feasibilities checks:

- **Economic:** Can we complete the project within the budget or not?
- **Legal:** Can we handle this project as cyber law and other regulatory framework/compliances.
- **Operation feasibility:** Can we create operations which is expected by the client?
- **Technical:** Need to check whether the current computer system can support the software
- **Schedule:** Decide that the project can be completed within the given schedule or not.

Phase 3: Design:

In this third phase, the system and software design documents are prepared as per the requirement specification document. This helps define overall system architecture.This design phase serves as input for the next phase of the model.There are two kinds of design documents developed in this phase:

High-Level Design (HLD)

- Brief description and name of each module
- An outline about the functionality of every module
- Interface relationship and dependencies between modules
- Database tables identified along with their key elements
- Complete architecture diagrams along with technology details

Low-Level Design(LLD)

- Functional logic of the modules

- Database tables, which include type and size
- Complete detail of the interface
- Addresses all types of dependency issues
- Listing of error messages
- Complete input and outputs for every module

Phase 4: Coding:

Once the system design phase is over, the next phase is coding. In this phase, developers start build the entire system by writing code using the chosen programming language. In the coding phase, tasks are divided into units or modules and assigned to the various developers. It is the longest phase of the Software Development Life Cycle process.

In this phase, Developer needs to follow certain predefined coding guidelines. They also need to use programming tools like compiler, interpreters, debugger to generate and implement the code.

Phase 5: Testing:

Once the software is complete, and it is deployed in the testing environment. The testing team starts testing the functionality of the entire system. This is done to verify that the entire application works according to the customer requirement.

During this phase, QA and testing team may find some bugs/defects which they communicate to developers. The development team fixes the bug and send back to QA for a re-test. This process continues until the software is bug-free, stable, and working according to the business needs of that system.

Phase 6: Installation/Deployment:

Once the software testing phase is over and no bugs or errors left in the system then the final deployment process starts. Based on the feedback given by the project manager, the final software is released and checked for deployment issues if any.

Phase 7: Maintenance:

Once the system is deployed, and customers start using the developed system, following 3 activities occur

- Bug fixing - bugs are reported because of some scenarios which are not tested at all
- Upgrade - Upgrading the application to the newer versions of the Software
- Enhancement - Adding some new features into the existing software

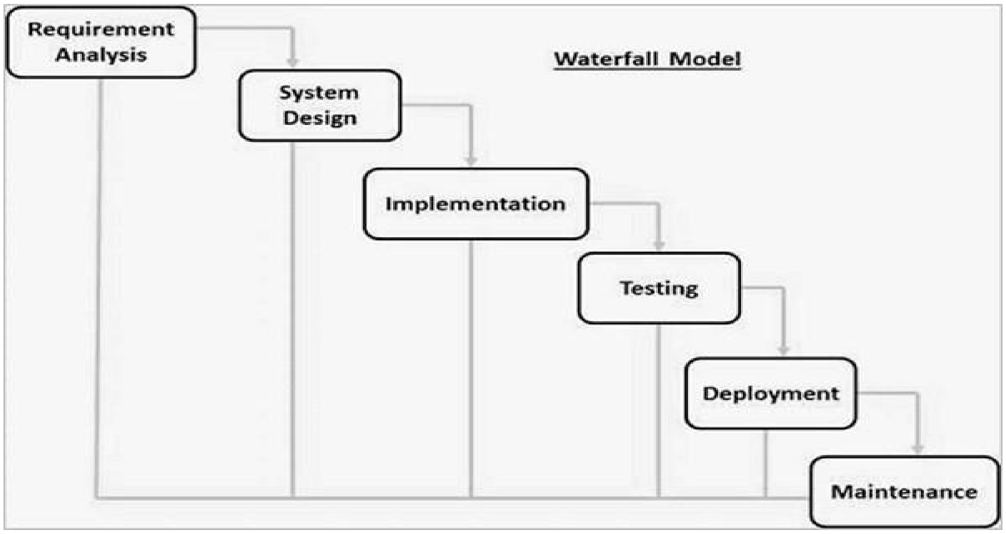
Iterative waterfall model:

The waterfall is a widely accepted SDLC model. In this approach, the whole process of the software development is divided into various phases of SDLC. In this SDLC model, the outcome of one phase acts as the input for the next phase.

This SDLC model is documentation-intensive, with earlier phases documenting what need be performed in the subsequent phases.

Waterfall approach was first SDLC Model to be used widely in Software Engineering to ensure success of the project. In "The Waterfall" approach, the whole process of software development is divided into separate phases. In this Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.

The following illustration is a representation of the different phases of the Waterfall Model.



The sequential phases in Waterfall model are −

- **Requirement Gathering and analysis** − All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
- **System Design** − The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.
- **Implementation** − With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.
- **Integration and Testing** − All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
- **Deployment of system** − Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.
- **Maintenance** − There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

Advantages and Disadvantages of Waterfall-Model

Advantages	Dis-Advantages
• Before the next phase of development, each phase must be completed	• Error can be fixed only during the phase
• Suited for smaller projects where requirements are well defined	• It is not desirable for complex project where requirement changes frequently
• They should perform quality assurance test (Verification and Validation) before completing each stage.	• Testing period comes quite late in the developmental process
• Elaborate documentation is done at every phase of the software's development cycle	• Documentation occupies a lot of time of developers and testers
• Project is completely dependent on project team with minimum client intervention	• Clients valuable feedback cannot be included with ongoing development phase
• Any changes in software is made during the process of the development	• Small changes or errors that arise in the completed software may cause a lot of problems

Prototyping Model:

Prototyping Model is a software development model in which prototype is built, tested, and reworked until an acceptable prototype is achieved. It also creates base to produce the final system or software. It works best in scenarios where the project's requirements are not known in detail. It is an iterative, trial and error method which takes place between developer and client.

Requirements→Quick Design→Build prototype→User Evaluation→Refining prototype→Implement and prototype

Step 1: Requirements gathering and analysis

A prototyping model starts with requirement analysis. In this phase, the requirements of the system are defined in detail. During the process, the users of the system are interviewed to know what is their expectation from the system.

Step 2: Quick design

The second phase is a preliminary design or a quick design. In this stage, a simple design of the system is created. However, it is not a complete design. It gives a brief idea of the system to the user. The quick design helps in developing the prototype.

Step 3: Build a Prototype

In this phase, an actual prototype is designed based on the information gathered from quick design. It is a small working model of the required system.

Step 4: Initial user evaluation

In this stage, the proposed system is presented to the client for an initial evaluation. It helps to find out the strength and weakness of the working model. Comment and suggestion are collected from the customer and provided to the developer.

Step 5: Refining prototype

If the user is not happy with the current prototype, you need to refine the prototype according to the user's feedback and suggestions.

This phase will not over until all the requirements specified by the user are met. Once the user is satisfied with the developed prototype, a final system is developed based on the approved final prototype.

Step 6: Implement Product and Maintain

Once the final system is developed based on the final prototype, it is thoroughly tested and deployed to production. The system undergoes routine maintenance for minimizing downtime and prevent large-scale failures.

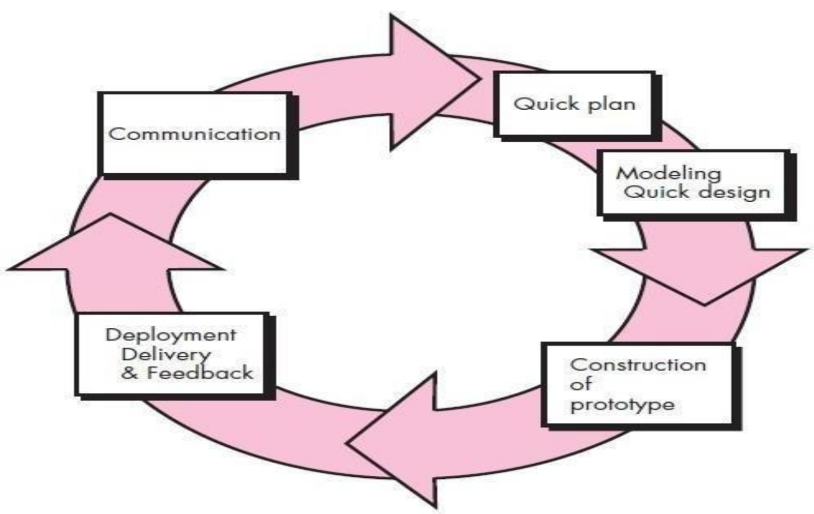


Fig : prototyping paradigm

Phases of prototyping Model

1. **Requirement Identification**: Here identification of product requirements is cleared in details. It is done through interview some product's future users and other members of the departments.

2. **Design Stage**: A first-round design is created in this stage for the new system.

3. **Build the Initial Prototype**: An initial prototype the target software is built from the original design. Working off all the product components may not be perfect or accurate. The first sample model is tailored as per the comments were given by the users and based on that the second prototype is built.

4. **Review of the Prototype**: After the product completes all the iterations of the update, it is presented to the customer or other stakeholders of the project. The response is accumulated in an organized way so that they can be used for further system enhancements.

5. **Iteration and Enhancement of Prototype**: Once the review of the product is done, it is set for further enhancement based on factors like - time, workforce as well as budget. Also, the technical feasibility of actual implementation is checked.

The final system is thoroughly evaluated and tested. Periodic maintenance is conceded on an ongoing basis for preventing large-scale breakdown as well as to minimize downtime.

There are four types of model available:

A) Rapid Throwaway Prototyping –

This technique offers a useful method of exploring ideas and getting customer feedback for each of them. In this method, a developed prototype need not necessarily be a part of the ultimately accepted prototype. Customer feedback helps in preventing unnecessary design faults and hence, the final prototype developed is of better quality.

B) Evolutionary Prototyping –

In this method, the prototype developed initially is incrementally refined on the basis of customer feedback till it finally gets accepted. In comparison to Rapid Throwaway Prototyping, it offers a better approach which saves time as well as effort. This is because developing a prototype from scratch for every iteration of the process can sometimes be very frustrating for the developers.

C) Incremental Prototyping – In this type of incremental Prototyping, the final expected product is broken into different small pieces of prototypes and being developed individually. In the end, when all individual pieces are properly developed, then the different prototypes are collectively merged into a single final product in their predefined order. It's a very efficient approach which reduces the complexity of the development process, where the goal is divided into sub-parts and each sub-part is developed individually. The time interval between the project begin and final delivery is substantially reduced because all parts of the system are prototyped and tested simultaneously.

D) Extreme Prototyping – This method is mainly used for web development. It is consists of three sequential independent phases:

D.1) In this phase a basic prototype with all the existing static pages are presented in the HTML format.

D.2) In the 2nd phase, Functional screens are made with a simulate data process using a prototype services layer.

D.3) This is the final step where all the services are implemented and associated with the final prototype.

Advantages –

- The customers get to see the partial product early in the life cycle. This ensures a greater level of customer satisfaction and comfort.
- New requirements can be easily accommodated as there is scope for refinement.
- Missing functionalities can be easily figured out.
- Errors can be detected much earlier thereby saving a lot of effort and cost, besides enhancing the quality of the software.
- The developed prototype can be reused by the developer for more complicated projects in the future.
- Flexibility in design.

Disadvantages –

- Costly w.r.t time as well as money.
- There may be too much variation in requirements each time the prototype is evaluated by the customer.
- Poor Documentation due to continuously changing customer requirements.
- It is very difficult for developers to accommodate all the changes demanded by the customer.
- There is uncertainty in determining the number of iterations that would be required before the prototype is finally accepted by the customer.
- After seeing an early prototype, the customers sometimes demand the actual product to be delivered soon.
- Developers in a hurry to build prototypes may end up with sub-optimal solutions.
- The customer might lose interest in the product if he/she is not satisfied with the initial prototype.

Use –
The Prototyping Model should be used when the requirements of the product are not clearly understood or are unstable. It can also be used if requirements are changing quickly. This model can be successfully used for developing user interfaces, high technology software-intensive systems, and systems with complex algorithms and interfaces. It is also a very good choice to demonstrate the technical feasibility of the product.

EVOLUTIONARY MODEL:

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.

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.

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:

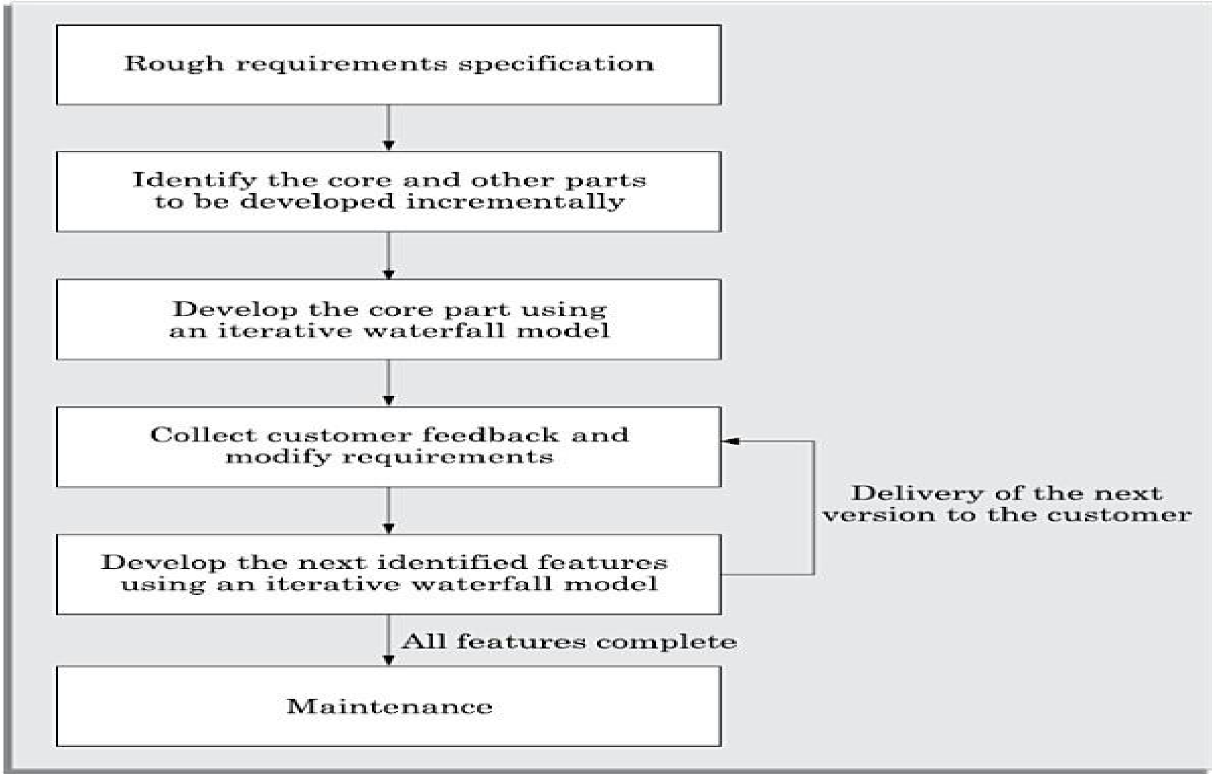
🕐 Large projects where you can easily find modules for incremental implementation. Often used when the customer wants to start using the core features rather than waiting for the full software.

🕐 Also used in object oriented software development because the system can be easily portioned into units in terms of objects.

Advantages: 1.User gets a chance to experiment partially developed system

2.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.



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

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

Advantages

The evolutionary model of development has several advantages.Two important advantages of using this model are the following:

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

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

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

- **Feature division into incremental parts can be non-trivial:** For many development projects, especially for small-sized projects, it is difficult to divide the required features into several parts that can be incrementally implemented and delivered. Further, even for larger problems, often the features are so interwined and dependent on each other that even an expert would need considerable effort to plan the incremental deliveries.
- ***Ad hoc* design:** Since at a time design for only the current increment is done, the design can become ad hoc without specific attention being paid to maintainability and optimality. Obviously, for moderate sized problems and for those for which the customer requirements are clear, the iterative waterfall model can yield a better solution.

The evolutionary model is well-suited to use in object-oriented software development projects.

Evolutionary model is appropriate for object-oriented development project, since it is easy to partition the software into stand alone units in terms of the classes. Also, classes are more or less self contained units that can be developed independently.

The Spiral Model : Originally proposed by **Barry Boehm**, the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software. Boehm describes the model in the following manner.

The spiral development model is a **risk-driven process model** generator that is used to **guide multi-stakeholder concurrent engineering** of software intensive systems. It has **two** main distinguishing features. One is a ***cyclic approach*** for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of ***anchor point milestones*** for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

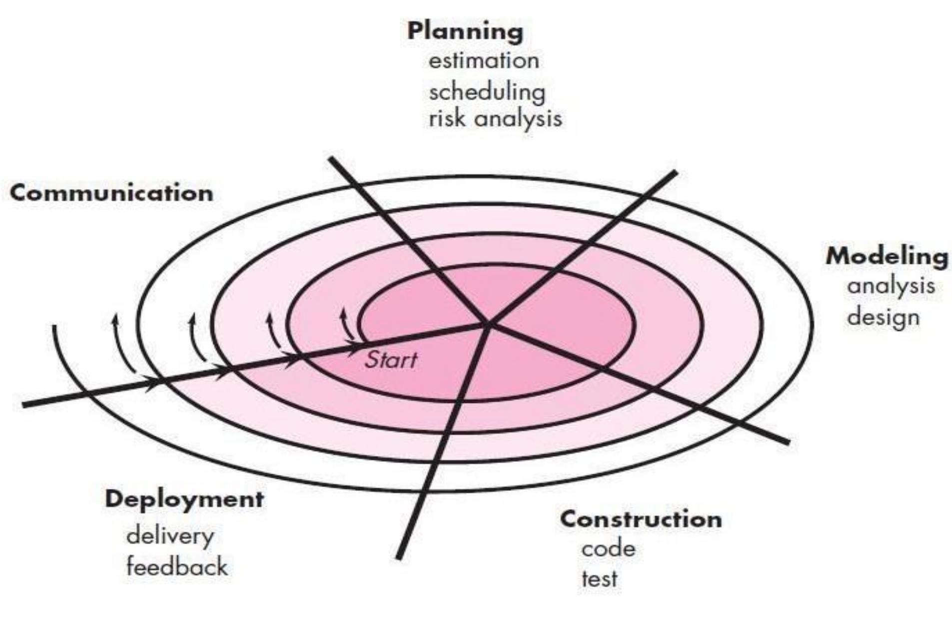


Fig : The Spiral Model

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.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.!!br0ken!!

During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a set of **framework activities** defined by the software engineering team. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a **clockwise** direction, beginning at the **center**. Risk is considered as each revolution is made. ***Anchor point milestones*** are a combination of work products and conditions that are attained along the path of the spiral are noted for each evolutionary pass.

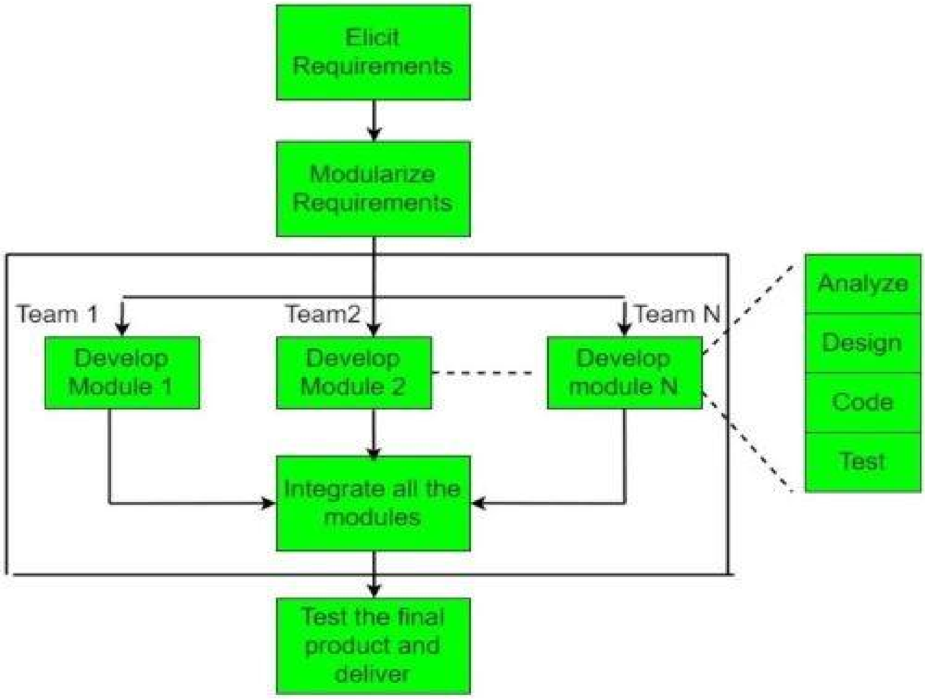
The first circuit around the spiral might result in the development of a **product** specification; subsequent passes around the spiral might be used to develop a **prototype** and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan.The spiral model can be adapted to apply throughout the life of the computer software. Therefore, the first circuit around the spiral might represent a "**concept development project**" that starts at the core of the spiral and continues for multiple iterations until concept development is complete. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a "**product enhancement project.**"

The spiral model is a **realistic approach** to the development of **large-scale systems** and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.

RAD (Rapid Application Development) Model:

The Rapid Application Development Model was first proposed by IBM in 1980's. The critical feature of this model is the use of powerful development tools and techniques.A software project can be implemented using this model if the project can be broken down into small modules wherein each module can be assigned independently to separate teams. These modules can finally be combined to form the final product.

Development of each module involves the various basic steps as in waterfall model i.e analyzing, designing, coding and then testing, etc. as shown in the figure.Another striking feature of this model is a short time span i.e the time frame for delivery(time-box) is generally 60-90 days.



The use of powerful developer tools such as JAVA, C++, Visual BASIC, XML, etc. is also an integral part of the projects.

This model consists of 4 basic phases:

1.Requirements Planning –It involves the use of various techniques used in requirements elicitation like brainstorming, task analysis, form analysis, user scenarios, FAST (Facilitated Application Development Technique), etc. It also consists of the entire structured plan describing the critical data, methods to obtain it and then processing it to form final refined model.

2.User Description –This phase consists of taking user feedback and building the prototype using developer tools. In other words, it includes re-examination and validation of the data collected in the first phase. The dataset attributes are also identified and elucidated in this phase.

3.Construction –In this phase, refinement of the prototype and delivery takes place. It includes the actual use of powerful automated tools to transform process and data models into the final working product. All the required modifications and enhancements are too done in this phase.

4.Cutover –All the interfaces between the independent modules developed by separate teams have to be tested properly. The use of powerfully automated tools and subparts makes testing easier. This is followed by acceptance testing by the user.

The process involves building a rapid prototype, delivering it to the customer and the taking feedback. After validation by the customer, SRS document is developed and the design is finalised.

Advantages –

- Use of reusable components helps to reduce the cycle time of the project.
- Feedback from the customer is available at initial stages.
- Reduced costs as fewer developers are required.
- Use of powerful development tools results in better quality products in comparatively shorter time spans.
- The progress and development of the project can be measured through the various stages.
- It is easier to accommodate changing requirements due to the short iteration time spans.

Disadvantages –

- The use of powerful and efficient tools requires highly skilled professionals.
- The absence of reusable components can lead to failure of the project.
- The team leader must work closely with the developers and customers to close the project in time.
- The systems which cannot be modularized suitably cannot use this model.
- Customer involvement is required throughout the life cycle.
- It is not meant for small scale projects as for such cases, the cost of using automated tools and techniques may exceed the entire budget of the project.

Applications –

1. This model should be used for a system with known requirements and requiring short development time.
2. It is also suitable for projects where requirements can be modularized and reusable components are also available for development.
3. The model can also be used when already existing system components can be used in developing a new system with minimum changes.
4. This model can only be used if the teams consist of domain experts. This is because relevant knowledge and ability to use powerful techniques is a necessity.
5. The model should be chosen when the budget permits the use of automated tools and techniques required.

Agile Model

In earlier days Iterative Waterfall model was very popular to complete a project. But nowadays developers face various problems while using it to develop a software. The main difficulties included handling change requests from customers during project development and the high cost and time required to incorporate these changes. To overcome these drawbacks of Waterfall model, in the mid-1990s the Agile Software Development model was proposed. "**Agile process model**" refers to a software development approach based on iterative development. Agile methods break tasks into smaller iterations, or parts do not directly involve long term planning. The project scope and requirements are laid down at the beginning of the development process. Plans regarding the number of iterations, the duration and the scope of each iteration are clearly defined in advance.

Each iteration is considered as a short time "frame" in the Agile process model, which typically lasts from one to four weeks. The division of the entire project into smaller parts helps to minimize the project risk and to reduce the overall project delivery time requirements. Each iteration involves a team working through a full software development life cycle including planning, requirements analysis, design, coding, and testing before a working product is demonstrated to the client.

Phases of Agile Model: Following are the phases in the Agile model are as follows:

1.Requirements gathering

2.Design the requirements

3.Construction/ iteration

4.Testing/ Quality assurance

5.Deployment

6.Feedback

1. Requirements gathering: In this phase, you must define the requirements. You should explain business opportunities and plan the time and effort needed to build the project. Based on this information, you can evaluate technical and economic feasibility.

2. Design the requirements: When you have identified the project, work with stakeholders to define requirements. You can use the user flow diagram or the high-level UML diagram to show the work of new features and show how it will apply to your existing system.

3. Construction/ iteration: When the team defines the requirements, the work begins. Designers and developers start working on their project, which aims to deploy a working product. The product will undergo various stages of improvement, so it includes simple, minimal functionality.

4. Testing: In this phase, the Quality Assurance team examines the product's performance and looks for the bug.

5. Deployment: In this phase, the team issues a product for the user's work environment.

6. Feedback: After releasing the product, the last step is feedback. In this, the team receives feedback about the product and works through the feedback.

Agility principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self– organizing teams.

12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Human Factors: Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that *the process molds to the needs of the people and team*

1.Competence. In an agile development context, "competence" encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.

2.Common focus. Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.

3.Collaboration. Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.

4.Decision-making ability. Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.

5.Fuzzy problem-solving ability. Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change.

6.Mutual trust and respect. The agile team must become what DeMarco and Lister call a "jelled" team. A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of the parts."

7.Self-organization. In the context of agile development, self-organization implies **three** things:

1. The agile team organizes itself for the work to be done,
2. The team organizes the process to best accommodate its local environment
3. The team organizes the work schedule to best achieve delivery of the software increment. Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale.

1.EXTREME PROGRAMMING (XP):

***Extreme Programming* (XP),** the most widely used approach to agile software development, emphasizes business results first and takes an incremental, get-something-started approach to building the product, using continual testing and revision.

1.1 XP Values

Beck defines a set of **five** *values* that establish a foundation for all work performed as part of XP—**communication, simplicity, feedback, courage, and respect**. Each of these values is used as a driver for specific XP **activities, actions, and tasks**.

Communication :In order to achieve effective **communication** between software engineers and other stakeholders, XP emphasizes close, yet informal collaboration between customers and developers, the establishment of effective metaphors3 for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.

Simplicity :To achieve **simplicity***,* XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code). If the design must be improved, it can be *refactored* at a later time.

Feedback: is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy the software provides the agile team with feedback. XP makes use of the ***unit test*** as its primary testing tactic. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality.

Courage : Beck argues that strict adherence to certain XP practices demands **courage.** A better word might be ***discipline***. An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.

Respect:By following each of these values, the agile team inculcates **respect** among it members, between other stakeholders and team members, and indirectly, for the software itself. As theyachieve successful delivery of software increments, the team develops growing respect for the XP process.

1.2 The XP Process

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Following figure illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity.

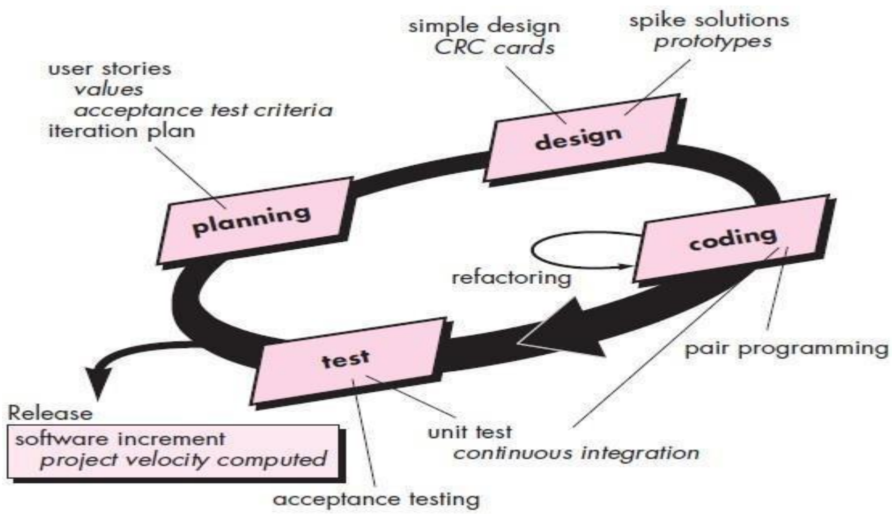


Fig : The Extreme Programming process

- **Planning:**The planning activity (also called *the planning game*) begins with *listening*—a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality.

- **Design:**XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a ***spike solution***, the design prototype is implemented and evaluated. XP encourages ***refactoring***—a construction technique that is also a method for design optimization.

- r**efactoring** in the following manner: Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure. It is a disciplined way to clean up code [that minimizes the chances of introducing bugs].
- **Coding.** After stories are developed and preliminary design work is done, the team does *not* move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

- A key concept during the coding activity is *pair programming*. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real time problem solving (two heads are often better than one) and real-time quality assurance.
- **Testing.** The creation of unit tests before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated. This encourages a regression testing strategy whenever code is modified. As the individual unit tests are organized into a "universal testing suite" integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells states: "Fixing small problems every few hours takes less time than fixing huge problems just before the deadline."
- XP ***acceptance tests***, also called ***customer tests,*** are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

1.3 Industrial XP:

Industrial Extreme Programming (IXP) in the following manner: "IXP is an organic evolution of XP. It is imbued with XP's minimalist, customer-centric, test- driven spirit. IXP differs most from the original XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices." IXP incorporates six new practices that are designed to help ensure that an XP project works successfully for significant projects within a large organization.

- **Readiness assessment.** Prior to the initiation of an IXP project, the organization should conduct a *readiness assessment.* The assessment ascertains whether (1) an appropriate development environment exists to support IXP, (2) the team will be populated by the proper set of stakeholders, (3) the organization has a distinct quality program and supports continuous improvement, (4) the organizational culture will support the new values of an agile team, and (5) the broader project community will be populated appropriately.
- **Project community.** Classic XP suggests that the right people be used to populate the agile team to ensure success. The implication is that people on the team must be well- trained, adaptable and skilled, and have the proper temperament to contribute to a self- organizing team. When XP is to be applied for a significant project in a large organization, the concept of the "team" should morph into that of a *community.* In IXP, the community members and their roles should be explicitly defined and mechanisms for communication and coordination between community members should be established.
- **Project chartering.** The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the organization. Chartering also examines the context of the project to determine how it complements, extends, or replaces existing systems or processes.
- **Test-driven management.** An IXP project requires measurable criteria for assessing the state of the project and the progress that has been made to date. Test-driven management establishes a series of measurable "destinations" and then defines mechanisms for determining whether or not these destinations have been reached.
- **Retrospectives.** An IXP team conducts a specialized technical review after a software increment is delivered. Called a *retrospective,* the review examines "issues, events, and lessons-learned" across a software increment and/or the entire software release. The intent is to improve the IXP process.
- **Continuous learning.** Because learning is a vital part of continuous process improvement, members of the XP team are encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher quality product.

1.4 THE XP DEBATE:

All new process models and methods spur worthwhile discussion and in some instances heated debate. Extreme Programming has done both. In an interesting book that examines the efficacy of XP Stephens and Rosenberg argue that many XP practices are worthwhile, but others have been overhyped, and a few are problematic.

- **Requirements volatility.** Because the customer is an active member of the XP team, changes to requirements are requested informally. As a consequence, the scope of the project can change and earlier work may have to be modified to accommodate current needs. Proponents argue that this happens regardless of the process that is applied and that XP provides mechanisms for controlling scope creep.

•**Conflicting customer needs.** Many projects have multiple customers, each with his own set of needs. In XP, the team itself is tasked with assimilating the needs of different customers, a job that may be beyond their scope of authority.

•***Requirements are expressed informally*.** User stories and acceptance tests are the only explicit manifestation of requirements in XP. Critics argue that a more formal model or specification is often needed to ensure that omissions, inconsistencies, and errors are uncovered before the system is built.

•**Lack of formal design.** XP deemphasizes the need for architectural design and in many instances, suggests that design of all kinds should be relatively informal. Critics argue that when complex systems are built, design must be emphasized to ensure that the overall structure of the software will exhibit quality and maintainability. XP proponents suggest that the incremental nature of the XP process limits complexity (simplicity is a core value) and therefore reduces the need for extensive design.

OTHER AGILE PROCESS MODELS

Other agile process models have been proposed and are in use across the industry.

•Adaptive Software Development (ASD)
•Scrum
•Dynamic Systems Development Method (DSDM)
•Crystal
•Feature Drive Development (FDD)
•Lean Software Development (LSD)
•Agile Modeling (AM)
•Agile Unified Process (AUP)

1.Adaptive Software Development (ASD)

Adaptive Software Development (ASD) has been proposed by Jim Highsmith as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization.

•High smith argues that an agile, adaptive development approach based on collaboration is "as much a source of *order* in our complex interactions as discipline and engineering." He defines an ASD "life cycle" that incorporates three phases, speculation, collaboration, and learning.

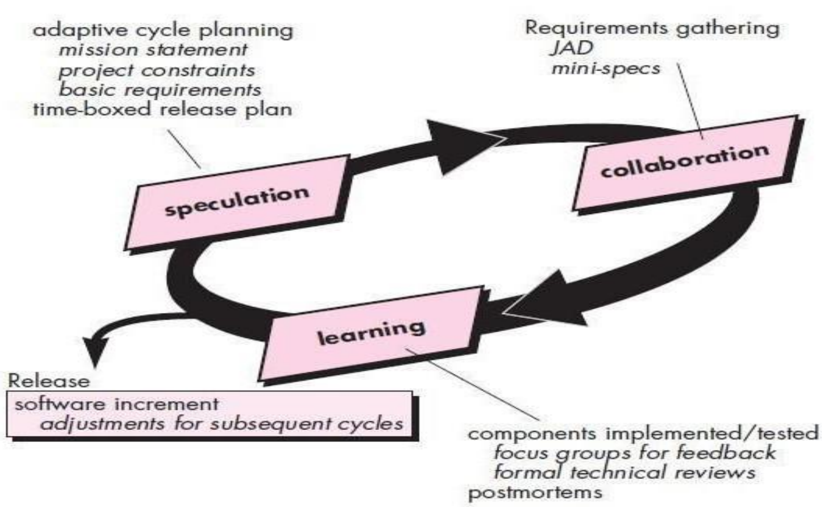


Fig : Adaptive software development

During ***speculation***, the project is initiated and ***adaptive cycle planning*** is conducted. Adaptive cycle planning uses project initiation information—the customer's mission statement, project constraints

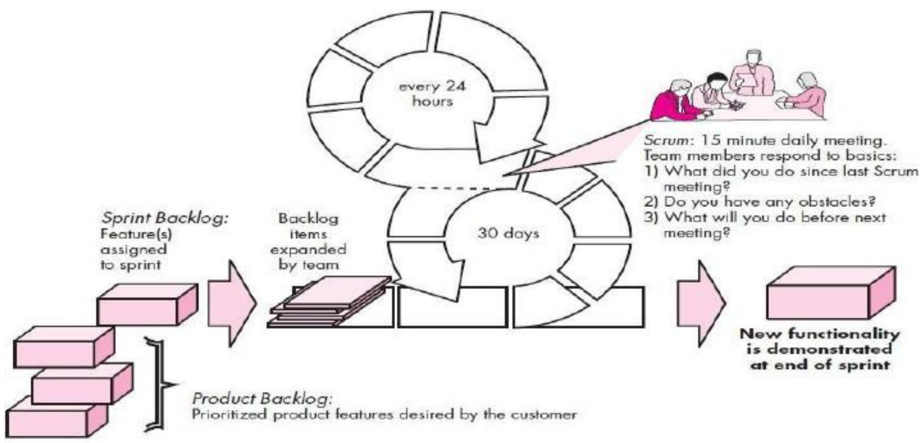
•motivated people use ***collaboration*** in a way that multiplies their talent and creative output beyond their absolute numbers. This approach is a recurring theme in all agile methods. But collaboration is not easy. It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking.

•It is, above all, a matter of trust. People working together must trust one another to (1) criticize without animosity, (2) assist without resentment, (3) work as hard as or harder than they do, (4) have the skill set to contribute to the work at hand, and (5) communicate problems or concerns in a way that leads to effective action.As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on **"learning**" as much as it is on progress toward a completed cycle.

•ASD teams learn in **three** ways: **focus groups, technical reviews , and project postmortems**. ASD's overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning yield software project teams that have a much higher likelihood of success.

2.SCRUM

Scrum is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks occur within a process pattern called a ***sprint.*** The work conducted within a sprint is adapted to the problem at hand and is defined and often modified.in real time by the Scrum team. The overall flow of the Scrum process is illustrated in following figure



•**Scrum** emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development actions:

•**Backlog**—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time. The product manager assesses the backlog and updates priorities as required.

•**Sprints**—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box (typically 30 days). Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.

•**Scrum meetings**—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members

•What did you do since the last team meeting?

•What obstacles are you encountering?

•What do you plan to accomplish by the next team meeting?

A team leader, called a ***Scrum master,*** leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to "**knowledge socialization**"

Demos—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

3.Dynamic Systems Development Method (DSDM)

The *Dynamic Systems Development Method* (DSDM) is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment" The DSDM philosophy is borrowed from a modified version of the **Pareto principle—80 percent of an application can be delivered in 20 percent of the time.** It would take to deliver the complete (100 percent) application. DSDM is an iterative software process in which each iteration follows the 80 percent rule. That is, only enough work is required for each increment tofacilitate movement to the next increment. The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

The *DSDM life cycle* that defines **three** different iterative cycles, preceded by **two** additional life cycle activities:

•**Feasibility study**—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process

•**Business study**—establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.

•**Functional model iteration**—produces a set of incremental prototypes that demonstrate functionality for the customer.

•**Design and build iteration**—revisits prototypes built during **functional model iteration** to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, functional model iteration and design and build iteration occur concurrently.

•**Implementation**—places the latest software increment into the operational environment. It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place. In either case, DSDM development work continues by returning to the functional model iteration activity.

4 .CRYSTAL

Alistair Cockburn and Jim Highsmith created the *Crystal family of agile methods* in order to achieve a software development approach that puts a premium on "maneuverability" during what Cockburn characterizes as "a resource limited, cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game"

•The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects. The intent is to allow agile teams to select the member of the crystal family that is most appropriate for their project and environment.

5.FEATURE DRIVEN DEVELOPMENT (FDD)

Feature Driven Development (FDD) was originally conceived by Peter Coad and his colleagues as a practical process model for object-oriented software engineering. Stephen Palmer and John Felsing have extended and improvedCoad's work, describing an adaptive, agile process that can be applied to moderately sized and larger software projects.

Like other agile approaches, FDD adopts a philosophy that (1) emphasizes collaboration among people on an FDD team; (2) manages problem and project complexity using feature- based decomposition followed by the integration of software increments, and (3) communication of technical detail using verbal, graphical, and text-based means.

FDD emphasizes software quality assurance activities by encouraging an incremental development strategy, the use of design and code inspections, the application of software quality assurance audits, the collection of metrics, and the use of patterns (for analysis, design, and construction).In the context of FDD, a *feature* "is a client-valued function that can be implemented in two weeks or less" The emphasis on the definition of features provides the following benefits: Because features are small blocks of deliverable functionality, users can describe them more easily; understand how they relate to one another more readily; and better review them for ambiguity, error, or omissions.Features can be organized into a hierarchical business-related grouping.

Since a feature is the FDD deliverable software increment, the team develops operational featuresevery two weeks.Because features are small, their design and code representations are easier to inspect effectively.

Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

- Coad and his colleagues suggest the following template for defining a feature:

•**<action>** the **<result> <by for of to>** a(n) **<object>**

•where an **<object>** is "a person, place, or thing

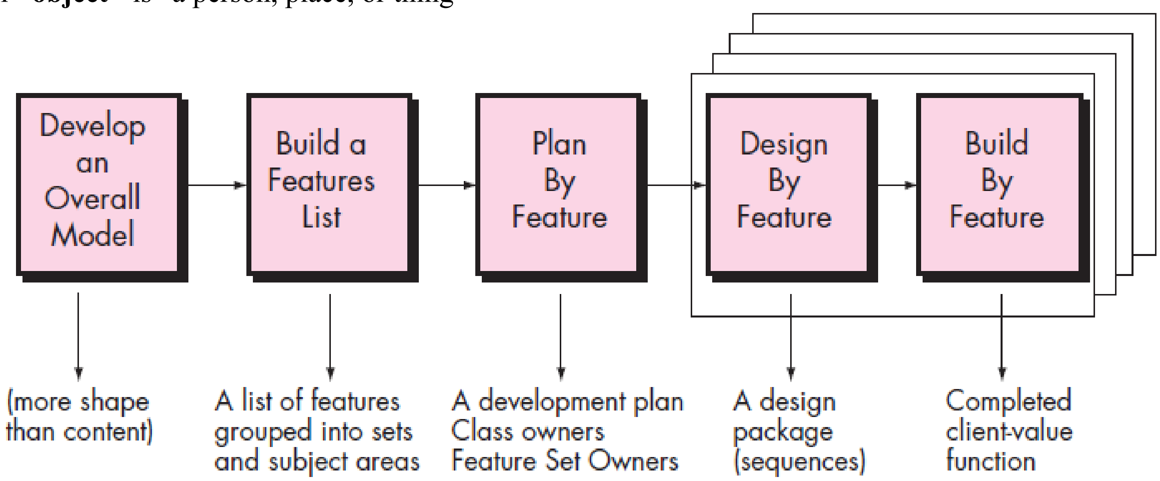


Fig : Feature Driven Development (FDD)

FDD provides greater emphasis on project management guidelines and techniques than many other agile methods. FDD defines **six** milestones during the design and implementation of a feature: "**design walkthrough, design, design inspection, code, code inspection, promote to build**"

6. LEAN SOFTWARE DEVELOPMENT (LSD)

Lean Software Development (LSD) has adapted the principles of lean manufacturing to the world of software engineering. The lean principles that inspire the LSD process can be summarized as **eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people, and optimize the whole**. Each of these principles can be adapted to the software process.

7.AGILE MODELING (AM)

Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. Agile models are more effective than traditional models because they are just barely good, they don't have to be perfect

Agile modeling adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refactor. The team must also have the humility to recognize that technologists do not have all the answers and that business experts and other stakeholders should be respected and embraced.

Agile Modeling suggests a wide array of "core" and "supplementary" modeling principles, those that make AM unique are :

- **Model with a purpose.** A developer who uses AM should have a specific goal in mind before creating the model. Once the goal for the model is identified, the type of notation to be used and level of detail required will be more obvious.
- **Use multiple models.** There are many different models and notations that can be used to describe software. Only a small subset is essential for most projects. AM suggests that to provide needed insight, each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.
- **Travel light.** As software engineering work proceeds, keep only those models that will provide long-term value and jettison the rest. Every work product that is kept must be maintained as changes occur. This represents work that slows the team down. Ambler notes that "Every time you decide to keep a model you trade-off agility for the convenience of having that information available to your team in an abstract manner
- **Content is more important than representation.** Modeling should impart information to its intended audience. A syntactically perfect model that imparts little useful content is not as valuable as a model with flawed notation that nevertheless provides valuable content for its audience.
- **Know the models and the tools you use to create them.** Understand the strengths and weaknesses of each model and the tools that are used to create it.
- **Adapt locally.** The modeling approach should be adapted to the needs of the agile team.

8.AGILE UNIFIED PROCESS (AUP)

The Agile Unified Process (AUP) adopts a "serial in the large" and "iterative in the small" philosophy for building computer-based systems. By adopting the classic UP phased activities— inception, elaboration, construction, and transition—AUP provides a serial overlay that enables a team to visualize the overall process flow for a software project. However, within each of the activities, the team iterates to achieve agility and to deliver meaningful software increments to end users as rapidly as possible. **Each AUP iteration addresses the following activities.**

- **Modeling**. UML representations of the business and problem domains are created.
- **Implementation.** Models are translated into source code.
- **Testing**. Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.
- **Deployment.** Like the generic process activity deployment in this context focuses on the delivery of a software increment and the acquisition of feedback from end users.
- **Configuration and project management.** In the context of AUP, configuration management addresses change management, risk management, and the control of any persistent work products that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.

Environment management. Environment management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

Software project management :project planning

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.

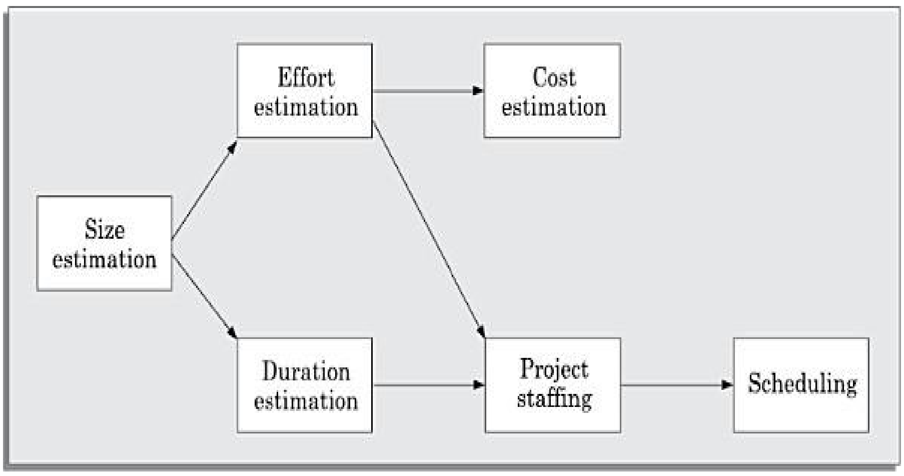


Fig:Precedence ordering among planning activities

Metrics for software project size estimation:Accurate estimation of the problem size is fundamental to satisfactory estimation of effort, time duration and cost of a software project. In order to be able to accurately estimate the project size, some important metrics should be defined in terms of which the project size can be expressed. The size of a problem is obviously not the number of bytes that the source code occupies. It is neither the byte size of the executable code. The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.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.

1.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.

2.Function point (FP): Function point metric was proposed by Albrecht [1983]. This metric overcomes many of the shortcomings of the LOC metric. Since its inception in late 1970s, 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.

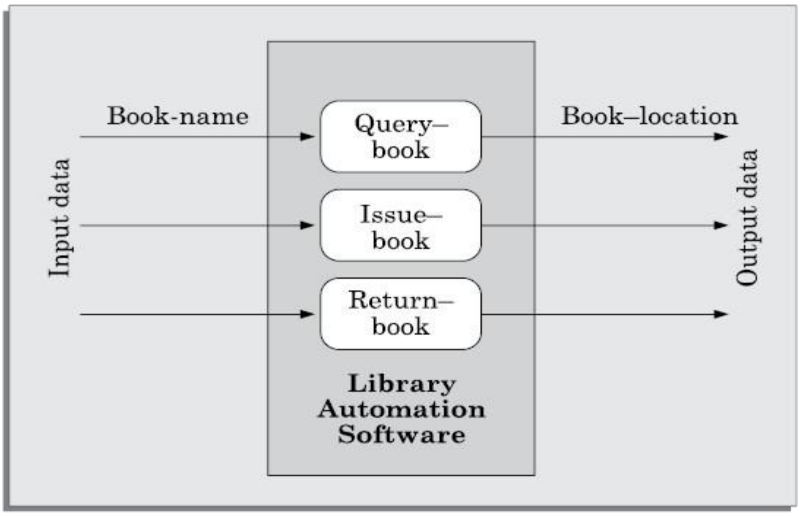


Fig: System function as a map of input data to output data

It is computed using the following three steps:

- **Step 1:** Compute the unadjusted function point (UFP) using a heuristic expression.
- **Step 2:** Refine UFP to reflect the actual complexities of the different parameters used in UFP computation.
- **Step 3:** Compute FP by further refining UFP to account for the specific characteristics of the project that can influence the entire development effort.

Step 1: UFP computation

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.

Step 2: Refine parameters

UFP computed at the end of step 1 is a gross indicator of the problem size. This UFP needs to be refined. This is possible, since each parameter (input, output, etc.) has been implicitly assumed to be of average complexity. However, this is rarely true. For example, some input values may be extremely complex, some very simple, etc. In order to take this issue into account, UFP is refined by taking into account the complexities of the parameters of UFP computation. The complexity of each parameter is graded into three broad categories—simple, average, or complex. The weights for the different parameters are determined based on the numerical values shown in Table 3.1. Based on these weights of the parameters, the parameter values in the UFP are refined. For example, rather than each input being computed as four FPs, very simple inputs are computed as three FPs and very complex inputs as six FPs

Table : Refinement of Function Point Entities			
Type	Simple	Average	Complex
Input(I)	3	4	6
Output (O)	4	5	7
Inquiry (E)	3	4	6
Number of files (F)	7	10	15
Number of interfaces	5	7	10

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

In the final step, several factors that can impact the overall project size are considered to refine the UFP computed in step 2. Examples of such project parameters that can influence the project sizes include high transaction rates, response time requirements, scope for reuse, etc. Albrecht identified 14 parameters that can influence the development effort.Each of these 14 parameters is assigned a value from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). A technical complexity factor (TCF) for the project is computed and the TCF is multiplied with UFP to yield FP. The TCF expresses the overall impact of the corresponding project parameters on the development effort. TCF is computed as (0.65+0.01*DI). As DI can vary from 0 to 84, TCF can vary from 0.65 to 1.49. Finally, FP is given as the product of UFP and TCF. That is, FP=UFP*TCF.

Example : Determine the function point measure of the size of the following supermarket software. A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number. Ea ch customer who registers for this scheme is assigned a unique customer number (CN) by the computer. Based on the generated CN, a clerk manually prepares a customer identity card after getting the market manager’s signature on it. A customer can present his customer identity card to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Rs. 10,000. The entries against the CN are reset on the last day of every year after the prize winners’ lists are generated. Assume that various project characteristics determining the complexity of software development to be average.

Answer:

Step 1: From an examination of the problem description, we find that there are two inputs, three outputs, two files, and no interfaces. Two files would be required, one for storing the customer details and another for storing the daily purchase records. Now, using equation ,we get:

$$UFP = 2 \times 4 + 3 \times 5 + 1 \times 4 + 10 \times 2 + 0 \times 10 = 47$$

Step 2: All the parameters are of moderate complexity, except the output parameter of customer registration, in which the only output is the CN value. Consequently, the complexity of the output parameter of the customer registration function can be categorized as simple. By consulting Table 3.1, we find that the value for simple output is given to be 4. The UFP can be refined as follows:

$$UFP = 3 \times 4 + 2 \times 5 + 1 \times 4 + 10 \times 2 + 0 \times 10 = 46$$

Therefore, the UFP will be 46.

Step 3: Since the complexity adjustment factors have average values, therefore the total degrees of influence would be: DI = 14 × 4 = 56

TCF = 0.65 + 0.01 + 56 = 1.21

Therefore, the adjusted FP=46*1.21=55.66

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 expertestimates 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:

$$\text{Estimated Parameter} = c_1 * e^{d1}$$

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. c_1 and d_1 are constants. The values of the constants c_1 and d_1 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:

$$\text{Estimated Resource} = c_1 * e_1^{\ 1} + c_2 * e_2^{\ 2} + \ldots$$

Where e_1, e_2, … are the basic (independent) characteristics of the software already estimated, and c_1, c_2, d_1, d_2, … 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 c_1, c_2, d_1, d_2, … . 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

Analytical estimation techniques derive the required results starting with basic assumptions regarding the project. Thus, unlike empirical and heuristic techniques, analytical techniques do have scientific basis. Halstead's software science is an example of an analytical technique. Halstead's software science can be used to derive some interesting results starting with a few simple assumptions. Halstead's software science is especially useful for estimating software maintenance efforts. In fact, it outperforms both empirical and heuristic techniques when used for predicting software maintenance efforts.

Three basic classes of software development 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. Roughly speaking, these three product classes correspond to application, utility and system programs, respectively. Normally, data processing programs are considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs. System programs interact directly with the hardware and typically involve meeting timing constraints and concurrent processing.

Boehm's [1981] definition of organic, semidetached, and embedded systems are elaborated below.

> Person-month (PM) is considered to be an appropriate unit for measuring effort, because developers are typically assigned to a project for a certain number of months.

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:

$$\text{Effort} = a_1 \times (\text{KLOC})^{a_2} \text{ PM}$$

$$\text{Tdev} = b_1 \times (\text{Effort})^{b_2} \text{ months}$$

- KLOC is the estimated size of the software product expressed in Kilo Lines Of Code.
- a_1, a_2, b_1, b_2 are constants for each category of software product.
- Tdev is the estimated time to develop the software, expressed in months

- Effort is the total effort required to develop the software product, expressed in person- months (PMs).

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

Organic : Effort = 2.4(KLOC)1.05 PM
Semi-detached : Effort = 3.0(KLOC)1.12 PM
Embedded : Effort = 3.6(KLOC)1.20 PM

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

Organic : Tdev = 2.5(Effort)0.38 Months
Semi-detached : Tdev = 2.5(Effort)0.35 Months Embedded :
Tdev = 2.5(Effort)0.32 Months

Example: Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of a software developer is Rs. 15,000 per month. Determine the effort required to develop the software product, the nominal development time, and the cost to develop the product.

From the basic COCOMO estimation formula for organic software:

Effort = $\mathbf{2.4 \times (32)^{1.05}}$ = **91 PM**
Nominal development time = $\mathbf{2.5 \times (91)^{0.38}}$ = **14 months**

Staff cost required to develop the product = 91 × Rs. 15, 000 = Rs.1,465,000

Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Thus, the effort required to develop a product increases very rapidly with project size.

Intermediate COCOMO model

The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort required to develop the product as well as the development time. Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. 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.

Halstead's Software Science – An Analytical Technique

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 over all program length, potential minimum value, actual volume, effort, and development time.

For a given program, let:

- η_1 be the number of unique operators used in the program,
- η_2 be the number of unique operands used in the program,
- N_1 be the total number of operators used in the program,
- N_2 be the total number of operands used in the program.

Operators and Operands for the ANSI C language

The following is a suggested list of operators for the ANSI C language:

- `( [ . , -> * + - ~ ! ++ -- * / % + - << >> < > <= >= != == & ^ | && || = *= /= %= += -= <<= >>= &= ^= |= : ? { ;`
- `CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN and a function name in a function call`

Operands are those variables and constants which are being used with operators in expressions. Note that variable names appearing in declarations are not considered as operands.

Example : The function name in a function definition is not counted as an operator.

```
int func ( int a, int b )
{
        . . .
}
```

For the above example code, the operators are: {}, ( ) We do not consider func, a, and b as operands, since these are part of the function definition.

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 $N = N_1 + N_2$. 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* $\eta = \eta_1 + \eta_2$.

Program Volume

The length of a program (i.e. the total number of operators and operands used in the code) depends on the choice of the operators and operands used. In other words, for the same programming problem, the length would depend on the programming style. This type of dependency would produce different measures of length for essentially the same problem when different programming languages are used. Thus, while expressing program size, the programming language used must be taken into consideration:

$$V = N\log_2\eta$$

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 $\log_2\eta$ bits (where η is the program vocabulary) will be needed. In this scheme, $N\log_2\eta$ 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.

Potential Minimum Volume

The potential minimum volume V* is defined as the volume of most succinct program in which a problem can be coded. The minimum volume is obtained when the program can be expressed using a single source code instruction., say a function call like foo( ) ;. In other words, the volume is bound from below due to the fact that a program would have at least two operators and no less than the requisite number of operands.

Thus, if an algorithm operates on input and output data d_1, d_2, … d_n, the most succinct program would be $f(d_1, d_2, \ldots d_n)$; for which $\eta_1 = 2$, $\eta_2 = n$. Therefore, $V^* = (2 + \eta_2)\log_2(2 + \eta_2)$.

The program level L is given by L = V*/V. The concept of program level L is introduced in an attempt to measure the level of abstraction provided by the programming language.Using this definition, languages can be ranked into levels that also appear intuitively correct. The above result implies that the higher the level of a language,the less effort it takes to develop a program using that language. This result agrees with the intuitive notion that it takes more effort to develop a program in assembly language than to develop a program in a high-level language to solve a problem.

Effort and Time

The effort required to develop a program can be obtained by dividing the program volume with the level of the programming language used to develop the code. Thus, effort E = V/L, where E is the number of mental discriminations required to implement the program and also the effort required to read and understand the program. Thus, the programming effort $E = V^2/V^*$ (since L = V*/V) varies as the square of the volume. Experience shows that E is well correlated to the effort needed for maintenance of an existing program.

The programmer's time T = E/S, where S the speed of mental discriminations. The value of S has been empirically developed from psychological reasoning, and its recommended value for programming applications is 18.

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. His method is summarized below.

Halstead assumed that it is quite unlikely that a program has several identical parts – in formal language terminology identicalsubstrings – of length greater than η (η being the program vocabulary). In fact, once a piece of code occurs identically at several places, it is made into a procedure or a function. Thus, it can be assumed that any program of length N consists of N/ η unique strings of length η. Now, it is standard combinatorial result that for any given alphabet of size K, there are exactly K^r different strings of length r.

$$N/\eta \le \eta^{\eta} \quad \text{Or, } N \le \eta^{\eta+1}$$

Since operators and operands usually alternate in a program, the upper bound can be further refined into $N \le \eta\, \eta_1^{\eta 1}\, \eta_2^{\eta 2}$. Also, N must include not only the ordered set of n elements, but it should also include all possible subsets of that ordered sets, i.e. the power set of N strings (This particular reasoning of Halstead is not very convincing!!!).

Therefore, $2^N = \eta\, \eta_1^{\eta 1}\, \eta_2^{\eta 2}$

Or, taking logarithm on both sides,

$$N = \log_2\eta + \log_2(\eta_1^{\eta 1}\, \eta_2^{\eta 2})$$

So we get,

$$N = \log_2(\eta_1^{\eta_1} \eta_2^{\eta_2})$$

(approximately, by ignoring $\log_2\eta$)

$$N = \log_2\eta_1^{\eta_1} + \log_2\eta_2^{\eta_2}$$

$$= \eta_1\log_2\eta_1 + \eta_2\log_2\eta_2$$

Experimental evidence gathered from the analysis of larger number of programs suggests that the computed and actual lengths match very closely. However, the results may be inaccurate when small programs when considered individually.

In conclusion, Halstead's theory tries to provide a formal definition and quantification of such qualitative attributes as program complexity, ease of understanding, and the level of abstraction based on some low-level parameters such as the number of operands, and operators appearing in the program. Halstead's software science provides gross estimation of properties of a large collection of software, but extends to individual cases rather inaccurately.

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,$\eta_1 = 12$, $\eta_2 = 11$

Estimated Length**= (12*log12 + 11*log11)**

= (12*3.58 + 11*3.45)

= (43+38) = 81

Volume **= Length*log(23)**

= 81*4.52

= 366

Staffing level estimation

Once the effort required to develop a software has been determined, it is necessary to determine the staffing requirement for the project. Putnam first studied the problem of what should be a proper staffing pattern for software projects. He extended the work of Norden who had earlier investigated the staffing pattern of research and development (R&D) type of projects. In order to appreciate the staffing pattern of software projects, Norden's and Putnam's results must be understood.

Norden's Work

Norden studied the staffing patterns of several R & D projects. He found that the staffing pattern can be approximated by the Rayleigh distribution curve. Norden represented the Rayleigh curve by the following equation:

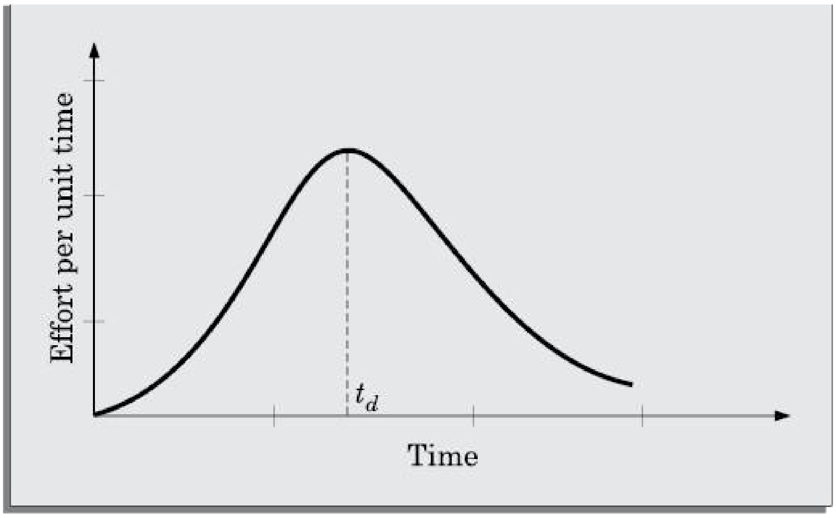


Figure: Rayleigh curve.

Norden represented the Rayleigh curve by the following equation:

$$E = \frac{K}{{t_d}^2} * t * e^{\frac{-t^2}{2{t_d}^2}}$$

where, E is the effort required at time t. E is an indication of the number of developers (or the staffing level) at any particular time during the duration of the project, K is the area under the curve, and td is the time at which the curve attains its maximum value. It must be remembered that the results of Norden are applicable to general R&D projects and were not meant to model the staffing pattern of software development projects.

Putnam's Work

Putnam studied the problem of staffing of software projects and found that the software development has characteristics very similar to other R & D projects studied by Norden and that the Rayleigh-Norden curve can be used to relate the number of delivered lines of code to the effort and the time required to develop the project. By analyzing a large number of army projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

The various terms of this expression are as follows:

- K is the total effort expended (in PM) in the product development and L is the product size in KLOC.
- t_d corresponds to the time of system and integration testing. Therefore, t_d can be approximately considered as the time required to develop the software.
- C_k is the state of technology constant and reflects constraints that impede the progress of the programmer. Typical values of C_k = 2 for poor development environment (no methodology, poor documentation, and review, etc.), C_k = 8 for good software development environment (software engineering principles are adhered to), C_k = 11 for an

excellent environment (in addition to following software engineering principles, automated tools and techniques are used). The exact value of C_k for a specific project can be computed from the historical data of the organization developing it.

Putnam suggested that optimal staff build-up on a project should follow the Rayleigh curve. Only a small number of engineers are needed at the beginning of a project to carry out planning and specification tasks. As the project progresses and more detailed work is required, the number of engineers reaches a peak. After implementation and unit testing, the number of project staff falls.

Effect of schedule change on cost

By analyzing a large number of army projects, Putnam derived the following expression:

$$K = \frac{L^3}{(C_k^3 t_d^4)}$$

or,

$$K = \frac{C}{{t_d}^4}$$

For the same product size, $C = \frac{L^3}{C_k^3}$ is a constant.

Or,

$$\frac{K_1}{K_2} = \frac{t_{d_2}^4}{t_{d_1}^4} \qquad (3.3)$$

Where, K is the total effort expended (in PM) in the product development and L is the product size in KLOC, t_d corresponds to the time of system and integration testing and C_k is the state of technology constant and reflects constraints that impede the progress of the programmer.

Example : The nominal effort and duration of a project have been estimated to be 1000PM and 15 months. The project cost has been negotiated to be Rs. 200,000,000. The needs the product to be developed and delivered in 12 month time. What should be the new cost to be negotiated?

Answer: The project can be classified as a large project. Therefore, the new cost to be negotiated can be given by the Putnam's formula:

new cost = Rs. 200, 000, 000 × $(\underline{15/12})^4$ = Rs. 488,281,250.

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:

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

- 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 are 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.It represents the WBS of an 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.

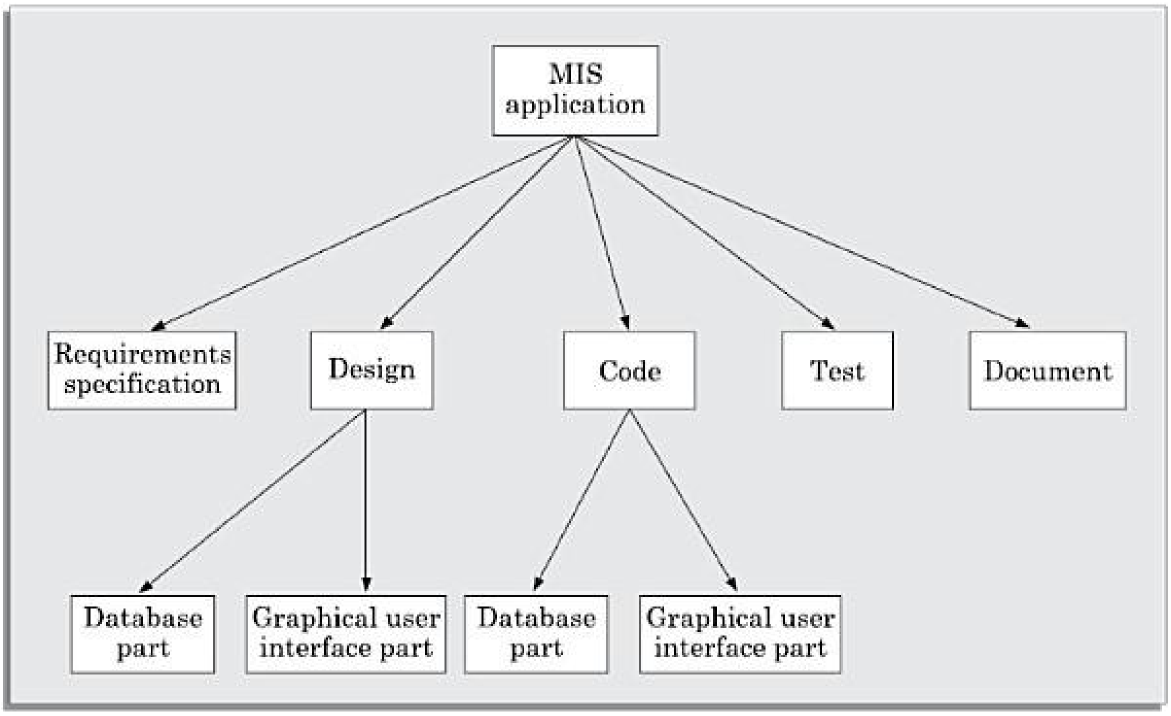


Figure: Work breakdown structure of an MIS problem.

How long to decompose?

The decomposition of the activities is carried out until any of the following is satisfied:

- A leaf-level subactivity (a task) requires approximately two weeks to develop.
- Hidden complexities are exposed, so that the job to be done is understood and can be assigned as a unit of work to one of the developers.
- Opportunities for reuse of existing software components is identified.

Activity Networks

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

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

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

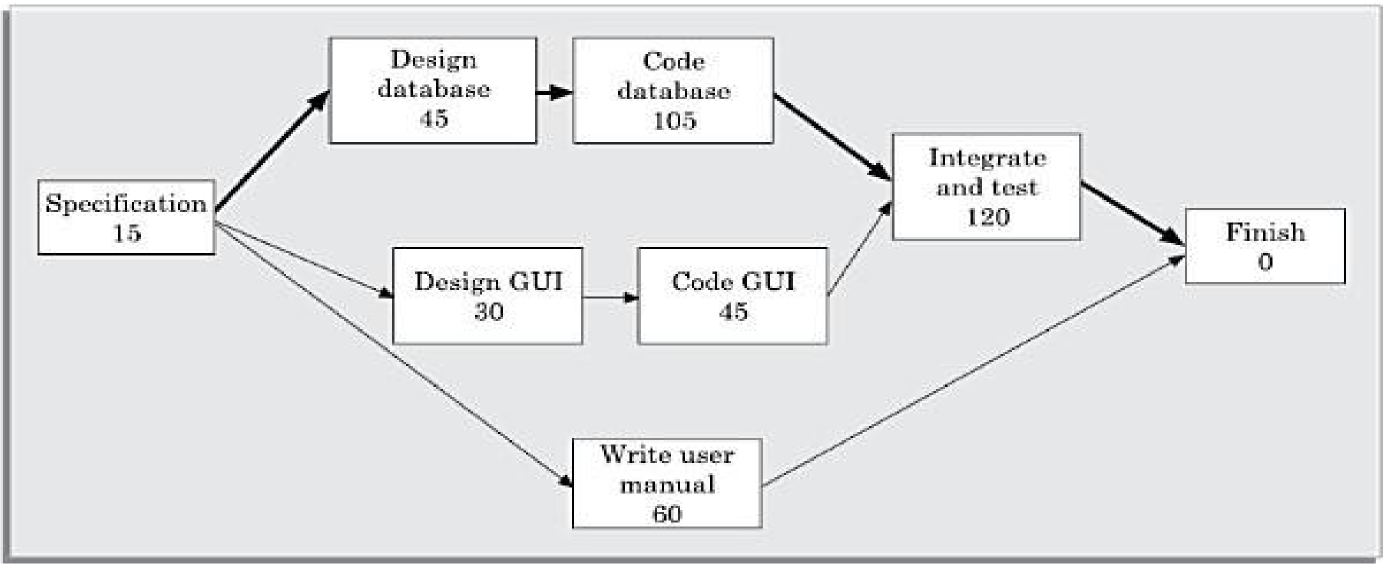


Figure: Activity network representation of the MIS problem.

Example: Determine the Activity network representation for the MIS development project of Example.Assume that the manager has determined the tasks to be represented from the work breakdown structure of Figure, and has determined the durations and dependencies for each task as shown in Table

Answer: The activity network representation has been shown in Figure

Table : Project Parameters Computed from Activity Network

Task Number	Task	Duration	Dependent on Tasks
T1	Specification	15	–
T2	Design database	45	T 1
T3	Design GUI	30	T 1
T4	Code database	105	T 2
T5	Code GUI part	45	T 3
T6	Integrate and test	120	T 4 and T 5
T7	Write user manual	60	T 1

Critical Path Method (CPM)

CPM and PERT are operation research techniques that were developed in the late 1950s. Since then, they have remained extremely popular among project managers. Of late, these two techniques have got merged and many project management tools support them as CPM/PERT. How ever, we point out the fundamental differences between the two and discuss CPM in this subsection and PERT in the next subsection.

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

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

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

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

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

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

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

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

Example: Use the Activity network of Figure to determine the ES and EF for every task for the MIS problem of Example.

Answer: The activity network with computed ES and EF values has been shown in Figure

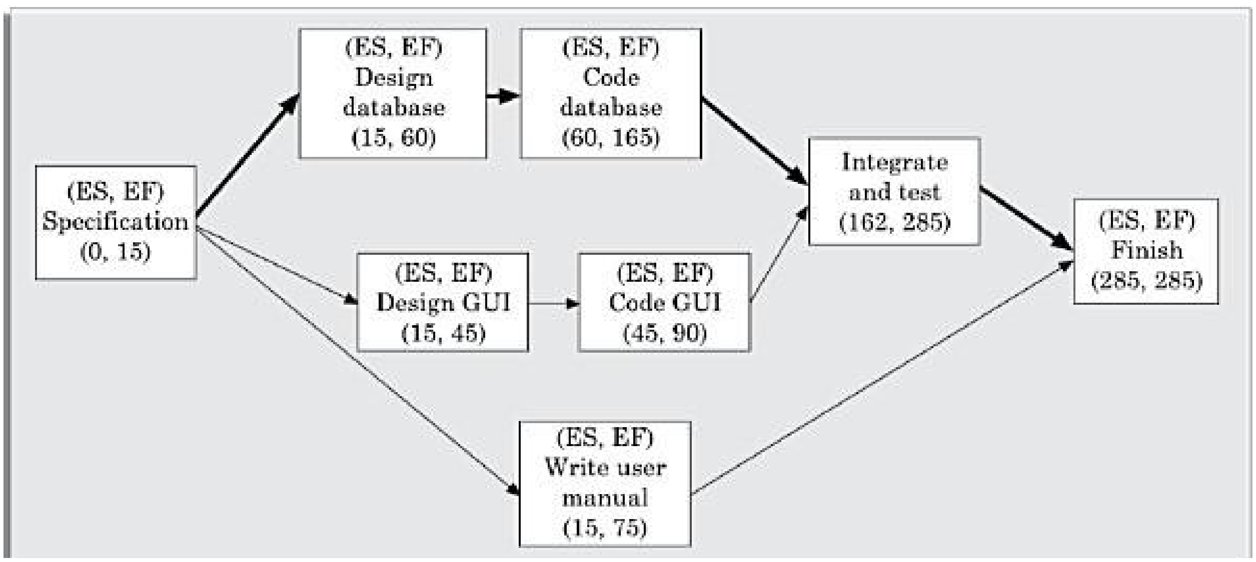


Figure 1 : AoN for MIS problem with (ES,EF).

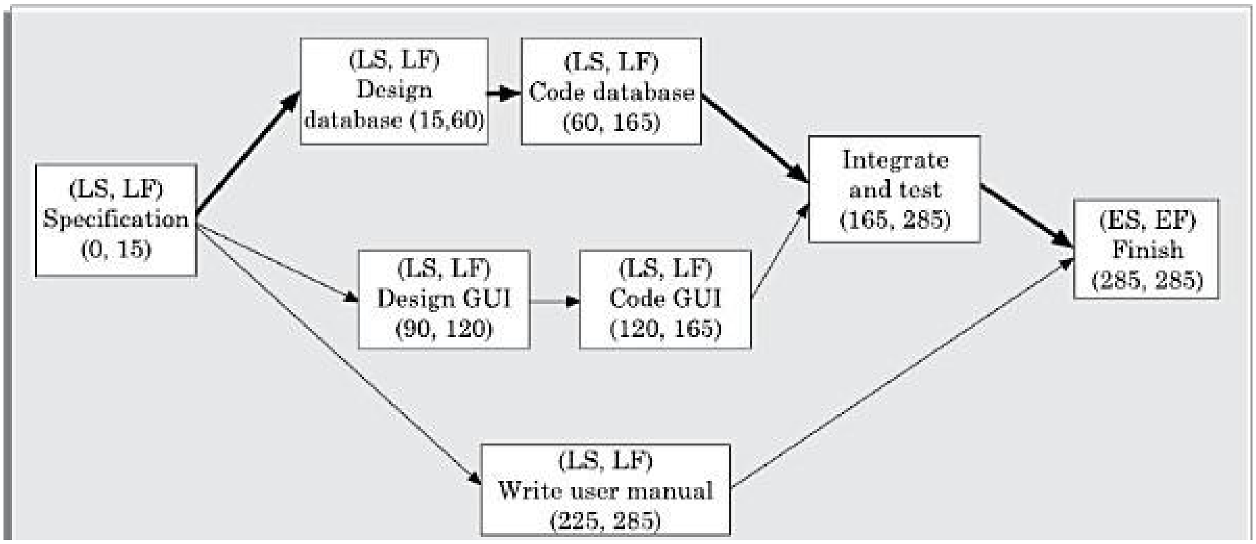


Figure 2: AoN of MIS problem with (LS,LF).

In Figure 1 and Figure 2, we show computation of (ES,EF) and (LS,LF) respectively. From this project parameters for different tasks for the MIS problem have been represented in Table 3.8.

Table : Project Parameters Computed From Activity Network

Task	ES	EF	LS	LF	ST
Specification	0	15	0	15	0
Design data base	15	60	15	60	0
Design GUI part	15	45	90	120	75
Code data base	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

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. 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.

Each task is annotated with three estimates:

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

The standard deviation for a task ST = (P – O)/6.

The mean estimated time is calculated as ET = (O + 4M + W)/6.

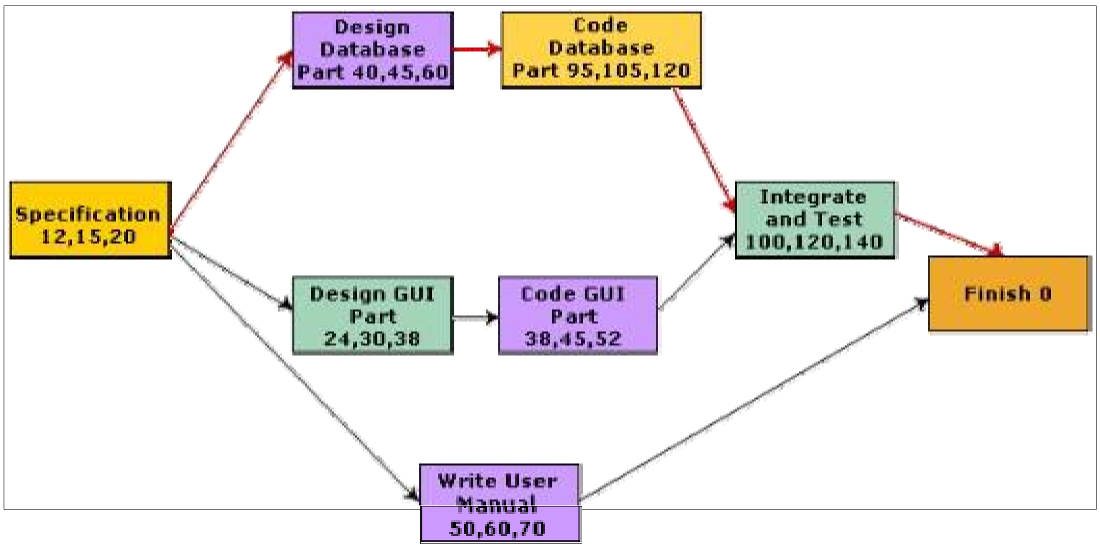


Fig: PERT chart representation of the MIS problem

Gantt Charts

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

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

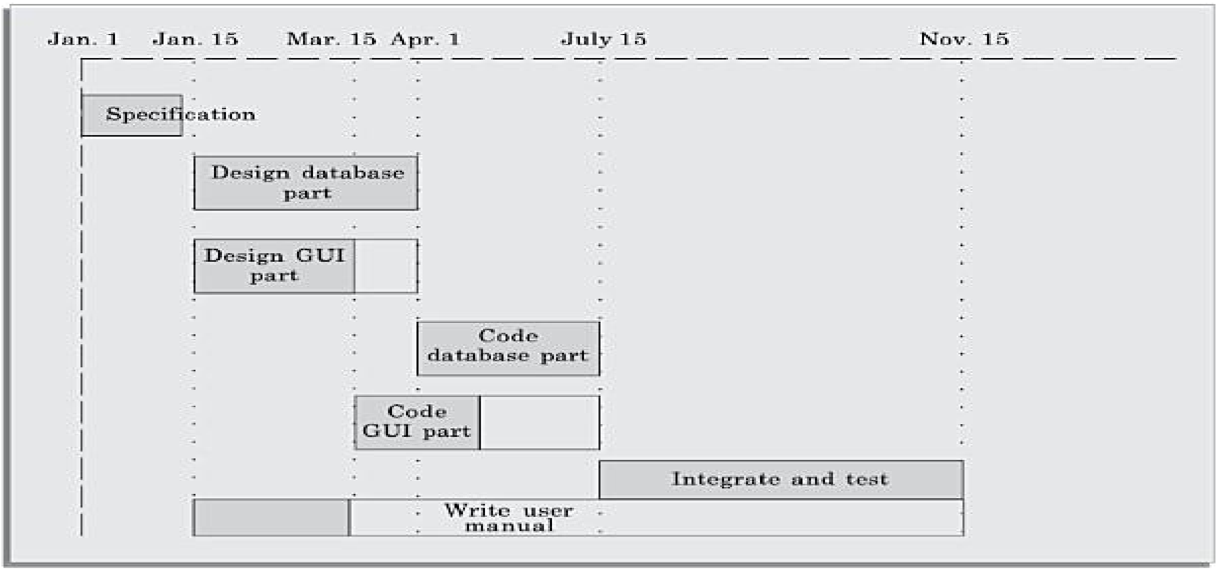


Figure 3.12: Gantt chart representation of the MIS problem.

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.

ORGANISATION AND TEAM STRUCTURES

Usually every software development organisation handles several projects at any time. Software organisations assign different teams of developers to handle different software projects. With regard to staff organisation, there are two important issues—How is the organisation as a whole structured? And, how are the individual project teams structured? There are a few standard ways in which software organisations and teams can be structured.

Functional format

In the functional format, the development staff are divided based on the specific functional group to which they belong to. This format has schematically.

The different projects borrow developers from various functional groups for specific phases of the project and return them to the functional group upon the completion of the phase. As a result, different teams of programmers from different functional groups 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 product evolves. Therefore, the functional format requires considerable communication among the different teams and development of good quality documents because the work of one team must be clearly understood by the subsequent teams working on the project. The functional organisation therefore mandates good quality documentation to be produced after every activity.

Project format

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

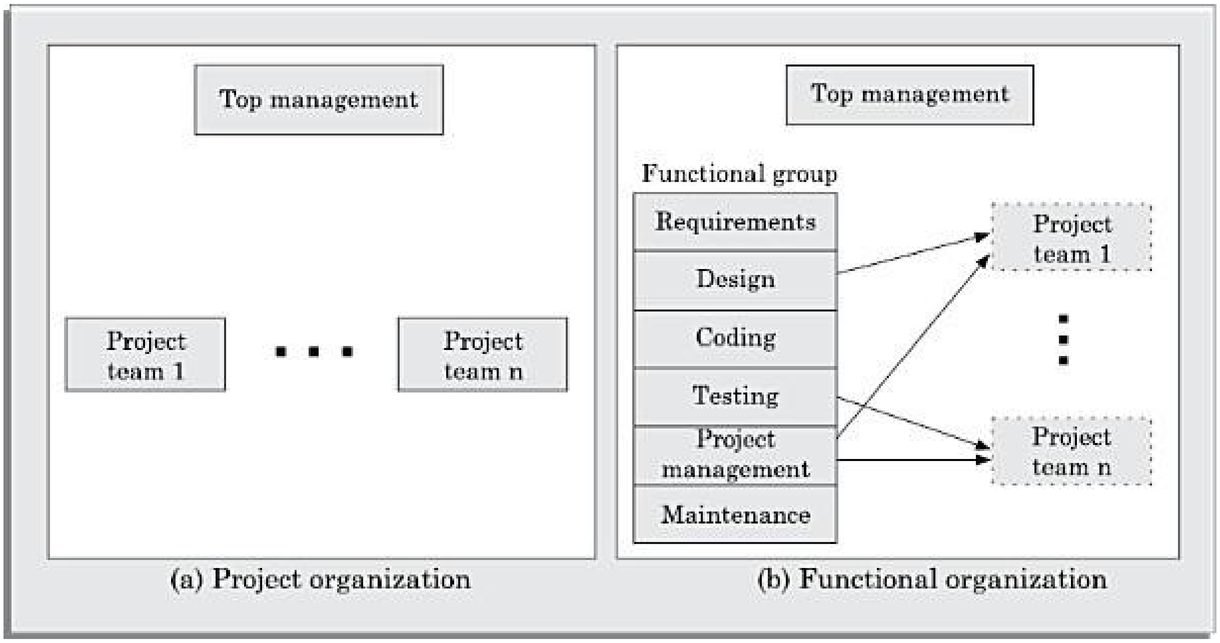


Figure: Schematic representation of the functional and project organisation.

Advantages of functional organization over project organization

Even though greater communication among the team members may appear as an avoidable overhead, the functional format has many advantages. The main advantages of a functional 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. We have already seen that the staffing pattern should approximately follow the Rayleigh distribution for efficient utilization of the personnel by minimizing their wait times. 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.

Matrix format

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

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

Figure: Matrix organisation.

Matrix organisations can be characterised as weak or strong, depending upon the relative authority of the functional managers and the project managers. In a strong functional matrix, the functional managers have authority to assign workers to projects and project managers have to accept the assigned personnel. In a weak matrix, the project manager controls the project budget, can reject workers from functional groups, or even decide to hire outside workers.

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

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

Team 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.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.

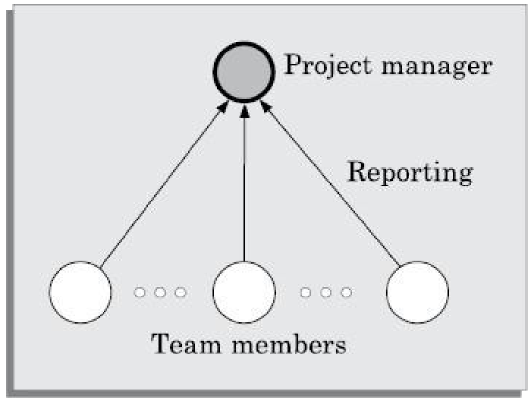


Figure: Chief programmer team structure.

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

Democratic team

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

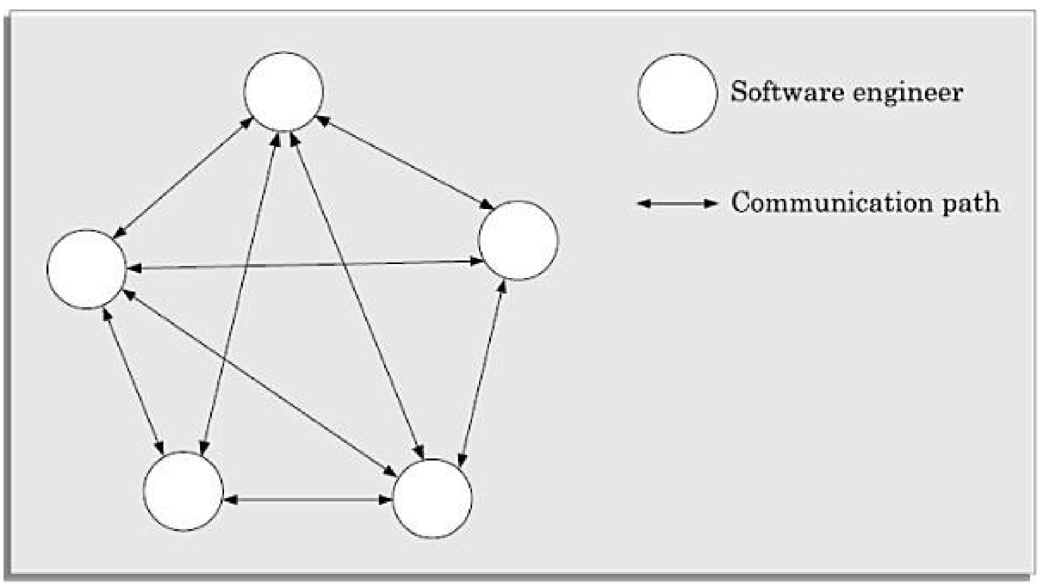


Figure: Democratic team structure.

In a democratic organisation, the team members have higher morale and job satisfaction. Consequently, it suffers from less manpower turnover. Though the democratic teams are less productive compared to the chief programmer team, the democratic team structure is appropriate for less understood problems, since a group of developers can invent better solutions than a single individual as in a chief programmer team. A democratic team structure is suitable for research-oriented projects requiring less than five or six developers. For large sized projects, a pure democratic organisation tends to become chaotic. The democratic team organisation encourages egoless programming as programmers can share and review each other's work. To appreciate the concept of egoless programming, we need to understand the concept of ego from a psychological perspective.

Mixed control team organisation

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

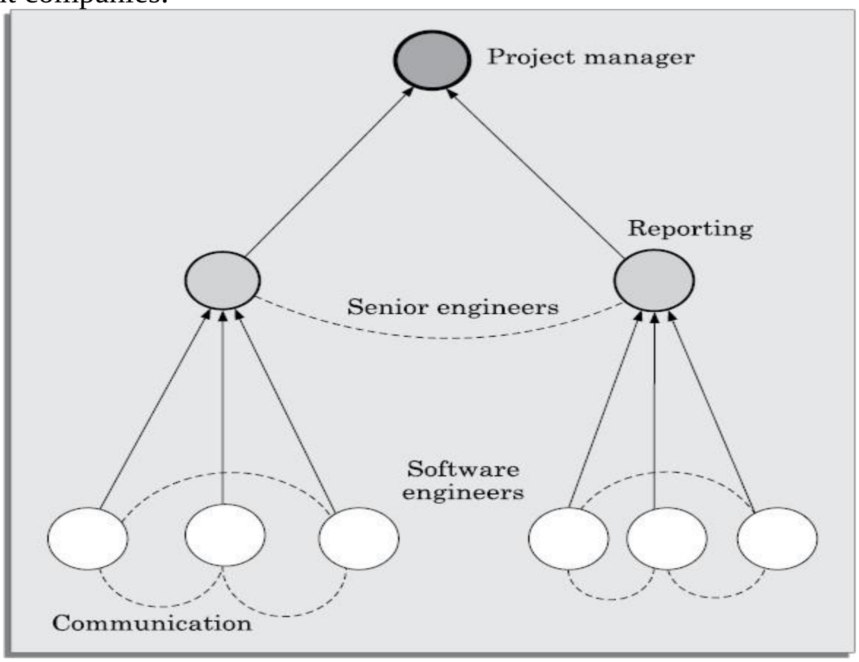


Figure: Mixed team structure.

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 democratic team to think that the design, code, and other deliverables to belong to the entire group. This is called egoless programming technique

STAFFING

Software project managers usually take the responsibility of choosing their team. Therefore, they need to identify good software developers for the success of the project. A common misconception held by managers as evidenced in their staffing, planning and scheduling practices, is the assumption that one software engineer is as productive as another. H owever, experiments have revealed that there exists a large variability of productivity between the worst and the best software developers in a scale of 1 to 30. In fact, the worst developers may sometimes even reduce the overall productivity of the team, and thus in effect exhibit negative productivity. Therefore, choosing good software developers is crucial to the success of a project.

Who is a good software engineer?

In the past, several studies concerning the traits of a good software engineer have been carried out. All these studies roughly agree on the following attributes that good software developers should possess:

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

Risk management

Every project is susceptible to a large number of risks. Without effective management of the risks, even the most meticulously planned project may go hay ware.

A risk is any anticipated unfavourable event or circumstance that can occur while a project is underway.

Risk Identification

A software project can be affected by a large variety of risks. In order to be able to systematically identify the important risks which might affect a software project, it is necessary to categorize risks into different classes. The project manager can then examine which risks from each class are relevant to the project. There are three main categories of risks which can affect a software project:

Project risks. Project risks concern varies forms of budgetary, schedule, personnel, resource, and customer-related problems. An important project risk is schedule slippage. Since, software is intangible, it is very difficult to monitor and control a software project. It is very difficult to control something which cannot be seen. For any manufacturing project, such as manufacturing of cars, the project manager can see the product taking shape. He can for instance, see that the engine is fitted, after that the doors are fitted, the car is getting painted, etc. Thus he can easily assess the progress of the work and control it. The invisibility of the product being developed is an important reason why many software projects suffer from the risk of schedule slippage.

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

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

Classification of risks in a project

Example : The project manager can identify several risks in this project. Let us classify them appropriately.

- What if the project cost escalates and overshoots what wasestimated?: **Project risk.**
- What if the mobile phones that are developed become too bulky in size to conveniently carry?: **Business risk.**
- What if it is later found out that the level of radiation coming from the phones is harmful to human being?: **Business risk.**
- What if call hand-off between satellites becomes too difficult to implement?: **Technical risk.**

Risk assessment

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

- The likelihood of a risk coming true (denoted as r).
- The consequence of the problems associated with that risk (denoted as s).

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

$$p = r * s$$

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

Risk Mitigation

After all the identified risks of a project are assessed, plans must be made to contain the most damaging and the most likely risks. Different risks require different containment procedures. In fact, most risks require ingenuity on the part of the project manager in tackling the risk.

There are three main strategies to plan for risk containment:

Avoid the risk: This may take several forms such as discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the engineers to avoid the risk of manpower turnover, etc.The different categories of constraints that usually give rise to risks are:

Process-related risk: These risks arise due to aggressive work schedule, budget, and resource utilisation.

Product-related risks: These risks arise due to commitment to challenging product features (e.g. response time of one second, etc.), quality, reliability etc.

Technology-related risks: These risks arise due to commitment to use certain technology (e.g., satellite communication).

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

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

Risk reduction: This involves planning ways to contain the damage due to a risk. For example, if there is risk that some key personnel might leave, new recruitment may be planned.

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

Risk related to schedule slippage

Even though there are three broad ways to handle any risk, but still risk handling requires a lot of ingenuity on the part of a project manager. As an example, it can be considered the options available to contain an important type of risk that occurs in many software projects – that of schedule slippage. Risks relating to schedule slippage arise primarily due to the intangible nature of software. Therefore, these can be dealt with by increasing the visibility of the software product. Visibility of a software product can be increased by producing relevant documents during the development process wherever meaningful and getting these documents reviewed by an appropriate team. Milestones should be placed at regular intervals through a software engineering process to provide a manager with regular indication of progress. Completion of a phase of the development process before followed need not be the only milestones. Every phase can be broken down to reasonable-sized tasks and milestones can be scheduled for these tasks too. A milestone is reached, once documentation produced as part of a software engineering task is produced and gets successfully reviewed. Milestones need not be placed for every activity. An approximate rule of thumb is to set a milestone every 10 to 15 days.

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 revision versus version

A new version of a software is created when there is a significant change in functionality, technology, or the hardware it runs on, etc. On the other hand a new revision of a software refers to minor bug fix in that software. A new release is created if there is only a bug fix, minor enhancements to the functionality, usability, etc.

For example, one version of a mathematical computation package might run on Unix-based machines, another on Microsoft Windows and so on. As a software is released and used by the customer, errors are discovered that need correction. Enhancements to the functionalities of the software may also be needed. A new release of software is an improved system intended to replace an old one. Often systems are described as version m, release n; or simple m.n. Formally, a history relation is version of can be defined between objects. This relation can be split into two sub relations *is revision of* and *is variant of*.

Necessity of software configuration management

There are several reasons for putting an object under configuration management. But, possibly the most important reason for configuration management is to control the access to the different deliverable objects. Unless strict discipline is enforced regarding updation and storage of different objects, several problems appear. The following are some of the important problems that appear if configuration management is not used.

Inconsistency problem when the objects are replicated. A scenario can be considered where every software engineer has a personal copy of an object (e.g. source code). As each engineer makes changes to his local copy, he is expected to intimate them to other engineers, so that the changes in interfaces are uniformly changed across all modules. However, many times an engineer makes changes to the interfaces in his local copies and forgets to intimate other teammates about the changes. This makes the different copies of the object inconsistent. Finally, when the product is integrated, it does not work. Therefore, when several team members work on developing an object, it is necessary for them to work on a single copy of the object, otherwise inconsistency may arise.

Problems associated with concurrent access. Suppose there is a single copy of a problem module, and several engineers are working on it. Two engineers may simultaneously carry out changes to different portions of the same module, and while saving overwrite each other. Though the problem associated with concurrent access to program code has been explained, similar problems occur for any other deliverable object.

Providing a stable development environment. When a project is underway, the team members need a stable environment to make progress. Suppose somebody is trying to integrate module A, with the modules B and C, he cannot make progress if developer of module C keeps changing C; this can be especially frustrating if a change to module C forces him to recompile A. When an effective configuration management is in place, the manager freezes the objects to form a base line. When anyone needs any of the objects under configuration control, he is provided with a copy of the base line item. The requester makes changes to his private copy. Only after the requester is through with all modifications to his private copy, the configuration is updated and a new base line gets formed instantly. This establishes a baseline for others to use and depend on. Also, configuration may be frozen periodically. Freezing a configuration may involve archiving everything needed to rebuild it. (Archiving means copying to a safe place such as a magnetic tape).

System accounting and maintaining status information. System accounting keeps track of who made a particular change and when the change was made.

Handling variants. Existence of variants of a software product causes some peculiar problems. Suppose somebody has several variants of the same module, and finds a bug in one of them. Then, it has to be fixed in all versions and revisions. To do it efficiently, he should not have to fix it in each and every version and revision of the software separately.

Software configuration management activities

Normally, a project manager performs the configuration management activity by using an automated configuration management tool. A configuration management tool provides automated support for overcoming all the problems mentioned above. In addition, a configuration management tool helps to keep track of various deliverable objects, so that the project manager can quickly and unambiguously determine the current state of the project. 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, precontrolled, and uncontrolled. Controlled objects are those which are already put under configuration control. One must follow some formal procedures to change them. Precontrolled objects are not yet under configuration control, but will eventually be under configuration control. Uncontrolled objects are not and will not be subjected to configuration control. Controllable objects include both controlled and precontrolled 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 fig. 12.5. Configuration management tools allow only one person to reserve a module at a time. Once an object is reserved, it does not allow any one else to reserve this module until the reserved module is restored as shown in fig. 12.5. Thus, by preventing more than one engineer to simultaneously reserve a module, the problems associated with concurrent access are solved.

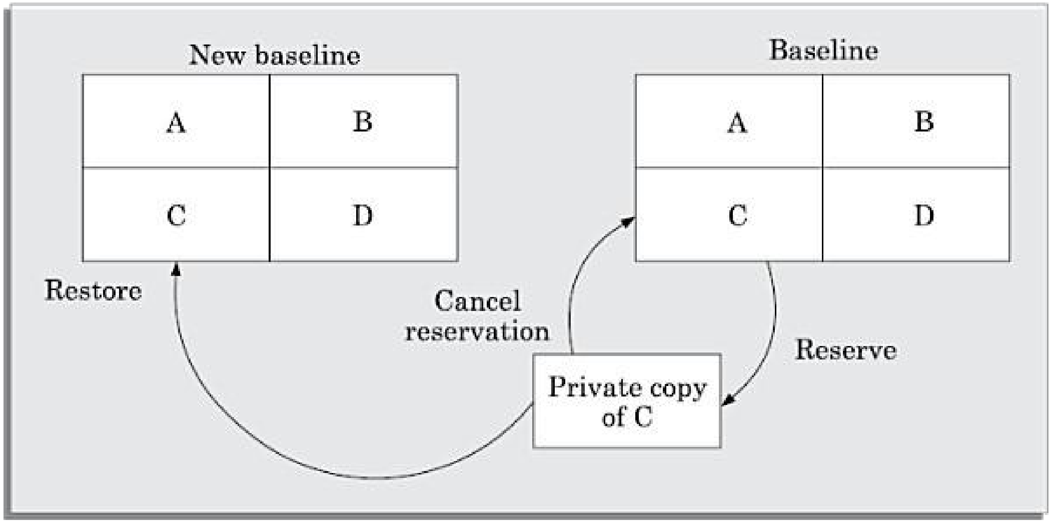


Fig: Reserve and restore operation in configuration control

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.

The change control board (CCB) sounds like a group of people. However, except for very large projects, the functions of the change control board are normally discharged by the project manager himself or some senior member of the development team. Once the CCB reviews the changes to the module, the project manager updates the old base line through a restore operation (as shown in fig. 12.5). A configuration control tool does not allow a developer to replace an object he has reserved with his local copy unless he gets an authorization from the CCB. By constraining the developers' ability to replace reserved objects, a stable environment is achieved. Since a configuration management tool allows only one engineer to work on one module at any one time, problem of accidental overwriting is eliminated. Also, since only the manager can update the baseline after the CCB approval, unintentional changes are eliminated.

Configuration management tools

SCCS and RCS are two popular configuration management tools available on most UNIX systems. SCCS or RCS can be used for controlling and managing different versions of text files. SCCS and RCS do not handle binary files (i.e. executable files, documents, files containing diagrams, etc.) SCCS and RCS provide an efficient way of storing versions that minimizes the amount of occupied disk space. Suppose, a module MOD is present in three versions MOD1.1, MOD1.2, and MOD1.3. Then, SCCS and RCS stores the original module MOD1.1 together with changes needed to transform MOD1.1 into MOD1.2 and MOD1.2 to MOD1.3. The changes needed to transform each base lined file to the next version are stored and are called deltas. The main reason behind storing the deltas rather than storing the full version files is to save disk space.

The change control facilities provided by SCCS and RCS include the ability to incorporate restrictions on the set of individuals who can create new versions, and facilities for checking components in and out (i.e. reserve and restore operations). Individual developers check out components and modify them. After they have made all necessary changes to a module and after the changes have been reviewed, they check in the changed module into SCCS or RCS. Revisions are denoted by numbers in ascending order, e.g., 1.1, 1.2, 1.3 etc. It is also possible to create variants or revisions of a component by creating a fork in the development history.

SOFTWARE AND SOFTWARE ENGINEERING:

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

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

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

1.THE NATURE OF SOFTWARE

Software takes Dual role of Software. It is a **Product** and at the same time a **Vehicle for delivering a product**.

Software delivers the most important product of our time is called **information**

Software is defined as

1. **Instructions**: Programs that when executed provide desired function, features, and performance
2. **Data structures**: Enable the programs to adequately manipulate information
3. **Documents**: Descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

1.1.Characteristics of software:

Software has characteristics that are considerably different than those of hardware:

1.Software is developed or engineered, it is not manufactured in the Classical Sense.

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both the activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent or easily corrected for software. Both the activities are dependent on people, but the relationship between people is totally varying. These two activities require the construction of a "**product**" but the approaches are different. Software costs are concentrated in engineering which means that software projects cannot be managed as if they were manufacturing.

2. Software doesn't "Wear Out"

The following figure shows the relationship between failure rate and time. Consider the failure rate as a function of time for hardware. The relationship is called **the bathtub curve**, indicates that hardware exhibits relatively high failure rates early in its life, defects are corrected and the failure rate drops to a steady-state level for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. So,stated simply, the hardware begins to wear out. Software is not susceptible to the environmental maladies that cause **hardware to wear out**

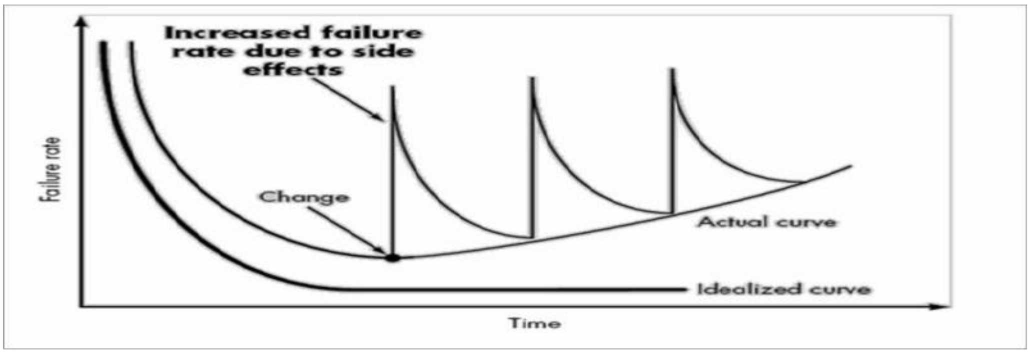


Fig: FAILURE CURVE FOR SOFTWARE

Fig: FAILURE CURVE FOR HARDWARE

3. Although the industry is moving toward component-based construction, most software continues to be custom built

A software component should be designed and implemented so that it can be reused in many Different programs . Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts

<u>1.2.Types of Software:</u>

1. System Software: It is a collection of programs written to service other programs. Some system software (example- compilers, editors and file management utilities). Some other system applications (operating system components, drivers, networking software, telecommunications processors). In either case, the system software area is characterized by heavy interaction with computer hardware, heavy usage by multiple users, scheduling, resource sharing, process management, complex data structure.

2. Application Software: It consists of standalone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision-making. Application software is mainly used to control business functions in real-time.

3. Engineering/Scientific Software: Formerly characterized by "number crunching" algorithms, engineering and scientific software applications range from as tronomy to volacanology.However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithm Computer-Aided Design (CAD), System simulation has begun to take on real-time.

4. Embedded Software: Embedded Software resides within a product or system and it is used to implement and control features and functions for the end-user and for the system itself. Embedded Software can perform limited and esoteric functions. It also provides significant function and control capability (Example- Digital Function, Dashboard displays, cracking system, etc).

5. Product-line Software: Designed to provide a specific Capability for use by many different customers, product-line. The software can focus on a limited and esoteric marketplace (Example- inventory control products) or address mass consumer markets (Example – Word Processing, Spreadsheet, Computer Graphics, Multimedia, entertainment, database management, personal and business applications).

6. Web-Applications: "WebApps", span a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics, such as- E-commerce and B2B Applications. WebApps are evolving into sophisticated computing functions and also are integrated with Corporate databases and business applications.

7. Artificial Intelligence Software: AI software makes use of non-numerical algorithms to solve complex problems that are not amenable to computations or straight-forward analysis. These applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks and game playing.

New Software Challenges

- **Open-world computing** : Creating software to allow machines of all sizes to communicate with each other across vast networks (Distributed computing—wireless networks)
- **Netsourcing** : Architecting simple and sophisticated applications that benefit targeted end-user markets worldwide (the Web as a computing engine)
- **Open Source** : Distributing source code for computing applications so customers can make local modifications easily and reliably ("free" source code open to the computing community)

1.3.LEGACY SOFTWARE:

- Legacy software is older programs that are developed decades ago.
- The quality of legacy software is poor because it has inextensible design, convoluted code, poor and nonexistent documentation, test cases and results that are not achieved.As time passes legacy systems evolve due to following reasons:
- The software must be adapted to meet the needs of new computing environment or technology.
- The software must be enhanced to implement new business requirements.
- The software must be extended to make it interoperable with more modern systems or database
- The software must be re-architected to make it viable within a network environment.

2. UNIQUE NATURE OF WEB APPS

In the early days of the World Wide Web, websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics. As time passed, the augmentation of HTML by development tools (e.g., XML, Java) enabled Web engineers to provide computing capability along with informational content. *Web-based systems and applications* (*WebApps*) were born. Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.WebApps are one of a number of distinct software categories. Web-based systems and applications "involve a mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology."

The following attributes are encountered in the vast majority of WebApps.

- **Network intensiveness.** A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).
- **Concurrency.** A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.

- **Unpredictable load.** The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.
- **Performance.** If a WebApp user must wait too long, he or she may decide to go elsewhere.
- **Availability.** Although expectation of 100 percent availability is un reasonable, users of popular WebApps often demand access on a 24/7/365 basis
- **Data driven.** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).
- **Content sensitive.** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.
- **Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously.
- **Immediacy.** Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time- to-market that can be a matter of a few days or weeks.
- **Security.** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes
- **Aesthetics**. An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

3.SOFTWARE MYTHS:

Software Myths- beliefs about software and the process used to build it - can be traced to the earliest days of computing. Myths have a number of attributes that have made them insidious. For instance, myths appear to be reasonable statements of fact, they have an intuitive feel, and they are often promulgated by experienced practitioners who “know the score”.

We are having 3 types of myths

- **Management Myths**
- **Customer Myths**
- **Practioner’s Myths**

1.Management Myths :

Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth.

Myth : We already have a book that’s full of standards and procedures for building software. Won’t That provide my people with everything they need to know?

Reality :

- The book of standards may very well exist, but is it used?
- Are software practitioners aware of its existence?

- Does it reflect modern software engineering practice?
- Is it complete?
- Is it adaptable?
- Is it streamlined to improve time to delivery while still maintaining a focus on Quality? In many cases, the answer to these entire question is NO.

Myth : *If we get behind schedule, we can add more programmers and catch up*

Reality : Software development is not a mechanistic process like manufacturing. "Adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, there by reducing the amount of time spent on productive development effort

Myth : If we decide to outsource the software project to a third party, I can just relax and let that firm build it.

Reality : If an organization does not understand how to manage and control software project internally, it will invariably struggle when it out sources software project.

2.Customer Myths

A customer who requests computer software may be a person at the next desk,technical g group down the hall, the marketing /sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths led to false expectations and ultimately, dissatisfaction with the developers.

Myth : A general statement of objectives is sufficient to begin writing programs - we can fill in details later.

Reality : Although a comprehensive and stable statement of requirements is not always possible,an ambiguous statement of objectives is a recipe for disaster. Unambiguous requirements are developed only through effective and continuous communication between customer and developer.

Myth : Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality : It's true that software requirement change, but the impact of change varies with the time at which it is introduced. When requirement changes are requested early, cost impact is relatively small. However, as time passes, cost impact grows rapidly – resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

3.Practitioner's myths.

Myths that are still believed by software practitioners have been fostered by 50 years of programming culture. During the early days of software, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: Once we write the program and get it to work, our job is done.

Reality: Someone once said that "the sooner you begin 'writing code', the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: Until I get the program "running" I have no way of assessing its quality.

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the formal technical review. Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth: The only deliverable work product for a successful project is the working program.

Reality: A working program is only one part of a software configuration that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.

Myth: Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

Reality: Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times. Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

REQUIREMENTS GATHERING AND ANALYSIS:

The complete set of requirements are almost never available in the form of a single document from the customer. In fact, it would be unrealistic to expect the customers to produce a comprehensive document containing a precise description of what he wants. Further, the complete requirements are rarely obtainable from any single customer representative. Therefore, the requirements have to be gathered by the analyst from several sources in bits and pieces. These gathered requirements need to be analysed to remove several types of problems that frequently occur in the requirements that have been gathered piecemeal from different sources.

For any type of software development project, availability of a good quality requirements document has been acknowledged to be a key factor in the successful completion of the project. A good requirements document not only helps to form a clear understanding of various features required from the software, but also serves as the basis for various activities carried out during later life cycle phases.After understanding the precise user requirements, the analysts analyse the requirements to weed out inconsistencies, anomalies and incompleteness. They then proceed to write the *software requirements specification* (SRS) document.

The project team to ensure that it accurately captures all the user requirements, and that it is understandable, consistent, unambiguous, and complete. The SRS document is then given to the customer for review. After the customer has reviewed the SRS document and agrees to it, it forms the basis for all future development activities and also serves as a contract document between the customer and the development organisation.

We can conceptually divide the requirements gathering and analysis activity into two separate tasks:

- Requirements gathering
- Requirements analysis

We discuss these two tasks in the following subsections.

Requirements Gathering

Requirements gathering is also popularly known as *requirements elicitation*.The primary objective of the requirements gathering task is to collect the requirements from the *stakeholders*.

> A stakeholder is a source of the requirements and is usually a person, or a group of persons who either directly or indirectly are concerned with the software.

Requirements gathering may sound like a simple task. However, in practice it is very difficult to gather all the necessary information from a large number of stakeholders and from information scattered across several pieces of documents. Gathering requirements turns out to be especially challenging if there is no working model of the software being developed.Typically even before visiting the customer site, requirements gathering activity is started by studying the existing documents to collect all possible information about the system to be developed. During visit to the customer site,the analysts normally interview the end-users and customer representatives,carry out requirements gathering activities such as questionnaire surveys, task analysis, scenario analysis, and form analysis.

The goal of the requirements analysis and specification phase is to clearly understand the customer requirements and to systematically organise the requirements into a document called the Software Requirements Specification (SRS) document.

Good analysts share their experience and expertise with the customer and give his suggestions to define certain functionalities more comprehensively, make the functionalities more general and more complete. In the following, we briefly discuss the important ways in which an experienced analyst gathers requirements:

1.Studying existing documentation: The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site. Customers usually provide statement of purpose (SoP) document to the developers. Typically these documents might discuss issues such as the context in which the software is required, the basic purpose, the stakeholders, features of any similar software developed elsewhere, etc.

2.Interview: Typically, there are many different categories of users of a software. Each category of users typically requires a different set of features from the software. Therefore, it is important for the analyst to first identify the different categories of users and then determine the requirements of each. For example, the different categories of users of a library automation software could be the library members, the librarians, and the accountants. The library members would like to use the software to query availability of books and issue and return books. The librarians might like to use the software to determine books that are overdue, create member accounts, delete member accounts, etc. The accounts personnel might use the software to invoke functionalities concerning financial aspects such as the total fee collected from the members, book procurement expenditures, staff salary expenditures, etc.

3.Task analysis: The users usually have a black-box view of a software and consider the software as something that provides a set of services (functionalities). A service supported by a software is also called a *task*. We can therefore say that the software performs various tasks of the users. In this context, the analyst tries to identify and understand the different tasks to be performed by the software. For each identified task, the analyst tries to formulate the different steps necessary to realise the required functionality in consultation with the users. For example, for the issue book service, the steps may be—authenticate user, check the number of books issued to the customer and determine if the maximum number of books that this member can borrow has been reached, check whether the book has been reserved, post the book issue details in the member's record, and finally print out a book issue slip that can be presented by the member at the security counter to take the book out of the library premises.

4.Scenario analysis: A task can have many scenarios of operation. The different scenarios of a task may take place when the task is invoked under different situations. For different types of scenarios of a task, the behaviour of the software can be different. For example, the possible scenarios for the book issue task of a library automation software may be:

- Book is issued successfully to the member and the book issue slip is printed.
- The book is reserved, and hence cannot be issued to the member.
- The maximum number of books that can be issued to the member is already reached, and no more books can be issued to the member.

- For various identified tasks, the possible scenarios of execution are identified and the details of each scenario is identified in consultation with the users. For each of the identified scenarios, details regarding system response, the exact conditions under which the scenario occurs, etc. are determined in consultation with the user.

5.Form analysis: Form analysis is an important and effective requirements gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system. During the operation of a manual system, normally several forms are required to be filled up by the stakeholders, and in turn they receive several notifications (usually manually filled forms). In form analysis the exiting forms and the formats of the notifications produced are analysed to determine the data input to the system and the data that are output from the system. For the different sets of data input to the system, how these input data would be used by the system to produce the corresponding output data is determined from the users.

Requirements Analysis:

After requirements gathering is complete, the analyst analyses the gathered requirements to form a clear understanding of the exact customer requirements and to weed out any problems in the gathered requirements. It is natural to expect that the data collected from various stakeholders to contain several contradictions, ambiguities, and incompleteness, since each stakeholder typically has only a partial and incomplete view of the software.

During requirements analysis,the analyst needs to identify and resolve three main types of problems in the requirements:

- Anomaly
- Inconsistency
- Incompleteness

Anomaly: It is an anomaly is an ambiguity in a requirement. When a requirement is anomalous, several interpretations of that requirement are possible. Any anomaly in any of the requirements can lead to the development of an incorrect system, since an anomalous requirement can be interpreted in the several ways during development.

Inconsistency: Two requirements are said to be inconsistent, if one of the requirements contradicts the other. The follo wing are two examples of inconsistent requirements:

Incompleteness: An incomplete set of requirements is one in which some requirements have been overlooked. The lack of these features would be felt by the customer much later, possibly while using the software. Often, incompleteness is caused by the inability of the customer to visualise the system that is to be developed and to anticipate all the features that would be required. An experienced analyst can detect most of these missing features and suggest them to the customer for his consideration and approval for incorporation in the requirements.

Example:one of the clerks expressed the following—If a student secures a *grade point average* (GPA) of less than 6, then the parents of the student must be intimated about the regrettable performance through a (postal) letter as well as through e-mail. However, on an examination of all requirements, it was found that there is no provision by which either the postal or e-mail address of the parents of the students can be entered into the system. The feature that would allow entering the e-mail ids and postal addresses of the parents of the students was missing, thereby making the requirements incomplete.

Role of a system analyst

The analyst starts requirements gathering and analysis activity by collecting all information from the customer which could be used to develop the requirements of the system. He then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to removing all ambiguities and inconsistencies from the initial customer perception of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst in order to obtain a good grasp of the problem

- What is the problem?
- Why is it important to solve the problem?
- What are the possible solutions to the problem?
- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the likely complexities that might arise while solving the problem?
- If there are external software or hardware with which the developed software has to interface, then what exactly would the data interchange formats with the external system be?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various requirements problems. The most important requirements problems that the analyst has to identify and eliminate are the problems of anomalies, inconsistencies, and incompleteness. When the analyst detects any inconsistencies, anomalies or incompleteness in the gather requirements, he resolves them by carrying out further discussions with the end- users and the customers.

SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

After the analyst has gathered all the required information regarding the software to be developed, and has removed all incompleteness, inconsistencies, and anomalies from the specification, he starts to systematically organise the requirements in the form of an SRS document.

Users of SRS Document

Usually a large number of different people need the SRS document for very different purposes. Some of the important categories of users of the SRS document and their needs for use are as follows:

- **Users, customers, and marketing personnel:** These stakeholders need to refer to the SRS document to ensure that the system as described in the document will meet their needs. Remember that the customer may not be the user of the software, but may be some one employed or designated by the user. For generic products, the marketing personnel need to understand the requirements that they can explain to the customers.

- **Software developers:** The software developers refer to the SRS document to make sure that they are developing exactly what is required by the customer.

- **Test engineers:** The test engineers use the SRS document to understand the functionalities, and based on this write the test cases to validate its working. They need that the required functionality should be clearly described, and the input and output data should have been identified precisely.

- **User documentation writers:** The user documentation writers need to read the SRS document to ensure that they understand the features of the product well enough to be able to write the users' manuals.

- **Project managers:** The project managers refer to the SRS document to ensure that they can estimate the cost of the project easily by referring to the SRS document and that it contains all the information required to plan the project.

- **Maintenance engineers:** The SRS document helps the maintenance engineers to under- stand the functionalities supported by the system. A clear knowledge of the functionalities can help them to understand the design and code.

A well-formulated SRS document finds a variety of usage other than the primary intended usage as a basis for starting the software development work. In the following subsection, we identify the important uses of a well-formulated SRS document:

- **Forms an agreement between the customers and the developers:** A good SRS document sets the stage for the customers to form their expectation about the software and the developers about what is expected from the software.

- **Reduces future reworks:** The process of preparation of the SRS document forces the stakeholders to rigorously think about all of the requirements before design and development get underway. This reduces later redesign, recoding, and retesting. Careful review of the SRS document can reveal omissions, misunderstandings, and inconsistencies early in the development cycle.

- **Provides a basis for estimating costs and schedules:** Project managers usually estimate the size of the software from an analysis of the SRS document. Based on this estimate they make other estimations such as the effort required to develop the software and the total cost of development. The SRS document also serves as a basis for price negotiations with the customer. The pr oject manager also uses the SRS document for work scheduling.

- **Provides a baseline for validation and verification:** The SRS document provides a baseline against which compliance of the developed software can be checked. It is also used by the test engineers to create the *test plan*.

- **Facilitates future extensions:** The SRS document usually serves as a basis for planning future enhancements.

Traceability: Traceability means that it would be possible to identify (trace) the specific design component which implements a given requirement, the code part that corresponds to a given design component, and test cases that test a given requirement. Thus, any given code component can be traced to the corresponding design component, and a design component can be traced to a specific requirement that it implements a nd *vice versa*. Traceability analysis is an important concept and is frequently used during software development.

To achieve traceability, it is necessary that each functional requirement should be numbered uniquely and consistently. Proper numbering of the requirements makes it possible for different documents to uniquely refer to specific requirements.

Problems without a SRS document

- The important problems that an organization would face if it does not develop an SRS document are as follows:

- Without developing the SRS document, the system would not be implemented according to customer needs.

- Software developers would not know whether what they are developing is what exactly required by the customer.

- Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system.

- It will be very much difficult for user document writers to write the users' manuals properly without understanding the SRS document.

Characteristics of a Good SRS Document

The skill of writing a good SRS document usually comes from the experience gained from writing SRS documents for many projects. However, the analyst should be aware of the desirable qualities that every good SRS document should possess. IEEE Recommended Practice for Software

Requirements Specifications[IEEE830] describes the content and qualities of a good software requirements specification (SRS).

Purpose: This section should describe where the software would be deployed and and how the software would be used.

Project scope: This section should briefly describe the overall context within which the software is being developed. For example, the parts of a problem that are being automated and the parts that would need to be automated during future evolution of the software.

Environmental Characteristics:This section should briefly outline the environment (hardware and other software) with which the software will interact.

Some of the identified desirable qualities of an SRS document are the following:

- **Concise:** The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase the possibilities of errors in the document.
- **Implementation-independent:** The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements. It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the externally visible behaviour of the system a nd not discuss the implementation issues.

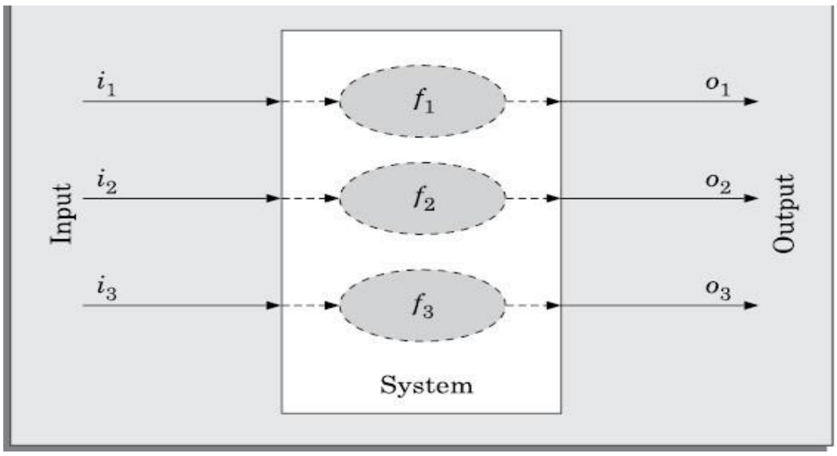


Fig:The black-box view of a system as performing a set of functions

The SRS document should describe the system to be developed as a black box, and should specify only the externally visible behaviour of the system. For this reason, the S R S document is also called the black-box specification of the software being

- **Traceable:** It should be possible to trace a specific requirement to the design elements that implement it and *vice versa*. Similarly, it should be possible to trace a requirement to the code segments that implement it and the test cases that test this requirement and *vice versa*. Traceability is also important to verify the results of a phase with respect to the previous phase and to analyse the impact of changing a requirement on the design elements and the code.
- **Modifiable:** Customers frequently change the requirements during the software development development due to a variety of reasons. Therefore, in practice the SRS document undergoes several revisions during software development. Also, an SRS document is often modified after the project completes to accommodate future enhancements and evolution. To cope up with the requirements changes, the SRS document should be easily modifiable. For this, an SRS document should be well-structured. A well- structured document is easy to understand and modify. Having the description of a requirement scattered across many places in the SRS document may not be wrong —but it tends to make the requirement difficult to understand and also any modification to the

requirement would become difficult as it would require changes to be made at large number of places in the document.

- **Identification of response to undesired events:** The SRS document should discuss the system responses to various undesired events and exceptional conditions that may arise.

- **Verifiable:** All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to design test cases based on the description of the functionality as to whether or not requirements have been met in an implementation. A requirement such as "the system should be user friendly" is not verifiable. On the other hand, the requirement —"When the name of a book is entered, the software should display whether the book is available for issue or it has been loaned out" is verifiable. Any feature of the required system that is not verifiable should be listed separately in the goals of the implementation section of the SRS document.

Attributes of Bad SRS Documents

SRS documents written by novices frequently suffer from a variety of problems. As discussed earlier, the most damaging problems are incompleteness, ambiguity, and contradictions.

- **Over-specification**
- **Forward references**
- **Wishful thinking**
- **Noise**

Important Categories of Customer Requirements

A good SRS document, should properly categorize and organise the requirements into different sections [IEEE830]. As per the IEEE 830 guidelines, the important categories of user requirements are the following.The IEEE 830 standard recommends that out of the various non-functional requirements, the external interfaces, and the design and implementation constraints should be documented in two different sections. The remaining non-functional requirements should be documented later in a section and these should include the performance and security requirements.

An SRS document should clearly document the following aspects of a software:

- Functional requirements
- Non-functional requirements
  - — Design and implementation constraints
  - — External interfaces required
  - — Other non-functional requirements
- Goals of implementation.

Functional requirements:-

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high- level functions $\{f_i\}$. The functional view of the system is shown in fig.Each function f_i of the system can be considered as a transformation of a set of input data (ii) to the corresponding set of output data (o_i). The user can get some meaningful piece of work done using a high-level function.

Identifying functional requirements from a problem description:

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem. Each high-level requirement characterizes a way of system usage by some user to perform some meaningful piece of work. There can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to identify the different types of users who might use the system and then try to identify the requirements from each user's perspective.

Here we list all functions $\{f_i\}$ that the system performs. Each function f_i as shown in fig is considered as a transformation of a set of input data to some corresponding output data.

Fig: Function f_i

For documenting the functional requirements, we need to specify the set of functionalities supported by the system. A function can be specified by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data. Let us first try to document the withdraw-cash function of an ATM (Automated Teller Machine) system. The withdraw-cash is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These different interaction sequences capture the different scenarios.

Example: - Withdraw Cash from ATM R1:

Description: The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash, otherwise it generates an error message.

R1.1 select withdraw amount option Input:

“withdraw amount” option

Output: user prompted to enter the account type

R1.2: select account type

Input: user option

Output: prompt to enter amount R1.3:

get required amount

Input: amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.

Output: The requested cash and printed transaction statement.

Processing: the amount is debited from the user's account if sufficient balance is available, otherwise an error message displayed.

Non-functional requirements:-

The non-functional requirements are non-negotiable obligations that must be supported by the software. The non-functional requirements capture those requirements of the customer that cannot be expressed as functions (i.e., accepting input data and producing output data). Non-functional requirements usually address aspects concerning external interfaces, user interfaces, maintainability, portability, usability, maximum number of concurrent users, timing, and throughput (transactions per second, etc.). The non-functional requirements can be critical in the sense that any failure by the developed software to achieve some minimum defined level in these requirements can be considered as a failure and make the software unacceptable by the customer.Nonfunctional requirements may include:

reliability issues,

accuracy of results,

#performance issues

human - computer interface issues,

#constraints on the system implementation,

security and maintainability of the system, etc.

In the following subsections, we discuss the different categories of non- functional requirements that are described under three different sections:

Design and implementation constraints: Design and implementation constraints are an important category of non-functional requirements describe any items or issues that will limit the options available to the developers. Some of the example constraints can be—corporate or regulatory policies that needs to be honoured; hardware limitations; interfaces with other applications; specific technologies, tools, and databases to be used; specific communications protocols to be used; security considerations; design conventions or programming standards to be followed, etc. Consider an example of a constraint that can be included in this section—Oracle DBMS needs to be used as this would facilitate easy interfacing with other applications that are already operational in the organisation.

External interfaces required: Examples of external interfaces are— hardware, software and communication interfaces, user interfaces, report formats, etc. To specify the user interfaces, each interface between the software and the users must be described. The description may include sample screen images, any GUI standards or style guides that are to be followed, screen layout constraints, standard buttons and functions(e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on. One example of a user interface requirement of a software can be that it should be usable by factory shop floor workers who may not even have a high school degree. The details of the user interface design such as screen designs, menu structure, navigation diagram, etc. should be documented in a separate user interface specification document.

External interface requirements:

User interfaces: This section should describe a high-level description of various interfaces and various principles to be followed.The details of the user interface design should be documented in a separate user interface specification document.

Hardware interfaces: This section should describe the interface between the software and the hardware components of the system. This section may include the description of the supported device types, the nature of the data and control interactions between the software and the hardware, and the communication protocols to be used.

Software interfaces: This section should describe the connections between this software and other specific software components, including databases,operating systems, tools, libraries, and integrated commercial components, etc. Identify the data items that would be input to the software and the data that would be output should be identified and the purpose of each should be described.

Communications interfaces: This section should describe the requirements associated with any type of communications required by the software, such as e-mail, web access, network server communications protocols, etc. This section should define any pertinent message formatting to be used. It should also identify any communication standards that will be used, such as TCP sockets, FTP, HTTP, or SHTTP. Specify any communication security or encryption issues that may be relevant, and also the data transfer rates, and synchronisation mechanisms.

Other non-functional requirements: This section contains a description of non- functional requirements that are neither design constraints and nor are external interface requirements. An important example is a performance requirement such as the number of transactions completed per unit time. Besides performance requirements, the other non-functional requirements to be described in this section may include reliability issues, accuracy of results, and security issues.

Performance requirements: Aspects such as number of transaction to be completed per second should be specified here. Some performance requirements may be specific to individual functional requirements or features. These should also be specified here.

Safety requirements: Those requirements that are concerned with possible loss or damage that could result from the use of the software are specified here. For example, recovery after power failure, handling software and hardware failures, etc. may be documented here.

Security requirements: This section should specify any requirements regarding security or privacy requirements on data used or created by the software. Any user identity authentication requirements should be described here. It should also refer to any external policies or regulations concerning the security issues. Define any security or privacy certifications that must be satisfied.

Goals of implementation:-

The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

The different scenarios occur depending on the amount entered for withdrawal. The different scenarios are essentially different behaviour exhibited by the system for the same high-level function. Typically, each user input and the corresponding system action may be considered as a sub-requirement of a high-level requirement. Thus, each high-level requirement can consist of several sub-requirements.

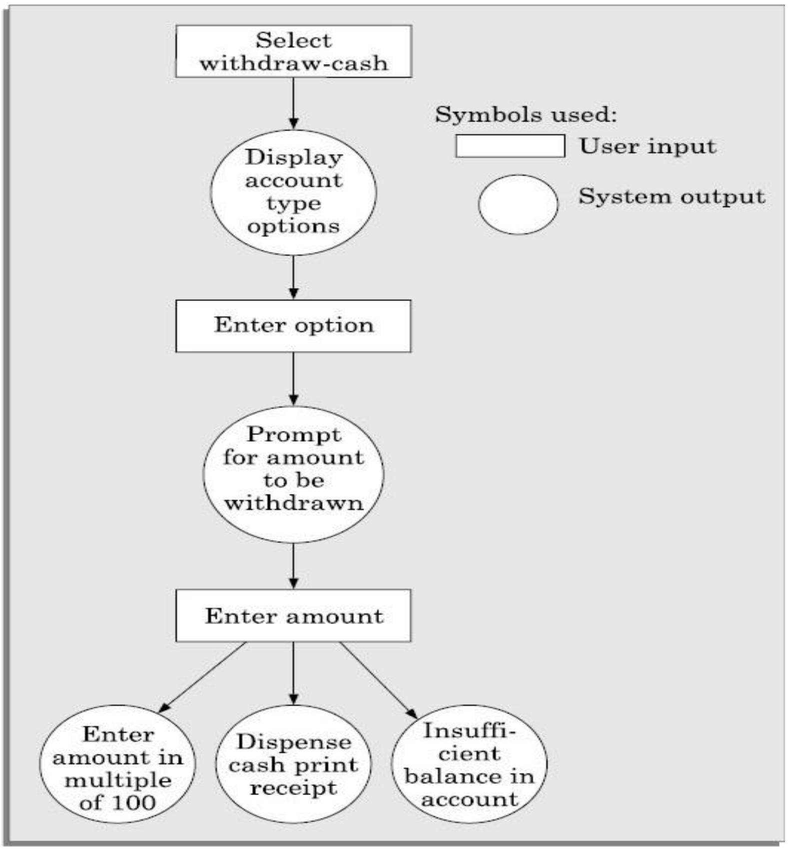


Figure: User and system interactions in high-level functional requirement.

Overall description of organisation of SRS document

Product perspective: This section needs to briefly state as to whether the software is intended to be a replacement for a certain existing systems, or it is a new software. If the software being developed would be used as a component of a larger system, a simple schematic diagram can be given to show the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.

Product features: This section should summarize the major ways in which the software would be used. Details should be provided in Section 3 of the document. So, only a brief summary should be presented here.

User classes: Various user classes that are expected to use this software are identified and described here. The different classes of users are identified by the types of functionalities that they are expected to invoke, or their levels of expertise in using computers.

Operating environment: This section should discuss in some detail the hardware platform on which the software would run, the operating system, and other application software with which the developed software would interact.

Design and implementation constraints: In this section, the different constraints on the design and implementation are discussed. These might include—corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; specific programming language to be used; specific communication protocols to be used; security considerations; design conventions or programming standards.

User documentation: This section should list out the types of user documentation, such as user manuals, on-line help, and trouble-shooting manuals that will be delivered to the customer along with the software.

A good SRS document should properly characterise the conditions under which different scenarios of interaction occur. That is, a high-level function might involve different steps to be undertaken as a consequence of some decisions made after each step. Sometimes the conditions can be complex and numerous and several alternative interaction and processing sequences may exist depending on the outcome of the corresponding condition checking. A simple text description in such cases can be difficult to comprehend and analyse. In such situations, a decision tree or a decision table can be used to represent the logic and the processing involved. Also, when the decision making in a functional requirement has been represented as a decision table, it becomes easy to automatically or at least manually design test cases.

There are two main techniques available to analyse and represent complex processing logic—decision trees and decision tables. Once the decision making logic is captured in the form of trees or tables, the test cases to validate these logic can be automatically obtained. It should, however, be noted that decision trees and decision tables have much broader applicability than just specifying complex processing logic in an SRS document. For instance, decision trees and decision tables find applications in information theory and switching theory.

Decision tree:

A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken. The edges of a decision tree represent conditions and the leaf nodes represent the actions to be performed depending on the outcome of testing the condition.

Example: -Consider Library Membership Automation Software (LMS) where it should support the following three options**:**

- New member
- Renewal
- Cancel membership

New member option-

Decision: When the 'new member' option is selected, the software asks details about the member like the member's name, address, phone number etc.

Action: If proper information is entered then a membership record for the member is created and a bill is printed for the annual membership charge plus the security deposit payable.

Renewal option-

Decision: If the 'renewal' option is chosen, the LMS asks for the member's name and his membership number to check whether he is a valid member or not.

Action: If the membership is valid then membership expiry date is updated and the annual membership bill is printed, otherwise an error message is displayed.

Cancel membership option-

Decision: If the 'cancel membership' option is selected, then the software asks for member's name and his membership number.

Action: The membership is cancelled, a cheque for the balance amount due to the member is printed and finally the membership record is deleted from the database.

Decision tree representation of the above example -

The following tree shows the graphical representation of the above example. After getting information from the user, the system makes a decision and then performs the corresponding actions.

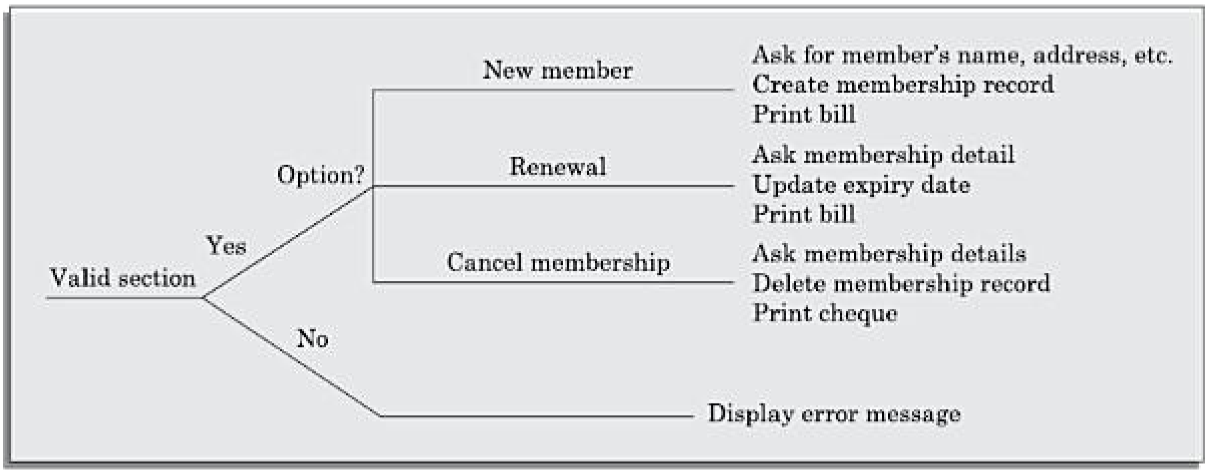


Fig: Decision tree for LMS

Decision table:

A decision table is used to represent the complex processing logic in a tabular or a matrix form. The upper rows of the table specify the variables or conditions to be evaluated. The lower rows of the table specify the actions to be taken when the corresponding conditions are satisfied. A column in a table is called a *rule*. A rule implies that if a condition is true, then the corresponding action is to be executed.

Example: -Consider the previously discussed LMS example. The following decision table (fig. 3.5) shows how to represent the LMS problem in a tabular form. Here the table is divided into two parts, the upper part shows the conditions and the lower part shows what actions are taken. Each column of the table is a rule.

Conditions

Valid selection	No	Yes	Yes	Yes
New member	-	Yes	No	No
Renewal	-	No	Yes	No
Cancellation	-	No	No	Yes
Actions				
Display error message	x	-	-	-
Ask member's details	-	x	-	-
Build customer record	-	x	-	-
Generate bill	-	x	x	-
Ask member's name & membership number	-	-	x	x
Update expiry date	-	-	x	-
Print cheque	-	-	-	x
Delete record	-	-	-	x

Fig: Decision table for LMS

From the above table you can easily understand that, if the valid selection condition is false then the action taken for this condition is 'display error message'. Similarly, the actions taken for other conditions can be inferred from the table.

Decision table *versus* decision tree

Even though both decision tables and decision trees can be used to represent complex program logic, they can be distinguishable on the following three considerations:

Readability: Decision trees are easier to read and understand when the number of conditions are small. On the other hand, a decision table causes the analyst to look at every possible combination of conditions which he might otherwise omit.

Explicit representation of the order of decision making: In contrast to the decision trees, the order of decision making is abstracted out in decision tables. A situation where decision tree is more useful is when multilevel decision making is required. Decision trees can more intuitively represent multilevel decision making hierarchically, whereas decision tables can only represent a single decision to select the appropriate action for execution.

Representing complex decision logic: Decision trees become very complex to understand when the number of conditions and actions increase. It may even be to draw the tree on a single page. When very large number of decisions are involved, the decision table representation may be preferred.

Formal technique

A formal technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realizable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc. The mathematical basis of a formal method is provided by the specification language.

Formal specification language

A formal specification language consists of two sets syn and sem, and a relation sat between them. The set syn is called the syntactic domain, the set sem is called the semantic domain, and the relation sat is called the satisfaction relation.For a given specification syn, and model of the system sem, if sat (syn, sem), as shown in figure then syn is said to be the specification of sem, and sem is said to be the specificand of syn.

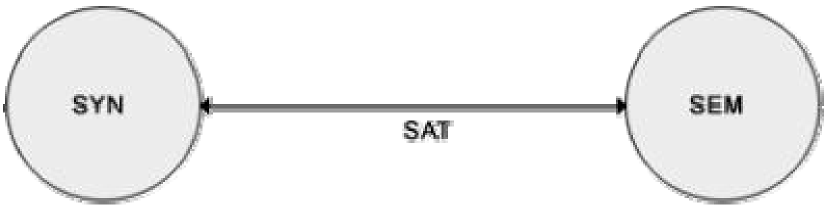


Fig: sat (syn, sem)

Syntactic Domains

The syntactic domain of a formal specification language consists of an alphabet of symbols and set of formation rules to construct well-formed formulas from the alphabet. The well-formed fsormulas are used to specify a system.

Semantic Domains

Formal techniques can have considerably different semantic domains. Abstract data type specification languages are used to specify algebras, theories, and programs. Programming languages are used to specify functions from input to output values. Concurrent and distributed system specification languages are used to specify state sequences, event sequences, state-transition sequences, synchronization trees, partial orders, state machines, etc.

Satisfaction Relation

Given the model of a system, it is important to determine whether an element of the semantic domain satisfies the specifications. This satisfaction is determined by using a homomorphism known as semantic abstraction function. The semantic abstraction function maps the elements of the semantic domain into equivalent classes. There can be different specifications describing different aspects of a system model, possibly using different specification languages. Some of these specifications describe the system's behavior and the others describe the system's structure. Consequently, two broad classes of semantic abstraction functions are defined: those that preserve a system's behavior and those that preserve a system's structure.

Model-oriented vs. property-oriented approaches

Formal methods are usually classified into two broad categories – model-oriented and property – oriented approaches. In a model-oriented style, one defines a system's behavior directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, sequences, etc.In the property-oriented style, the system's behavior is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy.

Example:-

Let us consider a simple producer/consumer example. In a property- oriented style, it is probably started by listing the properties of the system like: the consumer can start consuming only after the producer has produced an item, the producer starts to produce an item only after the consumer has consumed the last item, etc. A good example of a producer-consumer problem is CPU-Printer coordination. After processing of data, CPU outputs characters to the buffer for printing. Printer, on the other hand, reads characters from the buffer and prints them. The CPU is constrained by the capacity of the buffer, whereas the printer is constrained by an empty buffer.Examples of property-oriented specification styles are axiomatic specification and algebraic specification.

In a model-oriented approach, we start by defining the basic operations, p (produce) and c (consume). Then we can state that $S1 + p \rightarrow S$, $S + c \rightarrow S1$. Thus the model-oriented approaches essentially specify a program by writing another, presumably simpler program. Examples of popular model-oriented specification techniques are Z, CSP, CCS, etc.

Model-oriented approaches are more suited to use in later phases of life cycle because here even minor changes to a specification may lead to drastic changes to the entire specification. They do not support logical conjunctions (AND) and disjunctions (OR).

Property-oriented approaches are suitable for requirements specification because they can be easily changed. They specify a system as a conjunction of axioms and you can easily replace one axiom with another one.

Operational semantics

Informally, the operational semantics of a formal method is the way computations are represented. There are different types of operational semantics according to what is meant by a single run of the system and how the runs are grouped together to describe the behavior of the system. Some commonly used operational semantics are as follows:

Linear Semantics:-

In this approach, a run of a system is described by a sequence (possibly infinite) of events or states. The concurrent activities of the system are represented by non-deterministic interleavings of the automatic actions. For example, a concurrent activity $a \| b$ is represented by the set of sequential activities a;b and b;a. This is simple but rather unnatural representation of concurrency. The behavior of a system in this model consists of the set of all its runs. To make this model realistic, usually justice and fairness restrictions are imposed on computations to exclude the unwanted interleavings.

Branching Semantics:-

In this approach, the behavior of a system is represented by a directed graph as shown in the figure.The nodes of the graph represent the possible states in the evolution of a system. The descendants of each node of the graph represent the states which can be generated by any of the atomic actions enabled at that state. An example involving the transactions in an ATM is shown in fig. Although this semantic model distinguishes the branching points in a computation, still it represents concurrency by interleaving.

Maximally parallel semantics:-

In this approach, all the concurrent actions enabled at any state are assumed to be taken together. This is again not a natural model of concurrency since it implicitly assumes the availability of all the required computational resources.

Partial order semantics:-

Under this view, the semantics ascribed to a system is a structure of states satisfying a partial order relation among the states (events). The partial order represents a precedence ordering among events, and constraints some events to occur only after some other events have occurred; while the occurrence of other events (called concurrent events) is considered to be incomparable. This fact identifies concurrency as a phenomenon not translatable to any interleaved representation.

For example, figure shows the semantics implied by a simplified beverage selling machine. From the figure, we can infer that beverage is dispensed only if an inserted coin is accepted by the machine (precedence). Similarly, preparation of ingredients and milk are done simultaneously (concurrence). Hence, node Ingredient can be compared with node Brew, but neither can it be compared with node Hot/Cold nor with node Accepted.

Merits of formal requirements specification

Formal methods possess several positive features, some of which are discussed below.

- Formal specifications encourage rigour. Often, the very process of construction of a rigorous specification is more important than the formal specification itself. The construction of a rigorous specification clarifies several aspects of system behavior that are not obvious in an informal specification.
- Formal methods usually have a well-founded mathematical basis. Thus, formal specifications are not only more precise, but also mathematically sound and can be used to reason about the properties of a specification and to rigorously prove that an implementation satisfies its specifications.
- Formal methods have well-defined semantics. Therefore, ambiguity is specifications is automatically avoided when one formally specifies a system.
- The mathematical basis of the formal methods facilitates automating the analysis of specifications. For example, a tableau-based technique has been used to automatically check the consistency of specifications. Also, automatic theorem proving techniques can be used to verify that an implementation satisfies its specifications. The possibility of automatic verification is one of the most important advantages of formal methods.
- Formal specifications can be executed to obtain immediate feedback on the features of the specified system. This concept of executable specifications is related to rapid prototyping. Informally, a prototype is a “toy” working model of a system that can provide immediate feedback on the behavior of the specified system, and is especially useful in checking the completeness of specifications.

Limitations of formal requirements specification

- It is clear that formal methods provide mathematically sound frameworks within large, complex systems can be specified, developed and verified in a systematic rather than in an ad hoc manner. However, formal methods suffer from several shortcomings, some of which are the following:

- Formal methods are difficult to learn and use.

- The basic incompleteness results of first-order logic suggest that it is impossible to check absolute correctness of systems using theorem proving techniques.

- Formal techniques are not able to handle complex problems. This shortcoming results from the fact that, even moderately complicated problems blow up the complexity of formal specification and their analysis. Also, a large unstructured set of mathematical formulas is difficult to comprehend.

Axiomatic specification

- In axiomatic specification of a system, first-order logic is used to write the pre and post-conditions to specify the operations of the system in the form of axioms. The pre-conditions basically capture the conditions that must be satisfied before an operation can successfully be invoked. In essence, the pre-conditions capture the requirements on the input parameters of a function. The post-conditions are the conditions that must be satisfied when a function completes execution for the function to be considered to have executed successfully. Thus, the post- conditions are essentially constraints on the results produced for the function execution to be considered successful.

- The following are the sequence of steps that can be followed to systematically develop the axiomatic specifications of a function:

- Establish the range of input values over which the function should behave correctly. Also find out other constraints on the input parameters and write it in the form of a predicate.

- Specify a predicate defining the conditions which must hold on the output of the function if it behaved properly.

- Establish the changes made to the function's input parameters after execution of the function. Pure mathematical functions do not change their input and therefore this type of assertion is not necessary for pure functions.

- Combine all of the above into pre and post conditions of the function.

Example1: -Specify the pre- and post-conditions of a function that takes a real number as argument and returns half the input value if the input is less than or equal to 100, or else returns double the value.

$$f\ (x : real) : real$$

$$pre : x \in R$$

$$post : \{(x\leq 100) \wedge (f(x) = x/2)\} \vee \{(x>100) \wedge (f(x) = 2*x)\}$$

Example2: Axiomatically specify a function named search which takes an integer array and an integer key value as its arguments and returns the index in the array where the key value is present.

$$search(X : IntArray, key :Integer) :Integer$$

$$pre : \exists\ i \in [Xfirst....Xlast], X[i] = key$$

$$post : \{(X'[search(X, key)] = key) \wedge (X = X')\}$$

Here the convention followed is: If a function changes any of its input parameters and if that parameter is named X, then it is referred to as X′ after the function completes execution.mes faster.

Algebraic specification

In the algebraic specification technique an object class or type is specified in terms of relationships existing between the operations defined on that type. It was first brought into prominence by Guttag [1980, 1985] in specification of abstract data types. Various notations of algebraic specifications have evolved, including those based on OBJ and Larch languages.

Representation of algebraic specification:

Essentially, algebraic specifications define a system as a heterogeneous algebra. A heterogeneous algebra is a collection of different sets on which several operations are defined. Traditional algebras are homogeneous. A homogeneous algebra consists of a single set and several operations; {I, +, -, *, /}. In contrast, alphabetic strings together with operations of concatenation and length {A, I, con, len}, is not a homogeneous algebra, since the range of the length operation is the set of integers.

Each set of symbols in the algebra, in turn, is called a *sort* of the algebra. To define a heterogeneous algebra, we first need to specify its signature, the involved operations, and their domains and ranges. Using algebraic specification, we define the meaning of a set of interface procedure by using equations. An algebraic specification is usually presented in four sections.

Types section: In this section, the sorts (or the data types) being used is specified.

Exception section: This section gives the names of the exceptional conditions that might occur when different operations are carried out. These exception conditions are used in the later sections of an algebraic specification.For example, in a queue, possible exceptions are novalue (empty queue), underflow (removal from an empty queue), etc.

Syntax section: This section defines the signatures of the interface procedures. The collection of sets that form input domain of an operator and the sort where the output is produced are called the *signature* of the operator. For example, PUSH takes a stack and an element as its input and returns a new stack that has been created.

append : queue x element → queue

Equations section: This section gives a set of *rewrite rules* (or equations) defining the meaning of the interface procedures in terms of each other. In general, this section is allowed to contain conditional expressions.By convention each equation is implicitly universally quantified over all possible values of the variables. Names not mentioned in the syntax section such as ‘r’ or ‘e’ are variables. The first step in defining an algebraic specification is to identify the set of required operations. After having identified the required operators, it is helpful to classify them as either basic constructor operators, extra constructor operators, basic inspector operators, or extra inspection operators. The definition of these categories of operators is as follows:

Basic construction operators. These operators are used to create or modify entities of a type. The basic construction operators are essential to generate all possible element of the type being specified. For example, ‘create’ and ‘append’ are basic construction operators for a FIFO *queue*.

Extra construction operators. These are the construction operators other than the basic construction operators. For example, the operator ‘remove’ is an extra construction operator for a FIFO queue because even without using ‘remove’, it is possible to generate all values of the type being specified

Basic inspection operators. These operators evaluate attributes of a type without modifying them, e.g., eval, get, etc. Let S be the set of operators whose range is not the data type being specified. The set of the basic operators S1 is a subset of S, such that each operator from S-S1 can be expressed in terms of the operators from S1. For example, 'isempty' is a basic inspection operator because it does not modify the FIFO queue type.

Extra inspection operators. These are the inspection operators that are not basic inspectors.
A good rule of thumb while writing an algebraic specification, is to first establish which are the constructor (basic and extra) and inspection operators (basic and extra). Then write down an axiom for composition of each basic construction operator over each basic inspection operator and extra constructor operator. Also, write down an axiom for each of the extra inspector in terms of any of the basic inspectors. Thus, if there are m1 basic constructors, m2 extra constructors, n1 basic inspectors, and n2 extra inspectors, we should have m1 × (m2+n1) + n2 axioms are the minimum required and many more axioms may be needed to make the specification complete. Using a complete set of rewrite rules, it is possible to simplify an arbitrary sequence of operations on the interface procedures.

Develop algebraic specification of simple problems

- The first step in defining an algebraic specification is to identify the set of required operations. After having identified the required operators, it is helpful to classify them into different catgories.
- A simple way to determine whether an operator is a constructor (basic or extra) or an inspector (basic or extra) is to check the syntax expression for the operator. If the type being specified appears on the right hand side of the expression then it is a constructor, otherwise it is an inspection operator. For example, in a FIFO queue, 'create' is a constructor because the data type specified 'queue' appears on the right hand side of the expression. But, 'first' and 'isempty' are inspection operators since they do not modify the *queue* data type.

Example 1:-

Let us specify a data type point supporting the operations create, xcoord, ycoord, isequal; where the operations have their usual meaning.

```
Types:
            defines point
            uses boolean, integer
Syntax:
```

1.create : integer × integer → point

2.xcoord : point → integer

3.ycoord : point → integer

4.isequal : point × point → boolean

```
Equations:
```

1.xcoord(create(x, y)) = x

2.ycoord(create(x, y)) = y

3.isequal(create(x1, y1), create(x2, y2)) = ((x1 = x2)and(y1 = y2))

In this example, we have only one basic constructor (create), and three basic inspectors (xcoord, ycoord, and isequal). Therefore, we have only 3 equations.

The rewrite rules let you determine the meaning of any sequence of calls on the point type. Consider the following expression: *isequal (create (xcoord (create(2, 3)), 5),create (ycoord (create(2, 3)), 5))*. By applying the rewrite rule 1, you can simplify the given expression as *isequal (create (2, 5), create (ycoord (create(2, 3)), 5))*. By using rewrite rule 2, you can further simplify this as *isequal (create (2, 5),create (3, 5))*. This is false by rewrite rule 3.

Example 2:

Let us specify a FIFO queue supporting the operations *create, append, remove, first,* and *isempty* where the operations have their usual meaning.

Types:

defines queue

uses boolean, integer

Exceptions:

underflow, novalue

Syntax:

1.create : ✹ queue
2.append : queue x element ✹ queue
3.remove : queue ✹ queue + {underflow}
4.first : queue ✹ element + {novalue}
5.isempty : queue ✹ boolean

Equations:

1.isempty(create()) = true
2.isempty((append(q,e)) = false
3.first(create()) = novalue
4.first(append(q,e)) = is isempty(q) then e else first(q)
5.remove(create()) = underflow
6.remove(append(q,e)) = if isempty(q) then create() else append(remove(q),e)

In this example, there are two basic constructors (create and append), one extra construction operator (remove) and two basic inspectors (first and empty). Therefore, there are 2 x (1+2) + 0 = 6 equations.

Properties of algebraic specifications

Three important properties that every algebraic specification should possess are:

- **Completeness:** This property ensures that using the equations, it should be possible to reduce any arbitrary sequence of operations on the interface procedures. There is no simple procedure to ensure that an algebraic specification is complete.
- **Finite termination property:** This property essentially addresses the following question: Do applications of the rewrite rules to arbitrary expressions involving the interface procedures always terminate? For arbitrary algebraic equations, convergence (finite termination) is undecidable. But, if the right hand side of each rewrite rule has fewer terms than the left, then the rewrite process must terminate.
- **Unique termination property:** This property indicates whether application of rewrite rules in different orders always result in the same answer. Essentially, to determine this property, the answer to the following question needs to be checked: Can all possible sequence of choices in application of the rewrite rules to an arbitrary expression involving the interface procedures always give the same number? Checking the unique termination property is a very difficult problem.

Structured specification:

Developing algebraic specifications is time consuming. Therefore efforts have been made to device ways to ease the task of developing algebraic specifications. The following are some of the techniques that have successfully been used to reduce the effort in writing the specifications.

- **Incremental specification.** The idea behind incremental specification is to first develop the specifications of the simple types and then specify more complex types by using the specifications of the simple types.

- **Specification instantiation.** This involves taking an existing specification which has been developed using a generic parameter and instantiating it with some other sort.

Advantages and disadvantages of algebraic specifications

Algebraic specifications have a strong mathematical basis and can be viewed as heterogeneous algebra. Therefore, they are unambiguous and precise.Using an algebraic specification, the effect of any arbitrary sequence of operations involving the interface procedures can automatically be studied. A major shortcoming of algebraic specifications is that they cannot deal with side effects. Therefore, algebraic specifications are difficult to interchange with typical programming languages. Also, algebraic specifications are hard to understand.

Executable specification language (4GLs).

If the specification of a system is expressed formally or by using a programming language, then it becomes possible to directly execute the specification. However, executable specifications are usually slow and inefficient, 4GLs[3] (4^{th} Generation Languages) are examples of executable specification languages. 4GLs are successful because there is a lot of commonality across data processing applications. 4GLs rely on software reuse, where the common abstractions have been identified and parameterized. Careful experiments have shown that rewriting 4GL programs in higher level languages results in up to 50% lower memory usage and also the program execution time can reduce ten folds. Example of a 4GL is Structured Query Language (SQL).

Software design and its activities

Software design deals with transforming the customer requirements, as described in the SRS document, into a form (a set of documents) that is suitable for implementation in a programming language. A good software design is seldom arrived by using a single step procedure but rather through several iterations through a series of steps. Design activities can be broadly classified into two important parts:

- Preliminary (or high-level) design and
- Detailed design

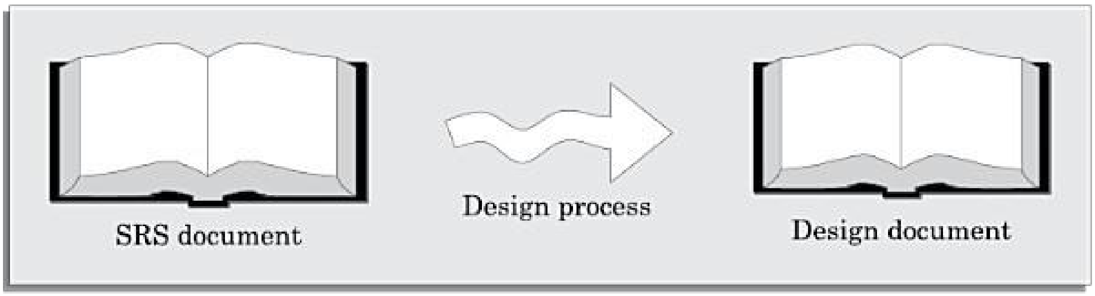


Fig:The design process.

Through high-level design, a problem is decomposed into a set of modules. The control relationships among the modules are identified, and also the interfaces among various modules are identified.

The following items are designed and documented during the design phase.

- **Different modules required:** The different modules in the solution should be clearly identified. Each module is a collection of functions and the data shared by the functions of the module. Each module should accomplish some well-defined task out of the overall responsibility of the software. Each module should be named according to the task it performs. For example, in an academic automation software, the module consisting of the functions and data necessary to accomplish the task of registration of the students should be named handle student registration.

- **Control relationships among modules:** A control relationship between two modules essentially arises due to function calls across the two modules. The control relationships existing among various modules should be identified in the design document.

- **Interfaces among different modules:** The interfaces between two modules identifies the exact data items that are exchanged between the two modules when one module invokes a function of the other module.

- **Data structures of the individual modules:** Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module. Suitable data structures for storing and managing the data of a module need to be properly designed and documented.

- **Algorithms required to implement the individual modules:** Each function in a module usually performs some processing activity. The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due considerations given to the accuracy of the results, space and time complexities.

The outcome of high-level design is called the program structure or the software architecture.A notation that is widely being used for procedural development is a tree-like diagram called the structure chart. Another popular design representation techniques called UML that is being used to document object-oriented design, involves developing several types of diagrams to document the object-oriented design of a systems.

Characteristics of a good software design

The definition of “a good software design” can vary depending on the application being designed. For example, the memory size used by a program may be an important issue to characterize a good solution for embedded software development – since embedded applications are often required to be implemented using memory of limited size due to cost, space, or power consumption considerations. For embedded applications, one may sacrifice design comprehensibility to achieve code compactness. For embedded applications, factors like design comprehensibility may take a back seat while judging the goodness of design. Therefore, the criteria used to judge how good a given design solution is can vary widely depending upon the application. Not only is the goodness of design dependent on the targeted application, but also the notion of goodness of a design itself varies widely across software engineers and academicians. However, most researchers and software engineers agree on a few desirable characteristics that every good software design for general application must possess. The characteristics are listed below:

- **Correctness:** A good design should correctly implement all the functionalities identified in the SRS document.
- **Understandability:** A good design is easily understandable
- **Efficiency:** It should be efficient.
- **Maintainability:** It should be easily amenable to change.

A design solution is correct, understandability of a design is possibly the most important issue to be considered while judging the goodness of a design. A design that is easy to understand is also easy to develop, maintain and change. Thus, unless a design is easily understandable, it would require tremendous effort to implement and maintain it.

Features of a design document:

In order to facilitate understandability, the design should have the following features:

- It should use consistent and meaningful names for various design components.
- The design should be modular. The term modularity means that it should use a cleanly decomposed set of modules.It should neatly arrange the modules in a hierarchy, e.g. in a tree-like diagram.

Cohesion

Most researchers and engineers agree that a good software design implies clean decomposition of the problem into modules, and the neat arrangement of these modules in a hierarchy. The primary characteristics of neat module decomposition are high cohesion and low coupling. Cohesion is a measure of functional strength of a module. A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

Classification of cohesion

The different classes of cohesion that a module may possess are depicted

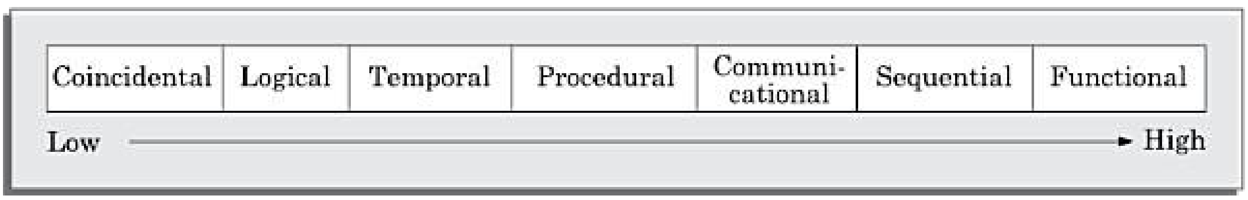


Fig:Classification of cohesion.

Coincidental cohesion: A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design. For example, in a transaction processing system (TPS), the get-input, print-error, and summarize-members functions are grouped into one module. The grouping does not have any relevance to the structure of the problem.

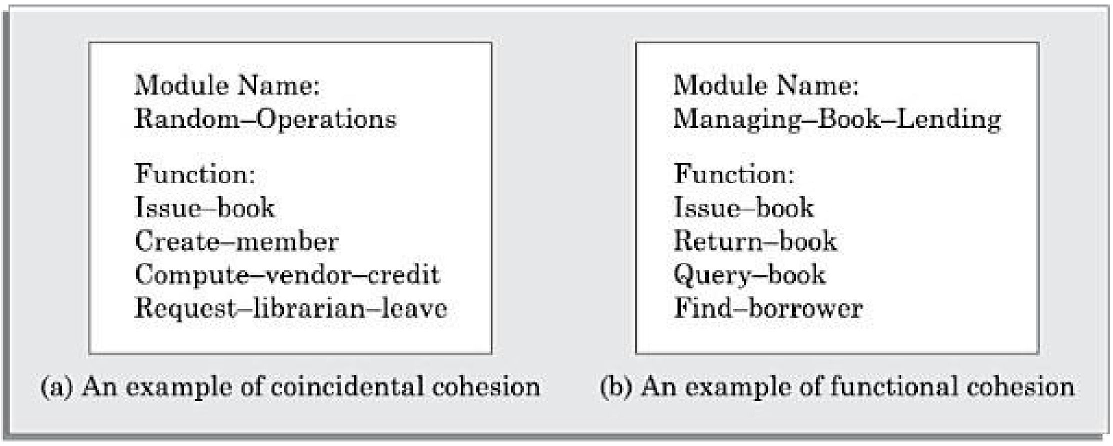


Fig: Examples of cohesion

Logical cohesion: A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc. An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.As an example of logical cohesion, consider a module that contains a set of print functions to generate various types of output reports such as grade sheets, salary slips, annual reports, etc.

Temporal cohesion: When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion. The set of functions responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion.

Procedural cohesion: A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message.The functions login(), place-order(), check-order(), print- bill(), place-order-on-vendor(), update-inventory(), and logout() all do different thing and operate on different data

Communicational cohesion: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack.As an example of procedural cohesion, consider a module named student in which the different functions in the module such as admitStudent, enterMarks, printGradeSheet, etc. access and manipulate data stored in an array named studentRecords defined within the module.

Sequential cohesion: A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next. For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.As an example consider the following situation. In an on-line store consider that after a customer requests for some item, it is first determined if the item is in stock. In this case, if the functions create-order(), check-item-availability(), place- order-on-vendor() are placed in a single module, then the module would exhibit sequential cohesion.

Functional cohesion: Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' pay-roll exhibits functional cohesion. Suppose a module exhibits functional cohesion and we are asked to describe what the module does, then we would be able to describe it using a single sentence.

Coupling

Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A module having high cohesion and low coupling is said to be functionally independent of other modules. If two modules interchange large amounts of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity.The interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.

Classification of Coupling

Even if there are no techniques to precisely and quantitatively estimate the coupling between two modules, classification of the different types of coupling will help to quantitatively estimate the degree of coupling between two modules. Five types of coupling can occur between any two modules.

Data	Stamp	Control	Common	Content

Low High

Fig: Classification of coupling

Data coupling: Two modules are data coupled, if they communicate through a parameter. An example is an elementary data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for the control purpose.

Stamp coupling: Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

Control coupling: Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another. An example of control coupling is a flag set in one module and tested in another module.

Common coupling: Two modules are common coupled, if they share data through some global data items.

Content coupling: Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.

Control Hierarchy:

Layered Arrangement Of Modules:

In a layered design solution, the modules are arranged into several layers based on their call relationships. A module is allowed to call only the modules that are at a lower layer. That is, a module should not call a module that is either at a higher layer or even in the same layer.In a layered design, the top-most module in the hierarchy can be considered as a manager that only invokes the services of the lower level module to discharge its responsibility.

Superordinate and subordinate modules: In a control hierarchy, a module that controls another module is said to be superordinate to it. Conversely, a module controlled by another module is said to be subordinate to the controller.

Visibility: A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visible to a module.

Control abstraction: In a layered design, a module should only invoke the functions of the modules that are in the layer immediately below it. In other words, the modules at the higher layers, should not be visible (that is, abstracted out) to the modules at the lower layers. This is referred to as control abstraction.

Depth and width: Depth and width of a control hierarchy provide an indication of the number of layers and the overall span of control respectively.

Fan-out: Fan-out is a measure of the number of modules that are directly controlled by a given module.In the fan-out of the module M1 is 3. A design in which the modules have very high fan-out numbers is not a good design. The reason for this is that a very high fan-out is an indication that the module lacks cohesion. A module having a large fan-out(greater than 7) is likely to implement several different functions and not just a single cohesive function.

Fan-in: Fan-in indicates the number of modules that directly invoke a given module. High fan-in represents code reuse and is in general, desirable in a good design. In the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.

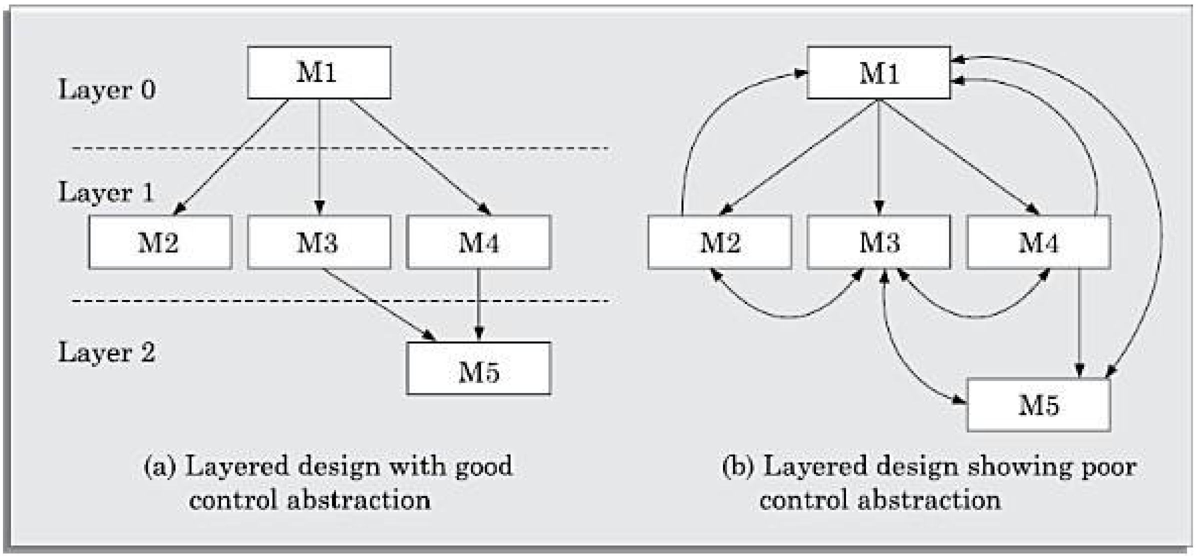


Figure: Examples of good and poor control abstraction.

Functional independence

A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules

Need for functional independence

Functional independence is a key to any good design due to the following reasons:

- **Error isolation:** Functional independence reduces error propagation. The reason behind this is that if a module is functionally independent, its degree of interaction with the other modules is less. Therefore, any error existing in a module would not directly effect the other modules.

- **Scope of reuse:** Reuse of a module becomes possible. Because each module does some well-defined and precise function, and the interaction of the module with the other modules is simple and minimal. Therefore, a cohesive module can be easily taken out and reused in a different program.
- **Understandability:** Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.

Function-oriented design

The following are the salient features of a typical function-oriented design approach:

Top-down decomposition: A system, to start with, is viewed as a black box that provides certain services (also known as high-level functions) to the users of the system.In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed functions.

A system is viewed as something that performs a set of functions.For example, consider a function create-new- library-member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge. This function may consist of the following sub- functions:

- Assign-membership-number
- create-member-record
- print-bill

Centralised system state: The system state can be defined as the values of certain data items that determine the response of the system to a user action or external event.Each of these sub-functions may be split into more detailed subfunctions and so on.

The system state is centralized and shared among different functions, e.g. data such as member-records is available for reference and updation to several functions such as:

- create-new-member
- delete-member
- update-member-record

Object-oriented design

In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities). The state is decentralized among the objects and each object manages its own state information. For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data. In fact, the functions defined for one object cannot refer or change data of other objects. Objects have their own internal data which define their state. Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from super class. Conceptually, objects communicate by message passing.

- **Data abstraction:** The principle of data abstraction implies that how data is exactly stored is abstracted away. This means that any entity external to the object (that is, an instance of an ADT) would have no knowledge about how data is exactly stored, organised, and manipulated inside the object.

- **Data structure:** A data structure is constructed from a collection of primitive data items. Just as a civil engineer builds a large civil engineering structure using primitive building materials such as bricks, iron rods, and cement; a programmer can construct a data structure as an organised collection of primitive data items such as integer, floating point numbers, characters, etc.

- **Data type:** A type is a programming language terminology that refers to anything that can be instantiated. For example, int, float, char etc., are the basic data types supported by C programming language. Thus, we can say that ADTs are user defined data types.

Function-oriented vs. object-oriented design approach

The following are some of the important differences between function-oriented and object- oriented design.

- Unlike function-oriented design methods, in OOD, the basic abstraction are not real-world functions such as sort, display, track, etc, but real- world entities such as employee, picture, machine, radar system, etc. For example in OOD, an employee pay-roll software is not developed by designing functions such as update-employee-record, get- employee-address, etc. but by designing objects such as employees, departments, etc. Grady Booch sums up this difference as "identify verbs if you are after procedural design and nouns if you are after object-oriented design"

- In OOD, state information is not represented in a centralized shared memory but is distributed among the objects of the system. For example, while developing an employee pay-roll system, the employee data such as the names of the employees, their code numbers, basic salaries, etc. are usually implemented as global data in a traditional programming system; whereas in an object-oriented system these data are distributed among different employee objects of the system. Objects communicate by message passing. Therefore, one object may discover the state information of another object by interrogating it. Of course, somewhere or other the real-world functions must be implemented. In OOD, the functions are usually associated with specific real-world entities (objects); they directly access only part of the system state information.

- Function-oriented techniques such as SA/SD group functions together if, as a group, they constitute a higher-level function. On the other hand, object-oriented techniques group functions together on the basis of the data they operate on.

Example: Fire-Alarm System

The owner of a large multi-stored building wants to have a computerized fire alarm system for his building. Smoke detectors and fire alarms would be placed in each room of the building. The fire alarm system would monitor the status of these smoke detectors. Whenever a fire condition is reported by any of the smoke detectors, the fire alarm system should determine the location at which the fire condition is reported by any of the smoke detectors, the fire alarm system should determine the location at which the fire condition has occurred and then sound the alarms only in the neighboring locations. The fire alarm system should also flash an alarm message on the computer console. Fire fighting personnel man the console round the clock. After a fire condition has been successfully handled, the fire alarm system should support resetting the alarms by the fire fighting personnel.

Function-Oriented Approach:

```
/* Global data (system state) accessible by various functions*/
BOOL detector_status[MAX_ROOMS];
int detector_locs[MAX_ROOMS];
BOOL alarm_status[MAX_ROOMS];
/* alarm activated when status is set */
```

```
int alarm_locs[MAX_ROOMS];

/* room number where alarm is located*/

int neighbor-alarm[MAX_ROOMS][10];

/* each detector has atmost 10 neighboring locations */
```

The functions which operate on the system state are:

```
interrogate_detectors();
get_detector_location();
determine_neighbor();
ring_alarm();
reset_alarm();
report_fire_location();
```

Object-Oriented Approach:

```
class detector
attributes:
status, location, neighbors
operations:
create, sense_status, get_location,
find_neighbors
class alarm
attributes:
location, status
operations:
create, ring_alarm, get_location, reset_alarm
```

- In the object oriented program, an appropriate number of instances of the class detector and alarm should be created. If the function-oriented and the object- oriented programs are examined, it can be seen that in the function-oriented program, the system state is centralized and several functions accessing this central data are defined. In case of the object-oriented program, the state information is distributed among various sensor and alarm objects.

- It is not necessary an object-oriented design be implemented by using an object-oriented language only. However, an object-oriented language such as C++ supports the definition of all the basic mechanisms of class, inheritance, objects, methods, etc. and also support all key object-oriented concepts that we have just discussed. Thus, an object-oriented language facilitates the implementation of an OOD. However, an OOD can as well be implemented using a conventional procedural language – though it may require more effort to implement an OOD using a procedural language as compared to the effort required for implementing the same design using an object-oriented language.

- Even though object-oriented and function-oriented approaches are remarkably different approaches to software design, yet they do not replace each other but complement each other in some sense. For example, usually one applies the top-down function-oriented techniques to design the internal methods of a class, once the classes are identified. In this case, though outwardly the system appears to have been developed in an object-oriented fashion, but inside each class there may be a small hierarchy of functions designed in a top-down manner.

The SA/SD technique can be used to perform the high-level design of a software. The details of SA/SD technique are discussed further.

OVERVIEW OF SA/SD METHODOLOGY

As the name itself implies, SA/SD methodology involves carrying out two distinct activities:

- Structured analysis (SA)
- Structured design (SD)

The roles of structured analysis (SA) and structured design (SD) have been shown schematically in Figure.Observe the following from the figure:

----> During structured analysis, the SRS document is transformed into a data flow diagram (DFD) model.

-----> During structured design, the DFD model is transformed into a structure chart.

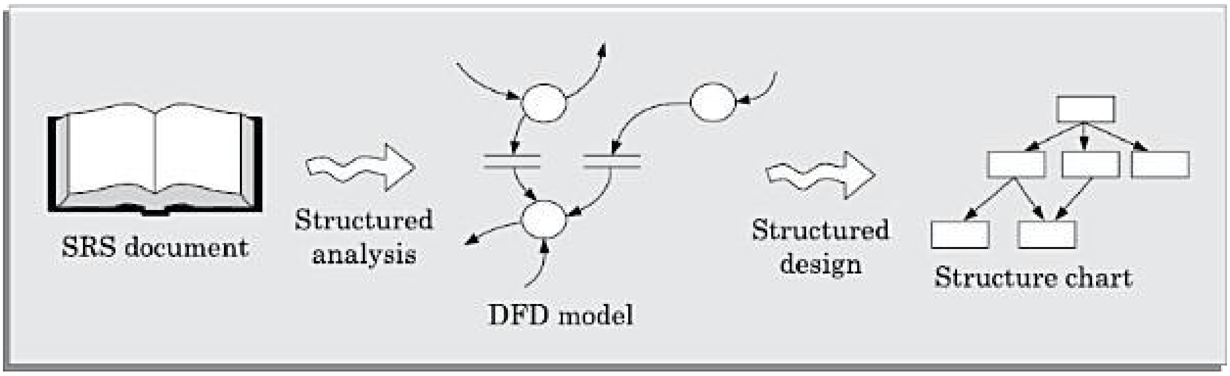


Figure: Structured analysis and structured design methodology.

The structured analysis activity transforms the SRS document into a graphic model called the DFD model. During structured analysis, functional decomposition of the system is achieved.Each function that the system needs to perform is analysed and hierarchically decomposed into more detailed functions. On the other hand, during structured design, all functions identified during structured analysis are mapped to a module structure. This module structure is also called the high-level design or the software architecture for the given problem. This is represented using a structure chart.

Structured Analysis

Structured analysis is used to carry out the top-down decomposition of a set of high-level functions depicted in the problem description and to represent them graphically. During structured analysis, functional decomposition of the system is achieved. That is, each function that the system performs is analyzed and hierarchically decomposed into more detailed functions. Structured analysis technique is based on the following essential underlying principles:

- Top-down decomposition approach.
- Divide and conquer principle.Each function is decomposed independently.
- Graphical representation of the analysis results using Data Flow Diagrams (DFDs).

Data Flow Diagram (DFD)

The DFD (also known as a bubble chart) is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among these functions. Each function is considered as a processing station (or process) that consumes some input data and produces some output data. The system is represented in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system. A DFD model uses a very limited number of primitive symbols to represent the functions performed by a system and the data flow among these functions.

Primitive symbols used for constructing DFDs

There are essentially five different types of symbols used for constructing DFDs. These primitive symbols are depicted in Figure. The meaning of these symbols are explained as follows:

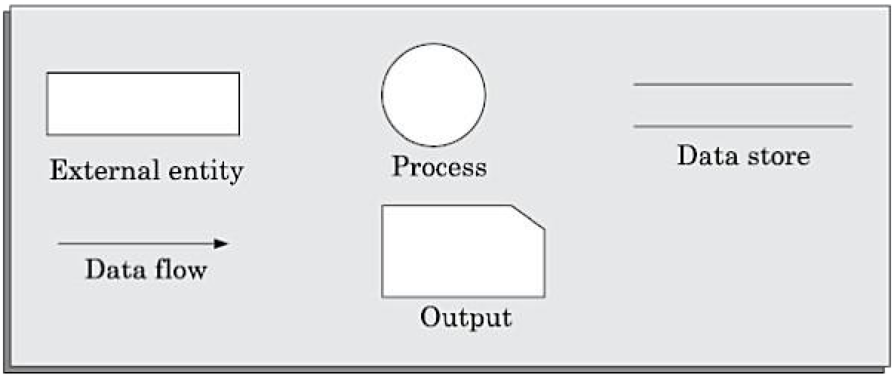


Figure: Symbols used for designing DFDs.

- **Function symbol:** A function is represented using a circle. This symbol is called a process ora bubble. Bubbles are annotated with the names of the corresponding functions .
- **External entity symbol:** An external entity such as a librarian, a library member, etc. is represented by a rectangle. The external entities are essentially those physical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system. In addition to the human users, the external entity symbols can be used to represent external hardware and software such as another application software that would interact with the software being modelled.
- **Data flow symbol:** A directed arc (or an arrow) is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow. Data flow symbols are usually annotated with the corresponding data names. For example the DFD. It shows three data flows—the data item number flowing from the process read-number to validate-number, data- item flowing into read-number, and valid-number flowing out of validate-number.
- **Data store symbol:** A data store is represented using two parallel lines. It represents a logical file. That is, a data store symbol can represent either a data structure or a physical file on disk. Each data store is connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store. An arrow flowing in or out of a data store implicitly represents the entire data of the data store and hence arrows connecting to a data store need not be annotated with the name of the corresponding data items. As an example of a data store, number is a data store.

- **Output symbol:** The output symbol i s as shown in Figure. The output symbol is used when a hard copy is produced.

Important concepts associated with constructing DFD models

Before we discuss how to construct the DFD model of a system, let us discuss some important concepts associated with DFDs:

Synchronous and asynchronous operations

If two bubbles are directly connected by a data flow arrow, then they are synchronous. This means that they operate at t he same speed. An example of such an arrangement .Here, the `validate-number` bubble can start processing only after t h e read- number bubble has supplied data to it; and the `read-number` bubble has to wait until the `validate-number` bubble has consumed its data.

However, if two bubbles are connected through a data store, then the speed of operation of the bubbles are independent. This statement can be explained using the following reasoning. The data produced by a producer bubble gets stored in the data store. It is therefore possible that the producer bubble stores several pieces of data items, even before the consumer bubble consumes any of them.

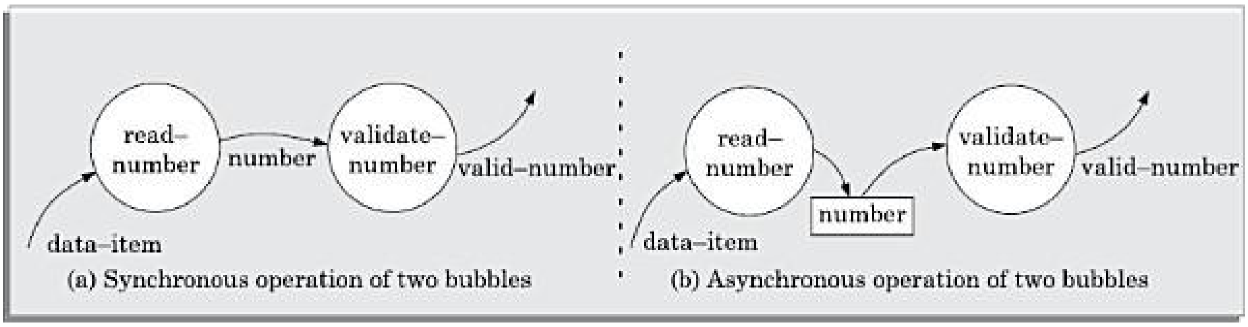


Figure: Synchronous and asynchronous data flow.

Data dictionary

A data dictionary lists all data items appearing in the DFD model of a system. The data items listed include all data flows and the contents of all data stores appearing on the DFDs in the DFD model of a system. A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items. For example, a data dictionary entry may represent that the data **grossPay** consists of the components regularPay and overtimePay.

grossPay = regularPay + overtimePay

For the smallest units of data items, the data dictionary lists their name and their type. Composite data items can be defined in terms of primitive data items using the following data definition operators. The dictionary plays a very important role in any software development process, especially for the following reasons:

- A data dictionary provides a standard terminology for all relevant data for use by the developers working in a project. A consistent vocabulary for data items is very important, since in large projects different developers of the project have a tendency to use different terms to refer to the same data, which unnecessarily causes confusion.

- The data dictionary helps the developers to determine the definition of different data structures in terms of their component elements while implementing the design.

- The data dictionary helps to perform impact analysis. That is, it is possible to determine the effect of some data on various processing activities and vice versa. Such impact analysis is especially useful when one wants to check the impact of changing an input value type, or a bug in some functionality, etc.

Data definition

Composite data items can be defined in terms of primitive data items using the following data definition operators.

- **+**: denotes composition of two data items, e.g. **a+b** represents data a and **b.**
- **[,,]**: represents selection, i.e. any one of the data items listed in the brackets can occur. For example, **[a,b]** represents either **a** occurs or **b** occurs.
- **()**: the contents inside the bracket represent optional data which may or may not appear. e.g. **a+(b)** represents either **a** occurs or **a+b** occurs.
- **{}**: represents iterative data definition, e.g. **{name}5** represents five **name** data. **{name}*** represents zero or more instances of **name** data.
- **=**: represents equivalence, e.g. **a=b+c** means that **a** represents **b** and **c**.

/* */:Anything appearing within **/*** and ***/** is considered as a comment.

DEVELOPING THE DFD MODEL OF A SYSTEM

A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs.

The DFD model of a problem consists of many of DFDs and a single data dictionary.

The DFD model of a system i s constructed by using a hierarchy of DFDs .The top level DFD is called the level 0 DFD or the context diagram. This is the most abstract (simplest) representation of the system (highest level). It is the easiest to draw and understand. At each successive lower level DFDs, more and more details are gradually introduced. To develop a higher-level DFD model, processes are decomposed into their subprocesses and the data flow among these subprocesses are identified.

To develop the data flow model of a system, first the most abstract representation (highest level) of the problem is to be worked out. Subsequently, the lower level DFDs are developed. Level 0 and Level 1 consist of only one DFD each. Level 2 may contain up to 7 separate DFDs, and level 3 up to 49 DFDs, and so on. However, there is only a single data dictionary for the entire DFD model. All the data names appearing in all DFDs are populated in the data dictionary and the data dictionary contains the definitions of all the data items.

Context diagram:

The context diagram is the most abstract data flow representation of a system. It represents the entire system as a single bubble. This bubble is labeled according to the main function of the system. The various external entities with which the system interacts and the data flow occurring between the system and the external entities are also represented. The data input to the system and the data output from the system are represented as incoming and outgoing arrows. These data flow arrows should be annotated with the corresponding data names. The name ‘context diagram’ is well justified because it represents the context in which the system is to exist, i.e. the external entities who would interact with the system and the specific data items they would be supplying the system and the data items they would be receiving from the system. The context diagram is also called as the level 0 DFD.

To develop the context diagram of the system, it is required to analyze the SRS document to identify the different types of users who would be using the system and the kinds of data they would be inputting to the system and the data they would be receiving the system. Here, the term “users of the system” also includes the external systems which supply data to or receive data from the system.

The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun). This is in contrast with the bubbles in all other levels which are annotated with verbs. This is expected since the purpose of the context diagram is to capture the context of the system rather than its functionality.

We can now describe how to go about developing the DFD model of a system more systematically.

Construction of context diagram: Examine the SRS document to determine:

- Different high-level functions that the system needs to perform.
- Data input to every high-level function.
- Data output from every high-level function.
- Interactions (data flow) among the identified high-level functions. Represent these aspects of the high-level functions in a diagrammatic form. This would form the top-level data flow diagram (DFD), usually called the DFD 0.

Construction of level 1 diagram: Examine the high-level functions described in the SRS document. If there are three to seven high-level requirements in the SRS document, then represent each of the high-level function in the form of a bubble. If there are more than seven bubbles, then some of them have to be combined. If there are less than three bubbles, then some of these have to be split.

Construction of lower-level diagrams: Decompose each high-level function into its constituent subfunctions through the following set of activities:

- Identify the different subfunctions of the high-level function.
- Identify the data input to each of these subfunctions.
- Identify the data output from each of these subfunctions.
- Identify the interactions (data flow) among these subfunctions. Represent these aspects in a diagrammatic form using a DFD.

Decomposition:-

Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into sub-functions at the successive levels of the DFD. Decomposition of a bubble is also known as factoring or exploding a bubble. Each bubble at any level of DFD is usually decomposed to anything between 3 to 7 bubbles. Too few bubbles at any level make that level superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes redundant. Also, too many bubbles, i.e. more than 7 bubbles at any level of a DFD makes the DFD model hard to understand. Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm.

Numbering of Bubbles:-

It is necessary to number the different bubbles occurring in the DFD. These numbers help in uniquely identifying any bubble in the DFD by its bubble number. The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc, etc. When a bubble numbered x is decomposed, its children bubble are numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

Balancing a DFD

The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as balancing a DFD. The concept of balancing a DFD has been illustrated in fig.In the level 1 of the DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1. In the next level, bubble 0.1 is decomposed. The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in.

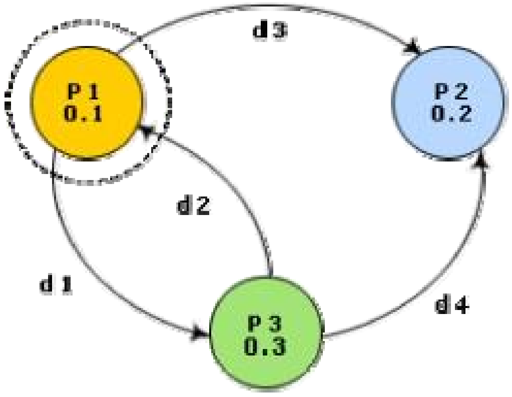


Fig:Level 1 DFD

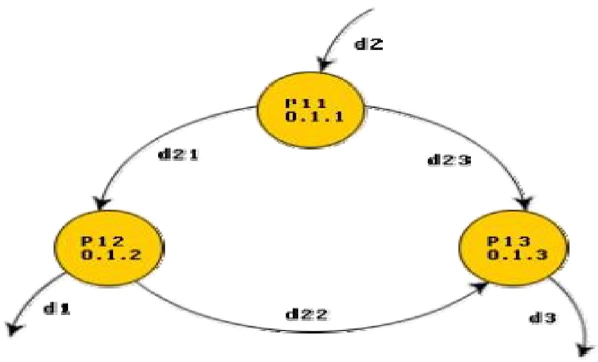


Fig:Level 2 DFD

some important shortcomings of the DFD model.

DFD models suffer from several shortcomings. The important shortcomings of the DFD models are the following:

- DFDs leave ample scope to be imprecise. In the DFD model, the function performed by a bubble is judged from its label. However, a short label may not capture the entire functionality of a bubble. For example, a bubble named find-book-position has only intuitive meaning and does not specify several things, e.g. what happens when some input information are missing or are incorrect. Further, the find-book-position bubble may not convey anything regarding what happens when the required book is missing.
- Control aspects are not defined by a DFD. For instance, the order in which inputs are consumed and outputs are produced by a bubble is not specified. A DFD model does not specify the order in which the different bubbles are executed. Representation of such aspects is very important for modeling real-time systems.
- The method of carrying out decomposition to arrive at the successive levels and the ultimate level to which decomposition is carried out are highly subjective and depend on the choice and judgment of the analyst. Due to this reason, even for the same problem, several alternative DFD representations are possible. Further, many times it is not possible to say which DFD representation is superior or preferable to another one.

- The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its sub- functions and we have to use subjective judgment to carry out decomposition.

Example.1(Tic-Tac-Toe Computer Game) Tic-tac-toe is a computer game in which a human player and the computer make alternate moves on a 3 × 3 square. A move consists of marking a previously unmarked square. The player who is first to place three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square wins. As soon as either of the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, and all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

The context diagram and the level 1 DFD are shown in Figure.

Data dictionary for the DFD model of Example

move: integer /* number between 1 to 9 */

display: game+result

game: board

board: {integer}9

result: [“computer won”, “human won”, “drawn”]

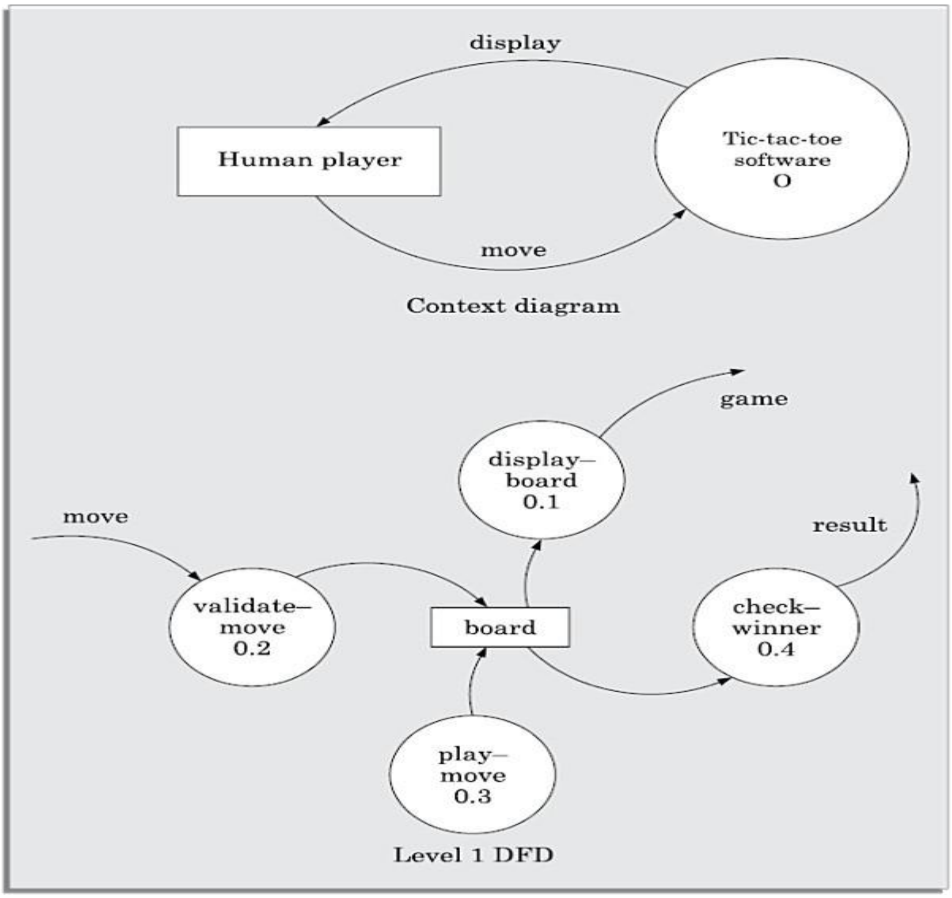


Figure: Context diagram and level 1 DFDs for Example 1

Example 2(Supermarket Prize Scheme) A super market needs to develop a software that would help it to automate a scheme that it plans to introduce to encourage regular customers. In this scheme, a customer would have first register by supplying his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Rs. 10,000. The entries against the CN are reset on the last day of every year after the prize winners' lists are generated.

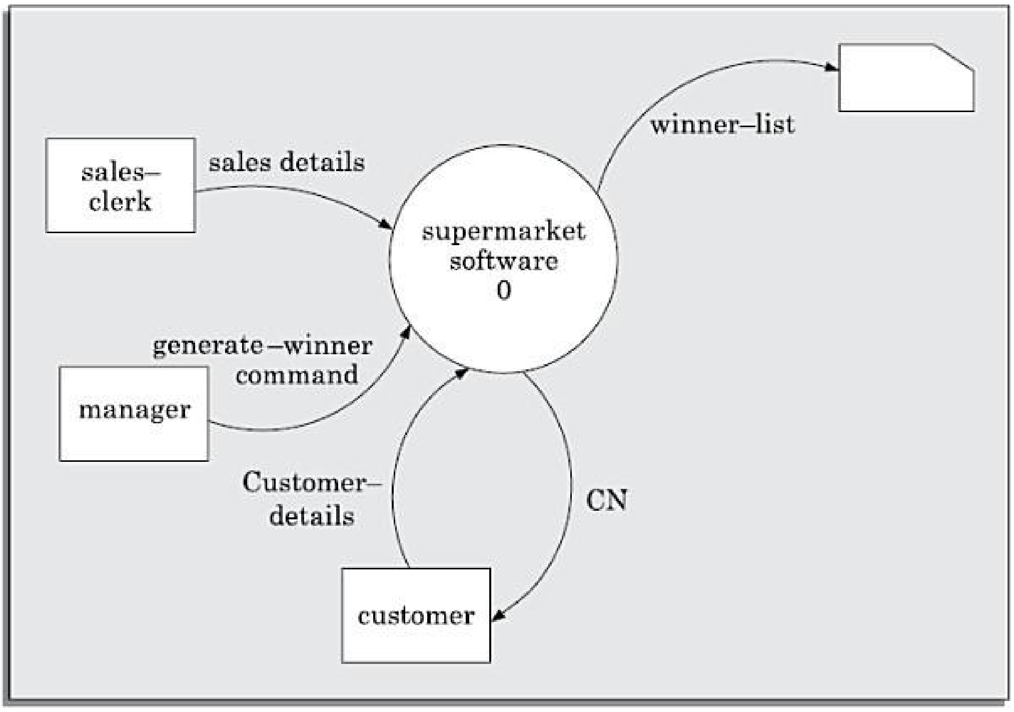


Figure: Context diagram for Example 2

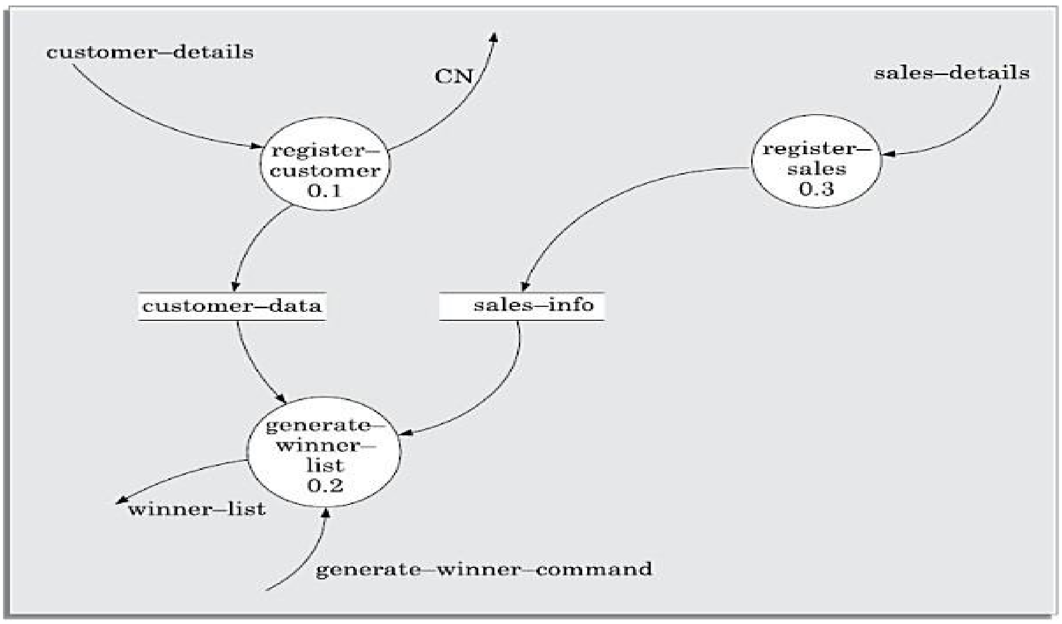


Figure: Level 1 diagram for Example 2