Software Engineering

KALIKI PHANI

SOFTWARE ENGINEERING

Unit – I: Basic concepts in software engineering and software project management

Basic concepts: abstraction versus decomposition, evolution of software engineering techniques, Software development life cycle (SDLC) models: Iterative waterfall model, Prototype model, Evolutionary model, Spiral model, RAD model, Agile models, software project management: project planning, project estimation, COCOMO, Halstead's Software Science, project scheduling, staffing, Organization and team structure, risk management, configuration management.

Important questions(unit:1):

- For the development of water level management system for skyscrapers. Suggest the suitable engineering process model. discuss this model with its phases and processes.
- For the development of online web-based hotel booking software, which software models will be more suitable? Discuss the suggested model with related phases associated with it.
- Explain how COCOMO model work for cost estimation and project estimation.
- With a neat diagram explain the following SDLC models with their advantages and disadvantages:
    - Iterative waterfall model
    - Prototype model
    - Evolutionary model
    - V model
    - RAD model
    - Spiral model
- Briefly discuss the agile process.
- Explain briefly about COCOMO – a heuristic estimation technique.
- Explain the various characteristics of software.
- Explain risk management and its principles.
- Infer Workout project cost estimates using COCOMO and schedules using PERT and GANTT charts.
- discuss about prototype model and state advantages of it.
- what are the umbrella activities of a software processes?
- which processes model lead to software reuse? Justify.
- illustrate the functioning of unified process.
- explain briefly about generic process model.
- discuss the process framework activities.
- explain unified process and elaborate on the unified process work products.
- explain personal and team process models.

Unit – II: Requirement's analysis and specification

The nature of software, The Unique nature of Webapps, Software Myths, Requirements gathering and analysis, software requirements specification, Traceability, Characteristics of a Good SRS Document, IEEE 830 guidelines, representing complex requirements using decision tables and decision trees, overview of formal system development techniques. axiomatic specification, algebraic specification.

Important questions(unit:2):

- Explain the unique nature of web apps.
- Describe software myths? discuss on various types of software myths and trust aspects of these myths.
- Discuss how requirements are facilitated and validated in software project.
- write the difficulty process for eliciting and understanding requirements from system stakeholders.
- Give a brief view on patterns for requirements modelling and also requirement modelling on webapps.
- Discuss the various requirement modelling strategies.
- what is SRS? How will you develop requirements model to finalize specifications for college admission system?
- write a short notes on roles and responsibilities of software project manager.

- Develop algebraic and axiomatic specifications for simple problems.
- Develop SRS document for sample problems using IEEE 830 format.

Unit – III: Software Design

Good Software Design, Cohesion and coupling, Control Hierarchy: Layering, Control Abstraction, Depth and width, Fan-out, Fan-in, Software design approaches, object oriented vs. function-oriented design. Overview of SA/SD methodology, structured analysis, Data flow diagram, Extending DFD technique to real life systems, Basic Object-oriented concepts, UML Diagrams, Structured design, Detailed design, Design review, Characteristics of a good user interface, User Guidance and Online Help, Mode-based Vs Mode-less Interface, Types of user interfaces, Component-based GUI development, User interface design methodology: GUI design methodology.

Important questions(unit:3):

- Explain the following
  - a. Design process.
  - b. Design model.
  - c. Design concepts.
- Discuss various steps involved in component-based development.
- Explain the rules of user interface design. (or) Explain the characteristics of good user interface.
- Explain the steps involved in web app interface design.
- Explain about content design and architecture design in detail.
- Explain the architectural context diagram for safe home security function.
- Draw the swim lane diagram for prescription refill function.
- Explain the basic design principles or concepts in software engineering.
- write and explain the interface design steps.
- Discuss the design pyramid for web apps.
- Develop UML model for development of portal for online store with various UML modelling diagrams.
- Develop user interface design for billing system for online shopping mail such as amazon. Apply user interface analysis and design steps properly.
- what is software architectural design? What are the building blocks? Develop software architecture for bus reservation system with proper styles.
- Design class-based component design with architectural model of UML for typical search engines used to search information over net.
- Explain software design? Explain data flow-oriented design?
- Explain how the design is evaluated.

Unit – IV: Coding and Testing

Coding standards and guidelines, code review, software documentation, Testing, Black Box Testing, White Box Testing, debugging, integration testing, Program Analysis Tools, system testing, performance testing, regression testing, Testing Object Oriented Programs.

Important questions(unit:3):

- Recognize the basic issues in software testing.
- Design test cases for black box and white box testing.
- Discuss the importance of test strategies for conventional software.
- Discuss various functional and unit testing techniques in detail.
- Explain the validation testing methodology.
- Explain about unit testing conditions and procedures.
- Explain the test strategies for object-oriented software.
- Briefly discuss about integration testing strategies.
- What is a decision table? How does it useful in testing? Explain it with the help of an example.

- What is testing strategy? Which type of tests are included in the testing strategy? Compare these testing methods with their pros and cons.
- Explain testing with OOA and OOD model.
- Discuss about all possible levels of software testing.
- Explain the types of black box testing in detail.
- Explain about debugging.
- Write a short note on regression testing.

Unit – V: Software quality, reliability, and other issues

Software reliability, Statistical testing, Software quality and management, ISO 9000, SEI capability maturity model (CMM), Personal software process (PSP), Six sigma, Software quality metrics, CASE and its scope, CASE environment, CASE support in software life cycle, Characteristics of software maintenance, Software reverse engineering, Software maintenance processes model, Estimation maintenance cost. Basic issues in any reuse program, Reuse approach, Reuse at organization level.

Unit Outcomes:

Student should be able to

1. Summarize various methods of software quality management.
2. Instruct the quality management standards ISO 9001, SEI CMM, PSP, and Six Sigma.
3. Outline software quality assurance, quality measures, and quality control.
4. Identify the basic issues in software maintenance, CASE support, and software reuse.

Unit – I
Basic concepts in software engineering and software project management

Software engineering discusses systematic and cost-effective techniques for software development. These techniques help develop software using an engineering approach. We mainly use these two fundamental principles in various forms and flavours in the different software development activities. A thorough understanding of these two principles is therefore needed.

Abstraction:

Def: Abstraction is the simplification of a problem by focusing on only one aspect of the problem while omitting all other aspects.
Abstraction refers to construction of a simpler version of a problem by ignoring the details. The principle of constructing an abstraction is popularly known as *modelling* (or *model construction)*.
When using the principle of abstraction to understand a complex problem, we focus our attention on only one or two specific aspects of the problem and ignore the rest. Whenever we omit some details of a problem to construct an abstraction, we construct a *model* of the problem.

A single level of abstraction can be sufficient for rather simple problems. However, more complex problems would need to be modelled as a hierarchy of abstractions. The most abstract representation would have only a few items and would be the easiest to understand. After one understands the simplest representation, one would try to understand the next level of abstraction where at most five or seven new information are added and so on until the lowest level is understood. By the time, one reaches the lowest level, he would have mastered the entire problem.

Decomposition:

Def: 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.
Decomposition is another important principle that is available to handle problem complexity. The decomposition principle is popularly known as the ***divide and conquer*** principle. A popular way to demonstrate the decomposition principle is by trying to break a large bunch of sticks tied together and then breaking them individually. The different parts after decomposition should be more or less independent of each other. That is, to solve one part you should not have to refer and understand other parts. If to solve one part you would have to understand other parts, then this would boil down to understanding all the parts together. This would effectively reduce the problem to the original problem before decomposition (the case when all the sticks tied together).

Why study software engineering?

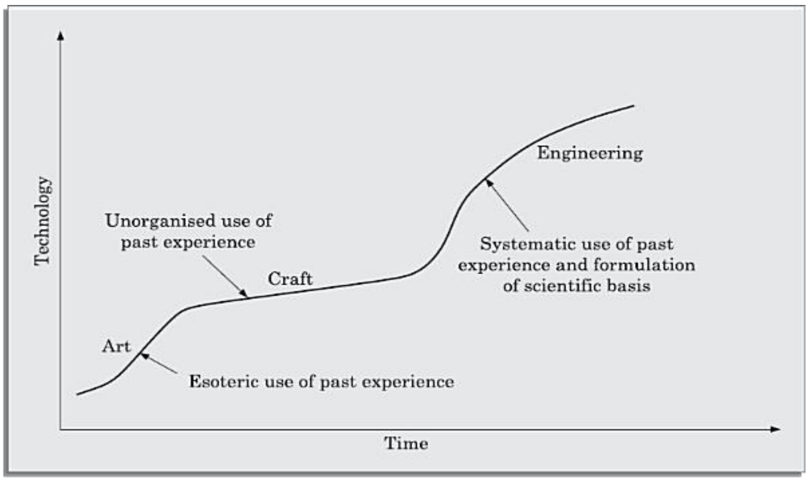
Let us examine the skills that you could acquire from a study of the software engineering principles. The following two are possibly the most important skill you could be acquiring after completing a study of software engineering:

- The skill to participate in development of large software. You can meaningfully participate in a team effort to develop a large software only after learning the systematic techniques that are being used in the industry.
- You would learn how to effectively handle complexity in a software development problem. In particular, you would learn how to apply the principles of abstraction and decomposition to handle complexity during various stages in software development such as specification, design, construction, and testing.

Evolution of software engineering-From an art to an engineering discipline:

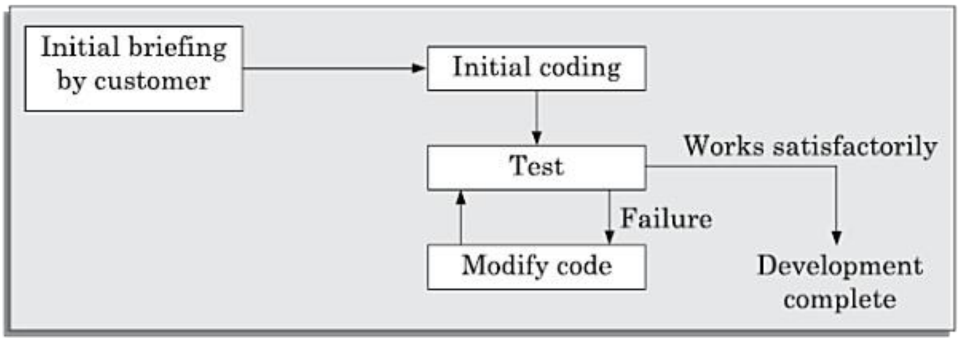
Over the years, software engineering principles have emerged from pure to a craft, and finally an engineering discipline. In early stages of programming, the style of program development is ad hoc, which is now variously being referred to as exploratory, build-and-fix, and code-and-fix styles.

The exploratory programming style is an informal technique used by programmers in the past, which has no set of rules but guided by their own intuition, experience, whims and fancies.

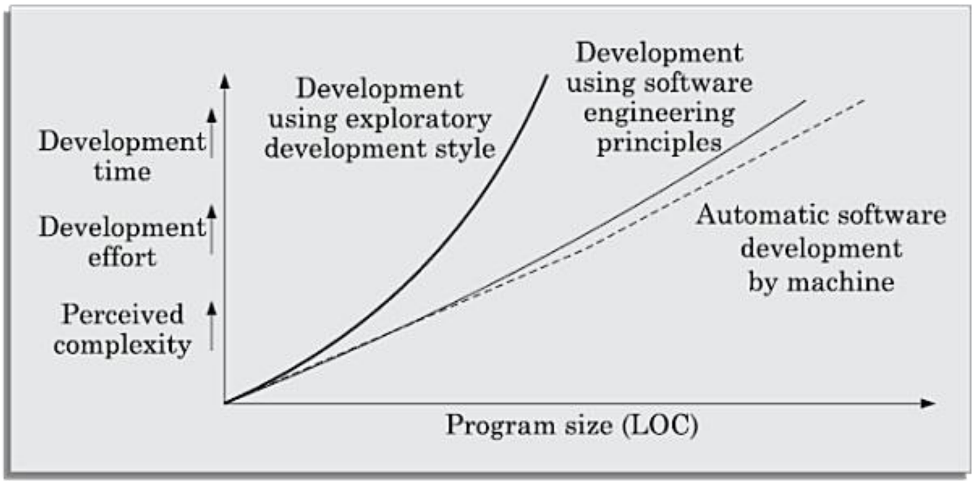


Evolution of technology development has followed strikingly similar patterns, that every technology in the initial technologies start as a form of art. Overtime it graduates to a craft and finally, emerges as an engineering discipline. We can consider the exploratory programming development style as an art and the programmers in the past who were like proficient artists (good programmers) knew certain principles that helped them to write good programs by some esoteric knowledge. The bad programmers were left to wonder how could some programmers effortlessly write elegant and correct programs each time.

The exploratory style usually leads poor quality and unmaintainable code and also makes program development very expensive as well as time consuming.



An exploratory development style can be successful when used for developing very small programs and not for professional software. In an exploratory development scenario, as the program size increases the required time and effort increases almost exponentially. For large problems, it would take too long and cost too much to be practically meaningful to develop program using exploratory programming style.



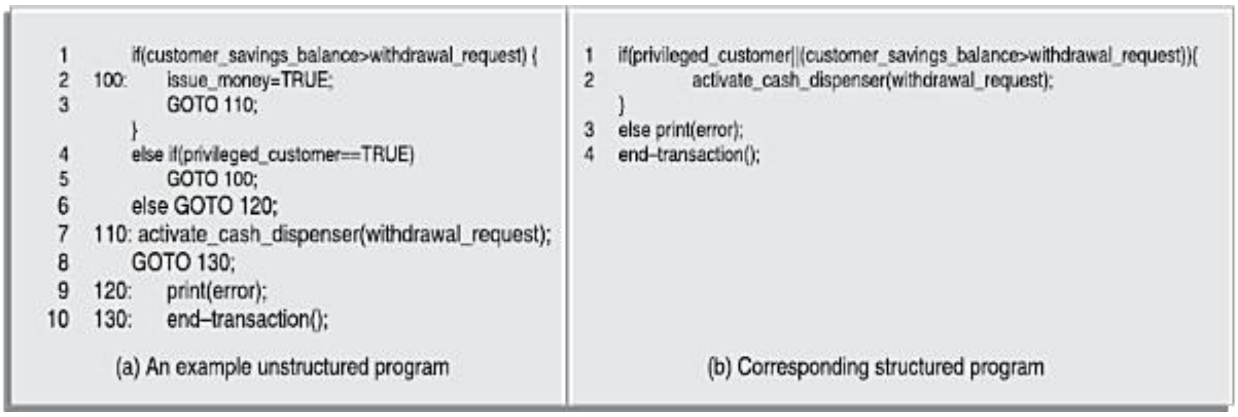
Control flow-based design:

In early 1960s, computers became faster by replacing vacuum tubes with semiconductor transistor-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.

As the size and complexity of programs kept increasing, the computer programming style proved to be insufficient. Programmers found it increasingly difficult not only to write cost-effective and correct programs, but also to understand and maintain programs written by others. So, the programmers paid attention to understand the control flow structure. A program's control flow structure indicates the sequence in which the program's instructions are executed.

Are GO TO statements the culprits?

In a landmark paper, Dijkstra [1968] published his (now famous) article "GO TO Statements Considered Harmful". He pointed out that unbridled use of GO TO statements is the main culprit in making the control structure of a program messy. GO TO statements alter the flow of control arbitrarily, resulting in too many paths. But, then why does use of too many GO TO statements makes a program hard to understand?



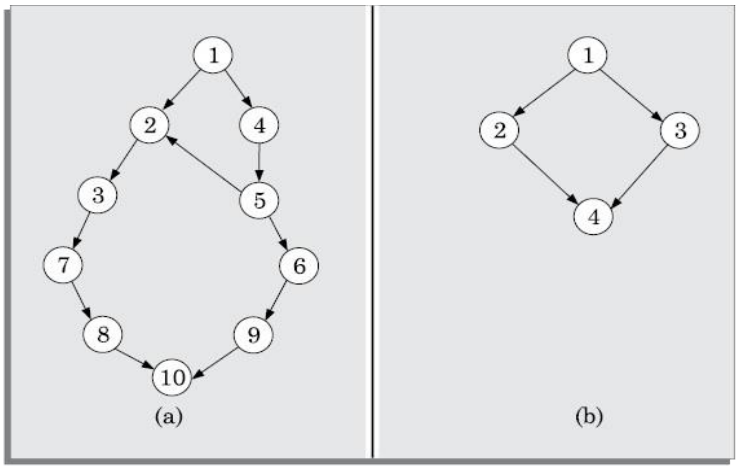
```
1        if(customer_savings_balance>withdrawal_request) {
2   100:     issue_money=TRUE;
3            GOTO 110;
         }
4        else if(privileged_customer==TRUE)
5            GOTO 100;
6        else GOTO 120;
7   110: activate_cash_dispenser(withdrawal_request);
8        GOTO 130;
9   120:     print(error);
10  130:     end–transaction();
```

(a) An example unstructured program

```
1  if(privileged_customer||(customer_savings_balance>withdrawal_request)){
2          activate_cash_dispenser(withdrawal_request);
   }
3  else print(error);
4  end–transaction();
```

(b) Corresponding structured program

An example of (a) Unstructured program (b) Corresponding structured program



Control flow graphs of the programs of Figures (a) and (b).

It became widely accepted that good programs should have very simple control structures. It is possible to distinguish good programs from bad programs by just visually examining their flow chart representations. The use of flow charts to design good control flow structures of programs became wide spread.

Structured programming- A logical extension:

A program is called structured when it uses only the sequence, selection, and iteration types of constructs and is modular. Structured programs avoid unstructured control flows by restricting the use of GO TO statements. Structured programming is facilitated, if the programming language being used supports single-entry, single-exit program constructs such as if-then-else, do-while, etc. Thus, an important feature of structured programs is the design of good control structures.

A structured program should be modular. A modular program is one which is decomposed into a set of modules such that the modules should have low interdependency among each other.

Data structure – object oriented:

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

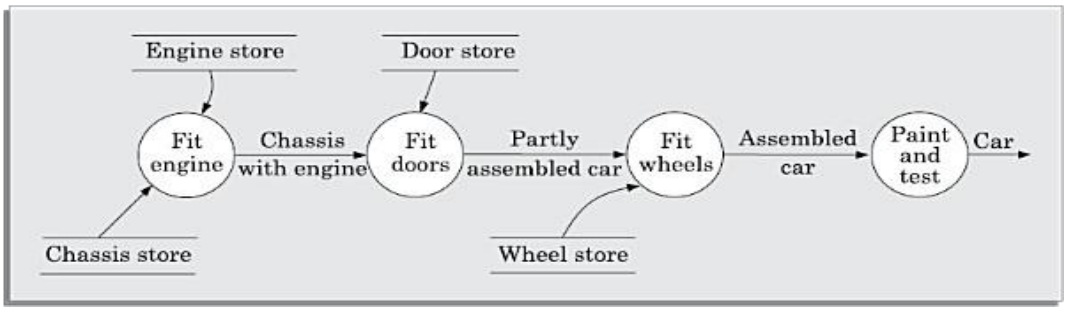
Using data structure-oriented design techniques, first a program's data structures are designed. The code structure is designed based on the data structure.

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 soon *data flow-oriented techniques* were proposed. The data flow-oriented techniques advocate that the major data items handled by a system must be identified and the processing required on these data items to produce the desired outputs should be determined. The functions (also called as processes) and the data items that are exchanged between the different functions are represented in a diagram known as a data flow diagram (DFD). The program structure can be designed from the DFD representation of the problem.

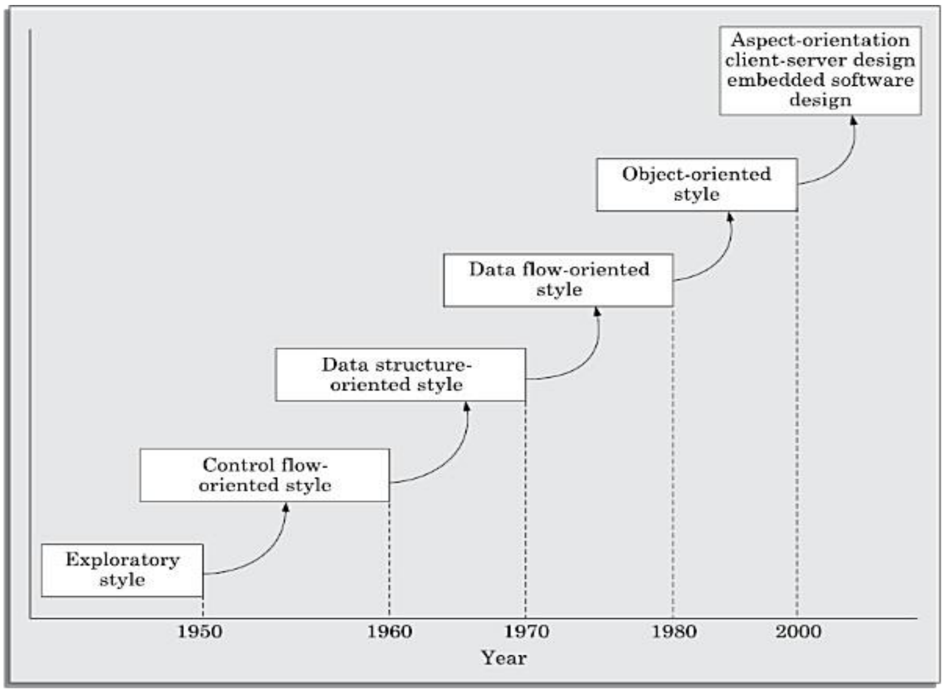
DFDs: A crucial program representation for procedural program design:

DFD has proven to be a generic technique which is being used to model all types of systems, and not just software systems. For example, the data-flow representation of an automated car assembly plant. In an automated car assembly plant, there are several processing stations (also called *workstations*) which are located along side of a conveyor belt (also called an *assembly line*). Each workstation is specialised to do jobs such as fitting of wheels, fitting the engine, spray painting the car, etc. As the partially assembled program moves along the assembly line, different workstations perform their respective jobs on the partially assembled software. Each circle in the DFD model of Figure represents a workstation (called a *process* or *bubble*). Each workstation consumes certain input items and produces certain output items. As a car under assembly arrives at a workstation, it fetches the necessary items to be fitted from the corresponding stores (represented by two parallel horizontal lines), and as soon as the fitting work is complete passes on to the next workstation. It is easy to understand the DFD model of the car assembly plant shown in Figure even without knowing anything regarding DFDs. In this regard, we can say that a major advantage of the DFDs is their simplicity.



Object oriented design:

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


Aspect-orientation client-server design embedded software design
Object-oriented style
Data flow-oriented style
Data structure-oriented style
Control flow-oriented style
Exploratory style
1950
1960
1970
1980
2000
Year

Evolution of software design techniques

Software development life cycle (SDLC):

The process of dividing software development work ingo to smaller, parallel or sequential steps or subprocesses to improve design, product management and project management is called software development life cycle (SDLC).

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

Software development life cycle contain a process framework, which is collection of activities, actions and tasks. Process framework is an adaptable approach that enables the software team to pick and choose the appropriate set of work actions and tasks.

Process *versus* methodology:

A **process** is a collection of activities, actions, tasks that are performed when some work product is to be created. An activity strives to achieve a broad objective.an action encompasses a set of tasks that produce a major work product. A task focuses on a small, but well-defined objective that produces a tangible outcome.

A **methodology**, on the other hand, prescribes a set of steps for carrying out a specific life cycle activity. It may also include the rationale and philosophical assumptions behind the set of steps through which the activity is accomplished.

Why document a development process?

It is not enough for an organisation to just have a well-defined development process, but the development process needs to be properly documented.

- A documented process model ensures that every activity in the life cycle is accurately defined. Without documentation, the activities and their ordering tend to be loosely defined, leading to confusion and misinterpretation by different teams in the organisation.
- An undocumented process gives a clear indication to the members of the development teams about the lack of seriousness on the part of the management of the organisation about following the process. Therefore, an undocumented process serves as a hint to the developers to loosely follow the process. The symptoms of an undocumented process are easily visible—designs are shabbily done, reviews are not carried out rigorously, etc.

Note: A documented development process forms a common understanding of the activities to be carried out among the software developers and helps them to develop software in a systematic and disciplined manner. A documented development process model, besides preventing the misinterpretations that might occur when the development process is not adequately documented, also helps to identify inconsistencies, redundancies, and omissions in the development process

A generic process framework for software engineering defines five framework activities –feasible study, requirement analysis and specification, design, construction and maintenance.

Feasibility study:

It's important to determine whether it would be *financially* and *technically feasible* to develop the software. The feasibility study involves carrying out several activities such as collection of basic information relating to the software, the processing required to be carried out on these data, the output data required to be produced by the system, as well as various constraints on the development. These collected data are analysed to perform at the following:

- Development of an overall understanding of the problem
- Formulation of the various possible strategies for solving the problem
- Evaluation of the different solution strategies

The outcome of the feasibility study phase by noting that other than deciding whether to take up a project or not, taking decisions regarding the solution strategy is defined. Therefore, feasibility study is a very crucial stage in software development.

Requirement analysis and specification:

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly.

- **Requirements gathering and analysis:** The goal of the requirements gathering activity is to collect all relevant information regarding the software to be developed from the customer with a view to clearly understand the requirements. The goal of the requirements analysis activity is to weed out the incompleteness and inconsistencies in these gathered requirements
- **Requirement's specification:** This is called a *software requirements specification* (SRS) document. The SRS document is written using end-user terminology to increase customer understandability. The SRS document normally serves as a contract between the development team and the customer.

Design:

After the analysis stage, it's time to create design or a blueprint for the software. Architects and senior developers create a high-level design of the software architecture along with a low-level design describing how each and every component in software should work.

Construction:

This is also called coding, testing and installation stage. The coding phase is also sometimes called the *implementation phase*, since the design is implemented into a workable solution in this phase. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually unit tested.

Unit testing is carried out to determine the correct working of the individual modules. The specific activities carried out during unit testing include designing test cases, testing, debugging to fix problems, and management of test cases. We shall discuss the coding and unit testing techniques

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

Maintenance:

The total effort spent on maintenance of a typical software during its operation phase is much more than that required for developing the software itself. Many studies carried out in the past confirm this and indicate that the ratio of relative effort of developing a typical software product and the total effort spent on its maintenance is roughly 40:60. Maintenance is required in the following three types of situations

- **Corrective maintenance:** This type of maintenance is carried out to correct errors that were not discovered during the product development phase.
- **Perfective maintenance:** This type of maintenance is carried out to improve the performance of the system, or to enhance the functionalities of the system based on customer's requests.

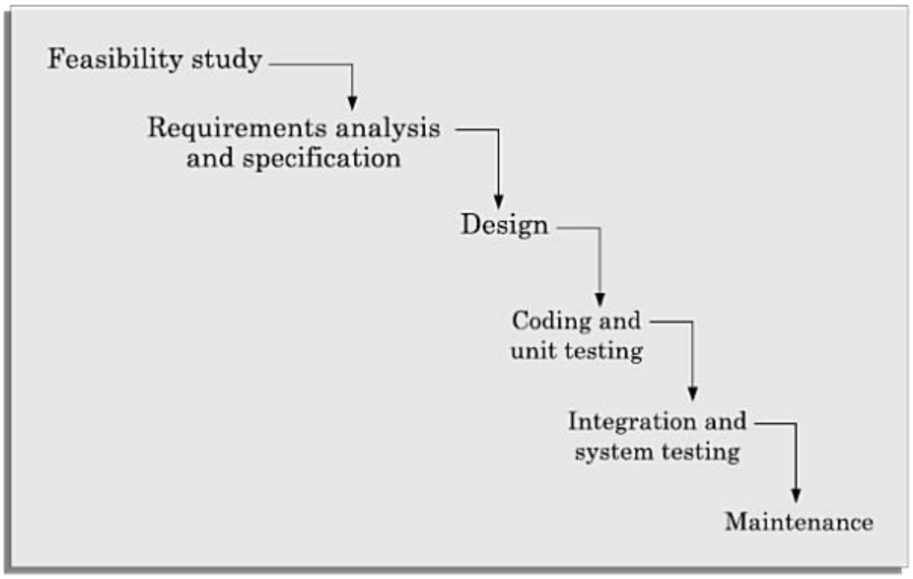
- **Adaptive maintenance:** Adaptive maintenance is usually required for porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system.

.

Most modern development processes can be vaguely described as agile. Other methodologies include waterfall prototyping, iterative and incremental development, spiral development, rapid application development and extreme programming.

Waterfall Model:

Waterfall model is the simplest model of software development paradigm. All the phases of SDLC will function one after another in linear manner. That is, when the first phase is finished then only the second phase will start and so on.


Feasibility study
Requirements analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

This model assumes that everything is carried out and taken place perfectly as planned in the previous stage and there is no need to think about the past issues that may arise in the next phase. This model does not work smoothly if there are some issues left at the previous step. The sequential nature of model does not allow us to go back and undo or redo our actions.
This model is best suited when developers already have designed and developed similar software in the past and are aware of all its domains.

Iterative waterfall model:

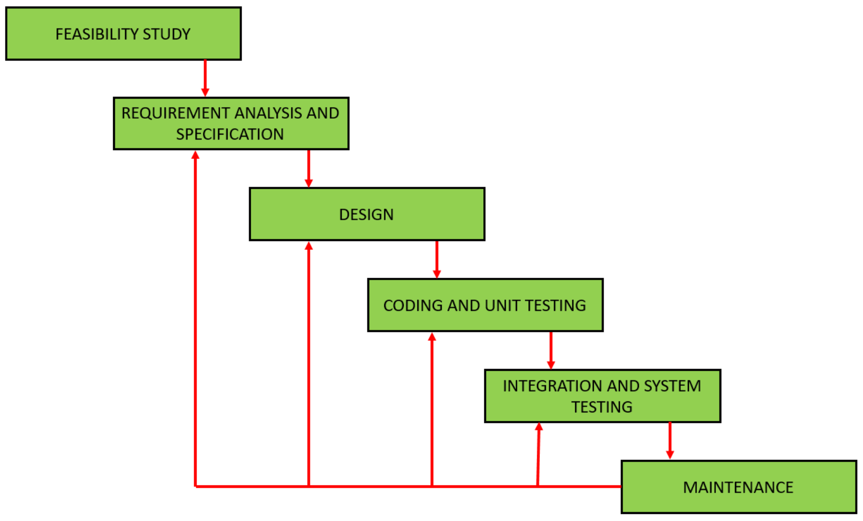
In practical software development project, the classical waterfall model is hard to use, as classical waterfall model as an idealistic model. In this context, the iterative waterfall model can be thought of as incorporating the necessary changes to the classical waterfall model to make it usable in practical software development projects.

The iterative waterfall model provides feedback paths from every phase to its preceding phases, which is the main difference from the classical waterfall model.

Phase containment of errors:

The principle of detecting errors as close to their points of commitment as possible is known as phase containment of errors.

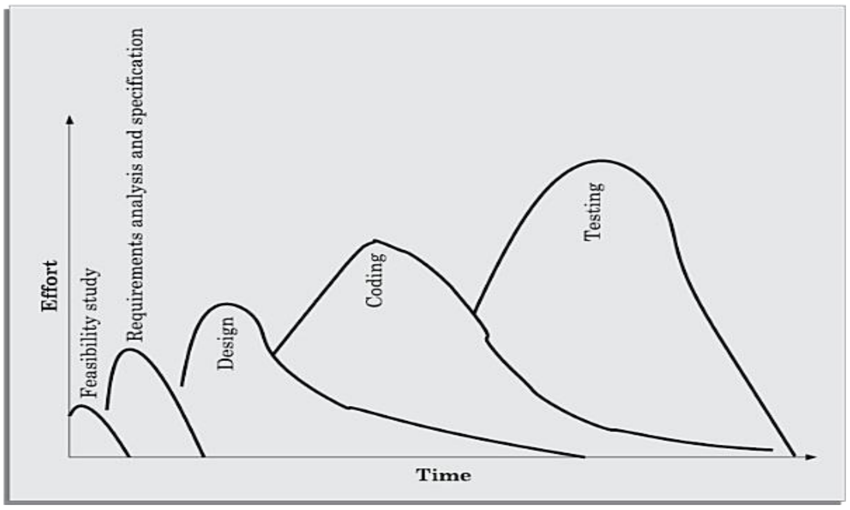
No matter how careful a programmer may be, he might end up committing some mistake(also called *errors* or *bugs*) in the work product. It is important to detect these errors in the same phase in which they take place, that reduces the effort and time required for correcting those.
It may not always be possible to detect all the errors in the same phase in which they are made. Nevertheless, the errors should be detected as early as possible.


FEASIBILITY STUDY
REQUIREMENT ANALYSIS AND SPECIFICATION
DESIGN
CODING AND UNIT TESTING
INTEGRATION AND SYSTEM TESTING
MAINTENANCE

Phase overlap:

Even though the strict waterfall model envisages sharp transitions to occur from one phase to the next, in practice the activities of different phases overlap due to two main reasons:

- In spite of the best effort to detect errors in the same phase in which they are committed, some errors escape detection and are detected in a later phase. If we rework after a phase is complete, then two phases will be overlapped.
- As we know each phase is assigned to a team, the work required to be carried out in a phase is divided among the team members. Some members may complete their part of the work earlier than other members. If strict phase transitions are maintained, then the team members who complete their work early would idle waiting for the phase to be complete, and are said to be in a *blocking state*. This causes wastage of resources and a source of cost escalation and inefficiency. As a result, in real projects, the phases are allowed to overlap.



Limitations of waterfall models:

Waterfall-based models have worked satisfactorily over last many years in the past. Now, not only software has become very large and complex, very few software project is being developed from scratch. In the present software development projects, use of waterfall model causes several problems.

Difficult to accommodate change requests: A major problem with the waterfall model is that the requirements need to be frozen before the development starts. Once requirements have been frozen, the waterfall model provides no scope for any modifications to the requirements.

Incremental delivery not supported: In the iterative waterfall model, the full software is completely developed and tested before it is delivered to the customer. There is no provision for any intermediate deliveries to occur. This may take several months or years to be completed and delivered to the customer. By the time the software is delivered, installed, and becomes ready for use, the customer's business process might have changed substantially. This makes the developed application a poor fit to the customer's requirements.

Phase overlap not supported: For most real-life projects, it becomes difficult to follow the rigid phase sequence prescribed by the waterfall model. By the term *a rigid phase sequence*, we mean that a phase can start only after the previous phase is complete in all respects. As already discussed, strict adherence to the waterfall model creates *blocking states*.

Error correction unduly expensive: In waterfall model, validation is delayed till the complete development of the software. As a result, the defects that are noticed at the time of validation incur expensive rework and result in cost escalation and delayed delivery.

Limited customer interactions: This model supports very limited customer interactions. It is generally accepted that software developed in isolation from the customer is the cause of many problems. In fact, interactions occur only at the start of the project and at project completion. As a result, the developed software usually turns out to be a misfit to the customer's actual requirements.

Heavy weight: The waterfall model overemphasises documentation. A significant portion of the time of the developers is spent in preparing documents, and revising them as changes occur over the life cycle. Heavy documentation though useful during maintenance and for carrying out review, is a source of team inefficiency.

No support for risk handling and code reuse: It becomes difficult to use the waterfall model in projects that are susceptible to various types of risks, or those involving significant reuse of existing development artifacts. Please recollect that software services types of projects usually involve significant reuse.

Prototype model:

The prototyping model can be considered to be an extension of the waterfall model. This model suggests building a working *prototype* of the system, before development of the actual software.

Necessity of the prototyping model:

The prototyping model is advantageous to use for specific types of projects. In the following, we identify three types of projects for which the prototyping model can be followed to advantage:

- The GUI part of a software system is almost always developed using the prototyping model.
- The prototyping model is especially useful when the exact technical solutions are unclear to the development team and helps to critically examine the technical issues of product development.
- An important reason for developing a prototype is that it is impossible to "get it right" the first time.

Life cycle activity of protype model:

The prototyping model of software development is developed through two major activities

- prototype construction
- Iterative waterfall-based software development.

Prototype development:

Prototype development starts with an initial requirements gathering phase. A quick design is carried out and a prototype is built. The developed prototype is submitted to the customer for evaluation. Based on the customer feedback, the requirements are refined and the prototype is suitably modified. This cycle of obtaining customer feedback and modifying the prototype continues till the customer approves the prototype.

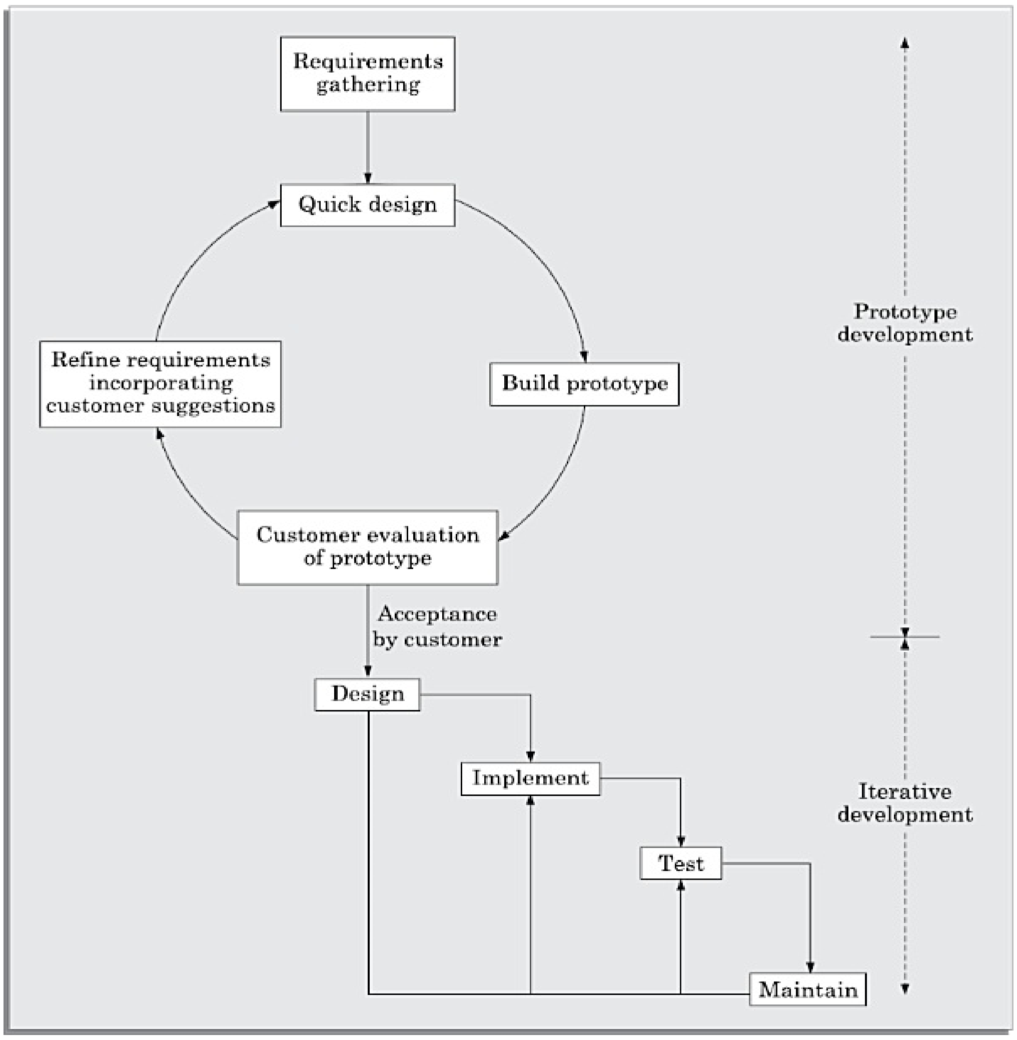
Iterative development:

Once the customer approves the prototype, the actual software is developed using the iterative waterfall approach. In spite of the availability of a working prototype, the SRS document is usually needed to be developed. The code for the prototype is usually thrown away. However, the experience gathered from developing the prototype helps a great deal in developing the actual system.

Advantages of the prototyping model:

Using a prototype model can bring multiple advantages, including:

- Customers get a say in the product early on, increasing customer satisfaction.
- Missing functionality and errors are detected easily.
- Prototypes can be reused in future, more complicated projects.
- It emphasizes team communication and flexible design practices.
- Users have a better understanding of how the product works.
- Quicker customer feedback provides a better idea of customer needs.



Disadvantages of the prototyping model:

- The main disadvantage of this methodology is that it is more costly in terms of time and money when compared to alternative development methods, such as the spiral or Waterfall model.
- The prototype is constructed only at the start of the project, the prototyping model is ineffective for risks identified later during the development cycle.

There are four types of prototype model available:

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.

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.

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. Of course, there might be the possibility that the pieces just not fit together due to some lack ness in the development phase – this can only be fixed by careful and complete plotting of the entire system before prototyping starts.

Extreme Prototyping:

This method is mainly used for web development. It is consisting of three sequential independent phases:

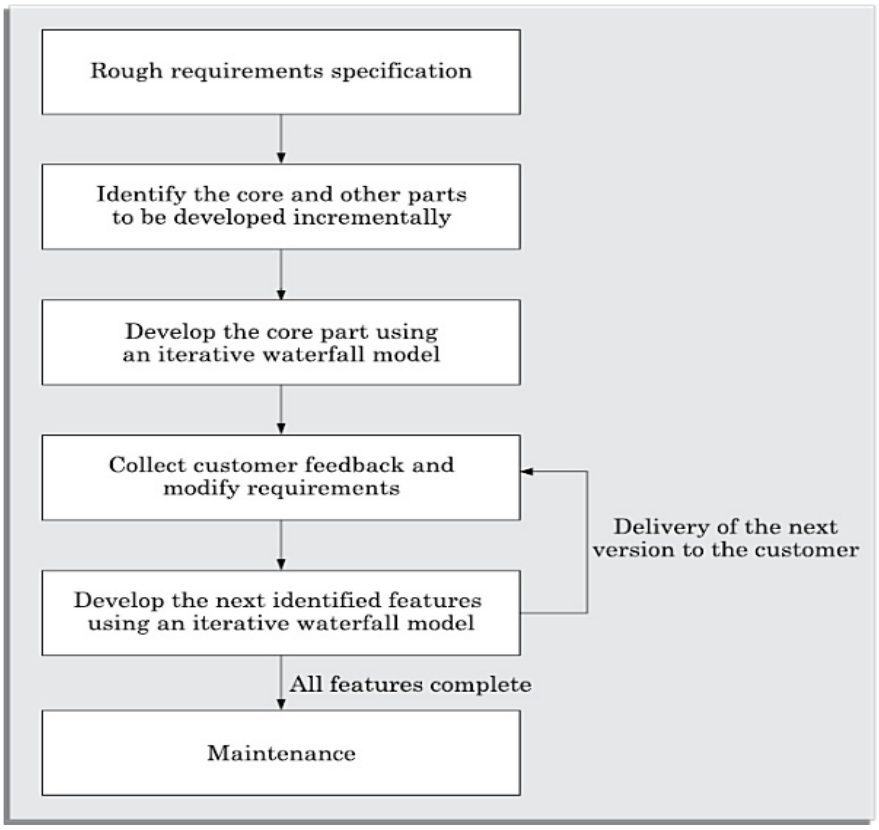
- In this phase a basic prototype with all the existing static pages are presented in the HTML format.
- In the 2nd phase, Functional screens are made with a simulate data process using a prototype services layer.
- This is the final step where all the services are implemented and associated with the final prototype.

Evolutionary model:

Evolutionary model is a combination of Iterative and Incremental model of software development life cycle. The prototype developed initially is incrementally refined on the basis of customer feedback till it finally gets accepted.

Feedback is provided by the users on the product for the planning stage of the next cycle and the development team responds, often by changing the product, plan or process. Therefore, the software product evolves with time.

Evolutionary model suggests breaking down the development cycle into smaller, incremental waterfall models as users are able to get access to the product at the end of each cycle. The number of chunks is huge and is the number of deliveries made to the customer. The main advantage is that the customer's confidence increases as he constantly gets quantifiable goods or services from the beginning of the project to verify and validate his requirements. The model allows for changing requirements as well as all work in broken down into maintainable work chunks.
A schematic representation of the evolutionary model of development has been shown below.


Rough requirements specification
Identify the core and other parts to be developed incrementally
Develop the core part using an iterative waterfall model
Collect customer feedback and modify requirements
Delivery of the next version to the customer
Develop the next identified features using an iterative waterfall model
All features complete
Maintenance

Application of Evolutionary Model:

1. It is used in large projects where you can easily find modules for incremental implementation. Evolutionary model is commonly used when the customer wants to start using the core features instead of waiting for the full software.

2. 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.

Advantages:

- **Effective elicitation of actual customer requirements:** In evolutionary model, a user gets a chance to experiment partially developed system.
- **Easy handling:** It reduces the error because the core modules get tested thoroughly.

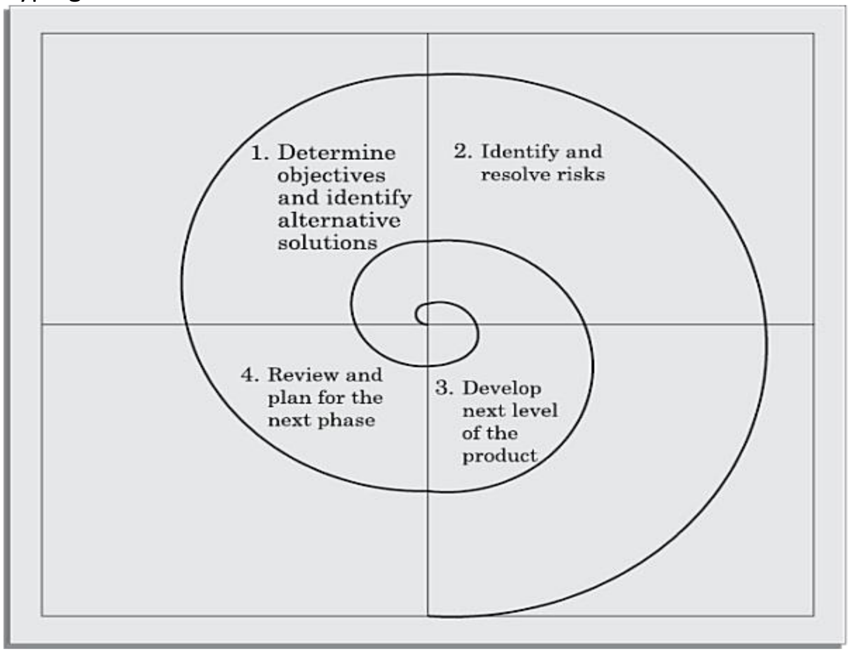
Disadvantages:

- **Feature division into incremental parts can be non-trivial:** Sometimes it is hard to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented and delivered.

Spiral model:

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

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


1. Determine objectives and identify alternative solutions
2. Identify and resolve risks
4. Review and plan for the next phase
3. Develop next level of the product

Risk handling in spiral model:

A risk is a circumstance that might hamper the successful completion of a software project. As an example, consider a project for which a risk can be that data access from a remote database might be too slow to be acceptable by the customer. This risk can be resolved by building a prototype of the data access subsystem and experimenting with the exact access rate. If the data access rate is too slow, possibly a caching scheme can be implemented or a faster communication scheme can be deployed to overcome the slow data access rate. Such risk resolutions are easier done by using a prototype. The spiral model supports coping up with risks by providing the scope to build a prototype at every phase of software development.

Phases of the Spiral Model:

Each phase in this model is split into four sectors (or quadrants) as shown in Figure.

Quadrant 1: The objectives are investigated, elaborated, and analyzed. Based on this, the risks involved in meeting the phase objectives are identified. In this quadrant, alternative solutions possible for the phase under consideration are proposed.
Quadrant 2: During the second quadrant, the alternative solutions are evaluated to select the best possible solution. To be able to do this, the solutions are evaluated by developing an appropriate prototype.
Quadrant 3: Activities during the third quadrant consist of developing and verifying the next level of the software. At the end of the third quadrant, the identified features have been implemented and the next version of the software is available.
Quadrant 4: Activities during the fourth quadrant concern reviewing the results of the stages traversed so far (i.e., the developed version of the software) with the customer and planning the next iteration of the spiral.
The radius of the spiral at any point represents the cost incurred in the project so far, and the angular dimension represents the progress made so far in the current phase.

In the spiral model of development, the project manager plays the crucial role, as he dynamically determines the number of phases as the project progresses. To make the model more efficient, the different features of the software that can be developed simultaneously through parallel cycles are identified.

Why Spiral Model is called Meta Model?
The Spiral model is called a Meta-Model because it subsumes all the other SDLC models. For example, a single loop spiral actually represents the Iterative Waterfall Model. The spiral model incorporates the stepwise approach of the Classical Waterfall Model. The spiral model uses the approach of the **Prototyping Model** by building a prototype at the start of each phase as a risk-handling technique. Also, the spiral model can be considered as supporting the evolutionary model – the iterations along the spiral can be considered as evolutionary levels through which the complete system is built.

Advantages of Spiral Model:
Below are some advantages of the Spiral Model.

1. **Risk Handling:** The projects with many unknown risks that occur as the development proceeds, in that case, Spiral Model is the best development model to follow due to the risk analysis and risk handling at every phase.
2. **Good for large projects:** It is recommended to use the Spiral Model in large and complex projects.
3. **Flexibility in Requirements:** Change requests in the Requirements at later phase can be incorporated accurately by using this model.
4. **Customer Satisfaction:** Customer can see the development of the product at the early phase of the software development and thus, they habituated with the system by using it before completion of the total product.

Disadvantages of Spiral Model:
Below are some main disadvantages of the spiral model.

1. **Complex:** The Spiral Model is much more complex than other SDLC models.
2. **Expensive:** Spiral Model is not suitable for small projects as it is expensive.
3. **Too much dependability on Risk Analysis:** The successful completion of the project is very much dependent on Risk Analysis. Without very highly experienced experts, it is going to be a failure to develop a project using this model.
4. **Difficulty in time management:** As the number of phases is unknown at the start of the project, so time estimation is very difficult.

RAD 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.

RAD model suggests that broken down of project into small modules whereas 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 analysing, designing, coding and then testing, etc. 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.

Main motivation:

In iterative waterfall model, the customers do not get to see the software, until the development is complete in all respects and the software has been delivered and installed. Naturally, the delivered software often does not meet the customer expectations. The changes are incorporated through subsequent maintenance efforts. It takes a long time to have a good product that meet the requirements of the customers. The RAD model tries to overcome this problem by inviting and incorporating customer feedback on successively developed and refined prototypes.

Working of RAD:

In the RAD model, development takes place in a series of short cycles or iterations. Development team focuses on present iteration and plans are made for that iteration only. The time planned for each iteration is called a time box. During each time box, a quick-and-dirty prototype for some functionality is developed. The prototype is refined based on the customer feedback. The development team usually consists of about five to six members, including a customer representative. Customer representative covers the communication gap between the customer and the development team.

The various phases of RAD are as follows:

1.Business Modelling: The information flow among business functions is defined by answering questions like what data drives the business process, what data is generated, who generates it, where does the information go, who process it and so on.

2. Data Modelling: The data collected from business modelling is refined into a set of data objects (entities) that are needed to support the business. The attributes (character of each entity) are identified, and the relation between these data objects (entities) is defined.

3. Process Modelling: The information object defined in the data modelling phase are transformed to achieve the data flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.

4. Application Generation: Automated tools are used to facilitate construction of the software; even they use the 4th GL techniques.

5. Testing & Turnover: Many of the programming components have already been tested since RAD emphasis reuse. This reduces the overall testing time. But the new part must be tested, and all interfaces must be fully exercised.

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.

Comparison of RAD with Other Models:

we compare the relative advantages and disadvantages of RAD with other life cycle models.

RAD versus prototyping model:

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

Though RAD is expected to lead to faster software development compared to the traditional models (such as the prototyping model), though the quality and reliability would be inferior.

RAD versus iterative waterfall model:

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

RAD versus evolutionary model:

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

How does RAD facilitate accommodation of change requests?

The customers usually suggest changes to a specific feature only after they have used it. Since the features are delivered in small increments, the customers are able to give their change requests to a feature already delivered. Incorporation of such change requests just after the delivery of an incremental feature saves cost as this is carried out before large investments have been made in development and testing of a large number of features.

How does RAD facilitate faster development?

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

Agile models:

Agile Methodology is a practice that promotes continuous iteration of development and testing throughout the software development lifecycle of the project. In the Agile model in software testing, both development and testing activities are concurrent. The agile software development model was proposed in the mid-1990s to overcome the limitations of the waterfall model.

In the agile model, the requirements are decomposed into many small parts that can be incrementally developed over an iteration. Each iteration is intended to be small and easily manageable and lasting for a couple of weeks only. At a time, only one increment is planned, developed, and then deployed at the customer site. No long-term plans are made. The time to complete an iteration is called a time box. The implication of the term time box is that the end date for an iteration does not change. That is, the delivery date is considered sacrosanct. The development team can, however, decide to reduce the delivered functionality during a time box if necessary.

Essential Idea behind Agile Models:

- For establishing close contact with the customer during development and to gain a clear understanding.

- Team size be deliberately kept small (5–9 people) to help the team members meaningfully engage in face-to-face communication and have collaborative work environment.

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

- Working software over comprehensive documentation.
- Frequent delivery of incremental versions of the software to the customer in intervals of few weeks.
- Requirement change requests from the customer are encouraged and are efficiently incorporated.
- Having competent team members and enhancing interactions among them, through face-to-face communication rather than through exchange of formal documents.
- Continuous interaction with the customer is considered much more important rather than effective contract negotiation. A customer representative is required to be a part of the development team, thus facilitating close, daily co-operation between customers and developers.
- Agile development projects usually deploy pair programming.

In pair programming, two programmers work together at one work station. One types in code while the other reviews the code as it is typed in. The two programmers switch their roles every hour or so. Several studies indicate that programmers working in pairs produce compact well-written programs and commit fewer errors as compared to programmers working alone.

Agile versus Other Models:

we compare the characteristics of the agile model with other models of development.

- Agile methodologies propose incremental and iterative approach whereas in waterfall model development flows sequentially from start to end.
- Agile process is broken down into smaller individual models that designers work on whereas waterfall model process is not broken into individual models.
- With agile model, even in midway of a project leaves the customer with some worthwhile code, that might possibly have already been put into live operation. Whereas in midway of project of waterfall models there is nothing to show beyond several documents.
- Agile development model's frequent re- evaluation of plans, emphasis on face-to-face communication, and relatively sparse use of documentation are similar to that of the exploratory style. Agile teams, however, do follow defined and disciplined processes and carry out systematic requirements capture, rigorous designs, compared to chaotic coding in exploratory programming.
- The central theme of RAD is based on designing quick and dirty prototypes, which are then refined into production quality code. Whereas Agile projects logically break down the solution into features that are incrementally developed and delivered. Developers using the RAD model focus on developing all the features of an application by first doing it badly and then successively improving the code over time.

The most popular Agile methods include Rational Unified Process (1994), Scrum (1995), Crystal Clear, Extreme Programming (1996), Adaptive Software Development, Feature Driven Development, and Dynamic Systems Development Method (DSDM) (1995). These are now collectively referred to as **Agile Methodologies.** Agile model is being used as an umbrella term to refer to a group of development processes.

Scrum:

SCRUM is an agile development process focused primarily on ways to manage tasks in team-based development conditions.

There are three roles in it, and their responsibilities are:

Scrum Master: The scrum can set up the master team, arrange the meeting and remove obstacles for the process

Product owner: The product owner makes the product backlog, prioritizes the delay and is responsible for the distribution of functionality on each repetition.

Scrum Team: The team manages its work and organizes the work to complete the sprint or cycle.

Extreme Programming(XP):

The extreme programming model recommends taking the best practices that have worked well in the past in program development projects to extreme levels.

Good practices needs to practiced extreme programming: Some of the good practices that have been recognized in the extreme programming model and suggested to maximize their use are given below:

- **Code Review:** Code review detects and corrects errors efficiently. It suggests pair programming as coding and reviewing of written code carried out by a pair of programmers who switch their works between them every hour.
- **Testing:** Testing code helps to remove errors and improves its reliability. XP suggests test-driven development (TDD) to continually write and execute test cases. In the TDD approach test cases are written even before any code is written.
- **Incremental development:** Incremental development is very good because customer feedback is gained and based on this development team come up with new increments every few days after each iteration.
- **Simplicity:** Simplicity makes it easier to develop good quality code as well as to test and debug it.
- **Design:** Good quality design is important to develop a good quality software. So, everybody should design daily.
- **Integration testing:** It helps to identify bugs at the interfaces of different functionalities. Extreme programming suggests that the developers should achieve continuous integration by building and performing integration testing several times a day.

Crystal:

There are three concepts of this method-

- **Chartering:** Multi activities are involved in this phase such as making a development team, performing feasibility analysis, developing plans, etc.
- **Cyclic delivery:** under this, two more cycles consist, these are:
    - Team updates the release plan.
    - Integrated product delivers to the users.
- **Wrap up:** According to the user environment, this phase performs deployment, post-deployment.

Dynamic Software Development Method(DSDM):

DSDM is a rapid application development strategy for software development and gives an agile project distribution structure. The essential features of DSDM are that users must be actively connected, and teams have been given the right to make decisions. The techniques used in DSDM are:

1. Time Boxing
2. MoSCoW Rules
3. Prototyping

The DSDM project contains seven stages:

1. Pre-project
2. Feasibility Study
3. Business Study
4. Functional Model Iteration
5. Design and build Iteration
6. Implementation
7. Post-project

Feature Driven Development(FDD):

This method focuses on "Designing and Building" features. In contrast to other smart methods, FDD describes the small steps of the work that should be obtained separately per function.

Lean Software Development:

Lean software development methodology follows the principle "just in time production." The lean method indicates the increasing speed of software development and reducing costs. Lean development can be summarized in seven phases.

1. Eliminating Waste
2. Amplifying learning
3. Defer commitment (deciding as late as possible)
4. Early delivery

5. Empowering the team
6. Building Integrity
7. Optimize the whole

When to use the Agile Model?

- When frequent changes are required.
- When a highly qualified and experienced team is available.
- When a customer is ready to have a meeting with a software team all the time.
- When project size is small.

Advantage(Pros) of Agile Method:

- Frequent Delivery
- Face-to-Face Communication with clients.
- Efficient design and fulfils the business requirement.
- Anytime changes are acceptable.
- It reduces total development time.

Disadvantages(Cons) of Agile Model:

- Due to the shortage of formal documents, it creates confusion and crucial decisions taken throughout various phases can be misinterpreted at any time by different team members.
- Due to the lack of proper documentation, once the project completes and the developers allotted to another project, maintenance of the finished project can become difficulty.

software project management:

The main goal of software project management is to enable a group of developers to work effectively towards the successful completion of a project. project management involves use of a set of techniques and skills to steer a project to success. A project manager is usually an experienced member of the team who essentially works as the administrative leader of the team and responsible for managing a project. For small software development projects, a single member of the team can take responsibilities for both project management and technical management. For large projects, a project manager is required to take responsibility of technical leadership.

SOFTWARE PROJECT MANAGEMENT COMPLEXITIES:

Management of software projects is much complex. The main factors contributing to the complexity of managing a software project, as identified by [Brooks75], are the following:

Invisibility: Software remains invisible, until its development is completed. The best that he can do perhaps is to monitor the milestones that have been completed by the development team and the documents that have been produced—which are rough indicators of the progress achieved.

Changeability: Because the software part of any system is easier to change as compared to the hardware part, the software part is the one that gets most frequently changed. These changes usually arise from changes to the business practices, changes to the hardware or underlying software (e.g., operating system, other applications), or just because the client changes his mind.

Frequent changes to the requirements and the invisibility of software are possibly the two major factors making software project management a complex task.

Complexity: Even a moderate sized software has millions of parts (functions) that interact with each other in many ways—data coupling, serial and concurrent runs, state transitions, control dependency, file sharing, etc. This makes managing these projects much more difficult as compared to many other kinds of projects.

Uniqueness: Every software project is usually associated with many unique features or situations. Due to the uniqueness of the software projects, a project manager in the course of a project faces many issues that are quite unlike the others he had encountered in the past. As a result, a software project manager has to confront many unanticipated issues in almost every project that he manages.

Exactness of the solution: The parameters of a function call in a program are required to be in complete conformity with the function definition. This requirement not only makes it difficult to get a software product up and working, but also makes reusing parts of one software product in another difficult. This requirement of exact conformity of

the parameters of a function introduces additional risks and contributes to the complexity of managing software projects.
Team-oriented and intellect-intensive work: Software development projects involve team-oriented, intellect-intensive work. In a software development project, the life cycle activities not only highly intellect intensive, but each member has to typically interact, review, and interface with several other members, constituting another dimension of complexity of software projects.

Responsibilities of a software manager:

A software project manager takes the overall responsibility of steering a project to success. We can broadly classify a project manager's varied responsibilities into the following two major categories:

- Project planning
- Project monitoring and control.

Project planning: Project planning is undertaken immediately after the feasibility study phase and before the starting of the requirements analysis and specification phase.
Project planning involves estimating several characteristics of a project and then planning the project activities based on these estimates made.
The initial project plans are revised from time to time as the project progresses and more project data become available.
Project monitoring and control: Project monitoring and control activities are undertaken once the development activities start.
The focus of project monitoring and control activities is to ensure that the software development proceeds as per plan.
While monitoring and controlling activities, a project manager may sometimes find it necessary to change the plan to cope up with specific situations at hand.
Three skills that are most critical to successful project management are the following:

- Knowledge of project management techniques.
- Decision taking capabilities.
- Previous experience in managing similar projects.

Project planning:

Project planning involves estimating several characteristics of a project and then planning the project activities based on these estimates made.
Complexities In project planning:

- unrealistic time and resource estimation
- Schedule delays can cause customer dissatisfaction and affect team morale.

Crucial project planning activities:

- Knowledge of project management techniques.
- Decision taking capabilities.
- Previous experience in managing similar projects.

During project planning, the project manager performs the following activities:

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

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

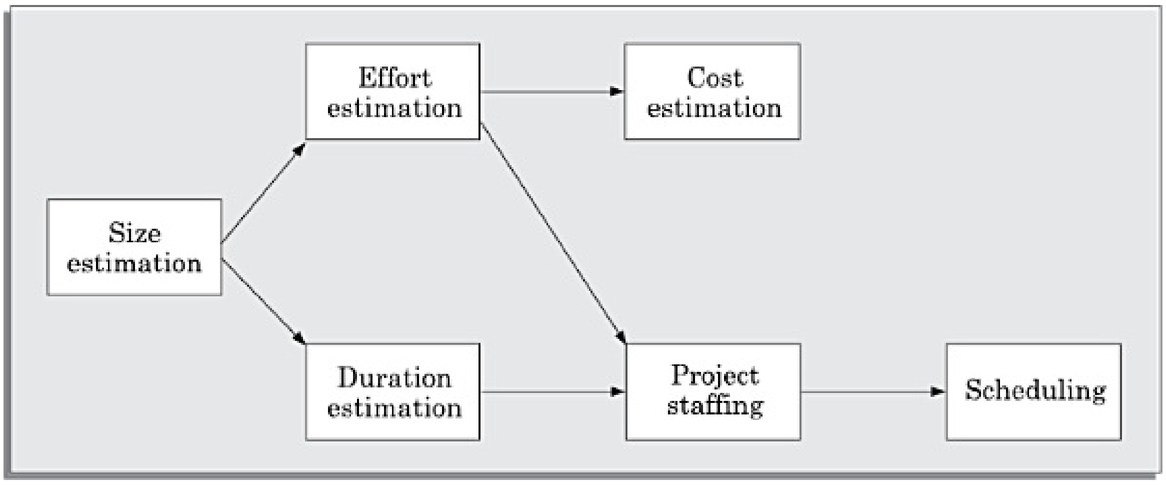


Fig: Precedence ordering among planning activities

Sliding Window Planning:

In the sliding window planning technique, starting with an initial plan, the project is planned more accurately over a number of stages. At the start of a project, the project manager has incomplete knowledge of the project. The project parameters are re-estimated periodically as understanding grows and also a periodically as project parameters change. By taking these developments into account, the project manager can plan the subsequent activities more accurately and with increasing levels of confidence. Planning a project over a number of stages protects managers from making big commitments at the start of the project. This technique of staggered planning is known as sliding window planning.

SPMP Document of Project Planning:

Once project planning is complete, project managers document their plans in a software project management plan (SPMP) document. Listed below are the different items that the SPMP document should discuss. Organisation of the software project management plan (SPMP) document

1.Introduction

(a) Objectives
(b) Major Functions
(c) Performance Issues
(d) Management and Technical Constraints

2.Project estimates

(a) Historical Data Used
(b) Estimation Techniques Used
(c) Effort, Resource, Cost, and Project Duration Estimates

3.Schedule

(a) Work Breakdown Structure
(b) Task Network Representation
(c) Gantt Chart Representation
(d) PERT Chart Representation

4.Project resources

(a) People
(b) Hardware and Software
(c) Special Resources

5.Staff organisation

(a) Team Structure
(b) Management Reporting

6.Risk management plan

(a) Risk Analysis
(b) Risk Identification
(c) Risk Estimation

(d) Risk Abatement Procedures

7.Project tracking and control plan

(a) Metrics to be tracked

(b) Tracking plan

(c) Control plan

8.Miscellaneous plans

(a) Process Tailoring

(b) Quality Assurance Plan

(c) Configuration Management Plan

(d) Validation and Verification

(e) System Testing Plan

(f) Delivery, Installation, and Maintenance Plan

METRICS FOR PROJECT SIZE ESTIMATION:

The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

Currently, two metrics are popularly being used to measure size

1.lines of code (LOC)

2.function point (FP)

1 .Lines of Code (LOC)

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

Accurate estimation of LOC count at the beginning of a project is a very difficult task. systematic guess work is used to count LOC at the starting of a project. Systematic guessing typically involves the following. The project manager divides the problem into modules, and each module into sub-modules and so on, until the LOC of the leaf-level modules are small enough to be predicted. LOC metric has several shortcomings when used to measure problem size.

shortcomings of the LOC metric:

LOC is a measure of coding activity alone. A good problem size measure should consider the total effort needed to carry out various life cycle activities (i.e., specification, design, code, test, etc.) and not just the coding effort. LOC, however, focuses on the coding activity alone—it merely computes the number of source lines in the final program. Code size, therefore, is obviously an improper indicator of the problem size.

LOC count depends on the choice of specific instructions: LOC gives a numerical value of problem size that can vary widely with coding styles of individual programmers. By coding style, we mean the choice of code layout, the choice of the instructions in writing the program, and the specific algorithms used.

Even for the same programming problem, different programmers might come up with programs having very different LOC counts. This situation does not improve, even if language tokens are counted instead of lines of code.

LOC measure correlates poorly with the quality and efficiency of the code: Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set or use improper algorithms. In fact, it is true that a piece of poor and sloppily written piece of code can have larger number of source instructions than a piece that is efficient and has been thoughtfully written.

LOC metric penalises use of higher-level programming languages and code reuse: A paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size, and in turn, would indicate lower effort! Thus,Modern programming methods such as object-oriented programming and reuse of components makes the relationships between LOC and other project attributes even less precise.

Difficult to accurately estimate LOC of the final program during problem specification: The LOC metric is of little use to the project managers during project planning because LOC count can accurately be computed only after the code has been fully developed.

2. Function Point (FP) Metric

Function point metric was proposed by Albrecht in 1983. This metric overcomes many of the shortcomings of the LOC metric. One of the advantages of the function point metric over the LOC metric is that it can easily be computed from the problem specification stage.

The size of a software product is directly dependent on the number of different high-level functions or features it supports. This assumption is reasonable, since each feature would take additional effort to implement.

Each different features may take very different amounts of efforts to develop. The more the number of data items that a function reads from the user and outputs and the more the number of files accessed, the higher is the complexity of the function. Each feature when invoked typically reads some input data and then transforms those to the required output data.

Example: The query book feature of a Library Automation Software takes the name of the book as input and displays its location in the library and the total number of copies available. Similarly, the issue book and the return book features produce their output based on the corresponding input data. It can therefore be argued that the computation of the number of input and output data items would give a more accurate indication of the code size compared to simply counting the number of high-level functions supported by the system.

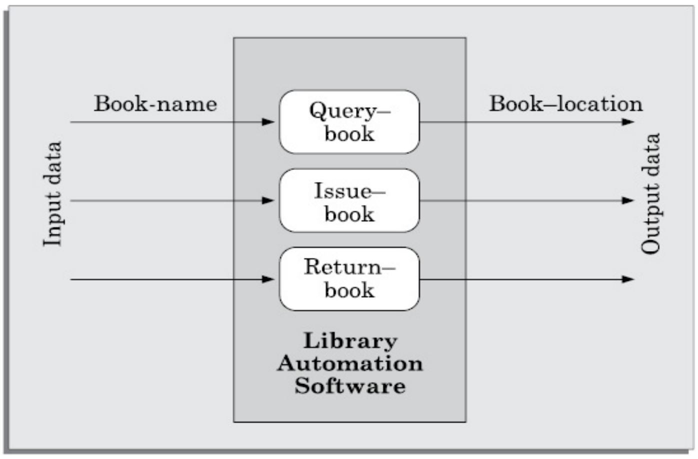


Fig: System function as a mapping of input data to output data.

Albrecht postulated that in addition to the number of basic functions that a software performs, size also depends on the number of files and the number of interfaces that are associated with the software. Here, interfaces refer to the different mechanisms for data transfer with external systems including the interfaces with the user, interfaces with external computers, etc.

Function point (FP) metric computation

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

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

We discuss these three steps in more detail in the following.

Step 1: UFP computation

The unadjusted function points (UFP) is computed as the weighted sum of five characteristics of a product as shown in the following expression.

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

1. **Number of inputs:** Each individual data input by the user are not simply added up to compute the number of inputs, but related inputs are grouped and considered as a single input. For example, the data of employee -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 describe a single employee.

2. **Number of outputs:** The outputs considered include reports printed, screen outputs, error messages produced, etc. While computing the number of outputs, the individual data items within a report are not considered; but a set of related data items is counted as just a single output.
3. **Number of inquiries:** An inquiry is a user command (without any data input) and only requires some actions to be performed by the system. Thus, the total number of inquiries is essentially the number of distinct interactive queries (without data input) which can be made by the users. Examples of such inquiries are print account balance, print all student grades, display rank holders' names, etc.
4. **Number of files:** The files referred to here are logical files. A logical file represents a group of logically related data. Logical files include data structures as well as physical files.
5. **Number of interfaces:** Here the interfaces denote the different mechanisms that are 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 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.

Table 3.1: 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. The list of these parameters have been shown in Table 3.2. 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.

Table 3.2: Function Point Relative Complexity Adjustment Factors

Requirement for reliable backup and recovery
Requirement for data communication
Extent of distributed processing
Performance requirements
Expected operational environment
Extent of online data entries
Extent of multi-screen or multi-operation online data input
Extent of online updating of master files
Extent of complex inputs, outputs, online queries and files
Extent of complex data processing
Extent that currently developed code can be designed for reuse
Extent of conversion and installation included in the design
Extent of multiple installations in an organisation and variety of customer organisations Extent of change and focus on ease of use

Example 3.1 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. Each 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 3.1, 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 = $14 \times 4 = 56$ TCF = $0.65 + 0.01 + 56 = 1.21$

Therefore, the adjusted FP=46*1.21=55.66

Feature point metric shortcomings:

- It does not take into account the algorithmic complexity of a function.
- The function point metric implicitly assumes that the effort required to design and develop any two different functionalities of the system is the same. But this is highly unlikely to be true. The effort required to develop any two functionalities may vary widely.
- FP only considers the number of functions that the system supports, without distinguishing the difficulty levels of developing the various functionalities. To overcome this problem, an extension to the function point metric called feature point metric has been proposed.

Feature point metric incorporates algorithm complexity as an extra parameter. This parameter ensures that the computed size using the feature point metric reflects the fact that higher the complexity of a function, the greater the effort required to develop it—therefore, it should have larger size compared to a simpler function.

Critical comments on the function point and feature point metrics

Proponents of function point and feature point metrics claim that these two metrics are language-independent and can be easily computed from the SRS document during project planning stage itself. On the other hand, opponents claim that these metrics are subjective and require a sleight of hand. An example of the subjective nature of the function point metric can be that the way one groups input and output data items into logically related groups can be very subjective. For example, consider that certain functionality requires the employee name and employee address to be input. It is possible that one can consider both these items as a single unit of data, since after all, these describe a single employee. It is also possible for someone else to consider an employee's address as a single unit of input data and name as another. Such ambiguities leave sufficient scope for debate and keep open the possibility for different project managers to arrive at different function point measures for essentially the same problem.

project estimation:

The different parameters of a project that need to be estimated include—project size, effort required to complete the project, project duration, and cost. Accurate estimation of these parameters is important, since these not only help in quoting an appropriate project cost to the customer, but also form the basis for resource planning and scheduling. A large number of estimation techniques have been proposed by researchers. These can broadly be classified into three main categories:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

Empirical estimation techniques:

Empirical estimation technique are based on the data taken from the previous project and some based on guesses and assumptions. There are many empirical estimation technique but most popular are

- **Expert Judgement Technique**
- **Delphi Cost Technique**

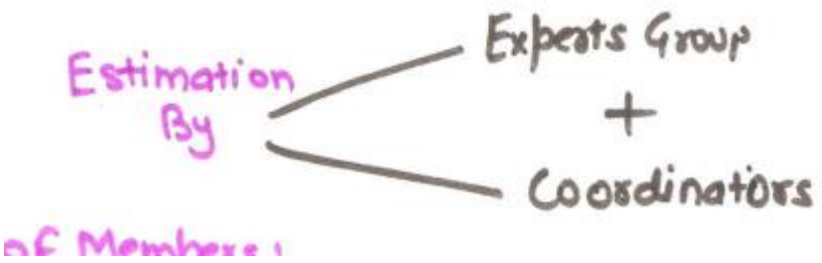
Expert judgement technique:

An expert makes an educated guess of the problem size after analysing the problem thoroughly. Expert estimate the cost of different components that is modules and sub modules of the system.

Disadvantages:

- Human error, considering not all factors and aspects of the project, individual bias, more chances of failure.
- Estimation by group of experts minimises factors such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias and desired to win a contract through overly optimistic estimates.

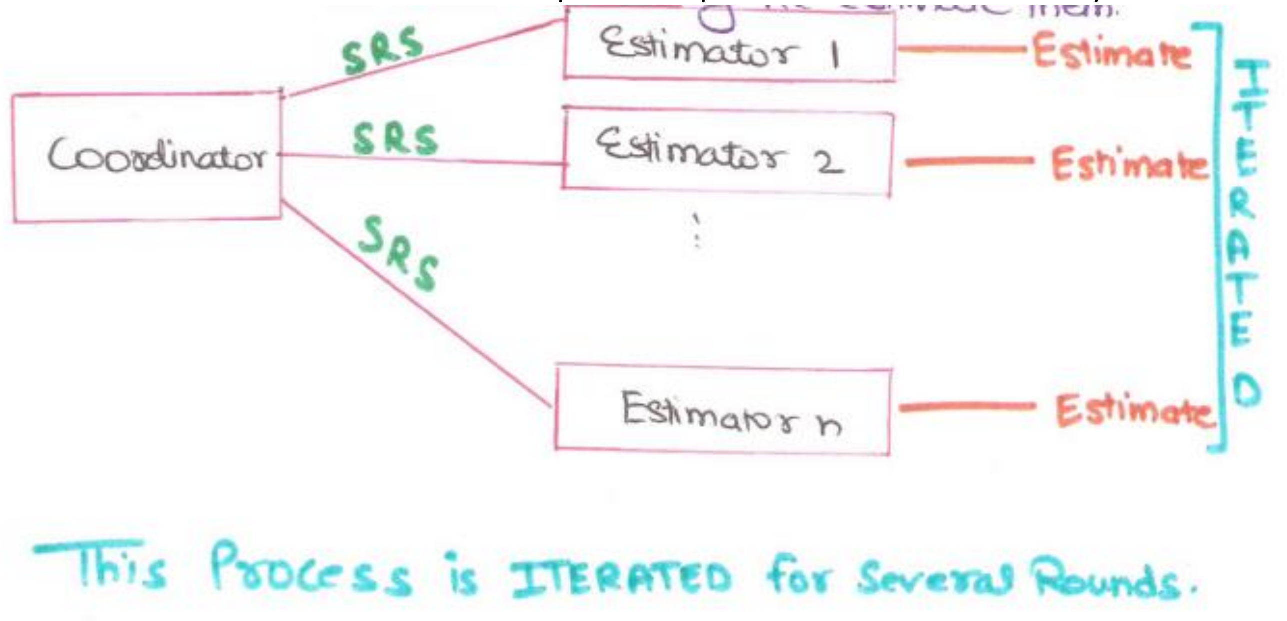
Delphi cost estimation:



Role of Members: Coordinator provide a copy of Software Requirement Specification(SRS) document and a form of recording it cost estimate to each estimator.

Estimator: It complete their individual estimate anomalously and submit to the coordinator with mentioning, if any, unusual characteristics of product which has influenced his estimation.

The coordinator and distribute the summary of the response to all estimator and they re-estimate them.



This process is Iterated for several rounds. No discussion is allowed among the estimator during the entire estimation process because there may be many estimators get easily influenced by rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimation, the coordinator takes the responsibility of compiling the result and preparing the final estimates.

Heuristic Techniques:

Heuristic techniques assume that the relationships that exist among the different project parameters can be satisfactorily modelled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the values of the independent parameters in the corresponding mathematical expression. Different heuristic estimation models can be divided into the following two broad categories—single variable and multivariable models.

Single variable estimation models assume that various project characteristic can be predicted based on a single previously estimated basic (independent) characteristic of the software such as its size. A single variable estimation model assumes that the relationship between a parameter to be estimated and the corresponding independent parameter can be characterised by an expression of the following form:

Estimated Parameter = $c_1 \; e^{d1}$

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

A multivariable cost estimation model assumes that a parameter can be predicted based on the values of more than one independent parameter. It takes the following form:

Estimated Resource = $c_1 \; p1^{d1} + c_2 \; p2^{d2} + \ldots$

where, p1, p2, ... are the basic (independent) characteristics of the software already estimated, and c1, c2, d1, d2, ... are constants.
Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters. The independent parameters influence the dependent parameter to different extents. This is modelled by the different sets of constants c1 , d1 , c2 , d2 , ... Values of these constants are usually determined from an analysis of historical data. The intermediate COCOMO model can be considered to be an example of a multivariable estimation model.

COCOMO:

Halstead's Software Science:

Project scheduling:

Project Scheduling in a project refers to roadmap of all activities to be done with specified order and within time slot allotted to each activity. Project managers tend to define various tasks, and project milestones and then arrange them keeping various factors in mind. They look for tasks like in critical path in the schedule, which are necessary to complete in specific manner (because of task interdependency) and strictly within the time allocated. Arrangement of tasks which lies out of critical path are less likely to impact over all schedule of the project.
For scheduling a project, it is necessary to -

- Break down the project tasks into smaller, manageable form
- Find out various tasks and correlate them
- Estimate time frame required for each task
- Divide time into work-units
- Assign adequate number of work-units for each task
- Calculate total time required for the project from start to finish

staffing:

Organization and team structure:

Risk management:

Risk management involves all activities pertaining to identification, analyzing and making provision for predictable and non-predictable risks in the project. Risk may include the following:

- Experienced staff leaving the project and new staff coming in.
- Change in organizational management.
- Requirement change or misinterpreting requirement.
- Under-estimation of required time and resources.
- Technological changes, environmental changes, business competition.

Risk Management Process:

There are following activities involved in risk management process:

- **Identification -** Make note of all possible risks, which may occur in the project.
- **Categorize -** Categorize known risks into high, medium and low risk intensity as per their possible impact on the project.

- **Manage -** Analyse the probability of occurrence of risks at various phases.
  Make plan to avoid or face risks. Attempt to minimize their side-effects.
- **Monitor -** Closely monitor the potential risks and their early symptoms.
  Also monitor the effective steps taken to mitigate or avoid them.

configuration management:

Unit-2: requirement analysis and specification

Experienced developers take considerable time to understand the exact requirements of the customer and to meticulously document those. They know that without a clear understanding of the problem and proper documentation of the same, it is impossible to develop a satisfactory solution.

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. When software is developed in a contract mode for some other organisation (that is, an outsourced project), the crucial role played by documentation of the precise requirements cannot be overstated. Even when an organisation develops a generic software product, the situation is not very different since some personnel from the organisation's own marketing department act as the customer. Therefore, for all types of software development projects, proper formulation of requirements and their effective documentation is vital. However, for very small software service projects, the agile methods advocate incremental development of the requirements.

An overview of requirements analysis and specification phase:

The requirements analysis and specification phase ends when the requirements specification document has been developed and reviewed. The requirements specification document is usually called as the software requirements specification (SRS) document. The goal of the requirements analysis and specification phase can be stated in a nutshell as follows.

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.

Who carries out requirements analysis and specification?

The engineers who gather a n d analyse customer requirements and then write the requirements specification document are known as system analysts in the software industry parlance. System analysts collect data pertaining to the product to be developed and analyse the collected data to conceptualise what exactly needs to be done. 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 SRS document is the final outcome of the requirements analysis and specification phase.

The nature of software:

Today, software takes on a dual role. It is a product at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or network of computers, accessible by a local hardware. Whether it resides with a mobile phone or operates inside a mainframe computer. Software is an information transformer- producing, managing, acquiring, modifying, displaying, or transmitting information. As the vehicle to deliver the product, software acts as the basics for the control of computer(operating systems), the communication of information(networks), and the creation and control of other programs(software tools and environment).

Software delivers most important product of our time -information. It transforms personal data(e.g., an individual's financial transactions) so that the data can be more useful in a local context; it provides a gate way to worldwide information networks(e.g., the network), and provides the means of acquiring information of all its forms.

Today, a huge software industry has become a dominant factor in the economies of the industrialised world.

The Unique nature of Webapps:

In the early days of the world wide web (1990 – 1995), websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics.

Today, webapps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also been integrated with corporate databases and business applications due to the development of HTML, Java, XML etc.

The following are the attributes encountered in the vast majority of the web apps:

- **Network insensitiveness:** A webapp resides on a network and must serve the need 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 internet).
- **Concurrency:** A large number of users may access the webapps 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 from day to day. 100 users may show up on Monday; 10,000 may use on Thursday.
- **Performance:** If a webapp user must wait too long, he or she may decide to go elsewhere.
- **Availability:** Users of popular webapps may demand access on a 24/7/365 basis.
- **Data driven:** The primary function of many webapps is to use hypermedia to text, graphics, audio and video content to the end user.
- **Content sensitive:**
- **Continuous evolution:** Unlike conventions application software, web applications evolve continuously.
- **Immediacy:** Immediacy is a need to get software to market quickly. Webapps often exhibit a time-to-market that can be a matter of a few days or weeks.
- **Security:** Webapps are available via network. In order to protect sensitive content and provide secure modes of data transmission, strong security measures must be implemented throughout the infrastructure of a webapp and within the application itself.

- **Aesthetics:** An undeniable part of 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 as technical design.

Software Myths:

Software myths: erroneous beliefs about software and the process that is used to build it. Myths have a number of attributes that make them insidious.

Today, most knowledgeable software engineering professionals recognise myths that have caused serious problems for managers and practitioners alike.

Management myths: Managers are often under pressure to maintain budgets, keep schedules from slipping and improve quality.

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? 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 all of these questions is "no".

Myth: If we get behind schedule, we can add more programmers and catchup (sometimes called the "Mongolian horde" concept).

Reality: Software development is not a mechanical process, like manufacturing. At first, the statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby it reduces the time of productivity.

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

Reality: If an organisation does not understand how to manage and control software project internally, it will invariably struggle when it outsources software projects.

Customer myths: In many cases, customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and ultimately dissatisfaction with the developer.

Myth: A general statement of objective 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 (usually derived iteratively) are developed only through effective and continuous communication between customer and developer.

Requirements gathering and analysis:

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

- Requirements gathering
- Requirement's analysis

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. 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.

To systematise this method of requirements gathering, the Delphi technique can be followed. In this technique, the analyst consolidates the requirements as understood by him in a document and then circulates it for the comments of the various categories of users. Based on their feedback, he refines his document. This procedure is repeated till the different users agree on the set of requirements.

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.

Task analysis helps the analyst to understand the nitty-gritty of various user tasks and to represent each task as a hierarchy of subtasks.

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.

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 b e 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.

Requirement's analysis:

The main purpose of the requirements analysis activity is to analyse the gathered requirements to remove all ambiguities, incompleteness, and inconsistencies from the gathered customer requirements and to obtain a clear understanding of the software to be developed.

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

- Anomaly
- Inconsistency
- Incompleteness

Let us examine these different types of requirements problems in detail.

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.

Example: While gathering the requirements for a process control application, the following requirement was expressed by a certain stakeholder: When the temperature becomes high, the heater should be switched off. Please note that words such as "high", "low", "good", "bad" etc. are indications of ambiguous requirements as these lack quantification and can be subjectively interpreted. If the threshold above which the temperature can be considered to be high is not specified, then it can be interpreted differently by different developers.

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

Example: Consider the following two requirements that were collected from two different stakeholders in a process control application development project.

- The furnace should be switched-off when the temperature of the furnace rises above 500 C.
- When the temperature of the furnace rises above 500 C, the water shower should be switched-on and the furnace should remain on.

The requirements expressed by the two stakeholders are clearly inconsistent.

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. The following are two examples of incomplete requirements:

Example: Suppose for the case study 4.1, 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.

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 it's 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. Also, a proper understanding of the functionalities supported enables them to determine the specific modifications to the system's functionalities would be needed for a specific purpose.

Many software engineers in a project consider the SRS document to be a reference document. However, it is often more appropriate to think of the SRS document as the documentation of a contract between the development team and the customer. In fact, the SRS document can be used to resolve any disagreements between the developers and the customers that may arise in the future. The SRS document can even be used as a legal document to settle disputes between the customers and the developers in a court of law. Once the customer agrees to the SRS document, the development team proceeds to develop the software and ensure that it conforms to all the requirements mentioned in the SRS document.

Software requirements specification:

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. The SRS document usually contains all the user requirements in a structured though an informal form.

Among all the documents produced during a software development life cycle, SRS document is probably the most important document and is the toughest to write. One reason for this difficulty is that the SRS document is expected to cater to the needs of a wide variety of audience. In the following subsection, we discuss the different categories of users of an SRS document and their needs from it.

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 and *vice versa*. Traceability analysis is an important concept and is frequently used during software development. For example, by doing a traceability analysis, we can tell whether all the requirements have been satisfactorily addressed in all phases. It can also be used to assess the impact of a requirements change. That is, traceability makes it easy to identify which parts of the design and code would be affected, when certain requirement change occurs. It can also be used to study the impact of a bug that is known to exist in a code part on various requirements, etc.

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. An example scheme of numbering the functional requirements is shown in Examples 4.7 and 4.8, where the functional requirements have been numbered R.1, R.2, etc. and the sub requirements for the requirement R.1 have been numbered R.1.1, R.1.2, etc.

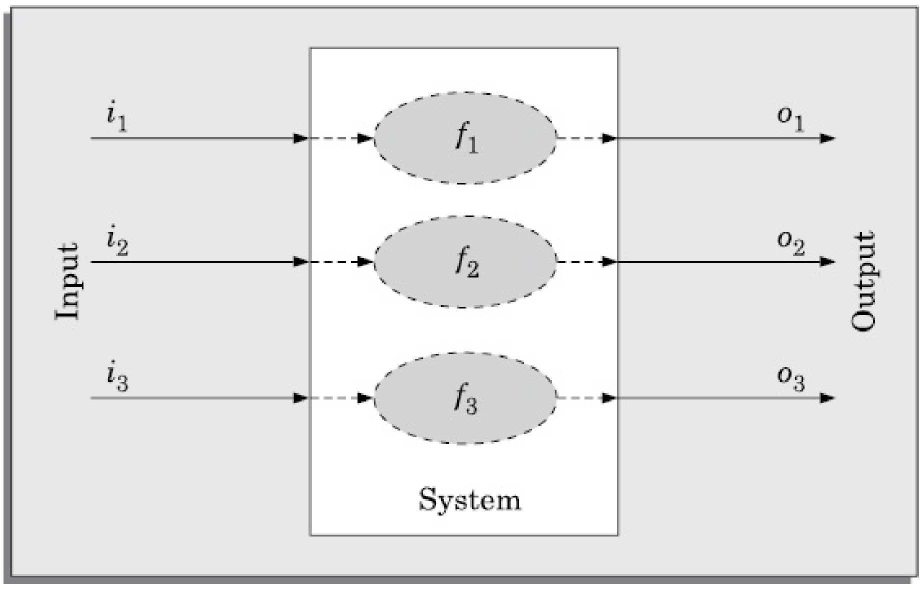
Characteristics of a Good SRS Document:

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). 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 and not discuss the implementation issues. This view with which a requirements specification is written, has been shown in Figure 4.1. Observe that in Figure 4.1, the SRS document describes the output produced for the different types of input and a description of the processing required to produce the output from the input (shown in ellipses) and the internal working of the software is not discussed at all.

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 SRS document is also called the black-box specification of the software being developed.



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

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 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.

IEEE 830 guidelines:

Representing complex requirements using decision tables and decision trees:

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 for it. However, use of decision trees

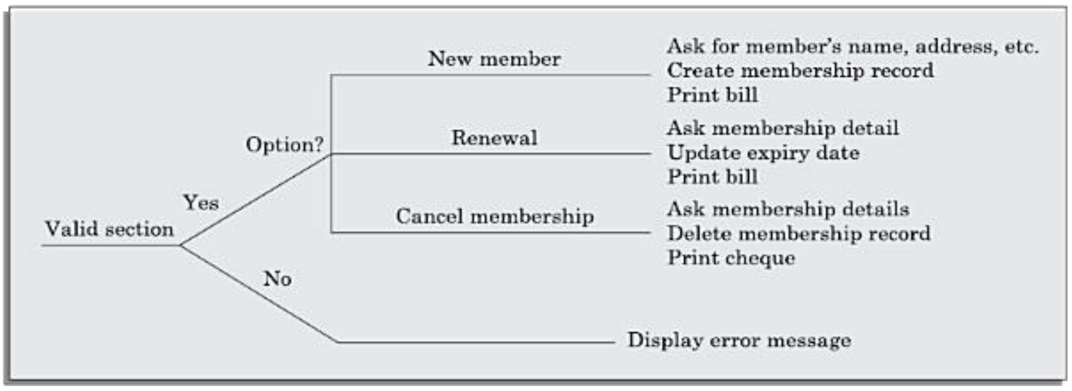
or tables would be superfluous in cases where the number of alternatives are few, or the decision logic is straightforward. In such cases, a simple text description would suffice.
There are two main techniques available to analyse and represent complex processing logic—decision trees and decision tables.

Decision tree:

A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken. Decision tables specify which variables are to be tested, and based on this what actions are to be taken depending upon the outcome of the decision making logic, and the order in which decision making is performed.

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 conditions. Instead of discussing how to draw a decision tree for a given processing logic, we shall explain through a simple example how to represent the processing logic in the form of a decision tree.

Example: A library membership management software (LMS) should support the following three options—new member, renewal, and cancel membership. When the *new member* option is selected, the software should ask the member's name, address, and phone number. If proper information is entered, the software should create a membership record for the new member and print a bill for the annual membership charge and the security deposit payable. If the *renewal* option is chosen, the LMS should ask the member's name and his membership number and check whether he is a valid member. If the member details entered are valid, then the membership expiry date in the membership record should be updated and the annual membership charge payable by the member should be printed. If the membership details entered are invalid, an error message should be displayed. If the *cancel membership* option is selected and the name of a valid member is entered, then the membership is cancelled, a choke for the balance amount due to the member is printed and his membership record is deleted.



Decision Tree for LMS.

The internal nodes represent conditions, the edges of the tree correspond to the outcome of the corresponding conditions. The leaf nodes represent the actions to be performed by the system. In the decision tree, first the user selection is checked. Based on whether the selection is valid, either further condition checking is undertaken or an error message is displayed. Observe that the order of condition checking is explicitly represented.

Decision table

A decision table shows the decision-making logic and the corresponding actions taken in a tabular or a matrix form. The upper rows of the table specify the variables or conditions to be evaluated and the lower rows specify the actions to be taken when an evaluation test is satisfied. A column in the table is called a *rule*. A rule implies that if a certain condition combination is true, then the corresponding action is executed. The decision table for the LMS problem of Example 4.10 is as shown in Table 4.1.

Table 4.1: Decision Table for the LMS Problem

Conditions				
Valid selection	NO	YES	YES	YES
New member	-	YES	NO	NO
Renewal	-	NO	YES	NO
Cancellation	-	NO	NO	YES
Actions				
Display error message	×			
Ask member's name, etc.		×		
Build customer record		×		
Generate bill		×	×	
Ask membership details			×	×
Update expiry date			×	
Print cheque				×
Delete record				×

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.

Overview of formal system development techniques:

A formal technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realisable, 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 its 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), then syn is said to be the specification of sem, and sem is said to be the specificand of syn.
The generally accepted paradigm for system development is through a hierarchy of abstractions. Each stage in this hierarchy is an implementation of its preceding stage and a specification of the succeeding stage. The different stages in this system development activity are requirements specification, functional design, architectural design, detailed design, coding, implementation, etc. In general, formal techniques can be used at every stage of the system development activity to verify that the output of one stage conforms to the output of the previous stage.
Syntactic domains
The syntactic domain of a formal specification language consists of an alphabet of symbols and a set of formation rules to construct wellformed formulas from the alphabet. The well-formed formulas 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, synchronisation 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 behaviour and the others describe the system's structure. Consequently, two broad classes of semantic abstraction functions are defined— those that preserve a system's behaviour and those that preserve a system's structure.
Model versus property-oriented methods
Formal methods are usually classified into two broad categories—the socalled model-oriented and the property-oriented approaches. In a modeloriented style, one defines a system's behaviour 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 behaviour is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy. Let us consider a simple producer/consumer example. In a property-oriented style, we would probably start 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. Two examples of property-oriented specification styles are axiomatic specification and algebraic specification.
In a model-oriented style, we would start by defining the basic operations, p (produce) and c (consume). Then we can state that $S 1 + p \Rightarrow S, S + c \Rightarrow S$
1. Thus model-oriented approaches essentially specify a program by writing another, presumably simpler program. A few notable examples of popular model-oriented specification techniques are Z, CSP,CCS, etc. It is alleged that property-oriented approaches are more suitable for requirements specification, and that the model-oriented approaches are more suited to system design specification. The reason for this distinction is the fact that property-oriented approaches specify a system behaviour not by what they say of the system but by what they do not say of the system. Thus, property-oriented specifications permit a large number of possible implementations. Furthermore, property-oriented approaches specify a system by a conjunction of axioms, thereby making it easier to alter/augment specifications at a later stage. On

the other hand, model-oriented methods do not support logical conjunctions and disjunctions, and thus even minor changes to a specification may lead to overhauling an entire specification. Since the initial customer requirements undergo several changes as the development proceeds, the property-oriented style is generally preferred for requirements specification. Later in this chapter, we have discussed two property-oriented specification techniques.

4.3.2 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 behaviour of the system. In the following subsection we discuss some of the commonly used operational semantics.

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 interleaving of the atomic actions. For example, a concurrent activity a || b is represented by the set of sequential activities a; b and b; a. This is a simple but rather unnatural representation of concurrency. The behaviour of a system in this model consists of the set of all its runs. To make this model more realistic, usually justice and fairness restrictions are imposed on computations to exclude the unwanted interleaving.

Branching semantics: In this approach, the behaviour of a system is represented by a directed graph. 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. 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 constrains 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.

Merits and limitations of formal methods

In addition to facilitating precise formulation of specifications, formal methods possess several positive features, some of which are discussed as follows:

Formal specifications encourage rigour. It is often the case that 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 behaviour that are not obvious in an informal specification. It is widely acknowledged that it is cost-effective to spend more efforts at the specification stage, otherwise, many flaws would go unnoticed only to be detected at the later stages of software development that would lead to iterative changes to occur in the development life cycle. According to an estimate, for large and complex systems like distributed real-time systems 80 per cent of project costs and most of the cost overruns result from the iterative changes required in a system development process due to inappropriate formulation of requirements specification. Thus, the additional effort required to construct a rigorous specification is well worth the trouble.

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. Informal specifications may be useful in understanding a system and its documentation, but they cannot serve as a basis of verification. Even carefully written specifications are prone to error, and experience has shown that unverified specifications are comparable in reliability to unverified programs. automatically avoided when one formally specifies a system.

The mathematical basis of the formal methods makes it possible for automating the analysis of specifications. For example, a tableaubased 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 behaviour of the specified system, and is especially useful in checking the completeness of specifications.

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

It has been pointed out by several researchers that formal specifications neither replace nor make the informal descriptions obsolete but complement them. In fact, the comprehensibility of formal specifications is greatly enhanced when the specifications are accompanied by an informal description. What is suggested is the use of formal techniques as a broad guideline for the use of the informal techniques. An interesting example of such an approach is reported by Jones in [1980]. In this approach, the use of a formal method identifies the necessary verification steps that need to be carried out, but it is legitimate to apply informal reasoning in presentation of correctness arguments and transformations. Any doubt or query relating to an informal argument is to be resolved by formal proofs.

In the following two sections, we discuss the axiomatic and algebraic specification styles. Both these techniques can be classified as the property-oriented specification techniques.

Axiomatic specification:

Algebraic specification:

Unit-3: Software design

The activities carried out during the design phase (called as design process) transform the SRS document into the design document.

Good Software Design:

In fact, the definition of a "good" software design can vary depending on the exact application being designed. most researchers and software engineers agree on a few desirable characteristics that every good software design for general applications must possess. These characteristics are listed below:
Correctness: A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.

Understandability: A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.

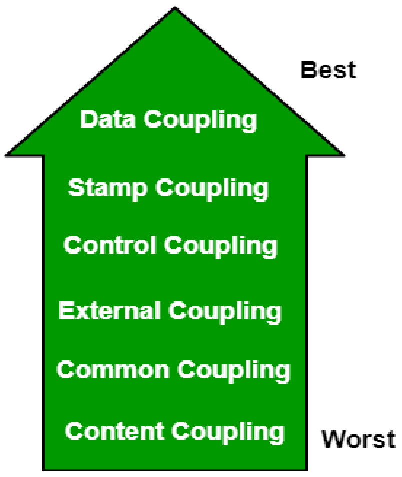
Efficiency: A good design solution should adequately address resource, time, and cost optimisation issues.

Maintainability: A good design should be easy to change. This is an important requirement, since change requests usually keep coming from the customer even after product release.

Modularity: A modular design is an effective decomposition of a problem. It is a basic characteristic of any good design solution. A modular design, in simple words, implies that the problem has been decomposed into a set of modules that have only limited interactions with each other.

A design solution is said to be highly modular, if the different modules in the solution have high cohesion and their inter-module couplings are low.

Coupling: Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.

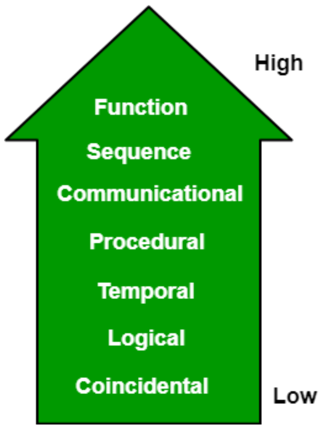


Types of Coupling:

- **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent to each other and communicating through data. Module communications don't contain tramp data. Example-customer billing system.
- **Stamp Coupling** In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice made by the insightful designer, not a lazy programmer.
- **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.
- **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.

- **Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses and reduced maintainability.
- **Content Coupling:** In a content coupling, one module can modify the data of another module or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

Cohesion: Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.



Types of Cohesion:

- **Functional Cohesion:** Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.
- **Sequential Cohesion:** An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
- **Communicational Cohesion:** Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer.
- **Procedural Cohesion:** Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.
- **Temporal Cohesion:** The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time-span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time.
- **Logical Cohesion:** The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.
- **Coincidental Cohesion:** The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component.

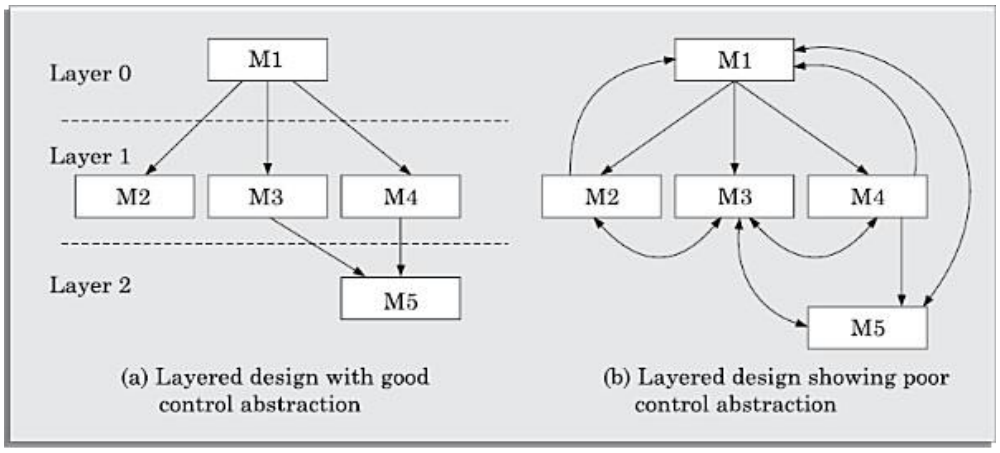
Control Hierarchy:

Layering:

A layered design is one in which when the call relations among different modules are represented graphically, it would result in a tree-like diagram with clear layering. In a layered design solution, the modules are arranged in a hierarchy of layers.

An important characteristic feature of a good design solution is layering of the modules. A layered design achieves control abstraction and is easier to understand and debug.

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. The modules at the intermediate layers offer services to their higher layer by invoking the services of the lower layer modules and also by doing some work themselves to a limited extent. The modules at the lowest layer are the worker modules.



(a) Layered design with good control abstraction
(b) Layered design showing poor control abstraction

In the following, we discuss some important concepts and terminologies associated with a layered design:

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.
From the above figure, the depth is 3 and width is also 3.

Fan-out: Fan-out is a measure of the number of modules that are directly controlled by a given module. In above Figure, 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 above Figure, the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.

Software design approaches:

There are two fundamentally different approaches to software design that are in use today—function-oriented design, and object-oriented design. Though these two design approaches are radically different, they are complementary rather than competing techniques. The object-oriented approach is a relatively newer technology and is still evolving. For development of large programs, the object- oriented approach is becoming increasingly popular due to certain advantages that it offers. On the other hand, function-oriented designing is a mature technology and has a large following.

Function-oriented Design:

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

Top-down decomposition:

In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed 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 high-level function may be refined into the following subfunctions:

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

Each of these subfunctions may be split into more detailed subfunctions and so on.

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.

For example, the set of books (i.e., whether borrowed by different users or available for issue) determines the state of a library automation system. Such data in procedural programs usually have global scope and are shared by many modules.

The system state is centralised and shared among different functions.

For example, in the library management system, several functions such as the following share data such as member-records for reference and update:

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

A large number of function-oriented design approaches have been proposed in the past. A few of the well-established function-oriented design approaches are as following:

- Structured design by Constantine and Yourdon, [1979]
- Jackson's structured design by Jackson [1975]
- Wargnier-Orr methodology [1977, 1981]
- Step-wise refinement by Wirth [1971]
- Hatley and Pirbhai's Methodology [1987]

Object-oriented Design:

In the object-oriented design (OOD) approach, a system is viewed as being made up of a collection of objects (i.e., entities). Each object is associated with a set of functions that are called its methods. Each object contains its own data and is responsible for managing it. The data internal to an object cannot be accessed directly by other objects and only through invocation of the methods of the object. The system state is decentralised since there is no globally shared data in the system and data is stored in each object. For example, in a library automation software, each library member may be a separate object with its own data and functions to operate on the stored data. The methods defined for one object cannot directly refer to or change the data of other objects.

The object-oriented design paradigm makes extensive use of the principles of abstraction and decomposition. Objects decompose a system into functionally independent modules. Objects can also be considered as instances of abstract data types (ADTs). ADT is an important concept that forms an important pillar of object orientation. There are three important concepts associated with an ADT—data abstraction, data structure, data type. We discuss these in the following subsection:

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. The entities external to the object can access the data internal to an object only by calling certain well-defined methods supported by the object. Consider an ADT such as a stack. The data of a stack object may internally be stored in an array, a linearly linked list, or a bidirectional linked list. The external entities have no knowledge of this and can access data of a stack object only through the supported operations such as push and pop.

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.

In object-orientation, classes are ADTs. Let us examine the three main advantages of using ADTs in programs:

The data of objects are encapsulated within the methods. The encapsulation principle is also known as data hiding. The encapsulation principle requires that data can be accessed and manipulated only through the methods supported by the object and not directly. This localises the errors. The reason for this is as follows. No program element is allowed to change a data, except through invocation of one of the methods. So, any error can easily be traced to the code segment changing the value. That is, the method that changes a data item, making it erroneous can be easily identified.

An ADT-based design displays high cohesion and low coupling. Therefore, object- oriented designs are highly modular.

Since the principle of abstraction is used, it makes the design solution easily understandable and helps to manage complexity.

Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from a super class. Conceptually, objects communicate by message passing. Objects have their own internal data. Thus, an object may exist in different states depending the values of the internal data. In different states, an object may behave differently.

Object oriented vs. function-oriented design:

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

- Unlike function-oriented design methods in OOD, the basic abstraction is not the services available to the users of the system such as issue book, display-book-details, find-issued-books, etc., but real-world entities such as member, book, book-register, 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.
- In OOD, state information exists in the form of data distributed among several objects of the system. In contrast, in a procedural design, the state information is available in a centralised shared data store. 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 design, 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 sending a message to it. Of course, somewhere or other the real-world functions must be implemented.

- Function-oriented techniques 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.

let us consider an example—that of an automated fire-alarm system for a large building.

Automated fire-alarm system—customer requirements:

The owner of a large multi-storied building wants to have a computerised fire alarm system designed, developed, and installed in 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 has been sensed and then sound the alarms only in the neighbouring locations. The fire alarm system should also flash an alarm message on the computer console. Firefighting personnel would 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 firefighting personnel.

Function-oriented approach: In this approach, the different high-level functions are first identified, and then the data structures are designed.

```
/* 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   neighbour-alarms[MAX-ROOMS][10]; /* each detector has at most */
                                              /* 10 neighbouring alarm locations */
        int   sprinkler[MAX_ROOMS];
```

The functions which operate on the system state are:

interrogate_detectors();

get_detector_location();

determine_neighbour_alarm();

determine_neighbour_sprinkler();

ring_alarm(); activate_sprinkler();

reset_alarm(); reset_sprinkler();

report_fire_location();

Object-oriented approach: In the object-oriented approach, the different classes of objects are identified. Subsequently, the methods and data for each object are identified. Finally, an appropriate number of instances of each class is created.

class detector attributes: status, location, neighbours' operations: create, sense-status, get-location, find-neighbours

class alarm attributes: location, status

operations: create, ring-alarm, get_location, reset_alarm class _sprinkler

attributes: location, status operations: create, activate-sprinkler, get_location, reset-sprinkler

By the following examples discussed above, we can easily observe the following differences:

In a function-oriented program, the system state (data) is centralised and several functions access and modify this central data. In case of an object-oriented program, the state information (data) is distributed among various objects.

In the object-oriented design, data is private in different objects and these are not available to the other objects for direct access and modification.

The basic unit of designing an object-oriented program is objects, whereas it is functions and modules in procedural designing. Objects appear as nouns in the problem description; whereas functions appear as verbs.

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 6.1. 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.

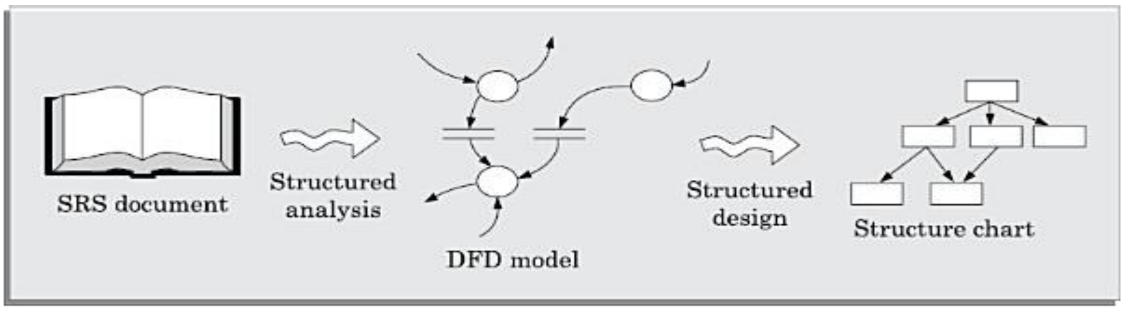


Figure 6.1: Structured analysis and structured design methodology.

As shown in Figure 6.1, 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. That is, 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 highlevel design or the software architecture for the given problem. This is represented using a structure chart.

The high-level design stage is normally followed by a detailed design stage. During the detailed design stage, the algorithms and data structures for the individual modules are designed. The detailed design can directly be implemented as a working system using a conventional programming language.

Structured analysis:

Significant contributions to the development of the structured analysis techniques have been made by Gane and Sarson [1979], and DeMarco and Yourdon [1978]. The structured analysis technique is based on the following underlying principles:

- Top-down decomposition approach.
- Application of divide and conquer principle. Through this each high-level function is independently decomposed into detailed functions.
- Graphical representation of the analysis results using data flow diagrams (DFDs).

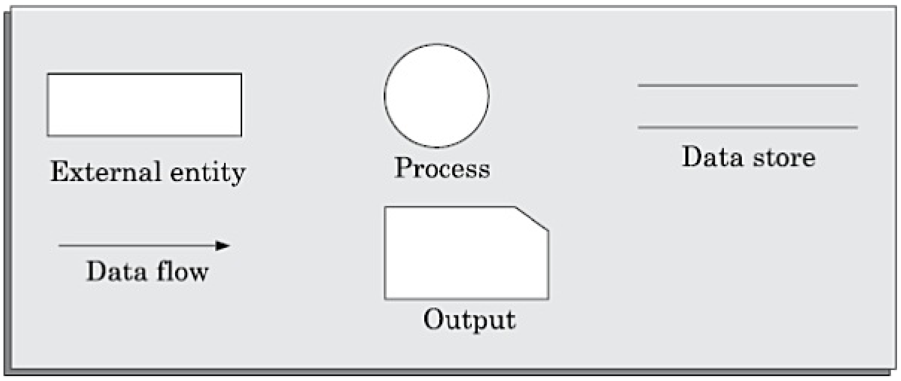
Data flow diagram:

A DFD 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 those functions.

The DFD (also known as the bubble chart) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on those data, and the output data generated by the system. The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism— it is simple to understand and use. 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.

Hierarchical model starts with a very abstract model of a system, various details of the system are slowly introduced through different levels of the hierarchy. The DFD technique is also based on a very simple set of intuitive concepts and rules. We now elaborate the different concepts associated with building a DFD model of a system.

There are essentially five different types of symbols used for constructing DFDs.



Function symbol: A function is represented using a circle. This symbol is called a process or a bubble. Bubbles are annotated with the names of the corresponding functions (see Figure 6.3).

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 in Figure 6.3(a) shows three data flows—the data item number flowing from the process read-number to validate-number, dataitem 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 t o 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 in Figure 6.3(b).

Output symbol: The output symbol i s as shown in Figure 6.2. The output symbol is used when a hard copy is produced.

The notations that we are following in this text are closer to the Yourdon’s notations than to the other notations. You may sometimes find notations in other books that are slightly different than those discussed here. For example, the data store may look like a box with one end open. That is because, they may be following notations such as those of Gane and Sarson [1979].

Extending DFD technique to real life systems:

In a real-time system, some of the high-level functions are associated with deadlines. Therefore, a function must not only produce correct results but also should produce them by some prespecified time. For real-time systems, execution time is an important consideration for arriving at a correct design. Therefore, explicit representation of control and event flow aspects are essential. One of the widely accepted techniques for extending the DFD technique to real-time system analysis is the Ward and Mellor technique [1985]. In the Ward and Mellor notation, a type of process that handles only control flows is introduced. These processes representing control processing are denoted using dashed bubbles. Control flows are shown using dashed lines/arrows.

Unlike Ward and Mellor, Hatley and Pirbhai [1987] show the dashed and solid representations on separate diagrams. To be able to separate the data processing and the control processing aspects, a control flow diagram (CFD) is defined. This reduces the complexity of the diagrams. In order to link the data processing and control processing diagrams, a notational reference (solid bar) to a control specification is used. The CSPEC describes the following:

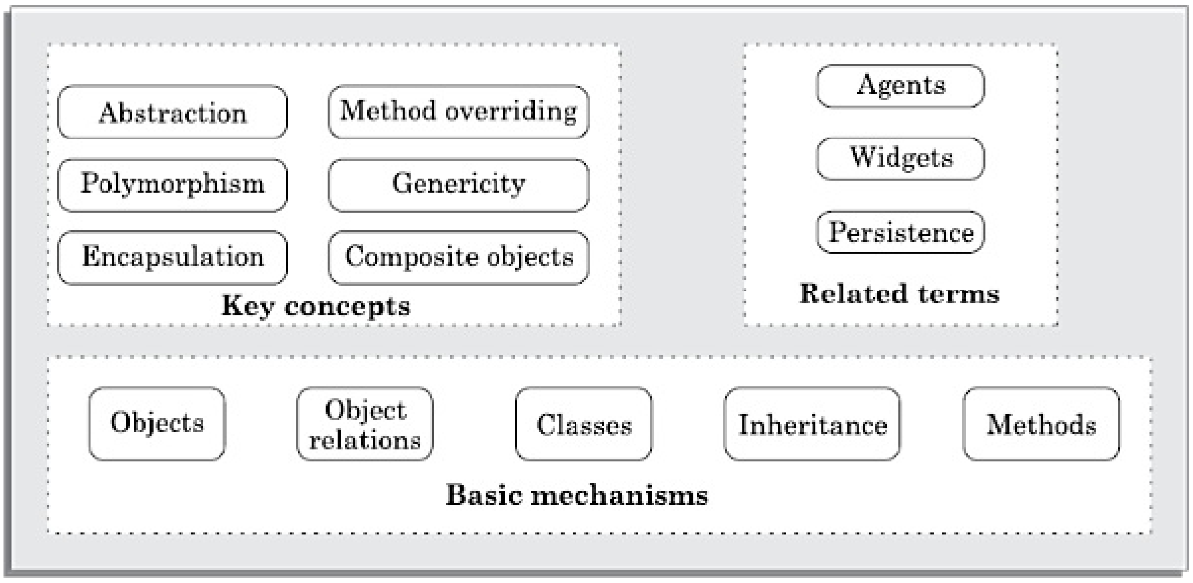
- The effect of an external event or control signal.
- The processes that are invoked as a consequence of an event.

Control specifications represents the behavior of the system in two different ways:

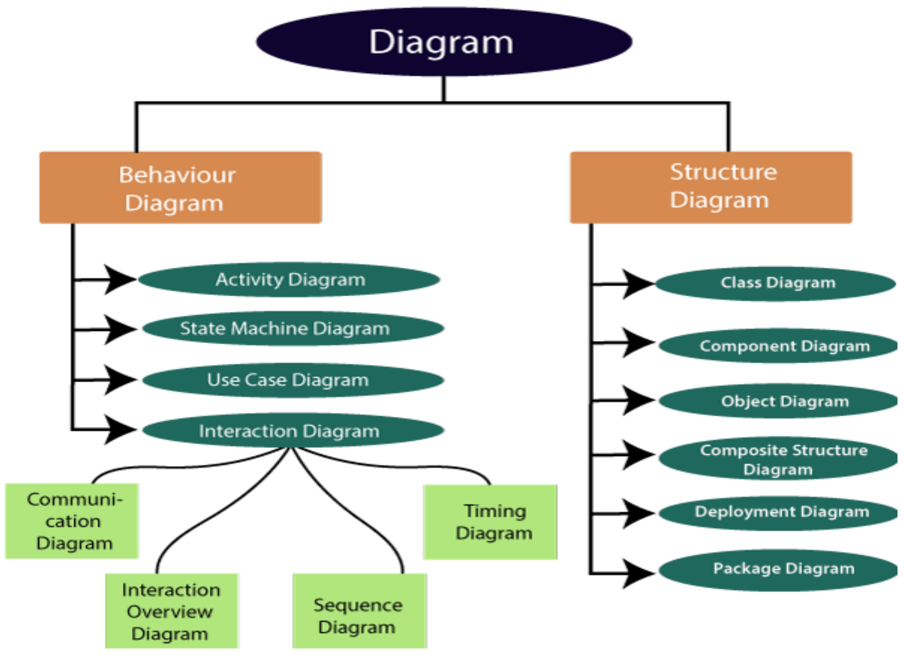
- It contains a state transition diagram (STD). The STD is a sequential specification of behaviour.
- It contains a program activation table (PAT). The PAT is a combinatorial specification of behaviour. PAT represents invocation sequence of bubbles in a DFD.

Basic Object-oriented concepts:

The principles of object-orientation have been founded on a few simple concepts.



UML Diagrams: The UML diagrams are categorized into **structural diagrams, behavioural diagrams,** and also interaction **overview diagrams.** The diagrams are hierarchically classified in the following figure:



1. Structural Diagrams

Structural diagrams depict a static view or structure of a system. It is widely used in the documentation of software architecture. It embraces class diagrams, composite structure diagrams, component diagrams, deployment diagrams, object diagrams, and package diagrams. It presents an outline for the system. It stresses the elements to be present that are to be modeled.

- **Class Diagram:** Class diagrams are one of the most widely used diagrams. It is the backbone of all the object-oriented software systems. It depicts the static structure of the system. It displays the system's class, attributes, and methods. It is helpful in recognizing the relation between different objects as well as classes.
- **Composite Structure Diagram:** The composite structure diagrams show parts within the class. It displays the relationship between the parts and their configuration that ascertain the behavior of the class. It makes full use of ports, parts, and connectors to portray the internal structure of a structured classifier. It is similar to class diagrams, just the fact it represents individual parts in a detailed manner when compared with class diagrams.
- **Object Diagram:** It describes the static structure of a system at a particular point in time. It can be used to test the accuracy of class diagrams. It represents distinct instances of classes and the relationship between them at a time.
- **Component Diagram:** It portrays the organization of the physical components within the system. It is used for modeling execution details. It determines whether the desired functional requirements have been considered by the planned development or not, as it depicts the structural relationships between the elements of a software system.
- **Deployment Diagram:** It presents the system's software and its hardware by telling what the existing physical components are and what software components are running on them. It produces information about system software. It is incorporated whenever software is used, distributed, or deployed across multiple machines with dissimilar configurations.

- **Package Diagram:** It is used to illustrate how the packages and their elements are organized. It shows the dependencies between distinct packages. It manages UML diagrams by making it easily understandable. It is used for organizing the class and use case diagrams.

2. Behavioral Diagrams:

Behavioral diagrams portray a dynamic view of a system or the behavior of a system, which describes the functioning of the system. It includes use case diagrams, state diagrams, and activity diagrams. It defines the interaction within the system.

- **State Machine Diagram:** It is a behavioral diagram. it portrays the system's behavior utilizing finite state transitions. It is also known as the **State-charts** diagram. It models the dynamic behavior of a class in response to external stimuli.
- **Activity Diagram:** It models the flow of control from one activity to the other. With the help of an activity diagram, we can model sequential and concurrent activities. It visually depicts the workflow as well as what causes an event to occur.
- **Use Case Diagram:** It represents the functionality of a system by utilizing actors and use cases. It encapsulates the functional requirement of a system and its association with actors. It portrays the use case view of a system.

3. Interaction Diagrams

Interaction diagrams are a subclass of behavioral diagrams that give emphasis to object interactions and also depicts the flow between various use case elements of a system. In simple words, it shows how objects interact with each other and how the data flows within them. It consists of communication, interaction overview, sequence, and timing diagrams.

- **Sequence Diagram:** It shows the interactions between the objects in terms of messages exchanged over time. It delineates in what order and how the object functions are in a system.
- **Communication Diagram:** It shows the interchange of sequence messages between the objects. It focuses on objects and their relations. It describes the static and dynamic behavior of a system.
- **Timing Diagram:** It is a special kind of sequence diagram used to depict the object's behavior over a specific period of time. It governs the change in state and object behavior by showing the time and duration constraints.
- **Interaction Overview diagram:** It is a mixture of activity and sequence diagram that depicts a sequence of actions to simplify the complex interactions into simple interactions.

Structured design:
Detailed design:
Design review:
Characteristics of a good user interface:
User Guidance and Online Help:
Mode-based Vs Mode-less Interface:
Types of user interfaces:

Component-based GUI development:
User interface design methodology: GUI design methodology

Unit – IV: Coding and Testing

CODING:

The input to the coding phase is the design document produced at the end of the design phase. Design document contains not only the high-level design of the system in the form of a module structure (e.g., a structure chart), but also the detailed design. During the coding phase, different modules identified in the design document are coded according to their respective module specifications.

The objective of the coding phase is to transform the design of a system into code in a high-level language, and then to unit test this code.

Software development organisations formulate their own coding standards that suit them the most, and require their developers to follow the standards rigorously. A coding standard gives a uniform appearance to the codes written by different engineers. It facilitates code understanding and code reuse. It promotes good programming practices.

It is mandatory for the programmers to follow the coding standards. Compliance of their code to coding standards is verified during code inspection. Any code that does not conform to the coding standards is rejected during code review and the code is reworked by the concerned programmer. In contrast, coding guidelines provide some general suggestions regarding the coding style to be followed but leave the actual implementation of these guidelines to the discretion of the individual developers.

Coding standards and guidelines:

Good software development organisations usually develop their own coding standards and guidelines depending on what suits their organisation best and based on the specific types of software they develop. To give an idea about the types of coding standards that are being used, we shall only list some general coding standards and guidelines that are commonly adopted by many software development organisations, rather than trying to provide an exhaustive list.

Representative coding standards:

Rules for limiting the use of global: These rules list what types of data can be declared global and what cannot, with a view to limit the data that needs to be defined with global scope.

Standard headers for different modules: The header of different modules should have standard format and information for ease of understanding and maintenance. The following is an example of header format that is being used in some companies:

- Name of the module.
- Date on which the module was created.
- Author’s name.
- Modification history.
- Synopsis of the module. This is a small writeup about what the module does.
- Different functions supported in the module, along with their input/output parameters.

- Global variables accessed/modified by the module.

Naming conventions for global variables, local variables, and constant identifiers: A popular naming convention is that variables are named using mixed case lettering. Global variable names would always start with a capital letter (e.g., Global Data) and local variable names start with small letters (e.g., local Data). Constant names should be formed using capital letters only (e.g., CONSTDATA).

Conventions regarding error return values and exception handling mechanisms: The way error conditions are reported by different functions in a program should be standard within an organisation. For example, all functions while encountering an error condition should either return a 0 or 1 consistently, independent of which programmer has written the code. This facilitates reuse and debugging.

Representative coding guidelines: The following are some representative coding guidelines that are recommended by many software development organisations. Wherever necessary, the rationale behind these guidelines is also mentioned.

Do not use a coding style that is too clever or too difficult to understand: Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can obscure meaning of the code and reduce code understandability; thereby making maintenance and debugging difficult and expensive.

Avoid obscure side effects: The side effects of a function call include modifications to the parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code. For example, suppose the value of a global variable is changed or some file I/O is performed obscurely in a called module. That is, this is difficult to infer from the function's name and header information. Then, it would be really hard to understand the code.

Do not use an identifier for multiple purposes: Programmers often use the same identifier to denote several temporary entities. For example, some programmers make use of a temporary loop variable for also computing and storing the final result. The rationale that they give for such multiple use of variables is memory efficiency, e.g., three variables use up three memory locations, whereas when the same variable is used for three different purposes, only one memory location is used. However, there are several things wrong with this approach and hence should be avoided. Some of the problems caused by the use of a variable for multiple purposes are as follows:

- Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable for multiple purposes can lead to confusion and make it difficult for somebody trying to read and understand the code.
- Use of variables for multiple purposes usually makes future enhancements more difficult. For example, while changing the final computed result from integer to float type, the programmer might subsequently notice that it has also been used as a temporary loop variable that cannot be a float type.

Code should be well-documented: As a rule of thumb, there should be at least one comment line on the average for every three source lines of code.

Length of any function should not exceed 10 source lines: A lengthy function is usually very difficult to understand as it probably has a large number of variables and carries out many different types of computations. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.

Do not use GO TO statements: Use of GO TO statements makes a program unstructured. This makes the program very difficult to understand, debug, and maintain.

Code review:

Code review is a very effective technique to remove defects from source code. In fact, review has been acknowledged to be more cost-effective in removing defects as compared to testing.
Code review for a module is undertaken after the module successfully compiles. That is, all the syntax errors have been eliminated from the module. Code review is designed to syntax errors, detect logical, algorithmic, and programming errors. Code review has been recognised as an extremely cost-effective strategy for eliminating coding errors and for producing high quality code.
The reason behind why code review is a much more cost-effective strategy to eliminate errors from code compared to testing is that reviews directly detect errors. On the other hand, testing only helps detect failures and significant effort is needed to locate the error during debugging.
The rationale behind the above statement is explained as follows. Eliminating an error from code involves three main activities—testing, debugging, and then correcting the errors. Testing is carried out to detect if the system fails to work satisfactorily for certain types of inputs and under certain circumstances. Once a failure is detected, debugging is carried out to locate the error that is causing the failure and to remove it. Of the three testing activities, debugging is possibly the most laborious and time-consuming activity. In code inspection, errors are directly detected, thereby saving the significant effort that would have been required to locate the error.
Normally, the following two types of reviews are carried out on the code of a module:

- Code inspection.
- Code walkthrough.

The procedures for conduction and the final objectives of these two review techniques are very different. In the following two subsections, we discuss these two code review techniques.

Code Walkthrough:

Code walkthrough is an informal code analysis technique. In this technique, a module is taken up for review after the module has been coded, successfully compiled, and all syntax errors have been eliminated. A few members of the development team are given the code a couple of days before the walkthrough meeting. Each member selects some test cases and simulates execution of the code by hand (i.e., traces the execution through different statements and functions of the code).
The main objective of code walkthrough is to discover the algorithmic and logical errors in the code.
The members note down their findings of their walkthrough and discuss those in a walkthrough meeting where the coder of the module is present.
Even though code walkthrough is an informal analysis technique, several guidelines have evolved over the years for making this naive but useful analysis technique more effective. These guidelines are based on personal experience, common sense, several other subjective factors. Therefore, these guidelines should be considered as examples rather than as accepted rules to be applied dogmatically. Some of these guidelines are following:
The team performing code walkthrough should not be either too big or too small. Ideally, it should consist of between three to seven members.
Discussions should focus on discovery of errors and avoid deliberations on how to fix the discovered errors.
In order to foster co-operation and to avoid the feeling among the engineers that they are being watched and evaluated in the code walkthrough meetings, managers should not attend the walkthrough meetings.

Code Inspection:

During code inspection, the code is examined for the presence of some common programming errors. This is in contrast to the hand simulation of code execution carried out during code walkthroughs. We can state the principal aim of the code inspection to be the following:

The principal aim of code inspection is to check for the presence of some common types of errors that usually creep into code due to programmer mistakes and oversights and to check whether coding standards have been adhered to.

The inspection process has several beneficial side effects, other than finding errors. The programmer usually receives feedback on programming style, choice of algorithm, and programming techniques. The other participants gain by being exposed to another programmer's errors.

As an example of the type of errors detected during code inspection, consider the classic error of writing a procedure that modifies a formal parameter and then calls it with a constant actual parameter. It is more likely that such an error can be discovered by specifically looking for this kinds of mistakes in the code, rather than by simply hand simulating execution of the code. In addition to the commonly made errors, adherence to coding standards is also checked during code inspection.

Good software development companies collect statistics regarding different types of errors that are commonly committed by their engineers and identify the types of errors most frequently committed. Such a list of commonly committed errors can be used as a checklist during code inspection to look out for possible errors.

Following is a list of some classical programming errors which can be checked during code inspection:

- Use of uninitialized variables.
- Jumps into loops.
- Non-terminating loops.
- Incompatible assignments.
- Array indices out of bounds.
- Improper storage allocation and deallocation.
- Mismatch between actual and formal parameter in procedure calls. Use of incorrect logical operators or incorrect precedence among operators.
- Improper modification of loop variables.
- Comparison of equality of floating-point values.
- Dangling reference caused when the referenced memory has not been allocated.

software documentation:

Testing:

Black Box Testing:

White Box Testing:

Debugging:

After a failure has been detected, it is necessary to first identify the program statement(s) that are in error and are responsible for the failure, the error can then be fixed. In this Section, we shall summarise the important approaches that are available to identify the error locations. Each of these approaches has its own advantages and disadvantages and therefore each will be useful in appropriate circumstances. We also provide some guidelines for effective debugging.

Debugging Approaches:

The following are some of the approaches that are popularly adopted by the programmers for debugging:

Brute force method:

This is the most common method of debugging but is the least efficient method. In this approach, print statements are inserted throughout the program to print the intermediate values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly. Single stepping using a symbolic debugger is another form of this approach, where the developer mentally computes the expected result after every source instruction and checks whether the same is computed by single stepping through the program.

Backtracking:

This is also a fairly common approach. In this approach, starting from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large for complex programs, limiting the use of this approach.

Cause elimination method:

In this approach, once a failure is observed, the symptoms of the failure (i.e., certain variable is having a negative value though it should be positive, etc.) are noted. Based on the failure symptoms, the causes which could possibly have contributed to the symptom is developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

Program slicing:

This technique is similar to back tracking. In the backtracking approach, one often has to examine a large number of statements. However, the search space is reduced by defining slices. A slice of a program for a particular variable and at a particular statement is the set of source lines preceding this statement that can influence the value of that variable [Mund2002]. Program slicing makes use of the fact that an error in the value of a variable can be caused by the statements on which it is data dependent.

Debugging Guidelines:

Debugging is often carried out by programmers based on their ingenuity and experience. The following are some general guidelines for effective debugging:

Many times, debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the program design may require an inordinate amount of effort to be put into debugging even for simple problems.

Debugging may sometimes even require full redesign of the system. In such cases, a common mistakes that novice programmers often make is attempting not to fix the error but its symptoms.

One must be beware of the possibility that an error correction may introduce new errors. Therefore after every round of error-fixing, regression testing (see Section 10.13) must be carried out.

integration testing: Integration testing is carried out after all (or at least some of) the modules have been unit tested. Successful completion of unit testing, to a large extent, ensures that the unit (or module) as a whole works satisfactorily. In this context, the objective of integration testing is to detect the errors at the module interfaces (call parameters). For example, it is checked that no parameter mismatch occurs when one module invokes the functionality of another module. Thus, the primary objective of integration testing is to test the module interfaces, i.e., there are no errors in parameter passing, when one module invokes the functionality of another module.

The objective of integration testing is to check whether the different modules of a program interface with each other properly.

During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realise the full system. After each integration step, the partially integrated system is tested.

An important factor that guides the integration plan is the module dependency graph.

We have already discussed in Chapter 6 that a structure chart (or module dependency graph) specifies the order in which different modules call each other. Thus, by examining the structure chart, the integration plan can be developed. Any one (or a mixture) of the following approaches can be used to develop the test plan:

- Big-bang approach to integration testing
- Top-down approach to integration testing
- Bottom-up approach to integration testing
- Mixed (also called sandwiched) approach to integration testing

In the following subsections, we provide an overview of these approaches to integration testing.

Big-bang approach to integration testing:

Big-bang testing is the most obvious approach to integration testing. In this approach, all the modules making up a system are integrated in a single step. In simple words, all the unit tested modules of the system are simply linked together and tested. However, this technique can meaningfully be used only for very small systems. The main problem with this approach is that once a failure has been detected during integration testing, it is very difficult to localise the error as the error may potentially lie in any of the modules. Therefore, debugging errors reported during big-bang integration testing are very expensive to fix. As a result, big-bang integration testing is almost never used for large programs.

Bottom-up approach to integration testing:

Large software products are often made up of several subsystems. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. In bottom-up integration testing, first the modules for each subsystem are integrated. Thus, the subsystems can be integrated separately and independently.

The primary purpose of carrying out the integration testing a subsystem is to test whether the interfaces among various modules making up the subsystem work satisfactorily. The test cases must be carefully chosen to exercise the interfaces in all possible manners.

In a pure bottom-up testing no stubs are required, and only test-drivers are required. Large software systems normally require several levels of subsystem testing, lower-level subsystems are successively combined to form higher-level subsystems. The principal advantage of bottom- up integration testing is that several disjoint subsystems can be tested simultaneously. Another advantage of bottom-up testing is that the low-level modules get tested thoroughly, since they are exercised in each integration step. Since the low-level modules do I/O and other critical functions, testing the low-level modules thoroughly increases the reliability of the system. A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems that are at the same level. This extreme case corresponds to the big-bang approach.

Top-down approach to integration testing:

Top-down integration testing starts with the root module in the structure chart and one or two subordinate modules of the root module. After the top-level ‘skeleton’ has been tested, the modules that are at the immediately lower layer of the ‘skeleton’ are combined with it and tested. Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not require any driver routines. An advantage of top-down integration testing is that it requires writing only stubs, and stubs are simpler to write compared to drivers. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, it becomes difficult to exercise the top-level routines in the desired manner since the lower-level routines usually perform input/output (I/O) operations.

Mixed approach to integration testing:

The mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches. In top down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. In the mixed testing approach, testing can start as and when modules become available after unit testing.

Therefore, this is one of the most commonly used integration testing approaches. In this approach, both stubs and drivers are required to be designed.

Program Analysis Tools:

A program analysis tool usually is an automated tool that takes either the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, adequacy of testing, etc. We can classify various program analysis tools into the following two broad categories:
Static analysis tools
Dynamic analysis tools
These two categories of program analysis tools are discussed in the following subsection.

Static Analysis Tools:
Static program analysis tools assess and compute various characteristics of a program without executing it. Typically, static analysis tools analyse the source code to compute certain metrics characterising the source code (such as size, cyclomatic complexity, etc.) and also report certain analytical conclusions. These also check the conformance of the code with the prescribed coding standards. In this context, it displays the following analysis results:
To what extent the coding standards have been adhered to?
Whether certain programming errors such as uninitialised variables, mismatch between actual and formal parameters, variables that are declared but never used, etc., exist? A list of all such errors is displayed.
Code review techniques such as code walkthrough and code inspection discussed in Sections 10.2.1 and 10.2.2 can be considered as static analysis methods since those target to detect errors based on analysing the source code. However, strictly speaking, this is not true since we are using the term static program analysis to denote automated analysis tools. On the other hand, a compiler can be considered to be a type of a static program analysis tool.
A major practical limitation of the static analysis tools lies in their inability to analyse run-time information such as dynamic memory references using pointer variables and pointer arithmetic, etc. In a high level programming languages, pointer variables and dynamic memory allocation provide the capability for dynamic memory references. However, dynamic memory referencing is a major source of programming errors in a program.
Static analysis tools often summarise the results of analysis of every function in a polar chart known as Kiviat Chart. A Kiviat Chart typically shows the analysed values for cyclomatic complexity, number of source lines, percentage of comment lines, Halstead’s metrics, etc.

Dynamic Analysis Tools:
Dynamic program analysis tools can be used to evaluate several program characteristics based on an analysis of the run time behaviour of a program. These tools usually record and analyse the actual behaviour of a program while it is being executed. A dynamic program analysis tool (also called a dynamic analyser) usually collects execution trace information by instrumenting the code. Code instrumentation is usually achieved by inserting additional statements to print the values of certain variables into a file to collect the execution trace of the program. The instrumented code when executed, records the behaviour of the software for different test cases.
An important characteristic of a test suite that is computed by a dynamic analysis tool is the extent of coverage achieved by the test suite.
After a software has been tested with its full test suite and its behaviour recorded, the dynamic analysis tool carries out a post execution analysis and produces reports which describe the coverage that has been achieved by the complete test suite for the program. For example, the dynamic analysis tool can report the statement, branch, and path coverage achieved by a test suite. If the coverage achieved is not

satisfactory more test cases can be designed, added to the test suite, and run. Further, dynamic analysis results can help eliminate redundant test cases from a test suite.
Normally the dynamic analysis results are reported in the form of a histogram or pie chart to describe the structural coverage achieved for different modules of the program. The output of a dynamic analysis tool can be stored and printed easily to provide evidence that thorough testing has been carried out.

system testing:

After all the units of a program have been integrated together and tested, system testing is taken up.
System tests are designed to validate a fully developed system to assure that it meets its requirements.
The test cases are therefore designed solely based on the SRS document.
The system testing procedures are the same for both object-oriented and procedural programs, since system test cases are designed solely based on the SRS document and the actual implementation (procedural or objectoriented) is immaterial.
There are essentially three main kinds of system testing depending on who carries out testing:

1. Alpha Testing: Alpha testing refers to the system testing carried out by the test team within the developing organisation.
2. Beta Testing: Beta testing is the system testing performed by a select group of friendly customers.
3. Acceptance Testing: Acceptance testing is the system testing performed by the customer to determine whether to accept the delivery of the system.

In each of the above types of system tests, the test cases can be the same, but the difference is with respect to who designs test cases and carries out testing.
The system test cases can be classified into functionality and performance test cases.
Before a fully integrated system is accepted for system testing, smoke testing is performed. Smoke testing is done to check whether at least the main functionalities of the software are working properly. Unless the software is stable and at least the main functionalities are working satisfactorily, system testing is not undertaken.
The functionality tests are designed to check whether the software satisfies the functional requirements as documented in the SRS document. The performance tests, on the other hand, test the conformance of the system with the non-functional requirements of the system. We have already discussed how to design the functionality test cases by using a black-box approach (in Section 10.5 in the context of unit testing). So, in the following subsection we discuss only smoke and performance testing.

performance testing:

Performance testing is an important type of system testing.
Performance testing is carried out to check whether the system meets the nonfunctional requirements identified in the SRS document.
There are several types of performance testing corresponding to various types of non-functional requirements. For a specific system, the types of performance testing to be carried out on a system depends on the different non-functional requirements of the system documented in its SRS document. All performance tests can be considered as black-box tests.
Stress testing
Stress testing is also known as endurance testing. Stress testing evaluates system performance when it is stressed for short periods of time. Stress tests are black-box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software. Input data volume, input data rate, processing time, utilisation of memory, etc., are tested beyond the designed capacity. For example, suppose an operating system is supposed to support fifteen concurrent transactions, then the system is stressed by attempting to initiate fifteen or more transactions simultaneously. A real-time system might be tested to determine the effect of simultaneous arrival of several high-priority interrupts.

Stress testing is especially important for systems that under normal circumstances operate below their maximum capacity but may be severely stressed at some peak demand hours. For example, if the corresponding nonfunctional requirement states that the response time should not be more than twenty secs per transaction when sixty concurrent users are working, then during stress testing the response time is checked with exactly sixty users working simultaneously.
Volume testing
Volume testing checks whether the data structures (buffers, arrays, queues, stacks, etc.) have been designed to successfully handle extraordinary situations. For example, the volume testing for a compiler might be to check whether the symbol table overflows when a very large program is compiled.
Configuration testing
Configuration testing is used to test system behaviour in various hardware and software configurations specified in the requirements. Sometimes systems are built to work in different configurations for different users. For instance, a minimal system might be required to serve a single user, and other extended configurations may be required to serve additional users during configuration testing. The system is configured in each of the required configurations and depending on the specific customer requirements, it is checked if the system behaves correctly in all required configurations.
Compatibility testing
This type of testing is required when the system interfaces with external systems (e.g., databases, servers, etc.). Compatibility aims to check whether the interfaces with the external systems are performing as required. For instance, if the system needs to communicate with a large database system to retrieve information, compatibility testing is required to test the speed and accuracy of data retrieval.
Regression testing
This type of testing is required when a software is maintained to fix some bugs or enhance functionality, performance, etc. Regression testing is also discussed in Section 10.13.
Recovery testing
Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc. The system is subjected to the loss of the mentioned resources (as discussed in the SRS document) and it is checked if the system recovers satisfactorily. For example, the printer can be disconnected to check if the system hangs. Or, the power may be shut down to check the extent of data loss and corruption.
Maintenance testing
This addresses testing the diagnostic programs, and other procedures that are required to help maintenance of the system. It is verified that the artifacts exist and they perform properly.
Documentation testing
It is checked whether the required user manual, maintenance manuals, and technical manuals exist and are consistent. If the requirements specify the types of audience for which a specific manual should be designed, then the manual is checked for compliance of this requirement.
Usability testing
Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface. During usability testing, the display screens, messages, report formats, and other aspects relating to the user interface requirements are tested. A GUI being just being functionally correct is not enough. Therefore, the GUI has to be checked against the checklist we discussed in Sec. 9.5.6.
Security testing
Security testing is essential for software that handle or process confidential data that is to be gurarded against pilfering. It needs to be tested whether the system is fool-proof from security attacks such as intrusion by hackers. Over the last few years, a large number of security testing techniques have been proposed, and these include password cracking, penetration testing, and attacks on specific ports, etc.

regression testing:

Regression testing spans unit, integration, and system testing. Instead, it is a separate dimension to these three forms of testing. Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix. However, if only a few statements are changed, then the entire test suite need not be run — only those test cases that test the functions and are likely to be affected by the change need to be run. Whenever a software is changed to either fix a bug, or enhance or remove a feature, regression testing is carried out.

Testing Object Oriented Programs:

Traditional procedural programs, procedures are the basic unit of testing. In contrast, objects are the basic unit of testing for object-oriented programs. Besides this, there are many other significant differences as well between testing procedural and object-oriented programs.
In fact, the different object- oriented features (inheritance, polymorphism, dynamic binding, state-based behaviour, etc.) require special test cases to be designed compared to the traditional testing As a result, the design model is a valuable artifact for testing object-oriented programs. Test cases are designed based on the design model. Therefore, this approach is considered to be intermediate between a fully white-box and a fully black-box approach, and is called a grey-box approach. Please note that grey-box testing is considered important for object-oriented programs. This is in contrast to testing procedural programs.
10.11.4 Grey-Box Testing of Object-oriented Programs
As we have already mentioned, model-based testing is important for objectoriented programs, as these test cases help detect bugs that are specific to the object-orientation constructs.
For object-oriented programs, several types of test cases can be designed based on the design models of object-oriented programs. These are called the grey-box test cases.
The following are some important types of grey-box testing that can be carried on based on UML models:
State-model-based testing
State coverage: Each method of an object are tested at each state of the object.
State transition coverage: It is tested whether all transitions depicted in the state model work satisfactorily.
State transition path coverage: All transition paths in the state model are tested.
Use case-based testing
Scenario coverage: Each use case typically consists of a mainline scenario and several alternate scenarios. For each use case, the mainline and all alternate sequences are tested to check if any errors show up.
Class diagram-based testing
Testing derived classes: All derived classes of the base class have to be instantiated and tested. In addition to testing the new methods defined in the derivec. lass, the inherited methods must be retested.
Association testing: All association relations are tested.
Aggregation testing: Various aggregate objects are created and tested.
Sequence diagram-based testing
Method coverage: All methods depicted in the sequence diagrams are covered. Message path coverage: All message paths that can be constructed from the sequence diagrams are covered.
10.11.5 Integration Testing of Object-oriented Programs
There are two main approaches to integration testing of object-oriented programs:

- Thread-based
- Use based

Thread-based approach: In this approach, all classes that need to collaborate to realise the behaviour of a single use case are integrated and tested. After all the required classes for a use case are integrated and tested, another use case is taken up and other classes (if any) necessary for execution of the second use case to run are integrated and tested. This is continued till all use cases have been considered.

Use-based approach: Use-based integration begins by testing classes that either need no service from other classes or need services from at most a few other classes. After these classes have been integrated and tested, classes that use the services from the already integrated classes are integrated and tested. This is continued till all the classes have been integrated and tested.

Unit – V: Software quality, reliability, and other issues

1.software maintenance process models.(R13,2016)
2.write about types of software maintenance.(R13,2016s)

Software reliability:

The reliability of a software product essentially denotes its trustworthiness or dependability. Alternatively, the reliability of a software product can also be defined as the probability of the product working "correctly" over a given period of time.

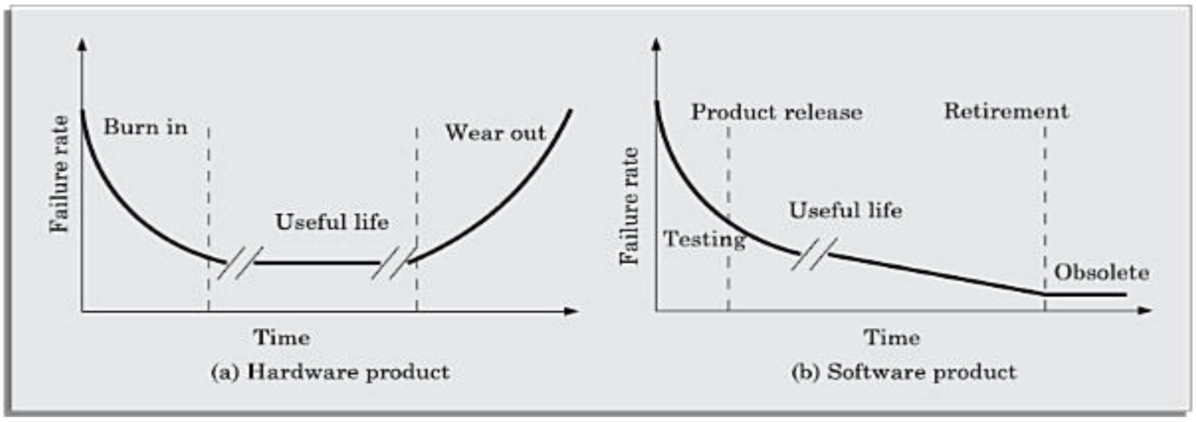
Removing errors from infrequently executed parts of software, makes little difference to the reliability of the product. It has been experimentally observed by analysing the behaviour of a large number of programs that 90 percent of the execution time of a typical program is spent in executing only 10 percent of the instructions in the program. The most used 10 percent instructions are often called the core of a program. The rest 90 percent of the program statements are called non-core. It is clear that the quantity by which the overall reliability of a program improves due to the correction of a single error depends on how frequently the instruction having the error is executed. If an error is removed from an instruction that is frequently executed (i.e., belonging to the core of the program), then this would show up as a large improvement to the reliability figure. On the other hand, removing errors from parts of the program that are rarely used, may not cause any appreciable change to the reliability of the product.

The main reasons that make software reliability more difficult to measure than hardware reliability:

- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

Hardware versus Software Reliability:

- Hardware components fail mostly due to wear and tear, whereas software components fail due to bugs.
- To fix a hardware fault, one has to either replace or repair the failed part, whereas a software product would continue to fail until the error is tracked down and either the design or the code is changed to fix the bug.
- change of reliability with time for a hardware component appears like a "bath tub". For a software component the failure rate is initially high, but decreases as the errors are corrected.



Reliability Metrics of Software Products:

we need some metrics to quantitatively express the reliability of a software product. A good reliability measure should be observer-independent, so that different people can agree on the degree of reliability a system has. However, in practice, it is very difficult to formulate a metric using which precise reliability measurement would be possible. In the absence of such measures, we discuss six metrics that correlate with reliability as follows:

Rate of occurrence of failure (ROCOF): ROCOF measures the frequency of occurrence of failures. ROCOF measure of a software product can be obtained by observing the behaviour of a software product in operation over a specified time interval and then calculating the ROCOF value as the ratio of the total number of failures observed and the duration of observation.

Mean time to failure (MTTF): MTTF is the time between two successive failures, averaged over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failures occur at the time instants $t_1, t_2, ..., t_n$. Then, MTTF can be calculated as

$$\sum_{i=1}^{n} \frac{t_{i+1}-t_i}{(n-1)}.$$

It is important to note that only run time is considered in the time measurements. That is, the time for which the system is down to fix the error, the boot time, etc. are not taken into account in the time measurements and the clock is stopped at these times.

Mean time to repair (MTTR): Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.

Mean time between failure (MTBF): The MTTF and MTTR metrics can be combined to get the MTBF metric: MTBF=MTTF+MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, the time measurements are real time and not the execution time as in MTTF

Probability of failure on demand (POFOD): This metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure. Thus, POFOD metric is very appropriate for software products that are not required to run continuously.

Availability: Availability of a system is a measure of how likely would the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs. This metric is important for systems such as telecommunication systems, and operating systems, and embedded controllers, etc. which are supposed to be never down and where repair and restart time are significant and loss of service during that time cannot be overlooked.

Shortcomings of reliability metrics of software product:

In order to estimate the reliability of a software product more accurately, it is necessary to classify various types of failures. Please note that the different classes of failures may not be mutually exclusive. A scheme of classification of failures is as follows:

Transient: Transient failures occur only for certain input values while invoking a function of the system.

Permanent: Permanent failures occur for all input values while invoking a function of the system.

Recoverable: When a recoverable failure occurs, the system can recover without having to shutdown and restart the system (with or without operator intervention).

Unrecoverable: In unrecoverable failures, the system may need to be restarted.

Cosmetic: These classes of failures cause only minor irritations, and do not lead to incorrect results. An example of a cosmetic failure is the situation where the mouse button has to be clicked twice instead of once to invoke a given function through the graphical user interface.

Reliability Growth Modelling:

A reliability growth model can be used to predict when (or if at all) a particular level of reliability is likely to be attained. Thus, reliability growth modelling can be used to determine when to stop testing to attain a given reliability level.

Although several different reliability growth models have been proposed, we will discuss only two very simple reliability growth models.

Jelinski and Moranda model:

The simplest reliability growth model is a step function model where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired. However, this simple model of reliability which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic since we already know that correction of different errors contribute differently to reliability growth.

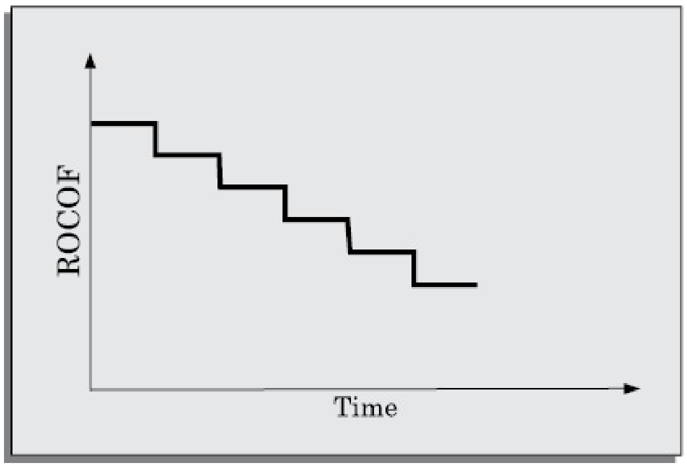


Figure 11.2: Step function model of reliability growth.

Littlewood and Veral's model:

This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It also models the fact that as errors are repaired, the average improvement to the product reliability per repair decreases. It treats an error's contribution to reliability improvement to be an independent random variable having Gamma distribution. This distribution models the fact that error corrections with large contributions to reliability growth are removed first. This represents diminishing return as test continues.

There are more complex reliability growth models, which give more accurate approximations to the reliability growth. However, these models are out of scope of this text.

Statistical testing:

Statistical testing is a testing process whose objective is to determine the reliability of the product rather than discovering errors. The test cases designed for statistical testing with an entirely different objective from those of conventional testing. To carry out statistical testing, we need to first define the operation profile of the product.

Operation profile: Different categories of users may use a software product for very different purposes. For example, a librarian might use the Library Automation Software to create member records, delete member records, add books to the library, etc., whereas a library member might use software to query about the availability of a book, and to issue and return books. Formally, we can define the operation profile of a software as the probability of a user selecting the different functionalities of the software. If we denote the set of various functionalities offered by the software by {fi}, the operational profile would

associate with each function {fi} with the probability with which an average user would select {fi} as his next function to use. Thus, we can think of the operation profile as assigning a probability value pi to each functionality fi of the software.

Defining the operation profile for a product:

We need to divide the input data into a number of input classes. For example, for a graphical editor software, we might divide the input into data associated with the edit, print, and file operations. We then need to assign a probability value to each input class; to signify the probability for an input value from that class to be selected. The operation profile of a software product can be determined by observing and analysing the usage pattern of the software by a number of users.

Steps in Statistical Testing:

The first step is to determine the operation profile of the software. The next step is to generate a set of test data corresponding to the determined operation profile. The third step is to apply the test cases to the software and record the time between each failure. After a statistically significant number of failures have been observed, the reliability can be computed.

For accurate results, statistical testing requires some fundamental assumptions to be satisfied. It requires a statistically significant number of test cases to be used. It further requires that a small percentage of test inputs that are likely to cause system failure to be included. Now let us discuss the implications of these assumptions.

It is straight forward to generate test cases for the common types of inputs, since one can easily write a test case generator program which can automatically generate these test cases. However, it is also required that a statistically significant percentage of the unlikely inputs should also be included in the test suite. Creating these unlikely inputs using a test case generator is very difficult.

Pros and cons of statistical testing:

Statistical testing allows one to concentrate on testing parts of the system that are most likely to be used. Therefore, it results in a system that the users can find to be more reliable (than actually it is!). Also, the reliability estimation arrived by using statistical testing is more accurate compared to those of other methods discussed. However, it is not easy to perform the statistical testing satisfactorily due to the following two reasons. There is no simple and repeatable way of defining operation profiles. Also, the number of test cases with which the system is to be tested should be statistically significant.

Software quality and management:

ISO 9000:

SEI capability maturity model (CMM):

CMM was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University in 1987.

- It is not a software process model. It is a framework that is used to analyze the approach and techniques followed by any organization to develop software products.
- It also provides guidelines to further enhance the maturity of the process used to develop those software products.
- It is based on profound feedback and development practices adopted by the most successful organizations worldwide.
- This model describes a strategy for software process improvement that should be followed by moving through 5 different levels.

- Each level of maturity shows a process capability level. All the levels except level-1 are further described by Key Process Areas (KPA's).

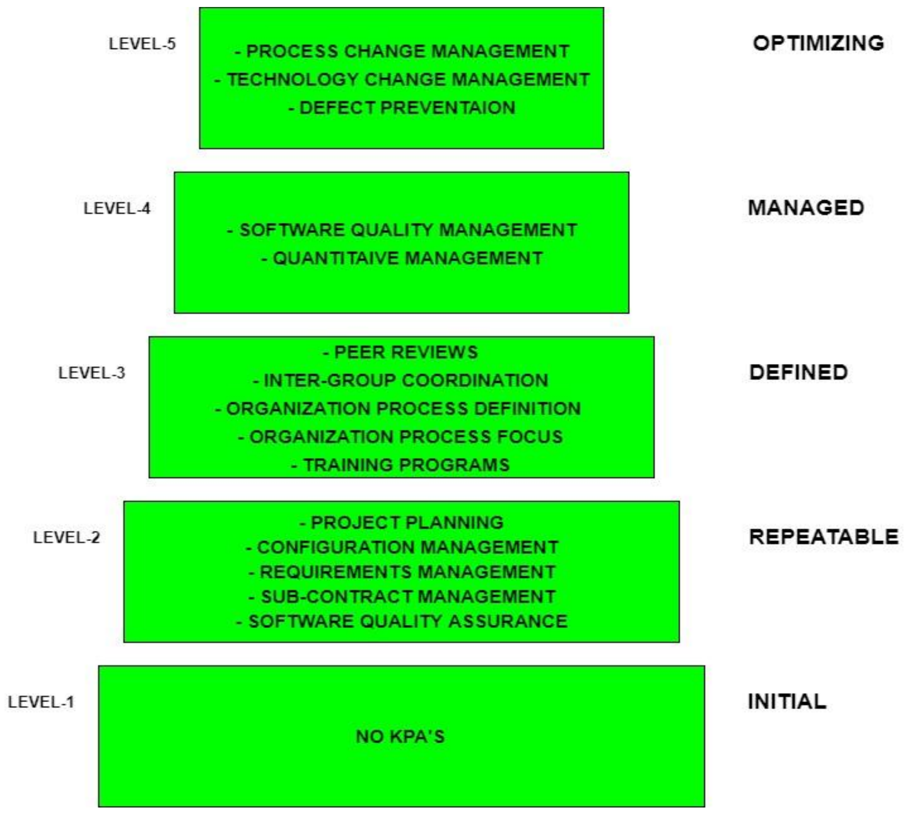
Shortcomings of SEI/CMM:

- It encourages the achievement of a higher maturity level in some cases by displacing the true mission, which is improving the process and overall software quality.
- It only helps if it is put into place early in the software development process.
- It has no formal theoretical basis and in fact is based on the experience of very knowledgeable people.
- It does not have good empirical support and this same empirical support could also be constructed to support other models.

Key Process Areas (KPA's):

Each of these KPA's defines the basic requirements that should be met by a software process in order to satisfy the KPA and achieve that level of maturity.

Conceptually, key process areas form the basis for management control of the software project and establish a context in which technical methods are applied, work products like models, documents, data, reports, etc. are produced, milestones are established, quality is ensured and change is properly managed.



LEVEL-5
- PROCESS CHANGE MANAGEMENT
- TECHNOLOGY CHANGE MANAGEMENT
- DEFECT PREVENTAION
OPTIMIZING
LEVEL-4
- SOFTWARE QUALITY MANAGEMENT
- QUANTITAIVE MANAGEMENT
MANAGED
LEVEL-3
- PEER REVIEWS
- INTER-GROUP COORDINATION
- ORGANIZATION PROCESS DEFINITION
- ORGANIZATION PROCESS FOCUS
- TRAINING PROGRAMS
DEFINED
LEVEL-2
- PROJECT PLANNING
- CONFIGURATION MANAGEMENT
- REQUIREMENTS MANAGEMENT
- SUB-CONTRACT MANAGEMENT
- SOFTWARE QUALITY ASSURANCE
REPEATABLE
LEVEL-1
NO KPA'S
INITIAL

The 5 levels of CMM are as follows:

Level-1: Initial –

- No KPA's defined.
- Processes followed are Adhoc and immature and are not well defined.
- Unstable environment for software development.
- No basis for predicting product quality, time for completion, etc.

Level-2: Repeatable –

- Focuses on establishing basic project management policies.
- Experience with earlier projects is used for managing new similar natured projects.
- Project Planning- It includes defining resources required, goals, constraints, etc. for the project. It presents a detailed plan to be followed systematically for the successful completion of good quality software.

- Configuration Management- The focus is on maintaining the performance of the software product, including all its components, for the entire lifecycle.
- Requirements Management- It includes the management of customer reviews and feedback which result in some changes in the requirement set. It also consists of accommodation of those modified requirements.
- Subcontract Management- It focuses on the effective management of qualified software contractors i.e.; it manages the parts of the software which are developed by third parties.
- Software Quality Assurance- It guarantees a good quality software product by following certain rules and quality standard guidelines while developing.

Level-3: Defined –

- At this level, documentation of the standard guidelines and procedures takes place.
- It is a well-defined integrated set of project-specific software engineering and management processes.
- Peer Reviews- In this method, defects are removed by using a number of review methods like walkthroughs, inspections, buddy checks, etc.
- Intergroup Coordination- It consists of planned interactions between different development teams to ensure efficient and proper fulfillment of customer needs.
- Organization Process Definition- Its key focus is on the development and maintenance of the standard development processes.
- Organization Process Focus- It includes activities and practices that should be followed to improve the process capabilities of an organization.
- Training Programs- It focuses on the enhancement of knowledge and skills of the team members including the developers and ensuring an increase in work efficiency.

Level-4: Managed –

- At this stage, quantitative quality goals are set for the organization for software products as well as software processes.
- The measurements made help the organization to predict the product and process quality within some limits defined quantitatively.
- Software Quality Management- It includes the establishment of plans and strategies to develop quantitative analysis and understanding of the product's quality.
- Quantitative Management- It focuses on controlling the project performance in a quantitative manner.

Level-5: Optimizing –

- This is the highest level of process maturity in CMM and focuses on continuous process improvement in the organization using quantitative feedback.
- Use of new tools, techniques, and evaluation of software processes is done to prevent recurrence of known defects.
- Process Change Management- Its focus is on the continuous improvement of the organization's software processes to improve productivity, quality, and cycle time for the software product.
- Technology Change Management- It consists of the identification and use of new technologies to improve product quality and decrease product development time.
- Defect Prevention- It focuses on the identification of causes of defects and prevents them from recurring in future projects by improving project-defined processes.

Personal software process (PSP):

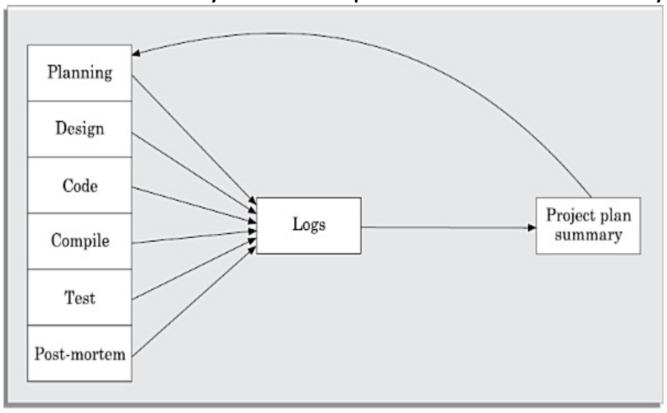
The SEI CMM which is reference model for raising the maturity levels of software and predicts the most expected outcome from the next project undertaken by the organizations does not tell software developers about how to analyse, design, code, test and document the software products, but expects that the developers use effectual practices. The Personal Software Process realized that the process of individual use is completely different from that required by the team.

Personal Software Process (PSP) is the skeleton or the structure that assist the engineers in finding a way to measure and improve the way of working to a great extent. It helps them in developing their respective skills at a personal level and the way of doing planning, estimations against the plans.

Objectives of PSP :
The aim of PSP is to give software engineers with the regulated methods for the betterment of personal software development processes.
The PSP helps software engineers to:

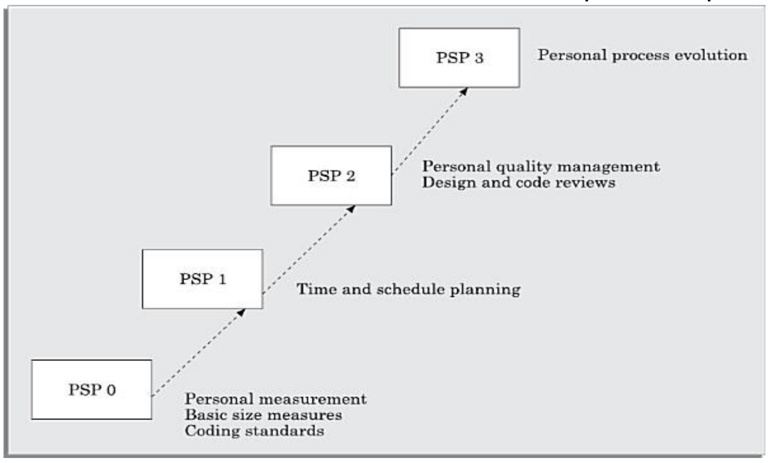
- Improve their approximating and planning skills.
- Make promises that can be fulfilled.
- Manage the standards of their projects.
- Reduce the number of faults and imperfections in their work.

Time measurement:
Personal Software Process recommend that the developers should structure the way to spend the time. The developer must measure and count the time they spend on different activities during the development.
PSP Planning :
The engineers should plan the project before developing because without planning a high effort may be wasted on unimportant activities which may lead to a poor and unsatisfactory quality of the result.



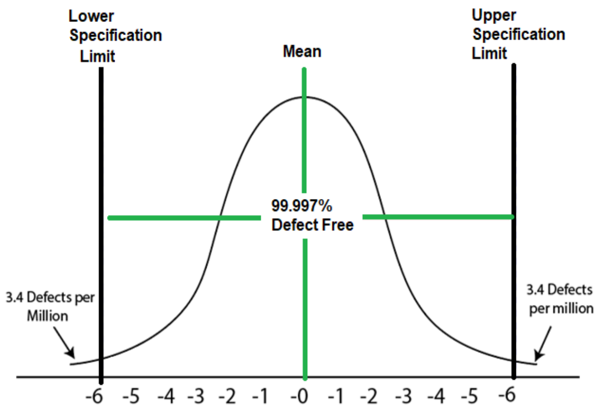
Levels of Personal Software Process :
Personal Software Process (PSP) has four levels-

1. **PSP 0 –**
   The first level of Personal Software Process, PSP 0 includes Personal measurement , basic size measures, coding standards.
2. **PSP 1 –**
   This level includes the planning of time and scheduling .
3. **PSP 2 –**
   This level introduces the personal quality management ,design and code reviews.
4. **PSP 3 –**
   The last level of the Personal Software Process is for the personal process evolution.



Six sigma:

Six Sigma is the process of producing high and improved quality output. This can be done in two phases – identification and elimination. The cause of defects is identified and appropriate elimination is done which reduces variation in whole processes. A six-sigma method is one in which 99.99966% of all the products to be produced have the same features and are of free from defects.

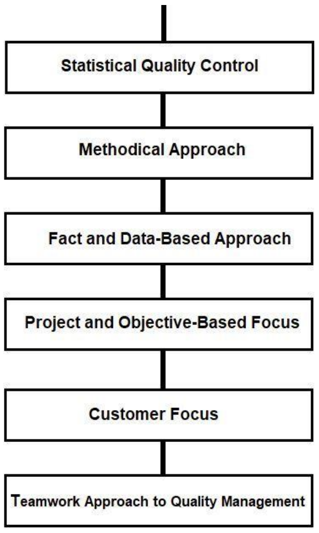


Six Sigma Curve

Characteristics of Six Sigma:

The Characteristics of Six Sigma are as follows:

1. **Statistical Quality Control:**
   Six Sigma is derived from the Greek Letter ? which denote Standard Deviation in statistics. Standard Deviation is used for measuring the quality of output.
2. **Methodical Approach:**
   The Six Sigma is a systematic approach of application in DMAIC and DMADV which can be used to improve the quality of production. DMAIC means for Design-Measure- Analyse-Improve-Control. While DMADV stands for Design-Measure-Analyse-Design-Verify.
3. **Fact and Data-Based Approach:**
   The statistical and methodical method shows the scientific basis of the technique.


Statistical Quality Control
Methodical Approach
Fact and Data-Based Approach
Project and Objective-Based Focus
Customer Focus
Teamwork Approach to Quality Management

Characteristics of Six Sigma

4. **Project and Objective-Based Focus:**
   The Six Sigma process is implemented to focus on the requirements and conditions.
5. **Customer Focus:**
   The customer focus is fundamental to the Six Sigma approach. The quality improvement and control standards are based on specific customer requirements.
6. **Teamwork Approach to Quality Management:**
   The Six Sigma process requires organizations to get organized for improving quality.

Six Sigma Methodologies:
Two methodologies used in the Six Sigma projects are DMAIC and DMADV.

- **DMAIC** is used to enhance an existing business process. The DMAIC project methodology has five phases:
    1. Define
    2. Measure
    3. Analyse
    4. Improve
    5. Control
- **DMADV** is used to create new product designs or process designs. The DMADV project methodology also has five phases:
    1. Define
    2. Measure
    3. Analyse
    4. Design
    5. Verify

Software quality metrics: Software quality metrics are a subset of software metrics that focus on the quality aspects of the product, process, and project. These are more closely associated with process and product metrics than with project metrics.
Software quality metrics can be further divided into three categories −

1. Product quality metrics
2. In-process quality metrics
3. Maintenance quality metrics

1.Product Quality Metrics
This metrics include the following −

- Mean Time to Failure
- Defect Density
- Customer Problems
- Customer Satisfaction

Mean Time to Failure
It is the time between failures. This metric is mostly used with safety critical systems such as the airline traffic control systems, avionics, and weapons.
Defect Density
It measures the defects relative to the software size expressed as lines of code or function point, etc. i.e., it measures code quality per unit. This metric is used in many commercial software systems.
Customer Problems
It measures the problems that customers encounter when using the product. It contains the customer's perspective towards the problem space of the software, which includes the non-defect oriented problems together with the defect problems.
The problems metric is usually expressed in terms of **Problems per User-Month (PUM)**.
PUM = Total Problems that customers reported (true defect and non-defect oriented
problems) for a time period + Total number of license months of the software during

the period
Where,
Number of license-month of the software = Number of install license of the software ×
Number of months in the calculation period
PUM is usually calculated for each month after the software is released to the market, and also for monthly averages by year.

Customer Satisfaction

Customer satisfaction is often measured by customer survey data through the five-point scale −

- Very satisfied
- Satisfied
- Neutral
- Dissatisfied
- Very dissatisfied

Satisfaction with the overall quality of the product and its specific dimensions is usually obtained through various methods of customer surveys. Based on the five-point-scale data, several metrics with slight variations can be constructed and used, depending on the purpose of analysis. For example −
Percent of completely satisfied customers
Percent of satisfied customers
Percent of dis-satisfied customers
Percent of non-satisfied customers
Usually, this percent satisfaction is used.

2.In-process Quality Metrics

In-process quality metrics deals with the tracking of defect arrival during formal machine testing for some organizations. This metric includes −

- Defect density during machine testing
- Defect arrival pattern during machine testing
- Phase-based defect removal pattern
- Defect removal effectiveness

Defect density during machine testing

Defect rate during formal machine testing (testing after code is integrated into the system library) is correlated with the defect rate in the field. Higher defect rates found during testing is an indicator that the software has experienced higher error injection during its development process, unless the higher testing defect rate is due to an extraordinary testing effort.
This simple metric of defects per KLOC or function point is a good indicator of quality, while the software is still being tested. It is especially useful to monitor subsequent releases of a product in the same development organization.

Defect arrival pattern during machine testing

The overall defect density during testing will provide only the summary of the defects. The pattern of defect arrivals gives more information about different quality levels in the field. It includes the following −
The defect arrivals or defects reported during the testing phase by time interval (e.g., week). Here all of which will not be valid defects.
The pattern of valid defect arrivals when problem determination is done on the reported problems. This is the true defect pattern.
The pattern of defect backlog overtime. This metric is needed because development organizations cannot investigate and fix all the reported problems immediately. This is a workload statement as well as a quality statement. If the defect backlog is large at the end of the development cycle and a lot of fixes have yet to be integrated into the system, the stability of the system (hence its quality) will be affected. Retesting (regression test) is needed to ensure that targeted product quality levels are reached.

Phase-based defect removal pattern
This is an extension of the defect density metric during testing. In addition to testing, it tracks the defects at all phases of the development cycle, including the design reviews, code inspections, and formal verifications before testing.
Because a large percentage of programming defects is related to design problems, conducting formal reviews, or functional verifications to enhance the defect removal capability of the process at the front-end reduces error in the software. The pattern of phase-based defect removal reflects the overall defect removal ability of the development process.
With regard to the metrics for the design and coding phases, in addition to defect rates, many development organizations use metrics such as inspection coverage and inspection effort for in-process quality management.
Defect removal effectiveness
It can be defined as follows −
DRE=DefectremovedduringadevelopmentphaseDefectslatentintheproduct×100%DRE=DefectremovedduringadevelopmentphaseDefectslatentintheproduct×100%
This metric can be calculated for the entire development process, for the front-end before code integration and for each phase. It is called **early defect removal** when used for the front-end and **phase effectiveness** for specific phases. The higher the value of the metric, the more effective the development process and the fewer the defects passed to the next phase or to the field. This metric is a key concept of the defect removal model for software development.
3.Maintenance Quality Metrics
Although much cannot be done to alter the quality of the product during this phase, following are the fixes that can be carried out to eliminate the defects as soon as possible with excellent fix quality.

- Fix backlog and backlog management index
- Fix response time and fix responsiveness
- Percent delinquent fixes
- Fix quality

Fix backlog and backlog management index
Fix backlog is related to the rate of defect arrivals and the rate at which fixes for reported problems become available. It is a simple count of reported problems that remain at the end of each month or each week. Using it in the format of a trend chart, this metric can provide meaningful information for managing the maintenance process.
Backlog Management Index (BMI) is used to manage the backlog of open and unresolved problems.
BMI=NumberofproblemsclosedduringthemonthNumberofproblemsarrivedduringthemonth×100%BMI=NumberofproblemsclosedduringthemonthNumberofproblemsarrivedduringthemonth×100%
If BMI is larger than 100, it means the backlog is reduced. If BMI is less than 100, then the backlog increased.
Fix response time and fix responsiveness
The fix response time metric is usually calculated as the mean time of all problems from open to close. Short fix response time leads to customer satisfaction.
The important elements of fix responsiveness are customer expectations, the agreed-to fix time, and the ability to meet one's commitment to the customer.
Percent delinquent fixes
It is calculated as follows −

$$Percent\ Delinquent\ Fixes =$$

$$\frac{Number\ of\ fixes\ that\ exceeded\ the\ response\ time\ criteria\ by\ ceverity\ level}{Number\ of\ fixes\ delivered\ in\ a\ specified\ time} \times 100\%$$

Fix Quality
Fix quality or the number of defective fixes is another important quality metric for the maintenance phase. A fix is defective if it did not fix the reported problem, or if it fixed the original problem but injected a new defect. For mission-critical software, defective fixes are detrimental to customer satisfaction. The metric of percent defective fixes is the percentage of all fixes in a time interval that is defective.
A defective fix can be recorded in two ways: Record it in the month it was discovered or record it in the month the fix was delivered. The first is a customer measure; the second is a process measure. The difference between the two dates is the latent period of the defective fix.
Usually the longer the latency, the more will be the customers that get affected. If the number of defects is large, then the small value of the percentage metric will show an optimistic picture. The quality goal for the maintenance process, of course, is zero defective fixes without delinquency.

CASE and its scope: CASE stands for **C**omputer **A**ided **S**oftware **E**ngineering. It means, development and maintenance of software projects with help of various automated software tools.
CASE environment:
CASE support in software life cycle:
Characteristics of software maintenance:
Software reverse engineering:
Software maintenance processes model:
Estimation maintenance cost:
Basic issues in any reuse program:
Reuse approach:
Reuse at organization level: