Software Engineering – UNIT I

1.1 INTRODUCTION

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

- Software engineering is an engineering discipline which is concerned with all aspects of software production
- Software engineers should adopt a systematic and organised approach to their work and use appropriate tools and techniques depending on the problem to be solved, the development constraints and the resources available **Software Engineering Definition** The seminal definition:

[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines. The IEEE definition:

Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

Importance of Software Engineering

- More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.
- It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of system, the majority of costs are the costs of changing the software after it has gone into use.

FAQ about software engineering

Question	Answer
What is software?	Computer programs, data structures and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.

What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.

1.2 Notion of Software as a Product

What is software?

The product that software professionals build and then support over the long term.
Software encompasses:

1.Instructions (computer programs) that when executed provide desired features, function, and performance;

2.Data structures that enable the programs to adequately store and manipulate information and

3.Documentation that describes the operation and use of the programs.

Software products:-

- Generic products

Stand-alone systems that are marketed and sold to any customer who wishes to buy them.

Examples – PC software such as editing, graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.

- Customized products

Software that is commissioned by a specific customer to meet their own needs.

Examples – embedded control systems, air traffic control software, traffic monitoring systems.

Why Software is Important?

- The economies of ALL developed nations are dependent on software.
- More and more systems are software controlled (transportation, medical, telecommunications, military, industrial, entertainment,)
- Software engineering is concerned with theories, methods and tools for professional software development. **Software costs**
- Software costs often dominate computer system costs. The costs of software on a PC are often greater than the hardware cost.

- Software costs **more to maintain** than it does to develop. For systems with a long life, maintenance costs may be several times development costs.
- Software engineering is concerned with cost-effective software development.

Features of Software

- Its characteristics that make it different from other things human being build.
- Software is developed or engineered, it is not manufactured in the classical sense which has quality problem.
- Software doesn't "wear out." but it deteriorates (due to change). Hardware has bathtub curve of failure rate (high failure rate in the beginning, then drop to steady state, then cumulative effects of dust, vibration, abuse occurs).
- Although the industry is moving toward component-based construction (e.g. standard screws and off-the-shelf integrated circuits), most software continues to be custom-built. Modern reusable components encapsulate data and processing into software parts to be reused by different programs. E.g. graphical user interface, window, pull-down menus in library etc.

Software Applications

1. System software: such as compilers, editors, file management utilities
2. Application software: stand-alone programs for specific needs.
3. Engineering/scientific software: Characterized by "number crunching"algorithms. such as automotive stress analysis, molecular biology, orbital dynamics etc
4. Embedded software resides within a product or system. (key pad control of a microwave oven, digital function of dashboard display in a car)
5. Product-line software focus on a limited marketplace to address mass consumer market. (word processing, graphics, database management)
6. WebApps (Web applications) network centric software. As web 2.0 emerges, more sophisticated computing environments is supported integrated with remote database and business applications.
7. AI software uses non-numerical algorithm to solve complex problem. Robotics, expert system, pattern recognition game playing

Software—New Categories

- Open world computing—pervasive, ubiquitous, distributed computing due to wireless networking. How to allow mobile devices, personal computer, enterprise system to communicate across vast network.
- Netsourcing—the Web as a computing engine. How to architect simple and sophisticated applications to target end-users worldwide.
- Open source—"free" source code open to the computing community

1.3 Characteristics of a good Software Product

Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilisation, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable and compatible with other systems that they use.

1.4 Engineering aspects of Software production

SOFTWARE ENGINEERING - A Layered Technology

- ✓ Software engineering is a layered technology.

- ✓ Any engineering approach must rest on organizational commitment to **quality** which fosters a continuous process improvement culture.
- ✓ **Process** layer as the foundation defines a framework with activities for effective delivery of software engineering technology. Establish the context where products (model, data, report, and forms) are produced, milestone are established, quality is ensured and change is managed.
- ✓ **Method** provides technical how-to's for building software. It encompasses many tasks including communication, requirement analysis, design modeling, program construction, testing and support.
- ✓ **Tools** provide automated or semi-automated support for the process and methods.

- ✓ When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.

Software Process

- A process is a collection of activities, actions and tasks that are performed when some work product is to be created. It is not a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work to pick and choose the appropriate set of work actions and tasks.
- Purpose of process is to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

Five Activities of a Generic Process framework

- Communication: communicate with customer to understand objectives and gather requirements
- Planning: creates a "map" defines the work by describing the tasks, risks and resources, work products and work schedule.
- Modeling: Create a "sketch", what it looks like architecturally, how the constituent parts fit together and other characteristics.
- Construction: code generation and the testing.
- Deployment: Delivered to the customer who evaluates the products and provides feedback based on the evaluation.
- These five framework activities can be used to all software development regardless of the application domain, size of the project, complexity of the efforts etc, though the details will be different in each case.
- For many software projects, these framework activities are applied iteratively as a project progresses. Each iteration produces a software increment that provides a subset of overall software features and functionality.

Umbrella Activities

- Complement the five process framework activities and help team manage and control progress, quality, change, and risk.
- Software project tracking and control: assess progress against the plan and take actions to maintain the schedule.
- Risk management: assesses risks that may affect the outcome and quality.
- Software quality assurance: defines and conduct activities to ensure quality.
- Technical reviews: assesses work products to uncover and remove errors before going to the next activity.
- Measurement: define and collects process, project, and product measures to ensure stakeholder's needs are met.
- Software configuration management: manage the effects of change throughout the software process.
- Reusability management: defines criteria for work product reuse and establishes mechanism to achieve reusable components.
- Work product preparation and production: create work products such as models, documents, logs, forms and lists.

Adapting a process model:

The process should be agile and adaptable to problems. Process adopted for one project might be significantly different than a process adopted from another project. (to the problem, the project, the team, organizational culture). Among the differences are:

- the overall flow of activities, actions, and tasks and the interdependencies among them
- the degree to which actions and tasks are defined within each framework activity

- the degree to which work products are identified and required
- the manner which quality assurance activities are applied
- the manner in which project tracking and control activities are applied
- the overall degree of detail and rigor with which the process is described
- the degree to which the customer and other stakeholders are involved with the project
- the level of autonomy given to the software team
- the degree to which team organization and roles are prescribed

Prescriptive and Agile Process Models

- The prescriptive process models stress detailed definition, identification, and application of process activates and tasks. Intent is to improve system quality, make projects more manageable, make delivery dates and costs more predictable, and guide teams of software engineers as they perform the work required to build a system.
- Unfortunately, there have been times when these objectives were not achieved. If prescriptive models are applied dogmatically and without adaptation, they can increase the level of bureaucracy.
- Agile process models emphasize project "agility" and follow a set of principles that lead to a more informal approach to software process. It emphasizes maneuverability and adaptability. It is particularly useful when Web applications are engineered.

George Polya outlines the essence of problem solving, suggests:

1.Understand the problem (communication and analysis).
2.Plan a solution (modeling and software design).
3.Carry out the plan (code generation).
4.Examine the result for accuracy (testing and quality assurance).

1.5 Necessity of automation

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 Tools

CASE tools are set of software application programs, which are used to automate SDLC activities. CASE tools are used by software project managers, analysts and engineers to develop software system.

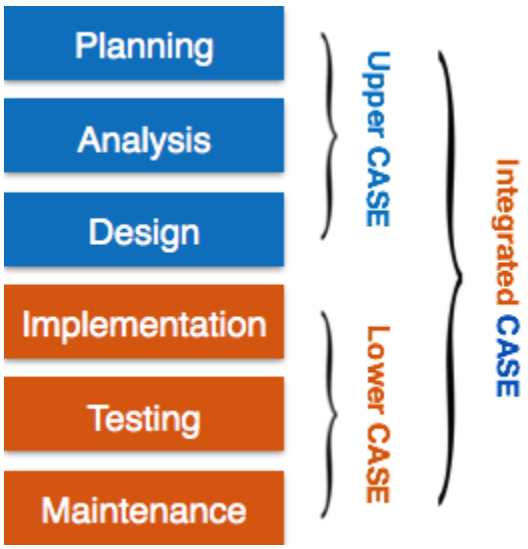
There are number of CASE tools available to simplify various stages of Software Development Life Cycle such as Analysis tools, Design tools, Project management tools, Database Management tools, Documentation tools are to name a few.

Use of CASE tools accelerates the development of project to produce desired result and helps to uncover flaws before moving ahead with next stage in software development.

Components of CASE Tools

CASE tools can be broadly divided into the following parts based on their use at a particular SDLC stage:

- **Central Repository** - CASE tools require a central repository, which can serve as a source of common, integrated and consistent information. Central repository is a central place of storage where product specifications, requirement documents, related reports and diagrams, other useful information regarding management is stored. Central repository also serves as data dictionary.



- **Upper Case Tools** - Upper CASE tools are used in planning, analysis and design stages of SDLC.
- **Lower Case Tools** - Lower CASE tools are used in implementation, testing and maintenance.
- **Integrated Case Tools** - Integrated CASE tools are helpful in all the stages of SDLC, from Requirement gathering to Testing and documentation.

CASE tools can be grouped together if they have similar functionality, process activities and capability of getting integrated with other tools.

Scope of Case Tools

The scope of CASE tools goes throughout the SDLC.

Case Tools Types

Now we briefly go through various CASE tools

Diagram tools

These tools are used to represent system components, data and control flow among various software components and system structure in a graphical form. For example, Flow Chart Maker tool for creating state-of-the-art flowcharts.

Process Modeling Tools

Process modeling is method to create software process model, which is used to develop the software. Process modeling tools help the managers to choose a process model or modify it as per the requirement of software product. For example, EPF Composer

Project Management Tools

These tools are used for project planning, cost and effort estimation, project scheduling and resource planning. Managers have to strictly comply project execution with every mentioned step in software project management. Project management tools help in storing and sharing project information in real-time throughout the organization. For example, Creative Pro Office, Trac Project, Basecamp.

Documentation Tools

Documentation in a software project starts prior to the software process, goes throughout all phases of SDLC and after the completion of the project.

Documentation tools generate documents for technical users and end users. Technical users are mostly in-house professionals of the development team who refer to system manual, reference manual, training manual, installation manuals etc. The end user documents describe the functioning and how-to of the system such as user manual. For example, Doxygen, DrExplain, Adobe RoboHelp for documentation.

Analysis Tools

These tools help to gather requirements, automatically check for any inconsistency, inaccuracy in the diagrams, data redundancies or erroneous omissions. For example, Accept 360, Accompa, CaseComplete for requirement analysis, Visible Analyst for total analysis.

Design Tools

These tools help software designers to design the block structure of the software, which may further be broken down in smaller modules using refinement techniques. These tools provides detailing of each module and interconnections among modules. For example, Animated Software Design

Configuration Management Tools

An instance of software is released under one version. Configuration Management tools deal with –

- Version and revision management

- Baseline configuration management
- Change control management

CASE tools help in this by automatic tracking, version management and release management. For example, Fossil, Git, Accu REV.

Change Control Tools

These tools are considered as a part of configuration management tools. They deal with changes made to the software after its baseline is fixed or when the software is first released. CASE tools automate change tracking, file management, code management and more. It also helps in enforcing change policy of the organization.

Programming Tools

These tools consist of programming environments like IDE (Integrated Development Environment), in-built modules library and simulation tools. These tools provide comprehensive aid in building software product and include features for simulation and testing. For example, Cscope to search code in C, Eclipse.

Prototyping Tools

Software prototype is simulated version of the intended software product. Prototype provides initial look and feel of the product and simulates few aspect of actual product.

Prototyping CASE tools essentially come with graphical libraries. They can create hardware independent user interfaces and design. These tools help us to build rapid prototypes based on existing information. In addition, they provide simulation of software prototype. For example, Serena prototype composer, Mockup Builder.

Web Development Tools

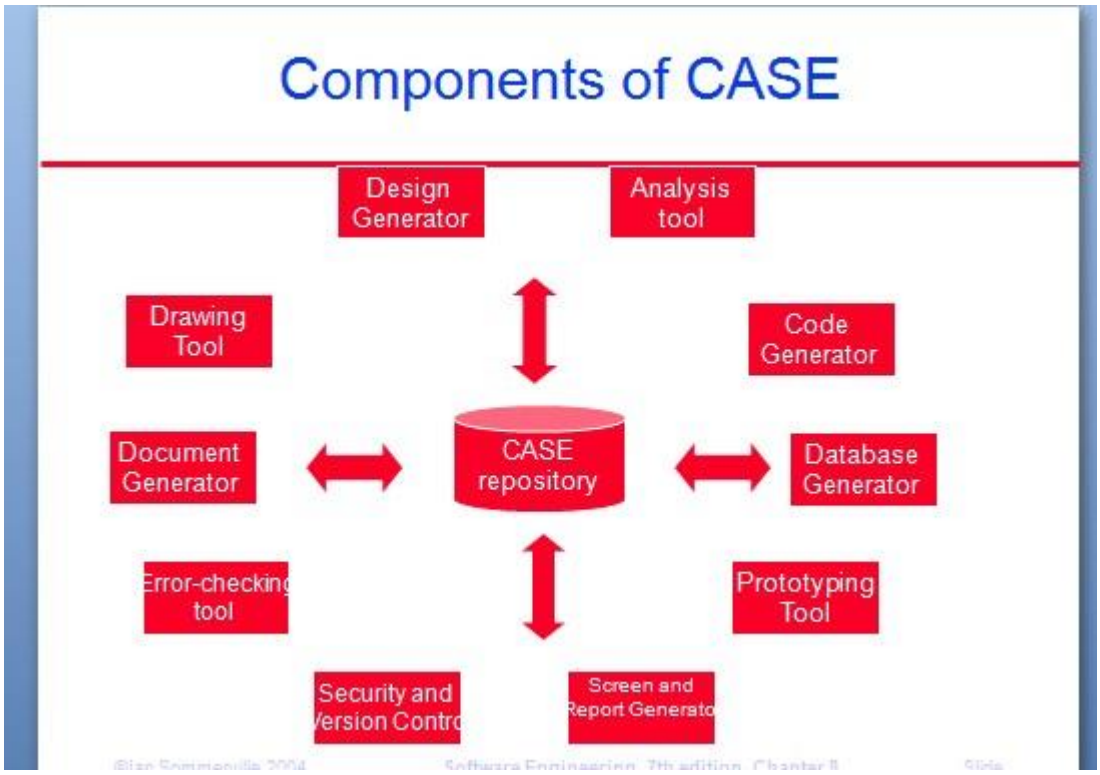
These tools assist in designing web pages with all allied elements like forms, text, script, graphic and so on. Web tools also provide live preview of what is being developed and how will it look after completion. For example, Fontello, Adobe Edge Inspect, Foundation 3, Brackets.

Quality Assurance Tools

Quality assurance in a software organization is monitoring the engineering process and methods adopted to develop the software product in order to ensure conformance of quality as per organization standards. QA tools consist of configuration and change control tools and software testing tools. For example, SoapTest, AppsWatch, JMeter.

Maintenance Tools

Software maintenance includes modifications in the software product after it is delivered. Automatic logging and error reporting techniques, automatic error ticket generation and root cause Analysis are few CASE tools, which help software organization in maintenance phase of SDLC. For example, Bugzilla for defect tracking, HP Quality Center.



Components of CASE

CASE repository

- Central component of any CASE tool
- Also known as the information repository or data dictionary

Components of CASE

- CASE repository
  - Centralized database
  - Allows easy sharing of information between tools and SDLC activities
  - Used to store graphical diagrams and prototype forms and reports during analysis and design workflows
  - Provides wealth of information to project manager and allows control over project
  - Facilitates reusability

Components of CASE

- CASE repository acts as:
  - Information repository
    - Combines information about organization's business information and application portfolio
    - Provides automated tools to manage and control access
  - Data dictionary
    - Used to manage and control access to information repository
    - Facilities for recording, storing and processing resources
    - Useful for cross-referencing

Components of CASE

- Analysis tools
  - Generate reports that help identify possible inconsistencies, redundancies and omissions

1.6 Job Responsibities of Software Engineers As Software Developers

- A passion for solving problems and providing workable solutions
- Knowledge of algorithms and data structures
- Strong analytical and reasoning skills with an ability to visualise processes and outcomes
- Proficiency in troubleshooting software issues and debugging a large codebase
- Outstanding all-round communication skills and ability to work collaboratively
- Researching, designing, implementing and managing software programs
- Testing and evaluating new programs
- Identifying areas for modification in existing programs and subsequently developing these modifications
- Writing and implementing efficient code
- Determining operational practicality
- Developing quality assurance procedures
- Deploying software tools, processes and metrics
- Maintaining and upgrading existing systems
- Training users
- Working closely with other developers, UX designers, business and systems analysts

Human Factors

Proponents of agile software development take great pains to emphasize the importance of "people factors." As Cockburn and Highsmith [Coc01al state, "Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that the process molds to the needs of the people and team, not the other way around.

2**.**If members of the software team are to drive the characteristics of the process that is applied to build software, a number of key traits must exist among the people on an agile team and the team itself:

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

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

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

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

Fuzzy problem-solving ability. Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change. In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs

to be solved tomorrow. However, lessons learned from any problem-solving Activity(Including those that solve the wrong Problem)May be or benefit to the team later in the project.

Mutual trust and respect. The agile team must become what DeMarco and Lister [DeM981 call a "jelled" team (Chapter 24). A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of the parts." [DeM981

Self-organization. In the context of agile development, self-organization implies three things: (I) the agile team organizes itself for the work to be done, (2) the team organizes the process to best accommodate its local environment, (3) the team organizes the work schedule to best achieve delivery of the software increment. Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale. In essence, the team serves as its own management. Ken Schwaber [Sch021 addresses these issues when he writes: "The team selects how much work it believes it can perform within the iteration, and the team commits to the work. Nothing demotivates a team as much as someone else making commitments for it. Nothing motivates a team as much as accepting the responsibility for fulfilling commitments that it made itself."

Software Engineering

Unit II

2.1 Software Process:

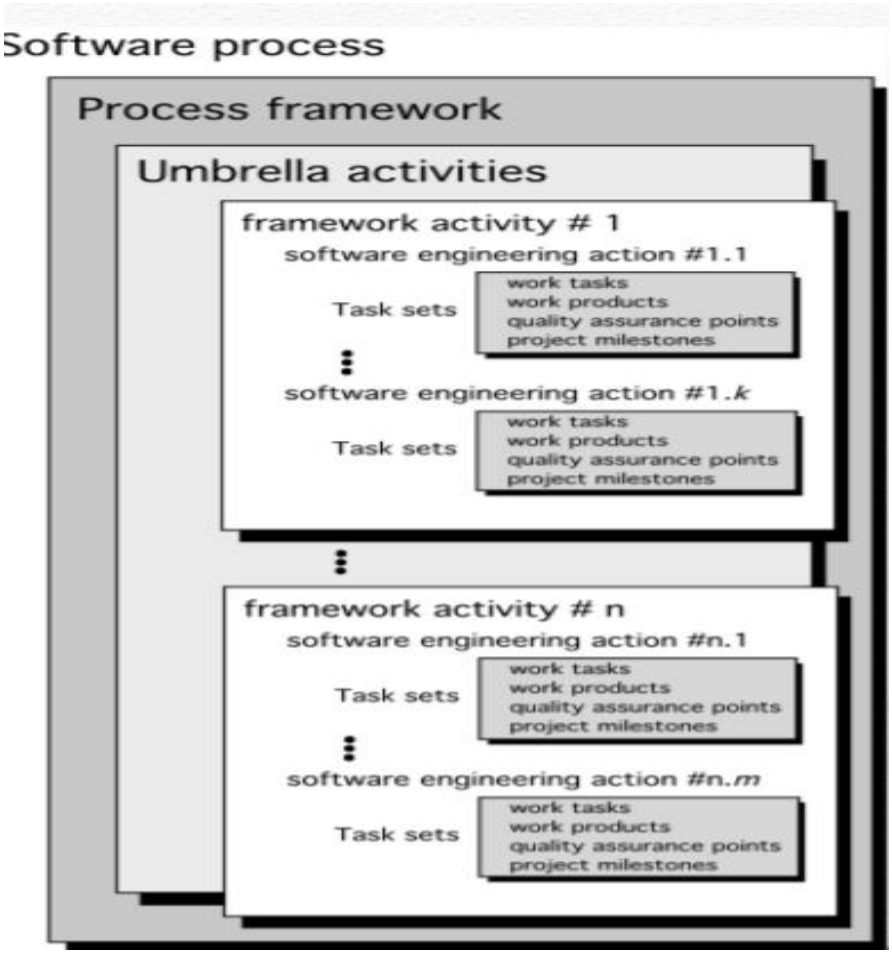
Definition of Software Process

- A framework for the activities, actions, and tasks that are required to build high-quality software.
- SP defines the approach that is taken as software is engineered.
- Is not equal to software engineering, which also encompasses technologies that populate the process– technical methods and automated tools.

A Generic Process Framework

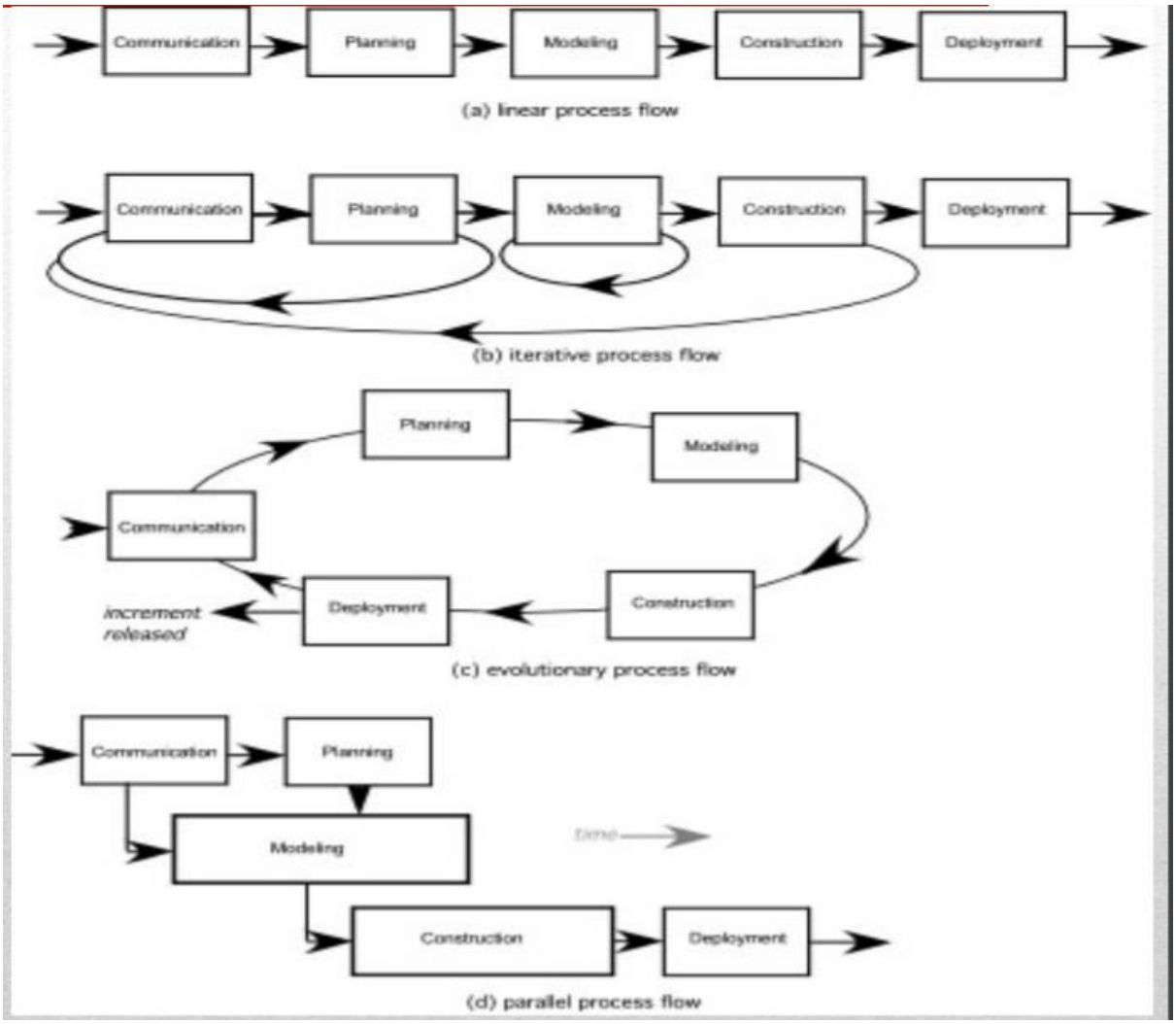
- A generic process framework for software engineering defines five framework activities - communication, planning, modeling, construction, and deployment.
- In addition, a set of umbrella activities- project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others are applied throughout the process.

A Generic Process Model



Process Flow:

- Linear process flow executes each of the five activities in sequence.
- An iterative process flow repeats one or more of the activities before proceeding to the next.
- An evolutionary process flow executes the activities in a circular manner. Each circuit leads to a more complete version of the software.
- A parallel process flow executes one or more activities in parallel with other activities (modeling for one aspect of the software in parallel with construction of another aspect of the software.)



Identifying a Task Set:

- A task set defines the actual work to be done to accomplish the objectives of a software engineering action.
- A list of the task to be accomplished
- A list of the work products to be produced
- A list of the quality assurance filters to be applied

Identifying Task set for communication activity

For example, a small software project requested by one person with simple requirements, the communication activity might encompass little more than a phone call with the stakeholder. Therefore, the only necessary action is phone conversation, the work tasks of this action are:

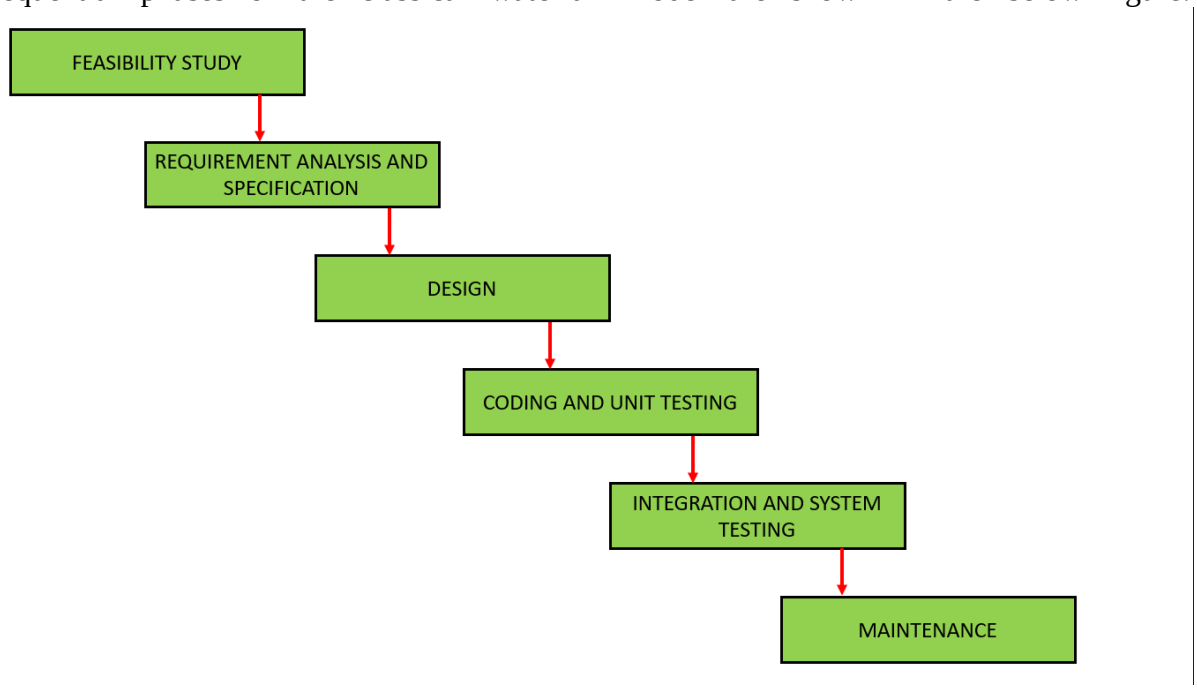
1. Make contact with stakeholder via telephone.
2. Discuss requirements and take notes.
3. Organize notes into a brief written statement of requirements.
4. E-mail to stakeholder for review and approval.

2.2 Process Models

2.2.1 Waterfall Model or Linear Sequential Model:

Classical waterfall model is the basic **software development life cycle** model. It is very simple but idealistic. Earlier this model was very popular but nowadays it is not used. But it is very important because all the other software development life cycle models are based on the classical waterfall model.

Classical waterfall model divides the life cycle into a set of phases. This model considers that one phase can be started after completion of the previous phase. That is the output of one phase will be the input to the next phase. Thus the development process can be considered as a sequential flow in the waterfall. Here the phases do not overlap with each other. The different sequential phases of the classical waterfall model are shown in the below figure:



Let us now learn about each of these phases in brief details:

1. **Feasibility Study**: The main goal of this phase is to determine whether it would be financially and technically feasible to develop the software. The feasibility study involves understanding the problem and then determine the various possible strategies to solve the problem. These different identified

solutions are analyzed based on their benefits and drawbacks, The best solution is chosen and all the other phases are carried out as per this solution strategy.

2. **Requirements analysis and specification**: The aim of the requirement analysis and specification phase is to understand the exact requirements of the customer and document them properly. This phase consists of two different activities.
    - **Requirement gathering and analysis:** Firstly all the requirements regarding the software are gathered from the customer and then the gathered requirements are analyzed. The goal of the analysis part is to remove incompleteness (an incomplete requirement is one in which some parts of the actual requirements have been omitted) and inconsistencies (inconsistent requirement is one in which some part of the requirement contradicts with some other part).
    - **Requirement specification:** These analyzed requirements are documented in a software requirement specification (SRS) document. SRS document serves as a contract between development team and customers. Any future dispute between the customers and the developers can be settled by examining the SRS document.
3. **Design**: The aim of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language.
4. **Coding and Unit testing**: In coding phase software design is translated into source code using any suitable programming language. Thus each designed module is coded. The aim of the unit testing phase is to check whether each module is working properly or not.
5. **Integration and System testing**: Integration of different modules are undertaken soon after they have been coded and unit tested. Integration of various modules is carried out incrementally over a number of steps. During each integration step, previously planned modules are added to the partially integrated system and the resultant system is tested. Finally, after all the modules have been successfully integrated and tested, the full working system is obtained and system testing is carried out on this.

    System testing consists three different kinds of testing activities as described below :

    - **Alpha testing:** Alpha testing is the system testing performed by the development team.
    - **Beta testing:** Beta testing is the system testing performed by a friendly set of customers.
    - **Acceptance testing:** After the software has been delivered, the customer performed the acceptance testing to determine whether to accept the delivered software or to reject it.
6. **Maintainence:** Maintenance is the most important phase of a software life cycle. The effort spent on maintenance is the 60% of the total effort spent to develop a full software. There are basically three types of maintenance :
    - **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 enhance the functionalities of the system based on the customer's request.

- **Adaptive Maintenance:** Adaptive maintenance is usually required for porting the software to work in a new environment such as work on a new computer platform or with a new operating system.

Advantages of Classical Waterfall Model

Classical waterfall model is an idealistic model for software development. It is very simple, so it can be considered as the basis for other software development life cycle models. Below are some of the major advantages of this SDLC model:

- This model is very simple and is easy to understand.
- Phases in this model are processed one at a time.
- Each stage in the model is clearly defined.
- This model has very clear and well undestood milestones.
- Process, actions and results are very well documented.
- Reinforces good habits: define-before- design, design-before-code.
- This model works well for smaller projects and projects where requirements are well understood.

Drawbacks of Classical Waterfall Model

Classical waterfall model suffers from various shortcomings, basically we can't use it in real projects, but we use other software development lifecycle models which are based on the classical waterfall model. Below are some major drawbacks of this model:

- **No feedback path:** In classical waterfall model evolution of a software from one phase to another phase is like a waterfall. It assumes that no error is ever committed by developers during any phases. Therefore, it does not incorporate any mechanism for error correction.
- **Difficult to accommodate change requests:** This model assumes that all the customer requirements can be completely and correctly defined at the beginning of the project, but actually customers' requirements keep on changing with time. It is difficult to accommodate any change requests after the requirements specification phase is complete.
- **No overlapping of phases:** This model recommends that new phase can start only after the completion of the previous phase. But in real projects, this can't be maintained. To increase the efficiency and reduce the cost, phases may overlap.

2.2.2 Incremental Model

There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.

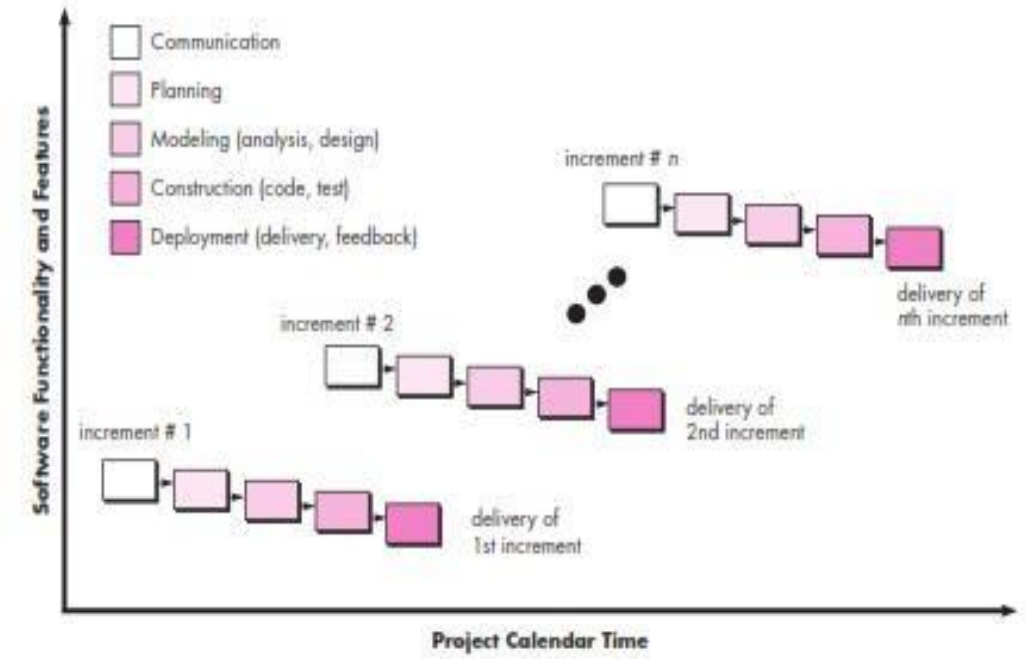
The *incremental* model combines elements of linear and parallel process flows discussed in Section 2.1. The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software [McD93] in a manner that is similar to the increments produced by an evolutionary process flow.

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

When an incremental model is used, the first increment is often a *core product.* That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/or evaluation, a

URE 2.5

mental



plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced. The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

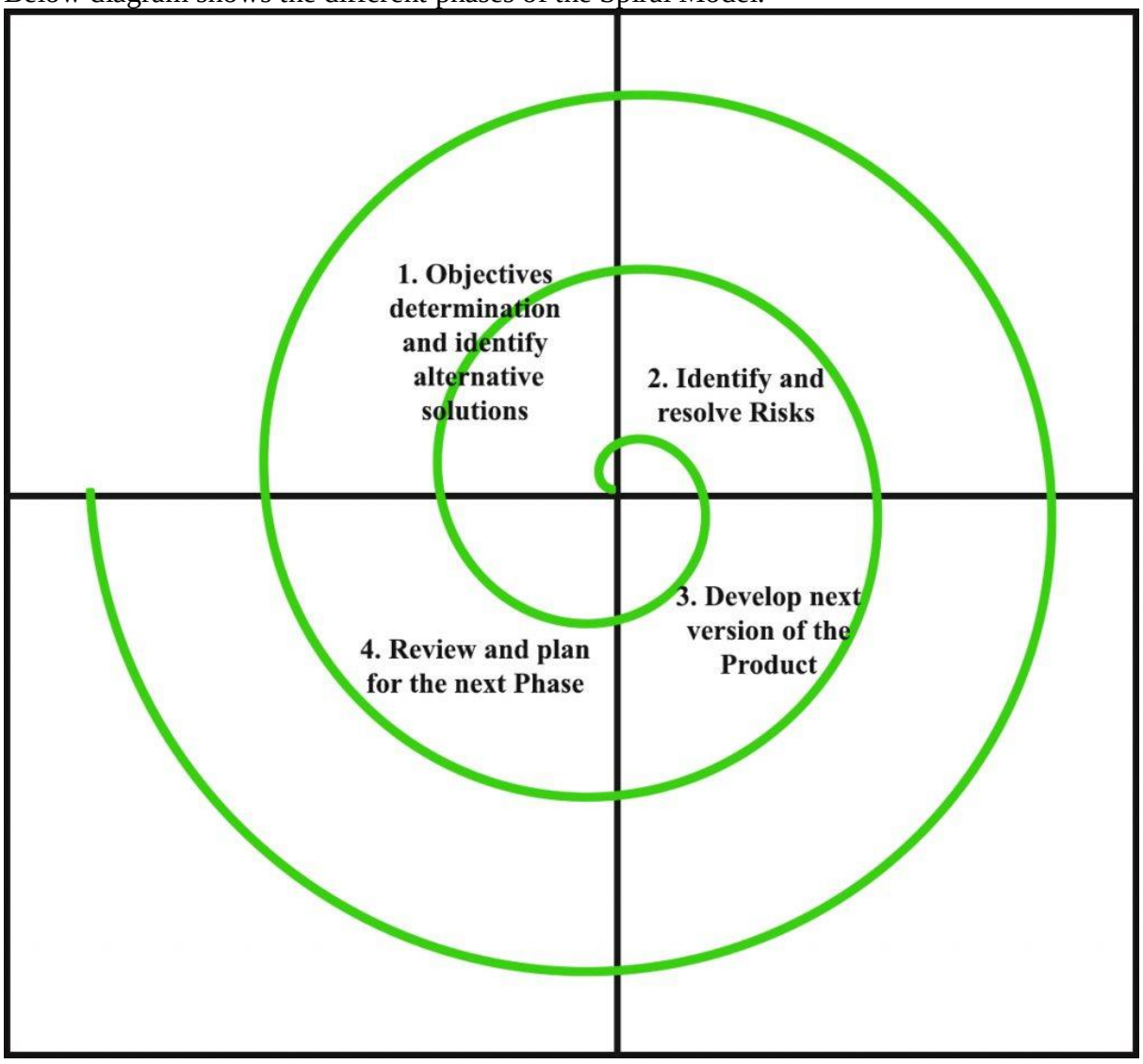
Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

2.2.3 Spiral Model:

Spiral model is one of the most important Software Development Life Cycle models, which provides support for **Risk Handling**. In its diagrammatic representation, it looks like a spiral with many loops. The exact number of loops of the spiral is unknown and can vary from project to project. **Each loop of the spiral is called a Phase of the software development process.** The exact number of phases needed to develop the product can be varied by the project manager depending upon the project risks. As the project manager dynamically determines the number of phases, so the project manager has an important role to develop a product using spiral model.

The Radius of the spiral at any point represents the expenses(cost) of the project so far, and the angular dimension represents the progress made so far in the current phase.

Below diagram shows the different phases of the Spiral Model:



Each phase of Spiral Model is divided into four quadrants as shown in the above figure. The functions of these four quadrants are discussed below-

1. **Objectives determination and identify alternative solutions:** Requirements are gathered from the customers and the objectives are identified, elaborated and

analyzed at the start of every phase. Then alternative solutions possible for the phase are proposed in this quadrant.

2. **Identify and resolve Risks:** During the second quadrant all the possible solutions are evaluated to select the best possible solution. Then the risks associated with that solution is identified and the risks are resolved using the best possible strategy. At the end of this quadrant, Prototype is built for the best possible solution.
3. **Develop next version of the Product:** During the third quadrant, the identified features are developed and verified through testing. At the end of the third quadrant, the next version of the software is available.
4. **Review and plan for the next Phase:** In the fourth quadrant, the Customers evaluate the so far developed version of the software. In the end, planning for the next phase is started.

Risk Handling in Spiral Model

A risk is any adverse situation that might affect the successful completion of a software project. The most important feature of the spiral model is handling these unknown risks after the project has started. Such risk resolutions are easier done by developing a prototype. The spiral model supports coping up with risks by providing the scope to build a prototype at every phase of the software development.

Prototyping Model also support risk handling, but the risks must be identified completely before the start of the development work of the project. But in real life project risk may occur after the development work starts, in that case, we cannot use Prototyping Model. In each phase of the Spiral Model, the features of the product dated and analyzed and the risks at that point of time are identified and are resolved through prototyping. Thus, this model is much more flexible compared to other SDLC models.

Why Spiral Model is called Meta Model ?

The Spiral model is called as 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 **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 of the advantages of the Spiral Model.

- **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.
- **Good for large projects:** It is recommended to use the Spiral Model in large and complex projects.
- **Flexibility in Requirements:** Change requests in the Requirements at later phase can be incorporated accurately by using this model.
- **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.

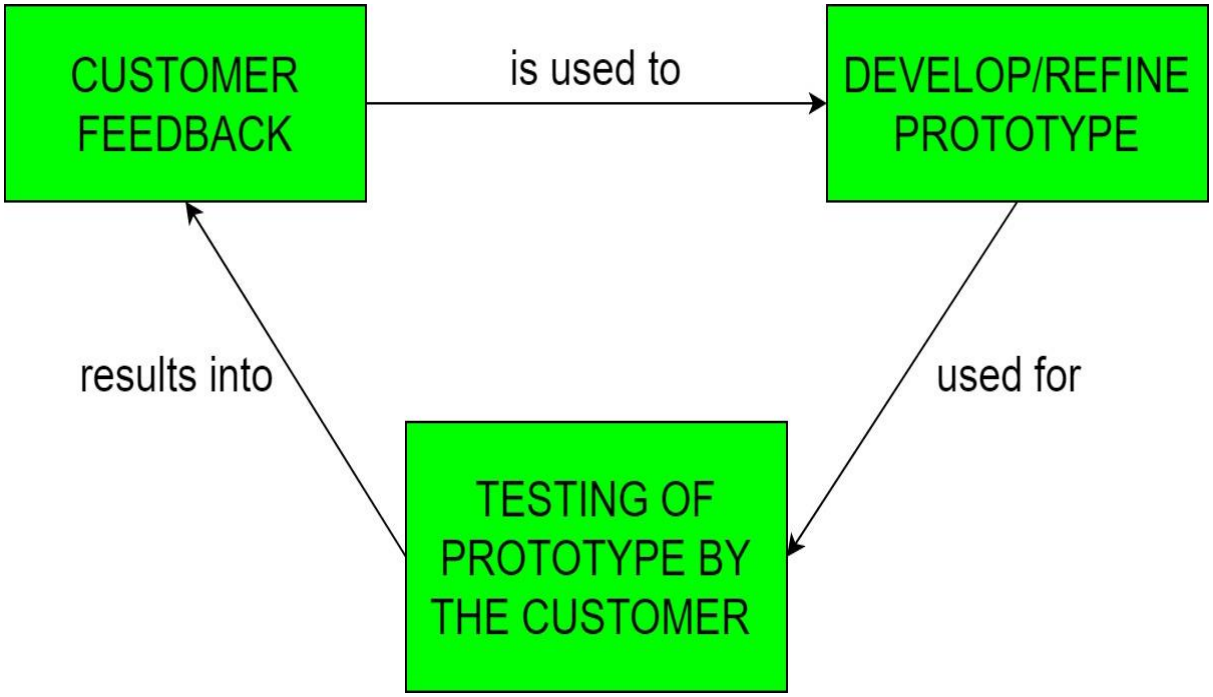
Disdvantages of Spiral Model: Below are some of the main disadvantages of the spiral model.

- **Complex:** The Spiral Model is much more complex than other SDLC models.

- **Expensive:** Spiral Model is not suitable for small projects as it is expensive.
- **Too much dependable on Risk Analysis:** The successful completion of the project is very much dependent on Risk Analysis. Without very highly experienced expertise, it is going to be a failure to develop a project using this model.
- **Difficulty in time management:** As the number of phases is unknown at the start of the project, so time estimation is very difficult.

2.2.4 Rapid Prototyping Model

Prototyping is defined as the process of developing a working replication of a product or system that has to be engineered. It offers a small scale facsimile of the end product and is used for obtaining customer feedback as described below:



The Prototyping Model is one of the most popularly used Software Development Life Cycle Models (SDLC models).This model is used when the customers do not know the exact project requirements beforehand. In this model, a prototype of the end product is first developed, tested and refined as per customer feedback repeatedly till a final acceptable prototype is achieved which forms the basis for developing the final product.

In this process model, the system is partially implemented before or during the analysis phase thereby giving the customers an opportunity to see the product early in the life cycle. The process starts by interviewing the customers and developing the incomplete high-level paper model. This document is used to build the initial prototype supporting only the basic functionality as desired by the customer. Once the customer figures out the problems, the prototype is further refined to eliminate them. The process continues till the user approves the prototype and finds the working model to be satisfactory.

There are 2 approaches for this model:

1. **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 a better quality.

2. **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.

Advantages –

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

Disadvantages –

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

Use –

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

2.2.5 Code and Fix Model:

The code and fix model probably is the most simple methodology in software engineering. It starts with little or no initial planning. Developers immediately start developing, fixing problems as they occur, until the project is complete.

Code and fix is a tempting choice when there is a tight development schedule because code development can be started right away and immediate results can be seen.

Unfortunately, if major architectural problems are found late in the process, usually large parts of the application have to be rewritten. Alternative development models can help in catching these problems in the early concept stages, when making changes is easier and less expensive.

The code and fix model is appropriate only for small projects that are not intended to serve as the basis for future development.

2.3 Good Program Design Techniques

- Design is a meaningful engineering representation of something that is to be built.
- It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for "good" design.
- In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces, and components.

2.3.1 Software Design

- The data design transforms the information domain model created during analysis into the data structures that will be required to implement the software
- The architectural design defines the relationship between major structural elements of the software, the "design patterns" that can be used to achieve the requirements
- The interface design describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it
- The component-level design transforms structural elements of the software architecture into a procedural description of software components

Design Principles

- The design process should not suffer from 'tunnel vision.'
- The design should be traceable to the analysis model.
- The design should be structured to accommodate change.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

Design and Quality

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

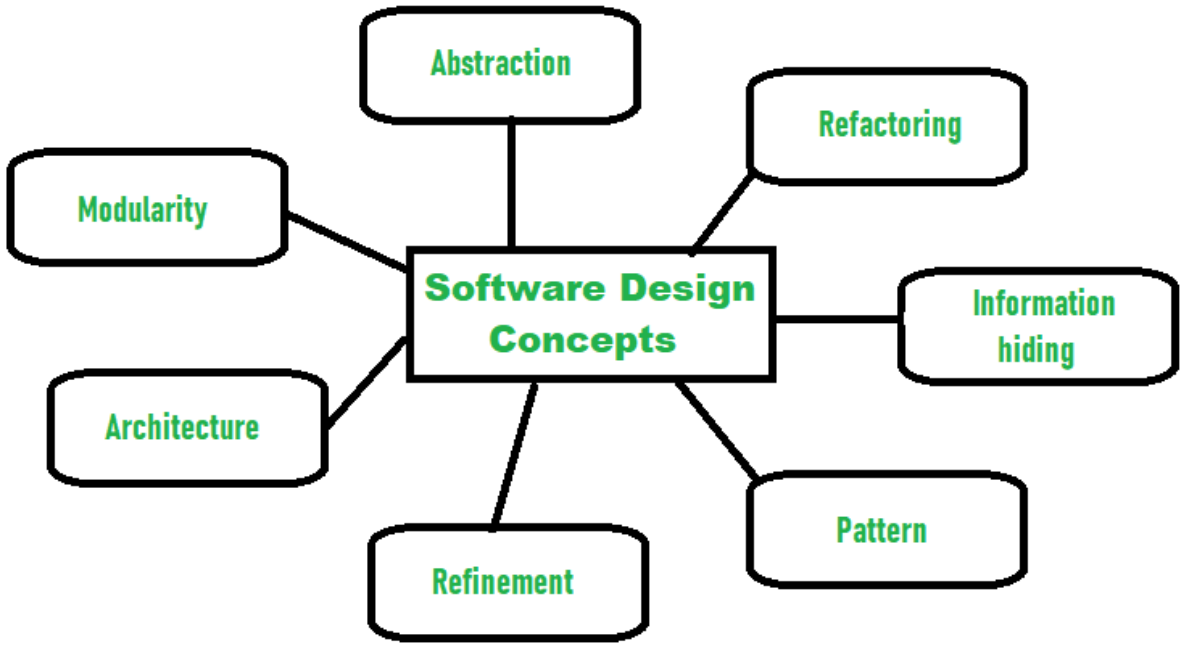
Quality Guidelines

- A design should exhibit an architecture that
    - (1) has been created using recognizable architectural styles or patterns,
    - (2) is composed of components that exhibit good design characteristics and

    - (3) can be implemented in an evolutionary fashion
    - For smaller systems, design can sometimes be developed linearly.
- A design should be modular; that is, the software should be logically partitioned into elements or subsystems
- A design should contain distinct representations of data, architecture, interfaces, and components.
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.

2.3.2 Software Design Concepts:

Concepts are defined as a principal idea or invention that comes in our mind or in thought to understand something. The **software design concept** simply means the idea or principle behind the design. It describes how you plan to solve the problem of designing software, the logic, or thinking behind how you will design software. It allows the software engineer to create the model of the system or software or product that is to be developed or built. The software design concept provides a supporting and essential structure or model for developing the right software. There are many concepts of software design and some of them are given below:



Following points should be considered while designing a Software:

1. **Abstraction- hide relevant data**
   Abstraction simply means to hide the details to reduce complexity and increases efficiency or quality. Different levels of Abstraction are necessary and must be applied

at each stage of the design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution. The solution should be described in broadways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.

2. **Modularity- subdivide the system**
   Modularity simply means to divide the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means to subdivide a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays there are different software available like Monolithic software that is hard to grasp for software engineers. So, modularity is design has now become a trend and is also important.
3. **Architecture- design a structure of something**
   Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.
4. **Refinement**
   The refinement concept of software design is actually a process of developing or presenting the software or system in a detailed manner that means to elaborate a system or software. Refinement is very necessary to find out any error if present and then to reduce it.
5. **Pattern- a repeated form**
   The pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.
6. **Information Hiding- hide the information**
   Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and it cant be accessed by any other modules.
7. **Refactoring- reconstruct something**
   Refactoring simply means to reconstruct something in such a way that it does not affect the behavior or any other features. Refactoring in software design means to reconstruct the design to reduce and complexity and simplify it without affecting the behavior or its functions. Fowler has defined refactoring as “the process of changing a software system in a way that it won’t affect the behavior of the design and improves the internal structure”.

Modularization: Modularization is the process of dividing a software system into multiple independent modules where each module works independently. There are many advantages of Modularization in software engineering. Some of these are given below:

- Easy to understand the system.
- System maintenance is easy.
- A module can be used many times as their requirements. No need to write it again and again.

- **Modularity**

The concept of modularity has been espoused for almost four decades.
Software is divided into separately named and addressable components, called modules.
Meyer [MEY88] defines five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

- Modular decomposability:

a design method provides a systematic mechanism for decomposing the problem into sub-problems --> reduce the complexity and achieve the modularity

- Modular composability:

a design method enables existing design components to be assembled into a new system.

- Modular understandability:

a module can be understood as a standalone unit it will be easier to build and easier to change.

- Modular continuity:

small changes to the system requirements result in changes to individual modules, rather than system-wide changes.

- Modular protection:

an aberrant condition occurs within a module and its effects are constrained within the module.

Let *C(x)* be a function that defines the perceived complexity of a problem *x*, and *E(x)* be a function that defines the effort (in time) required to solve a problem *x*. For two problems, *p1* and *p2*, if

$$C(p1) > C(p2) \qquad (13\text{-}1a)$$

it follows that

$$E(p1) > E(p2) \qquad (13\text{-}1b)$$

As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.
Another interesting characteristic has been uncovered through experimentation in human problem solving. That is,

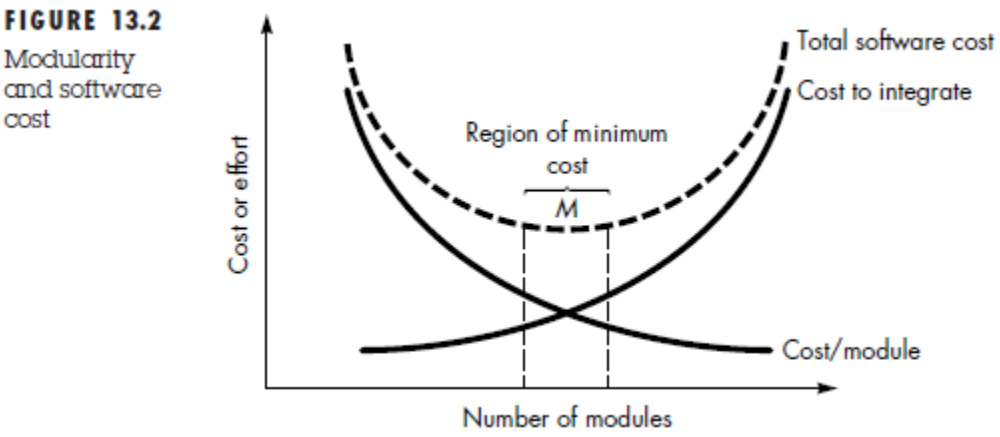
$$C(p1 + p2) > C(p1) + C(p2) \qquad (13\text{-}2)$$

Expression (13-2) implies that the perceived complexity of a problem that combines *p1* and *p2* is greater than the perceived complexity when each problem is considered separately. Considering Expression (13-2) and the condition implied by Expressions (13-1), it follows that

$$E(p1 + p2) > E(p1) + E(p2) \qquad (13\text{-}3)$$

This leads to a "divide and conquer" conclusion—it's easier to solve a complex problem when you break it into manageable pieces.

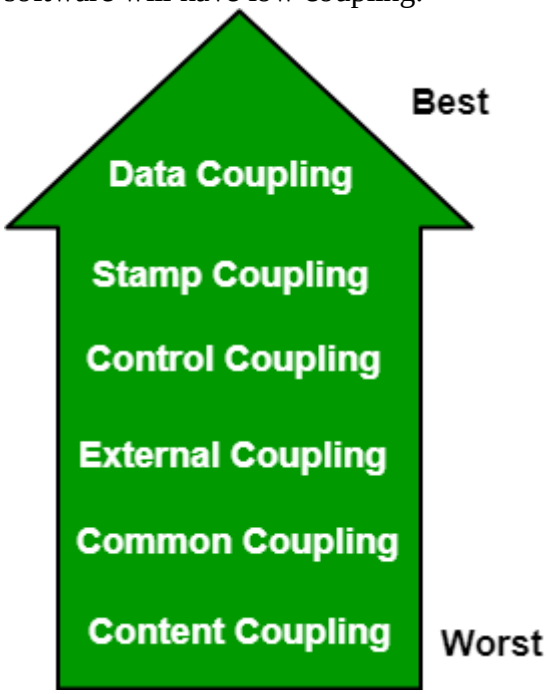
Referring to Figure 13.2, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, *M*, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict *M* with assurance.



2.3.3 Coupling and Cohesion

Coupling:

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



Types of Coupling:

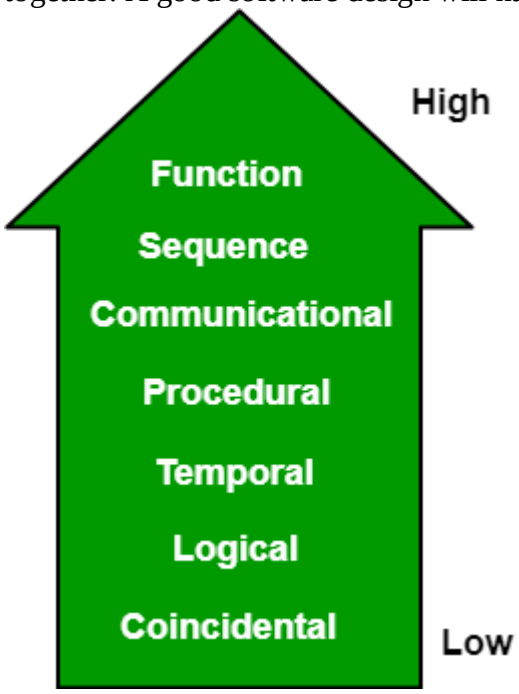
- **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent to each other and communicating through data. Module communications don't contain 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 int 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 init 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.

2.3.4 Abstraction and Information hiding:

- **Abstraction**

Each step in the software engineering process is a refinement in the level of abstraction of the software solution.

- Data abstractions: a named collection of data
- Procedural abstractions:
  A named sequence of instructions in a specific function
- Control abstractions:
  A program control mechanism without specifying internal details.

- Refinement: Refinement is actually a process of elaboration.

Stepwise refinement is a top-down design strategy proposed by Niklaus [WIR71].
The architecture of a program is developed by successively refining levels of procedural detail.

The process of program refinement is analogous to the process of refinement and partitioning that is used during requirements analysis. The major difference is in the level of implementation detail, instead of the approach.

Abstraction and refinement are complementary concepts.
Abstraction enables a designer to specify procedure and data w/o details.
Refinement helps the designer to reveal low-level details.

ABSTRACTION

When we consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken. Problem-oriented terminology is coupled with implementation- oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

Each step in the software process is a refinement in the level of abstraction of the software solution. During system engineering, software is allocated as an element of a computer-based system. During software requirements analysis, the software solution is stated in terms "that are familiar in the problem environment." As we move through the design

process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated.

Procedural abstraction

A *procedural abstraction* is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g.,walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

Data abstraction

A *data abstraction* is a named collection of data that describes a data object . In the context of the procedural abstraction *open*, we can define a data abstraction called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door**.

Many modern programming languages provide mechanisms for creating abstract data types. For example, the Ada package is a programming language mechanism that provides support for both data and procedural abstraction. The original abstract data type is used as a template or generic data structure from which other data structures can be instantiated.

Control abstraction

Control abstraction is the third form of abstraction used in software design. Like procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details. An example of a control abstraction is the *synchronization semaphore* used to coordinate activities in an operating system.

Information Hiding

Information hiding is the principle of segregation of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed.

2.3.5 Structured Programming

- **Software Architecture**

Software architecture is the hierarchical structure of program components and their interactions.

Shaw and Garlan [SHA95a] describe a set of properties of architecture design:

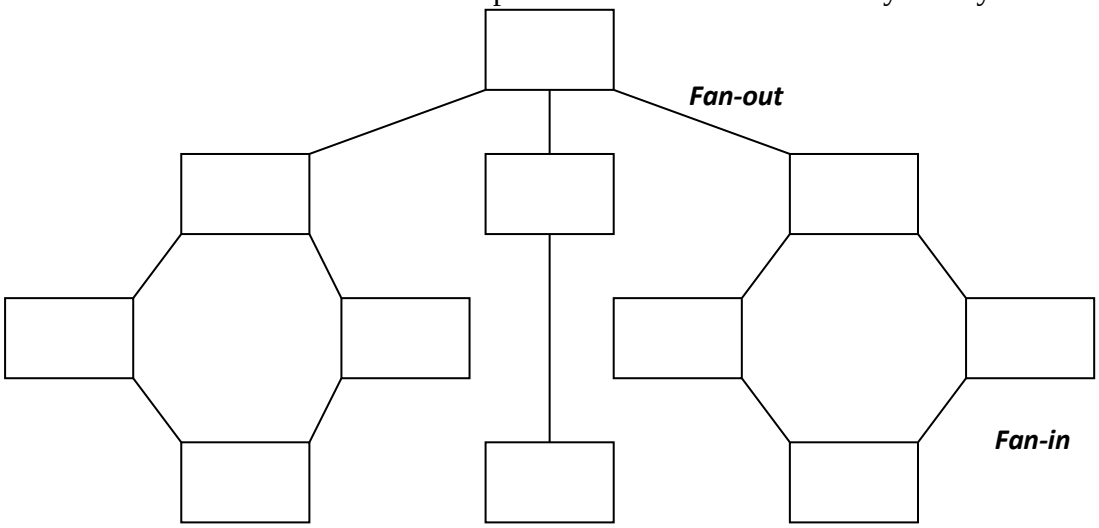
- Structural properties:
  The architecture design defines the system components and their interactions.
- Extra-functional properties:
  The architecture design should address how the design architecture achieves requirements for performance, capacity, reliability, adaptability, security.
- Families of related systems:
  The architecture design should draw upon repeatable patterns in the design of families of similar systems.

Different architectural design methods: (Figure 13.3)

- Structural models: represent architecture as an organized collection of components.
- Framework models: increase the level of design abstraction by identifying repeatable

  architecture design frameworks (patterns)
- Dynamic models: address the behavior aspects of the program architecture
- Process models: focus on the design of the business or technical process

- Functional models: can be used to represent the functional hierarchy of a system



Structural Partitioning
The program structure should be partitioned both horizontally and vertically. (Figure 13.4)

(1) **Horizontal partitioning** defines separate branches of the modular hierarchy for each major program function.
Simplest way is to partition a system into:
input, data transformation (processing), and output
Advantages of horizontal partition:
- easy to test, maintain, and extend
- fewer side effects in change propagation or error propagation
Disadvantage: more data to be passed across module interfaces
--> complicate the overall control of program flow

(2) **Vertical partitioning** suggests the control and work should be distributed top-down in program structure.
Advantages: good at dealing with changes:
- easy to maintain the changes
- reduce the change impact and and propagation

2.4 Automated Programming

- A type of computer programming in which some mechanism generates a computer program to allow human programmers to write the code at a higher abstraction level.
- *Generative programming* are concepts whereby programs can be written "to manufacture software components in an automated way
- The goal is to improve programmer productivity

It is often related to code-reuse topics such as component-based software engineering

- *Source-code generation* is the process of generating source code based on a description of the problem and is accomplished with a programming tool such as an integrated development environment (IDE).

2.5 Defensive Programming

- Defensive programming is a practice where developers anticipate failures in their code, then add supporting code to detect, isolate, and in some cases, recover from the anticipated failure.
- It can assist us by targeting defects in the source where they most commonly occur. By targeting defects in this way, we can find them sooner, leading to more stable and less exploitable software in a shorter period.
- Defensive programming operates on the premise that bugs exist in our code, so we should identify ways to more easily identify or counteract these problems.
- For example, watchdog timers are a common component of embedded systems that are designed to restart software and/or hardware after identifying anomalous behavior**.**

2.6 Redundant Programming

- To increase the reliability of software systems, researchers have investigated the use of various form of redundancy.
- A software system is redundant when it performs the same functionality through the execution of different elements.
- Redundancy has been extensively exploited in many software engineering techniques, for example for fault-tolerance and reliability engineering, and in self-adaptive and self-healing programs.

2.7 Aesthetics

- Aesthetics is a core design principle that defines a design's pleasing qualities.
- In visual terms, aesthetics includes factors such as balance, color, movement, pattern, scale, shape and visual weight.
- Designers use aesthetics to complement their designs' usability, and so enhance functionality with attractive layouts.
- It is a vital ingredient in user experience (UX) design and interaction design
- software systems are large and complex, often constructed by teams, intended to serve a useful function, and capable of causing injury and economic loss
- three dimensions along which designs may be evaluated: efficiency , economy , and elegance
- In the modern information economy many people spend much of their working lives interacting with one or a few software systems (e.g., a word processor, database system, or reservation system);
- In their recreational time, people may be engaged with the same or other software artifacts (e.g., a web browser or computer game).
- Therefore the external aesthetics of software systems can have a significant effect on the quality of many people's lives.
- most people would prefer to work with a beautiful tool
- the software system defines the work environment as fundamentally as the physical workspace does. Therefore, the aesthetics of the software systems deserves much attention

2.8 Software Modeling Tools

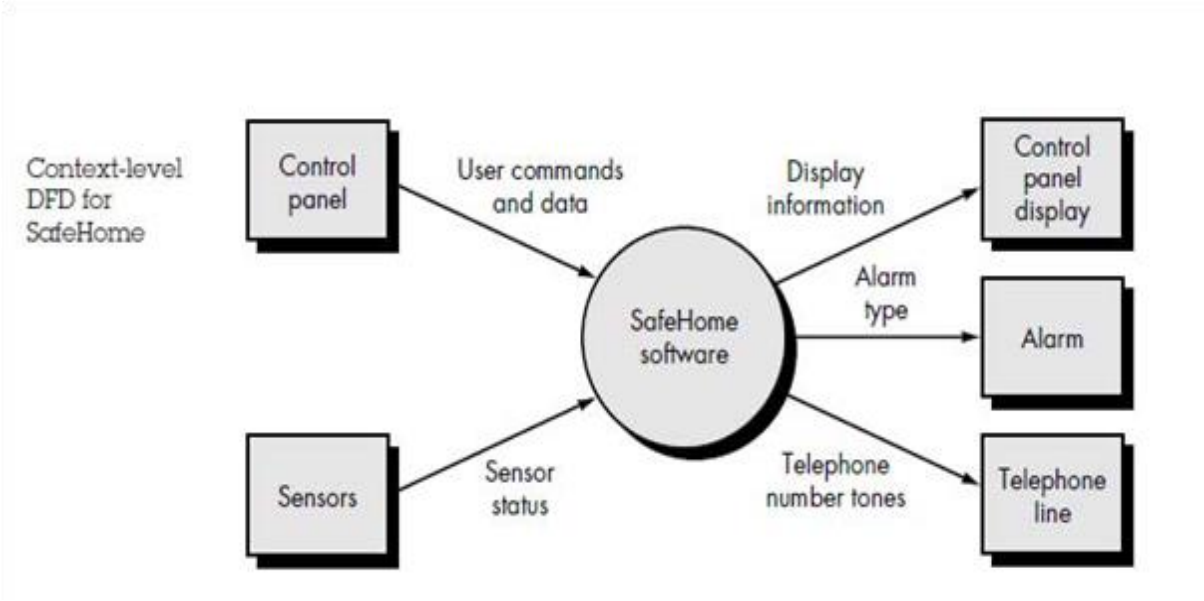
2.8.1 Data flow diagrams

- A data flow diagram is a graphical representation that depicts information flow and the transforms that are applied as data move from input to output.
- The data flow diagram may be used to represent a system or software at any level of abstraction.

- DFD provides a mechanism for functional modeling as well as information flow modeling.
- A DFD shows what kinds of data will be input to and output from the system, where the data will come from and go to, and where the data will be stored.
- It does not show information about the timing of processes, or information about whether processes will operate in sequence or in parallel (which is shown on a flowchart).

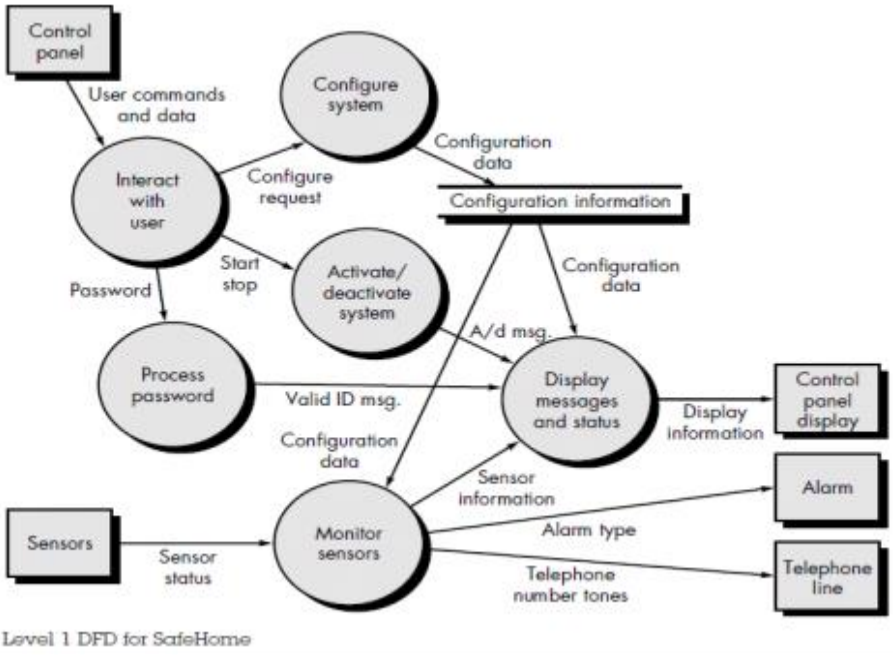
Context Level Data Flow Diagram

- Shows the interaction between the system and external agents
- The system's interactions with the outside world are modelled purely in terms of data flows across the system boundary.
- Shows the entire system as a single process, and gives no clues as to its internal organization.
- This context-level DFD is next "exploded", to produce a Level 1 DFD that shows some of the detail of the system being modeled.



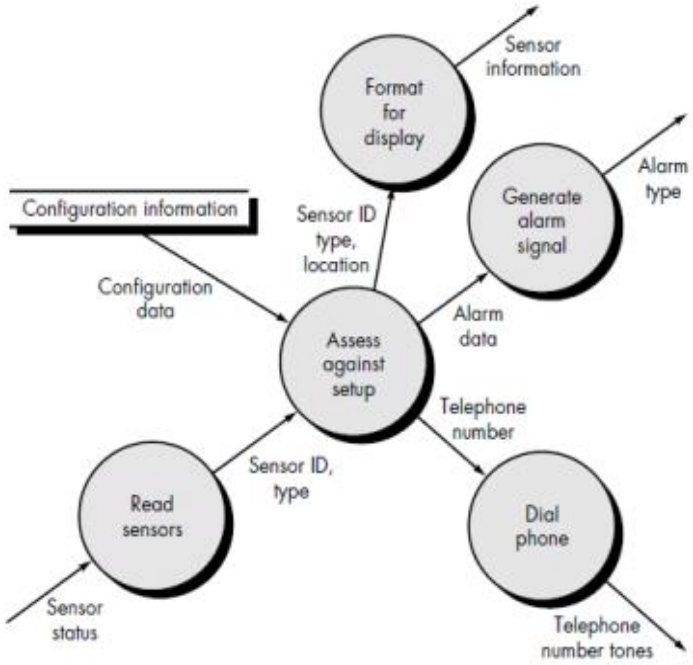
Level 1 DFD

- The Level 1 DFD shows how the system is divided into sub-systems (processes)
- It also identifies internal data stores that must be present in order for the system to do its job
- It shows the flow of data between the various parts of the system.



Level 1 DFD for SafeHome

Level 2 DFD



2.8.2 UML and XML

Unified Modeling Language (UML) is a general purpose modelling language. The main aim of UML is to define a standard way to **visualize** the way a system has been designed. It is quite similar to blueprints used in other fields of engineering.

UML is **not a programming language**, it is rather a visual language. We use UML diagrams to portray the **behavior and structure** of a system. UML helps software engineers, businessmen and system architects with modelling, design and analysis. The Object Management Group (OMG) adopted Unified Modelling Language as a standard in 1997. Its been managed by OMG ever since. International Organization for Standardization (ISO)

published UML as an approved standard in 2005. UML has been revised over the years and is reviewed periodically.

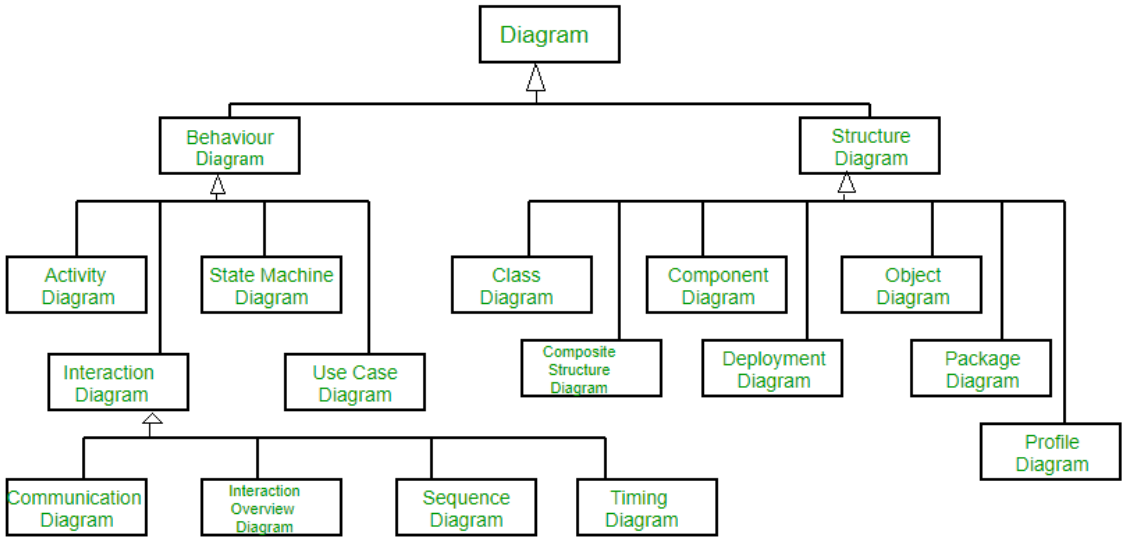
Do we really need UML?

- Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them.
- Businessmen do not understand code. So UML becomes essential to communicate with non programmers essential requirements, functionalities and processes of the system.
- A lot of time is saved down the line when teams are able to visualize processes, user interactions and static structure of the system.

UML is linked with **object oriented** design and analysis. UML makes the use of elements and forms associations between them to form diagrams. Diagrams in UML can be broadly classified as:

1. **Structural Diagrams –** Capture static aspects or structure of a system. Structural Diagrams include: Component Diagrams, Object Diagrams, Class Diagrams and Deployment Diagrams.
2. **Behavior Diagrams –** Capture dynamic aspects or behavior of the system. Behavior diagrams include: Use Case Diagrams, State Diagrams, Activity Diagrams and Interaction Diagrams.

The image below shows the hierarchy of diagrams according to UML 2.2



Object Oriented Concepts Used in UML –

1. **Class –** A class defines the blue print i.e. structure and functions of an object.
2. **Objects –** Objects help us to decompose large systems and help us to modularize our system. Modularity helps to divide our system into understandable components so that we can build our system piece by piece. An object is the fundamental unit (building block) of a system which is used to depict an entity.

3. **Inheritance –** Inheritance is a mechanism by which child classes inherit the properties of their parent classes.
4. **Abstraction –** Mechanism by which implementation details are hidden from user.
5. **Encapsulation –** Binding data together and protecting it from the outer world is referred to as encapsulation.
6. **Polymorphism –** Mechanism by which functions or entities are able to exist in different forms.

Additions in UML 2.0 –

- Software development methodologies like agile have been incorporated and scope of original UML specification has been broadened.
- Originally UML specified 9 diagrams. UML 2.x has increased the number of diagrams from 9 to 13. The four diagrams that were added are : timing diagram, communication diagram, interaction overview diagram and composite structure diagram. UML 2.x renamed statechart diagrams to state machine diagrams.
- UML 2.x added the ability to decompose software system into components and sub-components.

Structural UML Diagrams –

1. **Class Diagram –** The most widely use UML diagram is the class diagram. It is the building block of all object oriented software systems. We use class diagrams to depict the static structure of a system by showing system's classes,their methods and attributes. Class diagrams also help us identify relationship between different classes or objects.
2. **Composite Structure Diagram –** We use composite structure diagrams to represent the internal structure of a class and its interaction points with other parts of the system. A composite structure diagram represents relationship between parts and their configuration which determine how the classifier (class, a component, or a deployment node) behaves. They represent internal structure of a structured classifier making the use of parts, ports, and connectors. We can also model collaborations using composite structure diagrams. They are similar to class diagrams except they represent individual parts in detail as compared to the entire class.
3. **Object Diagram –** An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them. Since object diagrams depict behaviour when objects have been instantiated, we are able to study the behaviour of the system at a particular instant. An object diagram is similar to a class diagram except it shows the instances of classes in the system. We depict actual classifiers and their relationships making the use of class diagrams. On the other hand, an Object Diagram represents specific instances of classes and relationships between them at a point of time.
4. **Component Diagram –** Component diagrams are used to represent the how the physical components in a system have been organized. We use them for modelling implementation details. Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development. Component Diagrams become essential to use when we design and build complex systems. Interfaces are used by components of the system to communicate with each other.

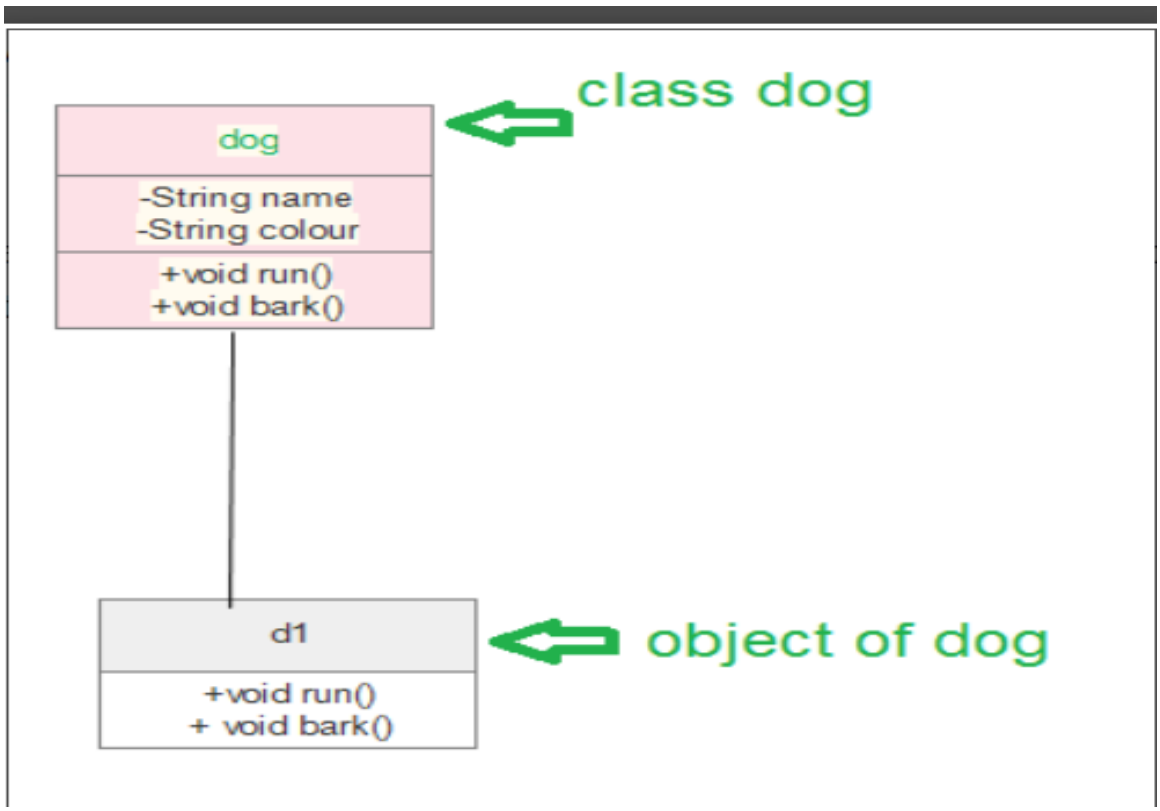
5. **Deployment Diagram –** Deployment Diagrams are used to represent system hardware and its software.It tells us what hardware components exist and what software components run on them.We illustrate system architecture as distribution of software artifacts over distributed targets. An artifact is the information that is generated by system software. They are primarily used when a software is being used, distributed or deployed over multiple machines with different configurations.
6. **Package Diagram –** We use Package Diagrams to depict how packages and their elements have been organized. A package diagram simply shows us the dependencies between different packages and internal composition of packages. Packages help us to organise UML diagrams into meaningful groups and make the diagram easy to understand. They are primarily used to organise class and use case diagrams.

Behavior Diagrams –

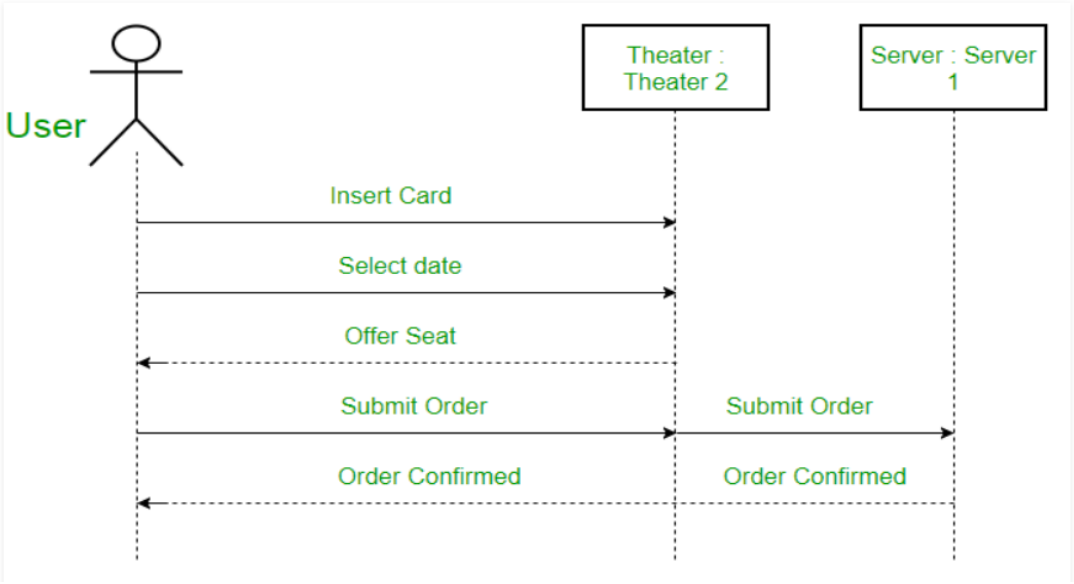
1. **State Machine Diagrams –** A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It's a behavioral diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as **State machines** and **State-chart Diagrams** . These terms are often used interchangeably.So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli.
2. **Activity Diagrams –** We use Activity Diagrams to illustrate the flow of control in a system. We can also use an activity diagram to refer to the steps involved in the execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram.An activity diagram focuses on condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.
3. **Use Case Diagrams –** Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors). A use case is basically a diagram representing different scenarios where the system can be used. A use case diagram gives us a high level view of what the system or a part of the system does without going into implementation details.
4. **Sequence Diagram –** A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place.We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.
5. **Communication Diagram –** A Communication Diagram(known as Collaboration Diagram in UML 1.x) is used to show sequenced messages exchanged between objects. A communication diagram focuses primarily on objects and their relationships. We can represent similar information using Sequence diagrams,however, communication diagrams represent objects and links in a free form.

6. **Timing Diagram –** Timing Diagram are a special form of Sequence diagrams which are used to depict the behavior of objects over a time frame. We use them to show time and duration constraints which govern changes in states and behavior of objects.
7. **Interaction Overview Diagram –** An Interaction Overview Diagram models a sequence of actions and helps us simplify complex interactions into simpler occurrences. It is a mixture of activity and sequence diagrams.

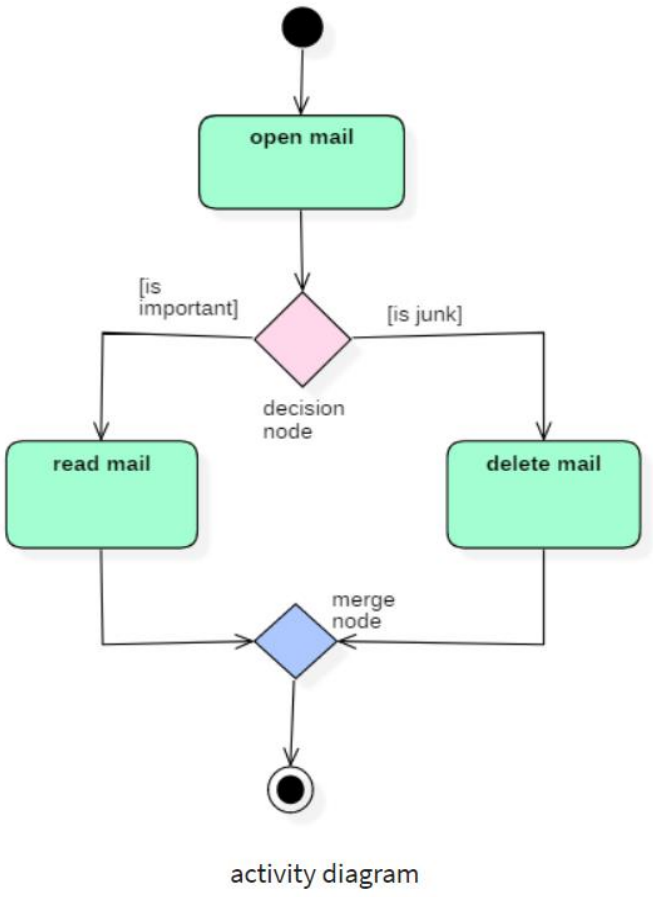
Class Diagram



Sequence Diagram:



Activity Diagram:



activity diagram

XML Metadata Interchange

- UML models can be exchanged among UML tools by using the XML Metadata Interchange (XMI) format.
- The **XML Metadata Interchange** (**XMI**) is an Object Management Group (OMG) standard for exchanging metadata information via Extensible Markup Language (XML)
- **Extensible Markup Language** (**XML**) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable

2.9 Jackson System Development (JSD)

Jackson System Development (JSD) is a method of system development that covers the software life cycle either directly or by providing a framework into which more specialized techniques can fit. JSD can start from the stage in a project when there is only a general statement of requirements.
However many projects that have used JSD actually started slightly later in the life cycle, doing the first steps largely from existing documents rather than directly with the users.

Phases of JDS:
JSD has 3 phases:

1. **Modelling Phase:**
   In the modelling phase of JSD the designer creates a collection of entity structure diagrams and identifies the entities in the system, the actions they perform, the attributes of the actions and time ordering of the actions in the life of the entities.
2. **Specification Phase:**
   This phase focuses on actually what is to be done? Previous phase provides the basic for this phase. An sufficient model of a time-ordered world must itself be time-ordered. Major goal is to map progress in the real world on progress in the system that models it.
3. **Implementation Phase:**
   In the implementation phase JSD determines how to obtain the required functionality. Implementation way of the system is based on transformation of specification into efficient set of processes. The processes involved in it should be designed in such a manner that it would be possible to run them on available software and hardware.

JSD Steps:
Initially there were six steps when it was originally presented by Jackson, they were as below:

1. Entity/action step
2. Initial model step
3. Interactive function step
4. Information function step
5. System timing step
6. System implementation step

Later some steps were combined to create method with only three steps:

1. Modelling Step
2. Network Step
3. Implementation Step

Merits of JSD:

- It is designed to solve real time problem.

- JSD modelling focuses on time.
- It considers simultaneous processing and timing.
- It is a better approach for micro code application.

Demerits of JSD:

- It is a poor methodology for high level analysis and data base design.
- JSD is a complex methodology due to pseudo code representation.
- It is less graphically oriented as compared to SA/SD or OMT.
- It is a bit complex and difficult to understand.

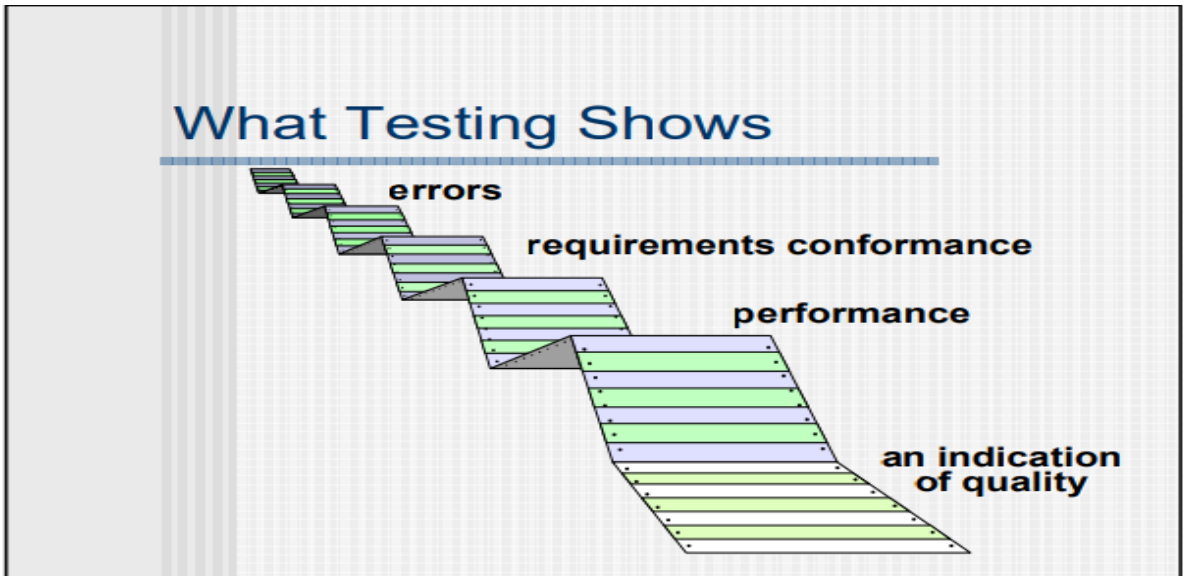
Unit III

Verification and Validation

3.1 Testing of Software Products

Testing

- Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.



Strategic Approach

- To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

Verification and validation

- **Verification** refers to the set of tasks that ensure that software correctly implements a specific function.
- **Validation** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
- Boehm [Boe81] states this another way:

- Verification: "Are we building the product right?"
- Validation: "Are we building the right product?"

Who Tests the Software?

- Developer:

Understands the system but, will test "gently" and, is driven by "delivery"

- Independent tester:

Must learn about the system, but, will attempt to break it and, is driven by quality

Testing

- Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer.
- Your goal is to design a series of test cases that have a high likelihood of finding errors
- These techniques provide systematic guidance for designing tests that
- (1) exercise the internal logic of software components, and
- (2) exercise the input and output domains of the program to uncover errors in program function, behavior and performance.
- Software is tested from two different perspectives: (1) internal program logic is exercised using “white box” test case design techniques.
- Software requirements are exercised using “black box” test case design techniques.
- In both cases, the intent is to find the maximum number of errors with the minimum amount of effort and time.

Testing objectives

- 1. Testing is a process of executing a program with the intent of finding an error.
- 2. A good test case is one that has a high probability of finding an as-yet undiscovered error.
- 3. A successful test is one that uncovers an as-yet-undiscovered error.

Testing principles

- All tests should be traceable to customer requirements
- Tests should be planned long before testing begins
- The Pareto principle applies to software testing
- Testing should begin “in the small” and progress toward testing “in the large.”
- Exhaustive testing is not possible

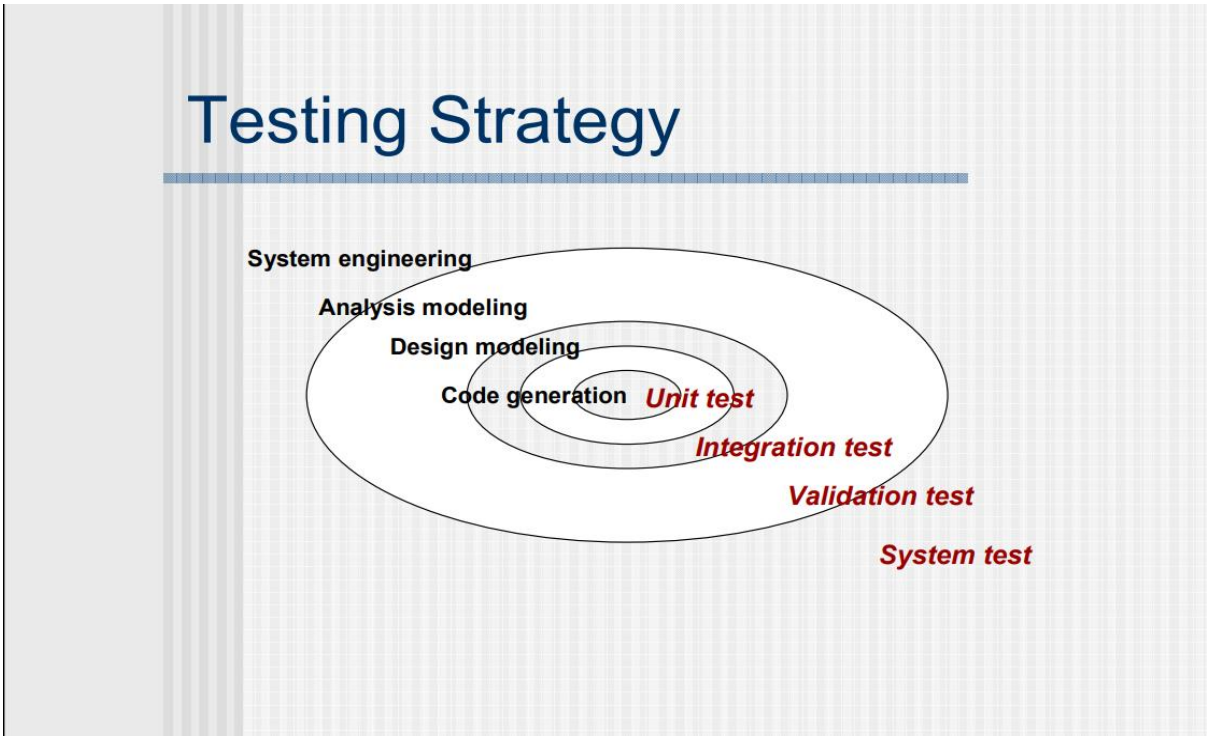
- To be most effective, testing should be conducted by an independent third party

Testability

- Software testability is simply how easily [a computer program] can be tested
- The checklist that follows provides a set of characteristics that lead to testable software.
- Operability
- Observability
- Controllability
- Decomposability
- Simplicity
- Stability
- Understandability

Testing Strategy

Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name unit testing. Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module's control structure to ensure complete coverage and maximum error detection. Next, components must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the dual problems of verification and program construction. Black-box test case design techniques are the most prevalent during integration, although a limited amount of white-box testing may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of high-order tests are conducted. Validation criteria (established during requirements analysis) must be tested. Validation testing provides final assurance that software meets all functional, behavioral, and performance requirements. Black-box testing techniques are used exclusively during validation.

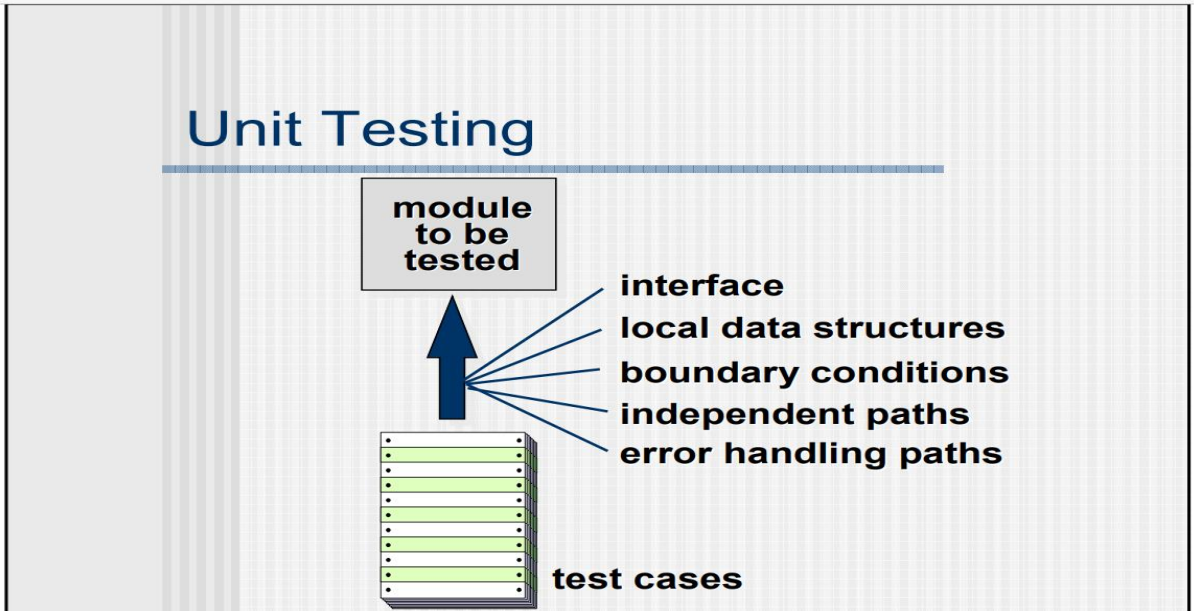


- We begin by ‘testing-in-the-small’ and move toward ‘testing-in-the-large’
- For conventional software, The module (component) is our initial focus Integration of modules follows
- For OO software, our focus when “testing in the small” changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

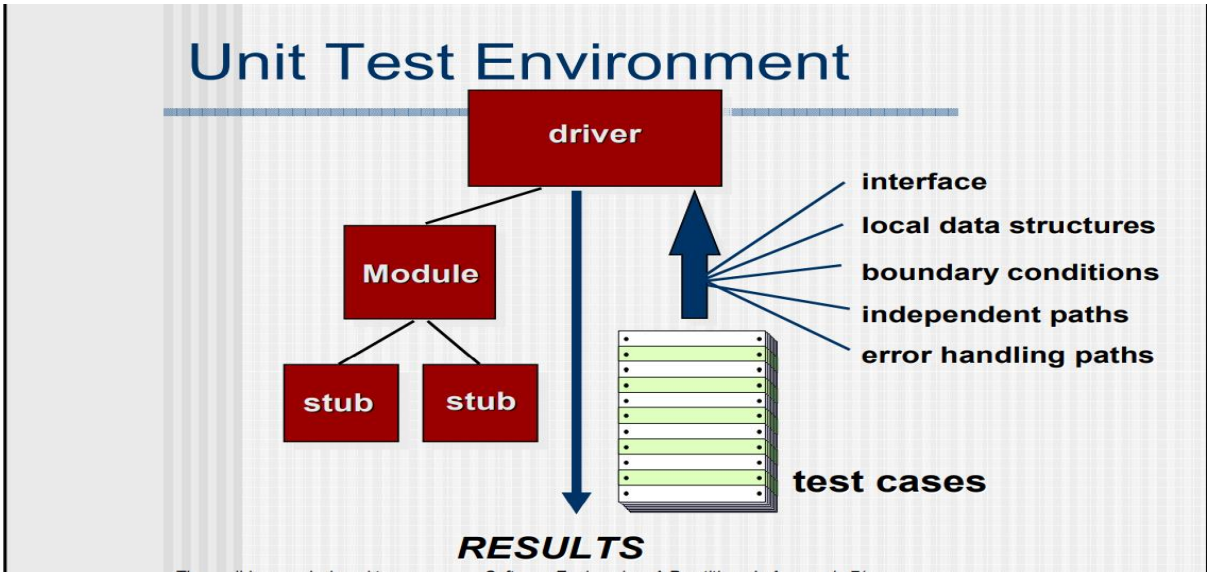
Unit testing

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and uncovered errors is limited by the constrained scope established for unit testing. The unit test is white-box oriented, and the step can be conducted in parallel for multiple components.

The module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.



- Among the more common errors in computation are (1) misunderstood or incorrect arithmetic precedence, (2) incorrect initialization, (3) precision inaccuracy, (4) incorrect symbolic representation of an expression



Integration testing

Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design.

Top-down Integration

Top-down integration testing is an incremental approach to construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate)

to the main control module are incorporated into the structure in either a depth-first or breadth-first manner

The integration process is performed in a series of five steps:

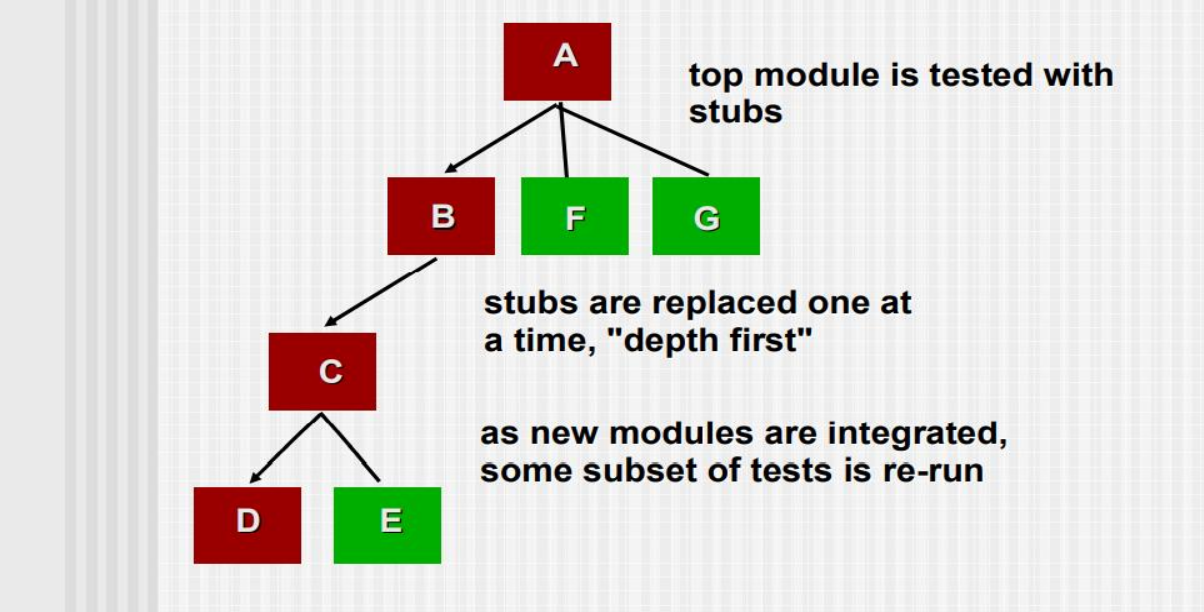
1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.

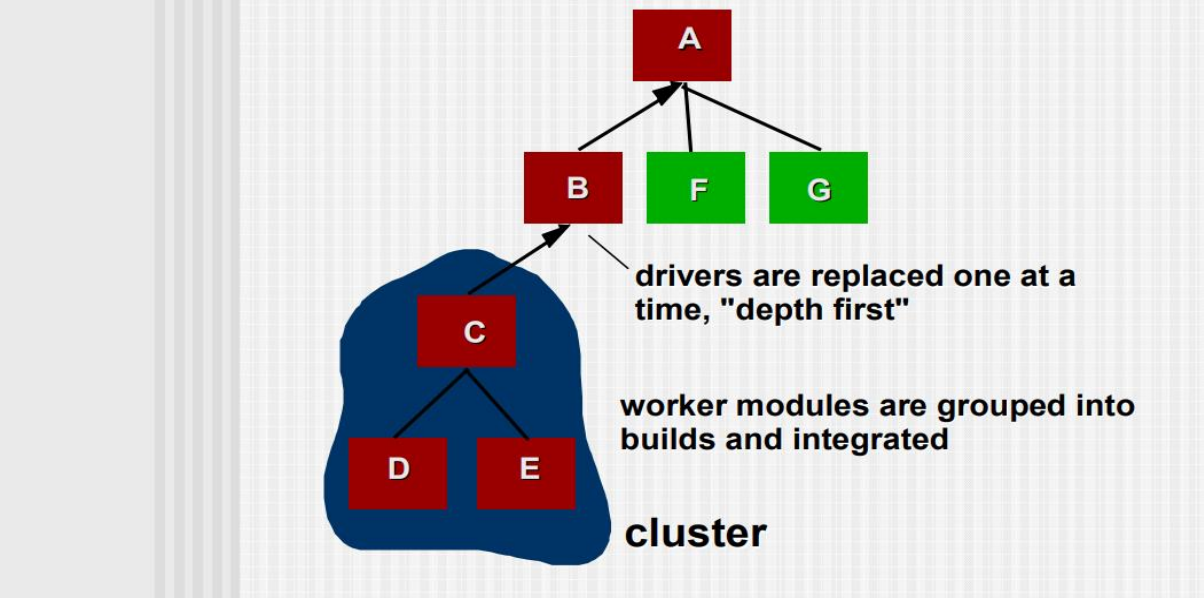
4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing may be conducted to ensure that new errors have not been introduced.



Bottom-up Integration

Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.



A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.

2. A driver (a control program for testing) is written to coordinate test case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

Regression Testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, regression testing is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

4 Smoke Testing

Smoke testing is an integration testing approach that is commonly used when "shrinkwrapped" software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis. In essence, the smoke testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a "build." A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.

2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.

3. The build is integrated with other builds and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

Object oriented testing

- begins by evaluating the correctness and consistency of the analysis and design models
- testing strategy changes
- the concept of the 'unit' broadens due to encapsulation
- integration focuses on classes and their execution across a 'thread' or in the context of a usage scenario
- validation uses conventional black box methods

High order testing

- **Validation testing** - Focus is on software requirements
- **System testing** - Focus is on system integration
- **Alpha/Beta testing** - Focus is on customer usage
- **Recovery testing** - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- **Security testing** - verifies that protection mechanisms built into a system will, in fact, protect it from improper attacks
- **Stress testing** executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- **Performance Testing** test the run-time performance of software within the context of an integrated system

3.2 Black-Box Testing and White-Box Testing

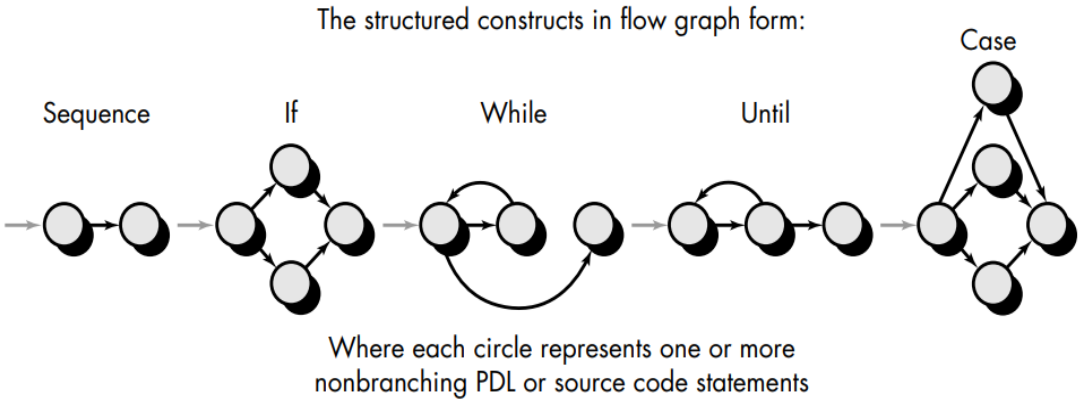
3.2.1 White box testing

- White-box testing, sometimes called glass-box testing, is a test case design method that uses the control structure of the procedural design to derive test cases.
- Using white-box testing methods, the software engineer can derive test cases that
- (1) guarantee that all independent paths within a module have been exercised at least once
- (2) exercise all logical decisions on their true and false sides
- (3) execute all loops at their boundaries and within their operational bounds
- (4) exercise internal data structures to ensure their validity

Basis path testing

- Basis path testing is a white-box testing technique first proposed by Tom McCabe .
- The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

Flow Graph Notation



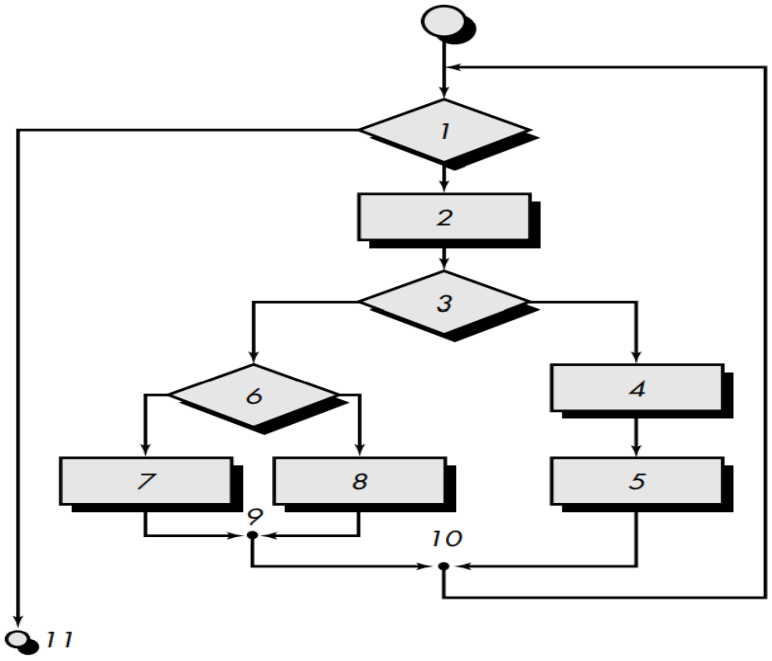
Cyclomatic Complexity

- Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.
- When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program
- It provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Independent path

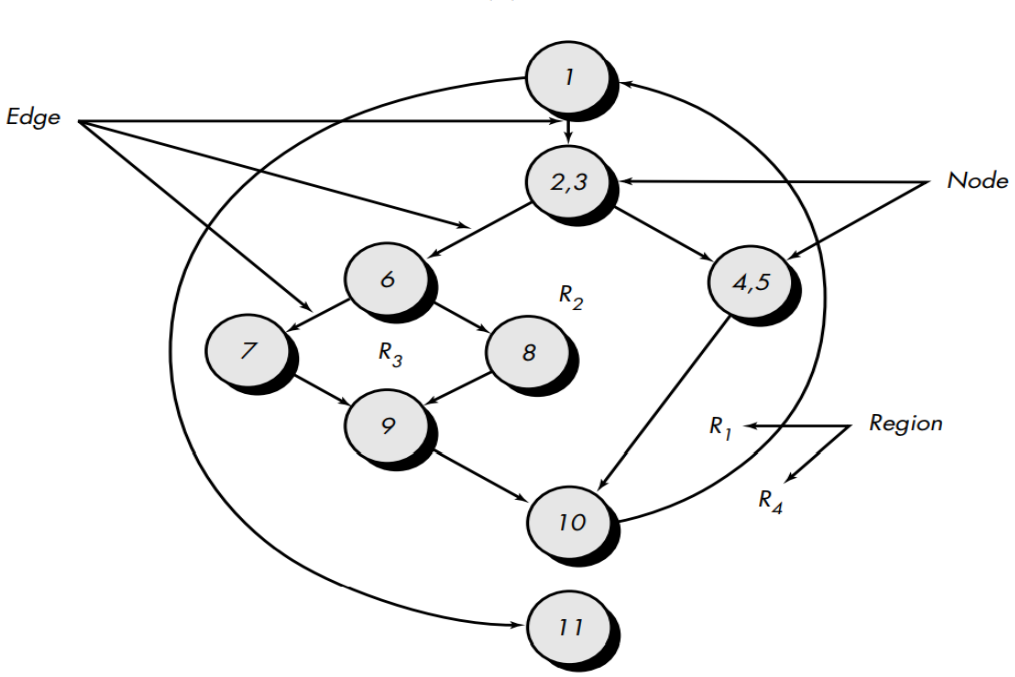
- An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.
- An independent path must move along at least one edge that has not been traversed before the path is defined.

Flow chart



(A)

Flow graph



- A set of independent paths for the flow graph:
- path 1: 1-11
- path 2: 1-2-3-4-5-10-1-11
- path 3: 1-2-3-6-8-9-10-1-11
- path 4: 1-2-3-6-7-9-10-1-11
- Note that each new path introduces a new edge.

- The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.
- Paths 1, 2, 3, and 4 constitute a basis set for the flow graph

Cyclomatic complexity

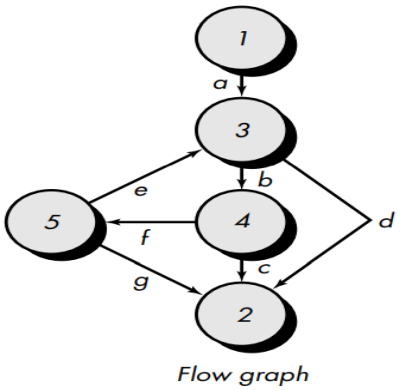
- Cyclomatic complexity has a foundation in graph theory and provides us with an extremely useful software metric.
- Complexity is computed in one of three ways:
- 1. The number of regions of the flow graph correspond to the cyclomatic complexity.
- 2. Cyclomatic complexity, V(G), for a flow graph, G, is defined as V(G) = E - N + 2 where E is the number of flow graph edges, N is the number of flow graph nodes.
- 3. Cyclomatic complexity, V(G), for a flow graph, G, is also defined as V(G) = P + 1 where P is the number of predicate nodes contained in the flow graph G.
- The cyclomatic complexity can be computed using each of the algorithms just noted:
- 1. The flow graph has four regions.
- 2. V(G) = 11 edges - 9 nodes + 2 = 4.
- 3. V(G) = 3 predicate nodes + 1 = 4.
- Therefore, the cyclomatic complexity of the flow graph is 4.

Deriving Test Cases

- Using the design or code as a foundation, draw a corresponding flow graph
- Determine the cyclomatic complexity of the resultant flow graph
- Determine a basis set of linearly independent paths
- Prepare test cases that will force execution of each path in the basis set
- Each test case is executed and compared to expected results.
- Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once

Graph Matrices

- A software tool that assists in basis path testing



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Link weight

- By adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing.
- The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist).
- But link weights can be assigned other, more interesting properties:
- The probability that a link (edge) will be executed.
- The processing time expended during traversal of a link.
- The memory required during traversal of a link.
- The resources required during traversal of a link.

Connection matrix

Node \ Connected to node	1	2	3	4	5	Connections
1			1			1 – 1 = 0
2						
3		1		1		2 – 1 = 1
4		1			1	2 – 1 = 1
5		1	1			2 – 1 = 1

Graph matrix

3 + 1 = 4 ← Cyclomatic complexity

CONTROL STRUCTURE TESTING

- Although basis path testing is simple and highly effective, it is not sufficient in itself.

- Other variations on control structure testing are needed.

Condition Testing

- Condition testing is a test case design method that exercises the logical conditions contained in a program module.
- A simple condition is a Boolean variable or a relational expression.
- A relational expression takes the form E1<relational-operator>E2 where E1 and E2 are arithmetic expressions and is one of the following:
- $<, \leq, =, \neq$ (nonequality), $>$, or $\geq$.
- A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses.

Types of errors in a condition

- Boolean operator error (incorrect/missing/extra Boolean operators).
- Boolean variable error.
- Boolean parenthesis error.
- Relational operator error.
- Arithmetic expression error.

condition testing strategies

- Branch testing is probably the simplest condition testing strategy. For a compound condition C, the true and false branches of C and every simple condition in C need to be executed at least once
- Domain testing requires three or four tests to be derived for a relational expression.
- BRO (branch and relational operator) testing technique guarantees the detection of branch and relational operator errors in a condition provided that all Boolean variables and relational operators in the condition occur only once and have no common variables

Data Flow Testing

- The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program
- Assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables
- A definition-use (DU) chain of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S')

- Simple data flow testing strategy is to require that every DU chain be covered at least once

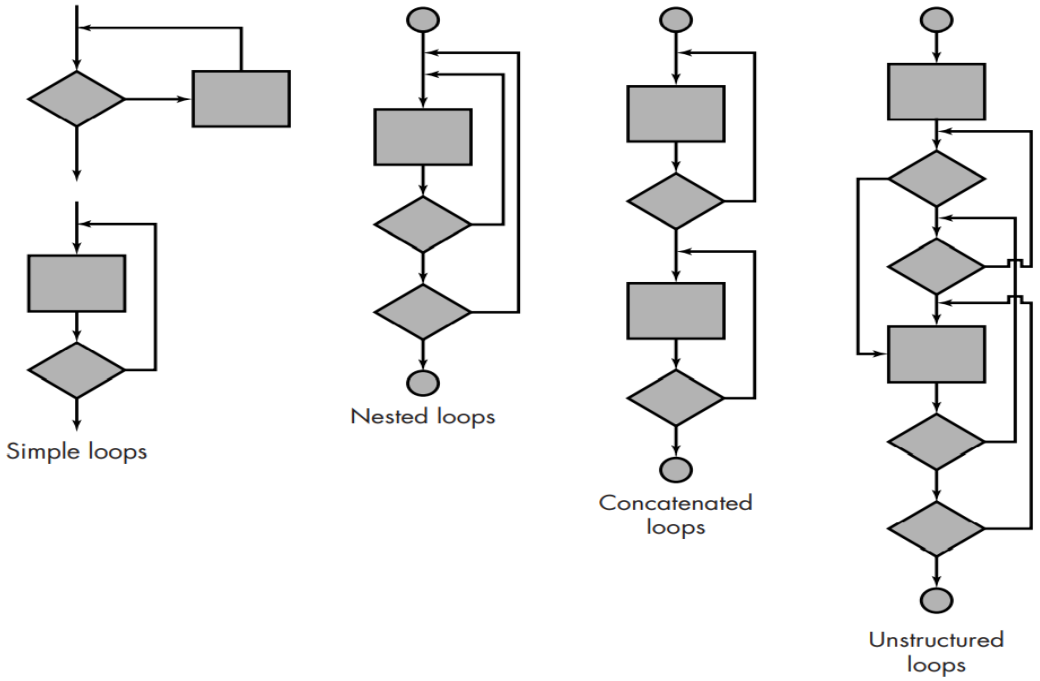
Loop Testing

- Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs
- Four different classes of loops can be defined:
- simple loops
- concatenated loops
- nested loops
- unstructured loops

Set of tests for simple loops

- 1. Skip the loop entirely.
- 2. Only one pass through the loop.
- 3. Two passes through the loop.
- 4. m passes through the loop where m < n.
- 5. n-1, n, n + 1 passes through the loop.

Types of loops



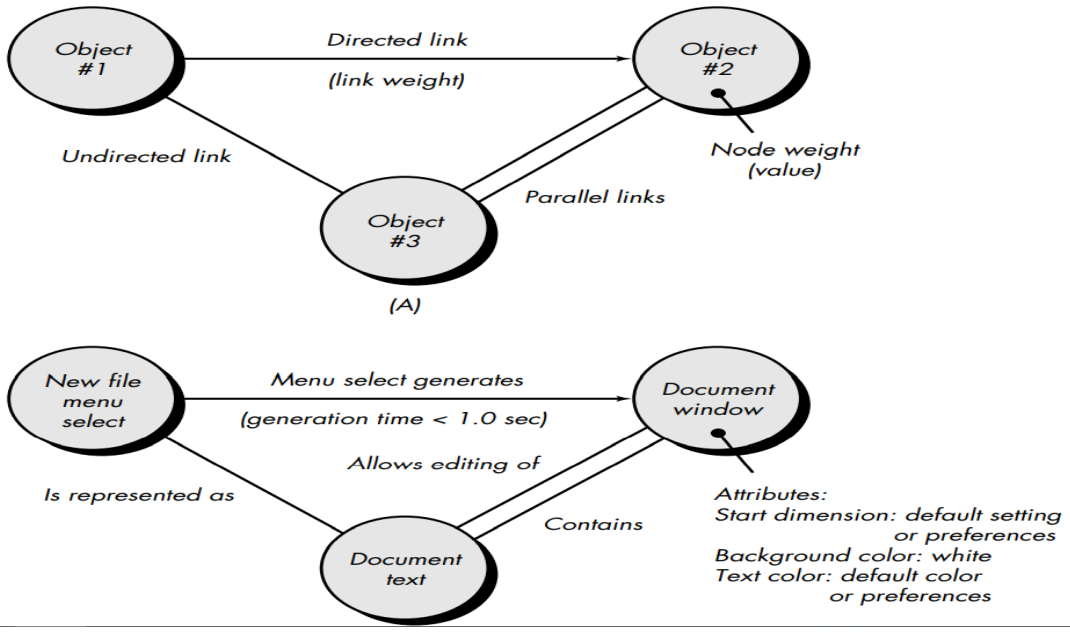
3.2.2 BLACK-BOX TESTING

- Black-box testing, also called behavioral testing, focuses on the functional requirements of the software.
- It enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.
- It is not an alternative to white-box techniques.
- Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods

Types of errors

- Black-box testing attempts to find errors in the following categories:
- (1) incorrect or missing functions
- (2) interface errors
- (3) errors in data structures or external data base access
- (4) behavior or performance errors
- (5) initialization and termination errors

Graph-Based Testing Methods



Equivalence Partitioning

- Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived
- Equivalence classes may be defined according to the following guidelines:
- 1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.

- 2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
- 3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
- 4. If an input condition is Boolean, one valid and one invalid class are defined.

Boundary Value Analysis

- Boundary value analysis leads to a selection of test cases that exercise bounding values
- Selection of test cases at the "edges" of the class

Guidelines:

- 1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
- 2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
- 3. If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Comparison Testing

- There are some situations (e.g., aircraft avionics, automobile braking systems) in which the reliability of software is absolutely critical.
- In such applications redundant hardware and software are often used to minimize the possibility of error.
- When redundant software is developed, separate software engineering teams develop independent versions of an application using the same specification.
- In such situations, each version can be tested with the same test data to ensure that all provide identical output.
- Then all versions are executed in parallel with real-time comparison of results to ensure consistency.

3.3 Static analysis

- Static analysis involves no dynamic execution of the software under test and can detect possible defects in an early stage, before running the program.
- Static analysis is done after coding and before executing unit tests.
- Static analysis can be done by a machine to automatically “walk through” the source code and detect noncomplying rules. The classic example is a compiler which finds lexical, syntactic and even some semantic mistakes.

- Static analysis can also be performed by a person who would review the code to ensure proper coding standards and conventions are used to construct the program. This is often called Code Review and is done by a peer developer, someone other than the developer who wrote the code.
- Static analysis is also used to force developers to not use risky or buggy parts of the programming language by setting rules that must not be used.

Things to be examined

- When developers performs code analysis, they usually look for
- Lines of code
- Comment frequency
- Proper nesting
- Number of function calls
- Cyclomatic complexity

Quality attributes

- Quality attributes that can be the focus of static analysis:
- Reliability
- Maintainability
- Testability
- Re-usability
- Portability
- Efficiency

Advantages of Static Analysis

- It can find weaknesses in the code at the exact location.
- It can be conducted by trained software assurance developers who fully understand the code.
- Source code can be easily understood by other or future developers
- Weaknesses are found earlier in the development life cycle, reducing the cost to fix.
- Less defects in later tests
- Unique defects are detected that cannot or hardly be detected using dynamic tests
    - Unreachable code
    - Variable use (undeclared, unused)
    - Uncalled functions

- Boundary value violations

3.4 Symbolic execution

- **Symbolic execution** (also **symbolic evaluation** or **symbex**) is a means of analyzing a program to determine what inputs cause each part of a program to execute.
- An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would.
- It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch.

Example

```
1 int f() {
2   ...
3   y = read();
4   z = y * 2;
5   if (z == 12) {
6     fail();
7   } else {
8     printf("OK");
9   }
10 }
```

- During symbolic execution, the program reads a symbolic value (e.g., λ) and assigns it to y.
- The program would then proceed with the multiplication and assign λ * 2 to z.
- When reaching the if statement, it would evaluate λ * 2 == 12.
- At this point of the program, λ could take any value, and symbolic execution can therefore proceed along both branches, by "forking" two paths.
- Each path gets assigned a copy of the program state at the branch instruction as well as a path constraint. In this example, the path constraint is λ * 2 == 12 for the then branch and λ * 2 != 12 for the else branch.

- Both paths can be symbolically executed independently. When paths terminate (e.g., as a result of executing fail() or simply exiting), symbolic execution computes a concrete value for λ by solving the accumulated path constraints on each path.

Limitations

- **Path explosion**
- **Program-dependent efficiency**
- **Memory aliasing**
- **Arrays**
- **Environment interactions**

3.5 Control flow graphs

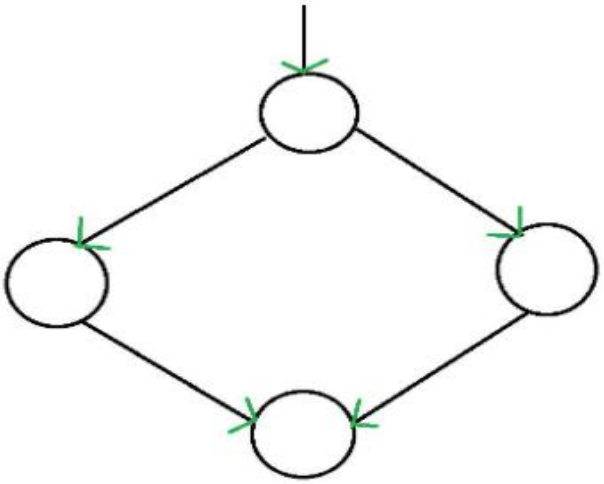
- A **Control Flow Graph (CFG)** is the graphical representation of control flow or computation during the execution of programs or applications.
- Control flow graphs are mostly used in static analysis as well as compiler applications, as they can accurately represent the flow inside of a program unit.

Characteristics of Control Flow Graph

- Control flow graph is process oriented.
- Control flow graph shows all the paths that can be traversed during a program execution.
- Control flow graph is a directed graph.
- Edges in CFG portray control flow paths and the nodes in CFG portray basic blocks.
- There exist 2 designated blocks in Control Flow Graph:

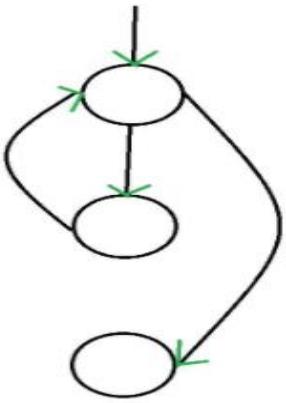
1. **Entry Block:**
   Entry block allows the control to enter into the control flow graph.
2. **Exit Block:**
   Control flow leaves through the exit block.

If-then-else



If-then-else

While loop



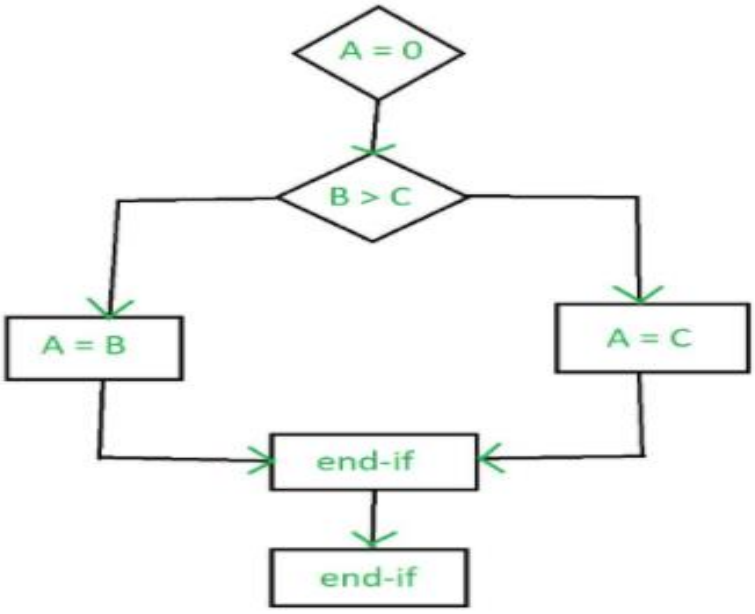
while

Example

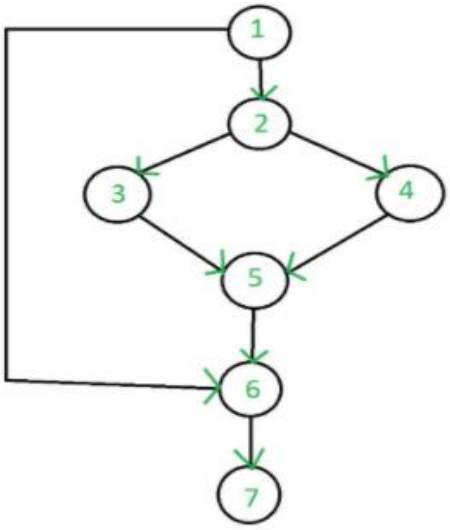
Example:

```
if  A = 10 then
  if B > C
     A = B
  else A = C
  endif
  endif
print A, B, C
```

Flow chart



Control flow graph



Control Flow Graph

Advantages of cfg

- It can easily encapsulate the information per each basic block.
- It can easily locate inaccessible codes of a program and syntactic structures such as loops are easy to find in a control flow graph.

3.6 Introduction to real time software systems

- The time-dependent nature of many real-time applications adds a new and potentially difficult element to the testing mix—time.
- Not only does the test case designer have to consider white- and black-box test cases but also event handling (i.e., interrupt processing), the timing of the data, and the parallelism of the tasks (processes) that handle the data
- In many situations, test data provided when a real time system is in one state will result in proper processing, while the same data provided when the system is in a different state may lead to error.
- The intimate relationship that exists between real-time software and its hardware environment can also cause testing problems.
- Software tests must consider the impact of hardware faults on software processing. Such faults can be extremely difficult to simulate realistically

Four-step strategy

- **Task testing**
- The first step in the testing of real-time software is to test each task independently.
- That is, white-box and black-box tests are designed and executed for each task.
- Each task is executed independently during these tests.

- Task testing uncovers errors in logic and function but not timing or behavior.
- **Behavioral testing**
- Using system models created with CASE tools, it is possible to simulate the behavior of a real-time system and examine its behavior as a consequence of external events.
- **Intertask testing**
- Once errors in individual tasks and in system behavior have been isolated, testing shifts to time-related errors. Asynchronous tasks that are known to communicate with one another are tested with different data rates and processing load to determine if intertask synchronization errors will occur
- **System testing**
- Most real-time systems process interrupts.
- Therefore, testing the handling of these Boolean events is essential.
- Using the state transition diagram and the control specification, the tester develops a list of all possible interrupts and the processing that occurs as a consequence of the interrupts

UNIT IV
SOFTWARE PROJECT MANAGEMENT

4.1 Management Functions and Processes

- Project management involves the planning, monitoring, and control of the people, process, and events that occur as software evolves from a preliminary concept to an operational implementation.
- Building computer software is a complex undertaking, particularly if it involves many people working over a relatively long time. That's why software projects need to be managed.

Project Management – 4 Ps

- People — the most important element of a successful project
- Product — the software to be built
- Process — the set of framework activities and software engineering tasks to get the job done
- Project — all work required to make the product a reality

Stakeholders

- *Senior managers* who define the business issues that often have significant influence on the project.
- *Project (technical) managers* who must plan, motivate, organize, and control the practitioners who do software work.
- *Practitioners* who deliver the technical skills that are necessary to engineer a product or application; often make poor team leaders
- *Customers* who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.
- *End-users* who interact with the software once it is released for production use.

The MOI Model of leadership for Team Leader:

- Motivation. The ability to encourage (by "push or pull") technical people to produce to their best ability.
- Organization. The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
- Ideas or innovation. The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Key traits of an effective project manager

- Problem solving
- Managerial identity
- Achievement
- Influence and team building

Responsibilities

- **Formal risk management**
- **Empirical cost and schedule estimation**
- **Metrics-based project management**
- **Tracking – amount of work done, costs, work remaining, etc…**
- **Defect tracking against quality targets**
- **People aware project management**

The product - Software Scope

- **Scope is defined by the following**
- **Context**
- **Information objectives**

- **Function and performance**

THE PROCESS

- Select the process model that is appropriate for the software to be engineered by a project team
- Some example paradigms –
- linear sequential model
- prototyping model
- RAD model
- Incremental model
- Spiral model

The project manager must decide which process model is most appropriate for

(1) the customers who have requested the product and the people who will do the work

(2) the characteristics of the product itself

(3) the project environment in which the software team works

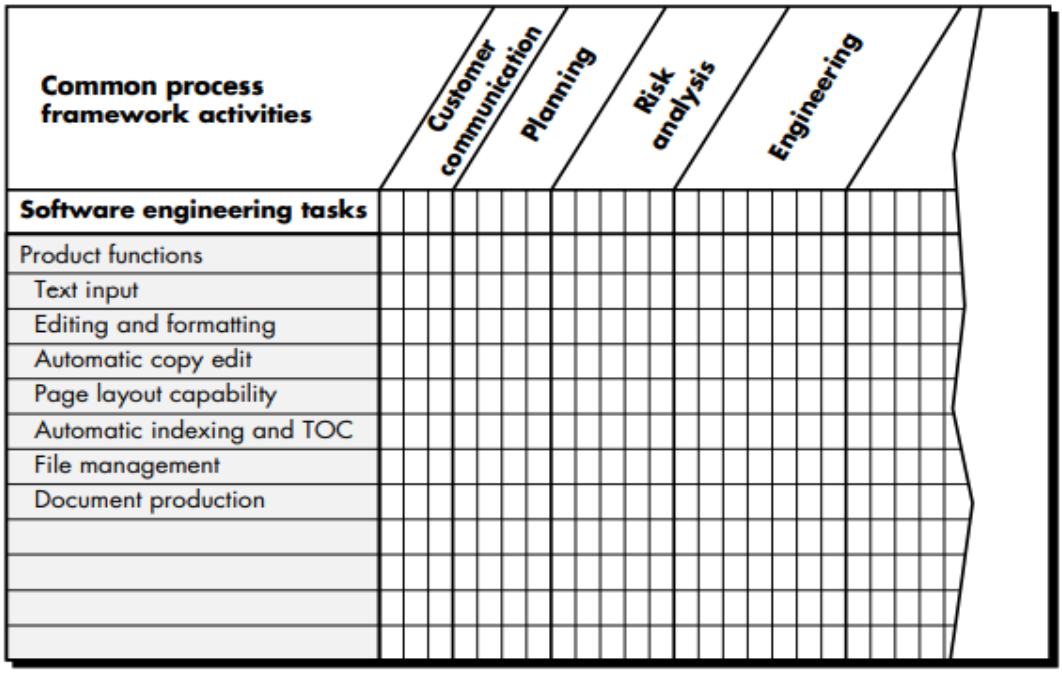
When a process model has been selected, the team then defines a preliminary project plan based on the set of common process framework activities.

Once the preliminary plan is established, process decomposition begins.

That is, a complete plan, reflecting the work tasks required to populate the framework activities must be created

Project planning begins with the melding of the product and the process.

- Each function to be engineered by the software team must pass through the set of framework activities that have been defined for a software organization
- Example set of framework activities:
- Customer communication
- Planning
- Risk analysis
- Engineering
- Construction and release
- Customer evaluation



Process Decomposition

Work tasks for the customer communication activity:

1. Develop list of clarification issues.
2. Meet with customer to address clarification issues.
3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required.

THE PROJECT

In order to manage a successful software project, we must understand what can go wrong

- 1.Software people don't understand their customer's needs.
- 2. The product scope is poorly defined.
- 3. Changes are managed poorly.
- 4. The chosen technology changes.
- 5. Business needs change [or are ill-defined].
- 6. Deadlines are unrealistic.
- 7. Users are resistant.
- 8. The project team lacks people with appropriate skills.
- 9.Managers [and practitioners] avoid best practices and lessons learned.

Commonsense approach to software projects

- Start on the right foot
- Maintain momentum
- Make smart decisions - the decisions of the project manager and the software team should be simple
- Conduct a postmortem analysis - Establish a consistent mechanism for extracting lessons learned for each project

4.2 Project planning and Control

- Planning involves estimation — how much money, how much effort, how many resources, and how much time it will take to build a specific software-based system or product
- Estimation begins with a description of the scope of the product.
- Until the scope is "bounded" it's not possible to develop a meaningful estimate.
- The problem is then decomposed into a set of smaller problems and each of these is estimated using historical data and experience as guides.
- It is advisable to generate your estimates using at least two different methods (as a cross check).
- Problem complexity and risk are considered before a final estimate is made

4.2.1 PROJECT PLANNING OBJECTIVES

- The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule.
- These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses.
- In addition, estimates should attempt to define best case and worst case scenarios so that project outcomes can be bounded.
- The planning objective is achieved through a process of information discovery that leads to reasonable estimates

Project Planning Task Set-I

- Establish project scope
- Determine feasibility
- Analyze risks
- Define required resources
    - Determine require human resources
    - Define reusable software resources
    - Identify environmental resources

Project Planning Task Set-II

- Estimate cost and effort
    - Decompose the problem
    - Develop two or more estimates using size, function points, process tasks or use-cases
    - Reconcile the estimates
- Develop a project schedule
    - Establish a meaningful task set
    - Define a task network
    - Use scheduling tools to develop a timeline chart
    - Define schedule tracking mechanisms

4.2.2 Project Control - Scheduling and tracking

- The software engineering tasks dictated by the software process model are refined for the functionality to be built.
- Effort and duration are allocated to each task and a task network (also called an "activity network") is created in a manner that enables the software team to meet the delivery deadline established**.**

Proper scheduling requires that
(1) all tasks appear in the network,
(2) effort and timing are intelligently allocated to each task,
(3) interdependencies between tasks are properly indicated,
(4) resources are allocated for the work to be done, and
(5) closely spaced milestones are provided so that progress can be tracked.

- Generalized project scheduling tools and techniques can be applied with little modification to software projects.
- Program evaluation and review technique (PERT) and critical path method (CPM) are two project scheduling methods that can be applied to software development.
- Both techniques are driven by information already developed in earlier project planning activities:
- 1)Estimates of effort
- 2) A decomposition of the product function
- 3) The selection of the appropriate process model and task set
- 4)Decomposition of tasks Interdependencies among tasks may be defined using a task network.
- Tasks, sometimes called the project work breakdown structure (WBS), are defined for the product as a whole or for individual functions.
- Both PERT and CPM provide quantitative tools that allow the software planner to
- (1) determine the critical path—the chain of tasks that determines the duration of the project;
- (2) establish "most likely" time estimates for individual tasks by applying statistical models;

- (3) calculate “boundary times” that define a time "window" for a particular task.

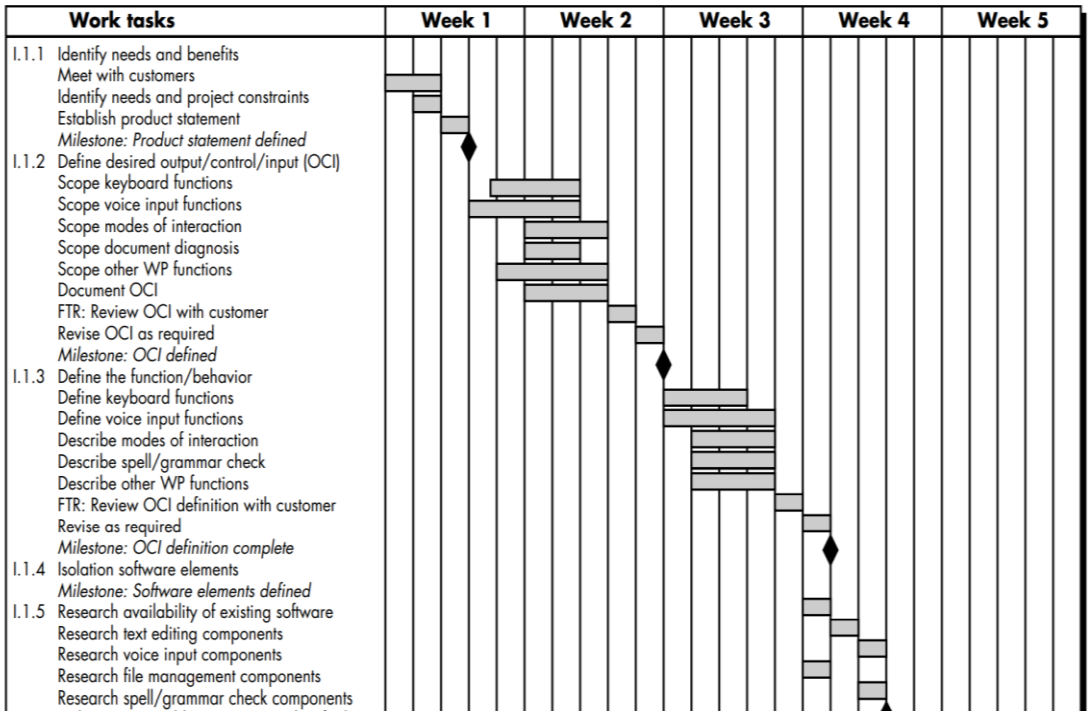
Timeline charts

- When creating a software project schedule, the planner begins with a set of tasks (the work breakdown structure).
- If automated tools are used, the work breakdown is input as a task network or task outline.
- Effort, duration, and start date are then input for each task. In addition, tasks may be assigned to specific individuals.
- As a consequence of this input, a timeline chart, also called a Gantt chart, is generated.
- A timeline chart can be developed for the entire project.
- Alternatively, separate charts can be developed for each project function or for each individual working on the project.

Project table

- Once the information necessary for the generation of a timeline chart has been input, the majority of software project scheduling tools produce project tables—
- a tabular listing of all project tasks,
- their planned and actual start- and end-dates,
- and a variety of related information

Gantt chart



Example project table

Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1 Identify needs and benefits							Scoping will require more effort/time
Meet with customers	wk1, d1	wk1, d1	wk1, d2	wk1, d2	BLS	2 p-d	
Identify needs and project constraints	wk1, d2	wk1, d2	wk1, d2	wk1, d2	JPP	1 p-d	
Establish product statement	wk1, d3	wk1, d3	wk1, d3	wk1, d3	BLS/JPP	1 p-d	
Milestone: Product statement defined	wk1, d3	wk1, d3	wk1, d3	wk1, d3			
I.1.2 Define desired output/control/input (OCI)							
Scope keyboard functions	wk1, d4	wk1, d4	wk2, d2		BLS	1.5 p-d	
Scope voice input functions	wk1, d3	wk1, d3	wk2, d2		JPP	2 p-d	
Scope modes of interaction	wk2, d1		wk2, d3		MLL	1 p-d	
Scope document diagnostics	wk2, d1		wk2, d2		BLS	1.5 p-d	
Scope other WP functions	wk1, d4	wk1, d4	wk2, d3		JPP	2 p-d	
Document OCI	wk2, d1		wk2, d3		MLL	3 p-d	
FTR: Review OCI with customer	wk2, d3		wk2, d3		all	3 p-d	
Revise OCI as required	wk2, d4		wk2, d4		all	3 p-d	
Milestone: OCI defined	wk2, d5		wk2, d5				
I.1.3 Define the function/behavior							

FIGURE 7.5 An example project table

Tracking the schedule

- The project schedule provides a road map for a software project manager.
- If it has been properly developed, the project schedule defines the tasks and milestones that must be tracked and controlled as the project proceeds.
- Tracking can be accomplished in a number of different ways:
- Conducting periodic project status meetings in which each team member reports progress and problems.
- Evaluating the results of all reviews conducted throughout the software engineering process.
- Determining whether formal project milestones have been accomplished by the scheduled date.
- Comparing actual start-date to planned start-date for each project task listed in the resource table
- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.
- Using earned value analysis to assess progress quantitatively

Control

- Control is employed by a software project manager to administer project resources, cope with problems, and direct project staff.
- If things are going well (i.e., the project is on schedule and within budget, reviews indicate that real progress is being made and milestones are being reached), control is light.
- But when problems occur, the project manager must exercise control to reconcile them as quickly as possible.
- After a problem has been diagnosed, additional resources may be focused on the problem area: staff may be redeployed or the project schedule can be redefined.

Time boxing

- When faced with severe deadline pressure, experienced project managers sometimes use a project scheduling and control technique called time-boxing
- The time-boxing strategy recognizes that the complete product may not be deliverable by the predefined deadline.
- Therefore, an incremental software paradigm is chosen and a schedule is derived for each incremental delivery.
- The tasks associated with each increment are then time-boxed.

- This means that the schedule for each task is adjusted by working backward from the delivery date for the increment.
- A “box” is put around each task.
- When a task hits the boundary of its time box (plus or minus 10 percent), work stops and the next task begins.

Earned value analysis

- Earned value is a measure of progress.
- It enables us to assess the “percent of completeness” of a project using quantitative analysis
- To determine the earned value, the following steps are performed:
- 1. The budgeted cost of work scheduled (BCWS) is determined for each work task represented in the schedule
- 2. The BCWS values for all work tasks are summed to derive the budget at completion, BAC. Hence, BAC = ∑(BCWSk) for all tasks k
- 3. Next, the value for budgeted cost of work performed (BCWP) is computed

Progress indicators

- Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:
- Schedule performance index, SPI = BCWP/BCWS
- Schedule variance, SV = BCWP – BCWS
- SPI is an indication of the efficiency with which the project is utilizing scheduled resources.
- An SPI value close to 1.0 indicates efficient execution of the project schedule.
- SV is simply an absolute indication of variance from the planned schedule.
- Percent scheduled for completion = BCWS/BAC provides an indication of the percentage of work that should have been completed by time t.
- Percent complete = BCWP/BAC provides a quantitative indication of the percent of completeness of the project at a given point in time, t.
- It is also possible to compute the actual cost of work performed, ACWP. The value for ACWP is the sum of the effort actually expended on work tasks that have been completed by a point in time on the project schedule.
- It is then possible to compute Cost performance index, CPI = BCWP/ACWP
- Cost variance, CV = BCWP – ACWP
- CPI value close to 1.0 provides a strong indication that the project is within its defined budget.
- CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project.

4.3 Organization and Intra-team Communication

Factors Affecting Team Organization

- Difficulty of problem to be solved
- Size of resulting program
- Team lifetime
- Degree to which problem can be modularized
- Required quality and reliability of the system to be built
- Rigidity of the delivery date

- Degree of communication required for the project

Agile Teams

- Teams have significant autonomy to make their own project management and technical decisions
- Planning kept to minimum and is constrained only by business requirements and organizational standards
- Team self-organizes as project proceeds to maximum contributes of each individual's talents
- May conduct daily (10 – 20 minute) meeting to synchronized and coordinate each day's work
    - What has been accomplished since the last meeting?
    - What needs to be accomplished by the next meeting?
    - How will each team member contribute to accomplishing what needs to be done?
    - What roadblocks exist that have to be overcome?

Coordination and Communication Issues

- Formal, impersonal approaches (e.g. documents, milestones, memos)
- Formal interpersonal approaches (e.g. review meetings, inspections)
- Informal interpersonal approaches (e.g. information meetings, problem solving)
- Electronic communication (e.g. e-mail, bulletin boards, video conferencing)
- Interpersonal networking (e.g. informal discussion with people other than project team members)

4.4 Risk Management

Definition of Risk

- A risk is a potential problem – it might happen and it might not
- Conceptual definition of risk
    - Risk concerns future happenings
    - Risk involves change in mind, opinion, actions, places, etc.
    - Risk involves choice and the uncertainty that choice entails
- Two characteristics of risk
    - Uncertainty – the risk may or may not happen, that is, there are no 100% risks (those, instead, are called constraints)
    - Loss – the risk becomes a reality and unwanted consequences or losses occur

Risk Categorization

- **Project risks**
    - They threaten the project plan
    - If they become real, it is likely that the project schedule will slip and that costs will increase

- **Technical risks**
  - They threaten the quality and timeliness of the software to be produced
  - If they become real, implementation may become difficult or impossible
- **Business risks**
  - They threaten the viability of the software to be built
  - If they become real, they jeopardize the project or the product
- **Sub-categories of Business risks**
  - **Market risk** – building an excellent product or system that no one really wants
  - **Strategic risk** – building a product that no longer fits into the overall business strategy for the company
  - **Sales risk** – building a product that the sales force doesn't understand how to sell
  - **Management risk** – losing the support of senior management due to a change in focus or a change in people
  - **Budget risk** – losing budgetary or personnel commitment

Risk Categorization – Another Approach

- Known risks
  - Those risks that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date)
- Predictable risks
  - Those risks that are extrapolated from past project experience (e.g., past turnover)
- Unpredictable risks
  - Those risks that can and do occur, but are extremely difficult to identify in advance

Reactive vs. Proactive Risk Strategies

- Reactive risk strategies
  - "Don't worry, I'll think of something"
  - The majority of software teams and managers rely on this approach
  - Nothing is done about risks until something goes wrong
    - The team then flies into action in an attempt to correct the problem rapidly (fire fighting)
  - Crisis management is the choice of management techniques
- Proactive risk strategies
  - Steps for risk management are followed (see next slide)
  - Primary objective is to avoid risk and to have a contingency plan in place to handle unavoidable risks in a controlled and effective manner

Steps for Risk Management

1) Identify possible risks; recognize what can go wrong
2) Analyze each risk to estimate the probability that it will occur and the impact (i.e., damage) that it will do if it does occur
3) Rank the risks by probability and impact
   - Impact may be negligible, marginal, critical, and catastrophic
4) Develop a contingency plan to manage those risks having high probability and high impact

Risk Identification

- Risk identification is a systematic attempt to specify threats to the project plan

- By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary
- Generic risks
  - Risks that are a potential threat to every software project
- Product-specific risks
  - Risks that can be identified only by those a with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built
  - This requires examination of the project plan and the statement of scope
  - "What special characteristics of this product may threaten our project plan?"

Risk Item Checklist

- Used as one way to identify risks
- Focuses on known and predictable risks in specific subcategories (see next slide)
- Can be organized in several ways
  - A list of characteristics relevant to each risk subcategory
  - Questionnaire that leads to an estimate on the impact of each risk
  - A list containing a set of risk component and drivers and their probability of occurrence

Known and Predictable Risk Categories

- Product size – risks associated with overall size of the software to be built
- Business impact – risks associated with constraints imposed by management or the marketplace
- Customer characteristics – risks associated with sophistication of the customer and the developer's ability to communicate with the customer in a timely manner
- Process definition – risks associated with the degree to which the software process has been defined and is followed
- Development environment – risks associated with availability and quality of the tools to be used to build the project
- Technology to be built – risks associated with complexity of the system to be built and the "newness" of the technology in the system
- Staff size and experience – risks associated with overall technical and project experience of the software engineers who will do the work

Questionnaire on Project Risk

1) Have top software and customer managers formally committed to support the project?
2) Are end-users enthusiastically committed to the project and the system/product to be built?
3) Are requirements fully understood by the software engineering team and its customers?
4) Have customers been involved fully in the definition of requirements?
5) Do end-users have realistic expectations?
6) Is the project scope stable?
7) Does the software engineering team have the right mix of skills?
8) Are project requirements stable?
9) Does the project team have experience with the technology to be implemented?
10) Is the number of people on the project team adequate to do the job?
11) Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

Risk Components and Drivers

- The project manager identifies the risk drivers that affect the following risk components

  - **Performance risk** - the degree of uncertainty that the product will meet its requirements and be fit for its intended use
  - **Cost risk** - the degree of uncertainty that the project budget will be maintained
  - **Support risk** - the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance
  - **Schedule risk** - the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time
- The impact of each risk driver on the risk component is divided into one of four impact levels
  - Negligible, marginal, critical, and catastrophic
- Risk drivers can be assessed as impossible, improbable, probable, and frequent

Risk Projection (Estimation)

- Risk projection (or estimation) attempts to rate each risk in two ways
  - The probability that the risk is real
  - The consequence of the problems associated with the risk, should it occur
- The project planner, managers, and technical staff perform four risk projection steps (see next slide)
- The intent of these steps is to consider risks in a manner that leads to prioritization

Be prioritizing risks, the software team can allocate limited resources where they will have the most impact

Risk Projection/Estimation Steps

1) Establish a scale that reflects the perceived likelihood of a risk (e.g., 1-low, 10-high)
2) Delineate the consequences of the risk
3) Estimate the impact of the risk on the project and product
4) Note the overall accuracy of the risk projection so that there will be no misunderstandings

Contents of a Risk Table

- A risk table provides a project manager with a simple technique for risk projection
- It consists of five columns
  - Risk Summary – short description of the risk
  - Risk Category – one of seven risk categories (slide 12)
  - Probability – estimation of risk occurrence based on group input
  - Impact – (1) catastrophic (2) critical (3) marginal (4) negligible
  - RMMM – Pointer to a paragraph in the Risk Mitigation, Monitoring, and management Plan

Risk Summary	Risk Category	Probability	Impact (1-4)	RMMM

Developing a Risk Table

- List all risks in the first column (by way of the help of the risk item checklists)
- Mark the category of each risk

- Estimate the probability of each risk occurring
- Assess the impact of each risk based on an averaging of the four risk components to determine an overall impact value (See next slide)
- Sort the rows by probability and impact in descending order
- Draw a horizontal cutoff line in the table that indicates the risks that will be given further attention

Assessing Risk Impact

- Three factors affect the consequences that are likely if a risk does occur
  - **Its nature** – This indicates the problems that are likely if the risk occurs
  - **Its scope** – This combines the severity of the risk (how serious was it) with its overall distribution (how much was affected)
  - **Its timing** – This considers when and for how long the impact will be felt
- The overall risk exposure formula is RE = P x C
  - P = the probability of occurrence for a risk
  - C = the cost to the project should the risk actually occur
- Example
  - P = 80% probability that 18 of 60 software components will have to be developed
  - C = Total cost of developing 18 components is $25,000
  - RE = .80 x $25,000 = $20,000

Risk Mitigation, Monitoring, and Management

- An effective strategy for dealing with risk must consider three issues (Note: these are not mutually exclusive)
  - Risk mitigation (i.e., avoidance)
  - Risk monitoring
  - Risk management and contingency planning
- Risk mitigation (avoidance) is the primary strategy and is achieved through a plan

Example: Risk of high staff turnover

- Strategy for Reducing Staff Turnover
  - ❑ Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market)
  - ❑ Mitigate those causes that are under our control before the project starts
  - ❑ Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave
  - ❑ Organize project teams so that information about each development activity is widely dispersed
  - ❑ Define documentation standards and establish mechanisms to ensure that documents are developed in a timely manner
  - ❑ Conduct peer reviews of all work (so that more than one person is "up to speed")
  - ❑ Assign a backup staff member for every critical technologist
- During risk monitoring, the project manager monitors factors that may provide an indication of whether a risk is becoming more or less likely
- Risk management and contingency planning assume that mitigation efforts have failed and that the risk has become a reality
- RMMM steps incur additional project cost
  - Large projects may have identified 30 – 40 risks
- Risk is not limited to the software project itself
  - Risks can occur after the software has been delivered to the user

- Software safety and hazard analysis
  - These are software quality assurance activities that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail
  - If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards

The RMMM Plan

- The RMMM plan may be a part of the software development plan (Paragraph 5.19.1) or may be a separate document
- Once RMMM has been documented and the project has begun, the risk mitigation, and monitoring steps begin
  - Risk mitigation is a problem avoidance activity
  - Risk monitoring is a project tracking activity
- Risk monitoring has three objectives
  - To assess whether predicted risks do, in fact, occur
  - To ensure that risk aversion steps defined for the risk are being properly applied
  - To collect information that can be used for future risk analysis
- The findings from risk monitoring may allow the project manager to ascertain what risks caused which problems throughout the project

Seven Principles of Risk Management

- **Maintain a global perspective**
  - View software risks within the context of a system and the business problem that is is intended to solve
- **Take a forward-looking view**
  - Think about risks that may arise in the future; establish contingency plans
- **Encourage open communication**
  - Encourage all stakeholders and users to point out risks at any time
- **Integrate risk management**
  - Integrate the consideration of risk into the software process
- **Emphasize a continuous process of risk management**
  - Modify identified risks as more becomes known and add new risks as better insight is achieved
- **Develop a shared product vision**
  - A shared vision by all stakeholders facilitates better risk identification and assessment
- **Encourage teamwork when managing risk**
  - Pool the skills and experience of all stakeholders when conducting risk management activities

4.5 Software Cost Estimation

- Estimation of resources, cost, and schedule for a software engineering effort requires experience
  - access to good historical information (metrics)
  - the courage to commit to quantitative predictions when qualitative information is all that exists
- Estimation carries inherent risk and this risk leads to uncertainty

4.5.1 Underlying factors of critical concern

Software Project Estimation Effective software project estimation is one of the most challenging and important activities in software development. Proper project planning and control is not possible without a sound and reliable estimate. As a whole, the software industry doesn't estimate projects well and doesn't use estimates appropriately. We suffer far more than we should as a result and we need to focus some effort on improving the situation. Under-estimating a project leads to under-staffing it (resulting in staff burnout), under-scoping the quality assurance effort (running the risk of low quality deliverables), and setting too short a schedule (resulting in loss of credibility as deadlines are missed). For those who figure on avoiding this situation by generously padding the estimate, over-estimating a project can be just about as bad for the organization! If you give a project more resources than it really needs without sufficient scope controls it will use them. The project is then likely to cost more than it should (a negative impact on the bottom line), take longer to deliver than necessary (resulting in lost opportunities), and delay the use of your resources on the next project.

4.5.2 Metrics for estimating costs of software products

The four basic steps in software project estimation are: 1) Estimate the size of the development product. This generally ends up in either Lines of Code (LOC) or Function Points (FP), but there are other possible units of measure. A discussion of the pros & cons of each is discussed in some of the material referenced at the end of this report. 2) Estimate the effort in person-months or person-hours. 3) Estimate the schedule in calendar months. 4) Estimate the project cost in dollars (or local currency)

Estimating size

An accurate estimate of the size of the software to be built is the first step to an effective estimate. Your source(s) of information regarding the scope of the project should, wherever possible, start with formal descriptions of the requirements - for example, a customer's requirements specification or request for proposal, a system specification, a software requirements specification. If you are estimating a project in later phases of the project's lifecycle, design documents can be used to provide additional detail. Don't let the lack of a formal scope specification stop you from doing an initial project estimate. A verbal description or a whiteboard outline are sometimes all you have to start with. In any case, you must communicate the level of risk and uncertainty in an estimate to all concerned and you must re-estimate the project as soon as more scope information is determined.

Estimating effort

Once you have an estimate of the size of your product, you can derive the effort estimate. This conversion from software size to total project effort can only be done if you have a defined software development lifecycle and development process that you follow to specify, design, develop, and test the software. A software development project involves far more than simply coding the software – in fact, coding is often the smallest part of the overall effort. Writing and reviewing documentation, implementing prototypes, designing the deliverables, and reviewing and testing the code take up the larger portion of overall project effort. The project effort estimate requires you to identify and estimate, and then sum up all the activities you must perform to build a product of the estimated size. There are two main ways to derive effort from size:

1) The best way is to use your organization's own historical data to determine how much effort previous projects of the estimated size have taken. This, of course, assumes (a) your organization has been documenting actual results from previous projects, (b) that you have at least one past project of similar size (it is even better if you have several projects of similar size as this reinforces that you consistently need a certain level of effort to develop projects of a given size), and (c) that you will follow a similar development lifecycle, use a similar development methodology, use similar tools, and use a team with similar skills and experience for the new project.

2) If you don't have historical data from your own organization because you haven't started collecting it yet or because your new project is very different in one or more key aspects, you can use a mature and generally accepted algorithmic approach such as Barry Boehm's COCOMO model or the Putnam Methodology to convert a size estimate into an effort estimate. These models have been derived by studying a significant number of completed projects from various organizations to see how their project sizes mapped into total project effort. These "industry data" models may not be as accurate as your own historical data, but they can give you useful ballpark effort estimates.

Estimating schedule

The third step in estimating a software development project is to determine the project schedule from the effort estimate. This generally involves estimating the number of people who will work on the project, what they will work on (the Work Breakdown Structure), when they will start working on the project and when they will finish (this is the "staffing profile"). Once you have this information, you need to lay it out into a calendar schedule. Again, historical data from your organization's past projects or industry data models can be used to predict the number of people you will need for a project of a given size and how work can be broken down into a schedule.

- Decomposition techniques use a "divide and conquer" approach to software project estimation
- Empirical estimation models can be used to offer a potentially valuable estimation approach in their own right. A model is based on experience (historical data) and takes the form d = f (vi) where d is one of a number of estimated values (e.g., effort, cost, project duration) and vi are selected independent parameters (e.g., estimated LOC or FP).
- Automated estimation tools implement one or more decomposition techniques or empirical models

DECOMPOSITION TECHNIQUES

- Software Sizing

- "Fuzzy logic" sizing

- Function point sizing

- Change sizing

- Problem-Based Estimation

- Process-Based Estimation

Problem-Based Estimation

- Regardless of the estimation variable that is used, the project planner begins by estimating a range of values for each function or information domain value.
- Using historical data or (when all else fails) intuition, the planner estimates an optimistic, most likely, and pessimistic size value for each function or count for each information domain value.
- A three-point or expected value can then be computed.
- The expected value for the estimation variable (size), S, can be computed as a weighted average of the optimistic (sopt), most likely (sm), and pessimistic (spess) estimates.
- For example, $S = (s_{opt} + 4s_m + s_{pess})/6$ gives heaviest credence to the "most likely" estimate and follows a beta probability distribution.

Process-Based Estimation

Activity →	CC	Planning	Risk analysis	Engineering		Construction release		CE	Totals
Task →				Analysis	Design	Code	Test		
Function ↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication CE = customer evaluation

EMPIRICAL ESTIMATION MODELS

- An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC or FP
- The empirical data that support most estimation models are derived from a limited sample of projects
- A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form $E = A + B \times (ev)^C$ where A, B, and C are empirically derived constants, E is effort in person-months, and ev is the estimation variable (either LOC or FP)

Loc and FP oriented models

$E = 5.2 \times (KLOC)^{0.91}$	Walston-Felix model
$E = 5.5 + 0.73 \times (KLOC)^{1.16}$	Bailey-Basili model
$E = 3.2 \times (KLOC)^{1.05}$	Boehm simple model
$E = 5.288 \times (KLOC)^{1.047}$	Doty model for KLOC > 9

- FP-oriented models have also been proposed. These include

$E = -13.39 + 0.0545\ FP$	Albrecht and Gaffney model
$E = 60.62 \times 7.728 \times 10^{-8}\ FP^3$	Kemerer model
$E = 585.7 + 15.12\ FP$	Matson, Barnett, and Mellichamp model

4.5.3 Metrics for estimating costs of software products – Function points

- Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value.
- Since 'functionality' cannot be measured directly, it must be derived indirectly using other direct measures.
- Function-oriented metrics were first proposed by Albrecht, who suggested a measure called the function point.
- Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

		Weighting factor			
Measurement parameter	**Count**	**Simple**	**Average**	**Complex**	
Number of user inputs		× 3	4	6	=
Number of user outputs		× 4	5	7	=
Number of user inquiries		× 3	4	6	=
Number of files		× 7	10	15	=
Number of external interfaces		× 5	7	10	=
Count total	→				

- Five information domain characteristics are determined and counts are provided in the appropriate table location. Information domain values are defined in the following manner:
- **Number of user inputs**. Each user input that provides distinct applicationoriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

- **Number of user outputs**. Each user output that provides applicationoriented information to the user is counted. In this context output refers to reports, screens, error messages, etc.
- **Number of user inquiries**. An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.
- **Number of files**. Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.
- **Number of external interfaces**. All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.
  - ✓ Once these data have been collected, a complexity value is associated with each count.
  - ✓ Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex.
  - ✓ Nonetheless, the determination of complexity is somewhat subjective.
  - ✓ To compute function points (FP), the following relationship is used:
  - ✓ FP = count total [0.65 + 0.01 ∑(Fi)] where count total is the sum of all FP entries
  - ✓ The Fi (i = 1 to 14) are "complexity adjustment values" based on responses to the following questions
  - ✓ 1. Does the system require reliable backup and recovery?
  - ✓ 2. Are data communications required?
  - ✓ 3. Are there distributed processing functions?
  - ✓ 4. Is performance critical?
  - ✓ 5. Will the system run in an existing, heavily utilized operational environment?
  - ✓ 6. Does the system require on-line data entry?
  - ✓ 7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
  - ✓ 8. Are the master files updated on-line?
  - ✓ 9. Are the inputs, outputs, files, or inquiries complex?
  - ✓ 10. Is the internal processing complex?
  - ✓ 11. Is the code designed to be reusable?
  - ✓ 12. Are conversion and installation included in the design?
  - ✓ 13. Is the system designed for multiple installations in different organizations?

- ✓ 14. Is the application designed to facilitate change and ease of use by the user?

4.6 Techniques for software cost estimation

4.6.1 Expert Judgment

Expert Judgment is a technique in which judgment is provided based upon a specific set of criteria and/or expertise that has been acquired in a specific knowledge area, application area, or product area, a particular discipline, an industry, etc. Such expertise may be provided by any group or person with specialized education, knowledge, skill, experience, or training.[1]. This knowledge base can be provided by a member of the project team, or multiple members of the project team, or by a team leader or team leaders. However, typically expert judgment requires an expertise that is not present within the project team and, as such, it is common for an external group or person with a specific relevant skill set or knowledge base to be brought in for a consultation,
Such expertise can be provided by any group or individual with specialized knowledge or training and is available from many sources, including:

- Units within the organization;
- Consultants;
- Stakeholders, including customers or sponsors;
- Professional and technical associations;
- Industry groups;
- Subject matter experts (SME);
- Project management office (PMO);
- Suppliers.

Application

Expert Judgment is use for situations which require recourse to expert judgment by completing, validating, interpreting and integrating existing data, assessing the impact of a change, predicting the occurrence of future events and the consequences of a decision, determining the present state of knowledge in one field, providing the elements needed for decision-making in the presence of several options.

Procedures

1. Select and confirm activity to be analyzed;
2. Create a list of statements/questions;
3. Select the experts;
4. Have the experts give their ratings/answers/etc.;
5. Make a report - send it out to everyone;
6. Have the experts revise their answers;
7. Make the second report.

Instructions

Create a contact list & skill inventory for each stakeholder on the subject expertise and make sure you have adequate communication system in place to contact experts on time. Ensure you seek expert judgment at appropriate time.

4.6.2 Delphi cost estimation

The **delphi technique** is the most used tool in securing Expert Judgment. Under this method the group's estimates are returned to the individual experts for review and a second round of forecasts is received from the experts. With each round the degree of consensus improves. The use of the delphi technique helps to reduce biased decisions

Other tools can be used, as:

Interviews. This tool is best used when knowledgeable, experienced people are available at an affordable cost and specific information is needed. Interview can be on a one-to-one or a many-to-one basis wherein conducted by asking a series of questions that will increase your knowledge of the project or a particular project activity.

Brainstorming. It's the kind of expert judgment tool is usually best use when input from multiple experts is needed or when experienced people aren't available. Brainstorming works by getting a group to focus on a problem and then coming up with as many solutions as possible. Once the session has resulted in a number of solutions, the results can be analyzed.

Historical data. Is best used when records are accurate and both projects are similar. Since a variety of documentations in project management matters most, so, it is very nice to have historical record data in every expert judgment to ensure that you are in line with what really it should be. Historical data uses the knowledge gained on a similar past project.

Delphi Method is a structured communication technique, originally developed as a systematic, interactive forecasting method which relies on a panel of experts. The experts answer questionnaires in two or more rounds. After each round, a facilitator provides an anonymous summary of the experts' forecasts from the previous round with the reasons for their judgments. Experts are then encouraged to revise their earlier answers in light of the replies of other members of the panel.

It is believed that during this process the range of answers will decrease and the group will converge towards the "correct" answer. Finally, the process is stopped after a predefined stop criterion (e.g. number of rounds, achievement of consensus, and stability of results) and the mean or median scores of the final rounds determine the results.

Delphi Method was developed in the 1950-1960s at the RAND Corporation.

Wideband Delphi Technique

In the 1970s, Barry Boehm and John A. Farquhar originated the Wideband Variant of the Delphi Method. The term "wideband" is used because, compared to the Delphi Method, the Wideband Delphi Technique involved greater interaction and more communication between the participants.

In Wideband Delphi Technique, the estimation team comprise the project manager, moderator, experts, and representatives from the development team, constituting a 3-7 member team. There are two meetings −

- Kickoff Meeting
- Estimation Meeting

Wideband Delphi Technique – Steps

Step 1 − Choose the Estimation team and a moderator.

Step 2 − The moderator conducts the kickoff meeting, in which the team is presented with the problem specification and a high level task list, any assumptions or project constraints. The team discusses on the problem and estimation issues, if any. They also decide on the units of estimation. The moderator guides the entire discussion, monitors time and after the kickoff meeting, prepares a structured document containing problem specification, high level task list, assumptions, and the units of estimation that are decided. He then forwards copies of this document for the next step.

Step 3 − Each Estimation team member then individually generates a detailed WBS, estimates each task in the WBS, and documents the assumptions made.

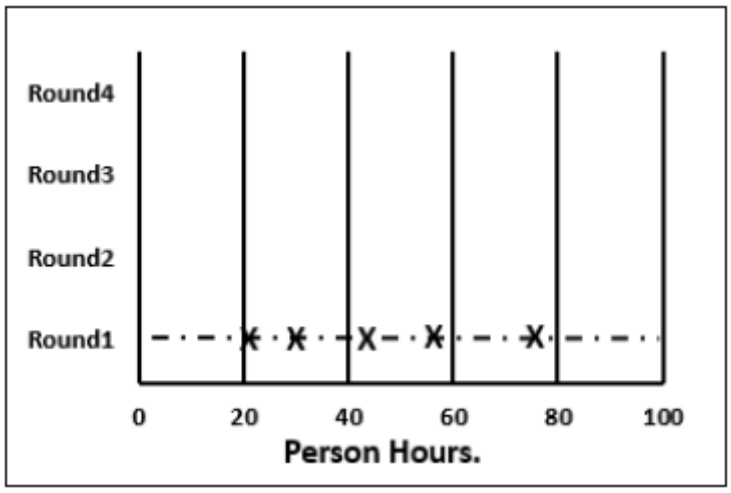
Wideband Delphi Estimation Sheet						
Project: <Project Name>			**Estimation Units:**	Person Hours		
Estimation Team Member:	<Name>			**Date:**	<MM-DD-YY>	
Task	**Initial Estimate**	**Change 1**	**Change 2**	**Change 3**	**Change 4**	**Final**
Task1	n_1					
Task2	n_2					
Task3	n_3					
Task4	n_4					
Task5	n_5					
Task6	n_6					
Task7	n_7					
Task8	n_8					
Net Change						
Total	$\sum n_i$					

Step 4 − The moderator calls the Estimation team for the Estimation meeting. If any of the Estimation team members respond saying that the estimates are not ready, the moderator gives more time and resends the Meeting Invite.

Step 5 − The entire Estimation team assembles for the estimation meeting.

Step 5.1 − At the beginning of the Estimation meeting, the moderator collects the initial estimates from each of the team members.

Step 5.2 − He then plots a chart on the whiteboard. He plots each member's total project estimate as an X on the Round 1 line, without disclosing the corresponding names. The Estimation team gets an idea of the range of estimates, which initially may be large.



Step 5.3 − Each team member reads aloud the detailed task list that he/she made, identifying any assumptions made and raising any questions or issues. The task estimates are not disclosed.

The individual detailed task lists contribute to a more complete task list when combined.

Step 5.4 − The team then discusses any doubt/problem they have about the tasks they have arrived at, assumptions made, and estimation issues.

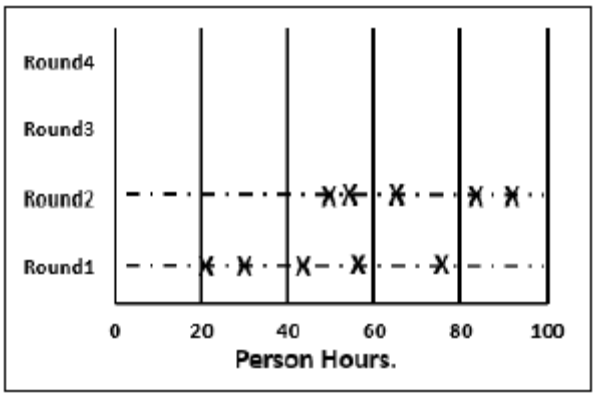
Step 5.5 − Each team member then revisits his/her task list and assumptions, and makes changes if necessary. The task estimates also may require adjustments based on the discussion, which are noted as +N Hrs. for more effort and –N Hrs. for less effort.

The team members then combine the changes in the task estimates to arrive at the total project estimate.

Wideband Delphi Estimation Sheet						
Project: <Project Name>			**Estimation Units:**	Person Hours		
Estimation Team Member:	<Name>			**Date:**	<MM-DD-YY>	
Task	**Initial Estimate**	**Change 1**	**Change 2**	**Change 3**	**Change 4**	**Final**
Task1	n_1	-1				
Task2	n_2	-2				
Task3	n_3	-4				
Task4	n_4	5				
Task5	n_5	0				
Task6	n_6	0				
Task7	n_7	2				
Task8	n_8	-3				
Net Change		**-3**				
Total	$\sum n_i$	$\sum n_i$-3				

Step 5.6 − The moderator collects the changed estimates from all the team members and plots them on the Round 2 line.

In this round, the range will be narrower compared to the earlier one, as it is more consensus based.



Step 5.7 − The team then discusses the task modifications they have made and the assumptions.

Step 5.8 − Each team member then revisits his/her task list and assumptions, and makes changes if necessary. The task estimates may also require adjustments based on the discussion.

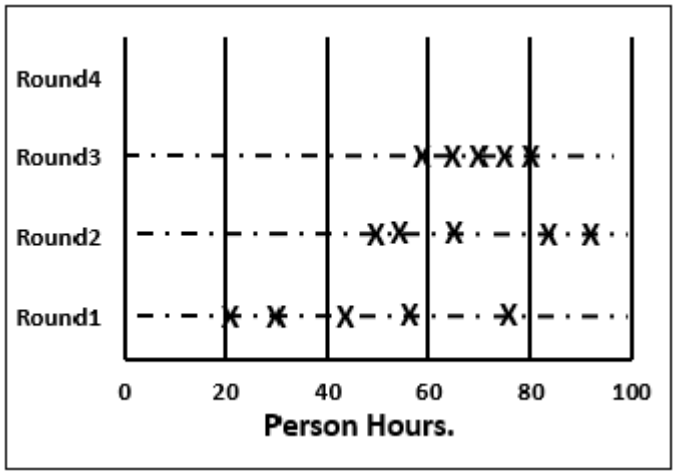
The team members then once again combine the changes in the task estimate to arrive at the total project estimate.

Step 5.9 − The moderator collects the changed estimates from all the members again and plots them on the Round 3 line.

Again, in this round, the range will be narrower compared to the earlier one.

Step 5.10 − Steps 5.7, 5.8, 5.9 are repeated till one of the following criteria is met −

- Results are converged to an acceptably narrow range.
- All team members are unwilling to change their latest estimates.
- The allotted Estimation meeting time is over.



Step 6 − The Project Manager then assembles the results from the Estimation meeting.

Step 6.1 − He compiles the individual task lists and the corresponding estimates into a single master task list.

Step 6.2 − He also combines the individual lists of assumptions.

Step 6.3 − He then reviews the final task list with the Estimation team.

Advantages and Disadvantages of Wideband Delphi Technique

Advantages

- Wideband Delphi Technique is a consensus-based estimation technique for estimating effort.
- Useful when estimating time to do a task.
- Participation of experienced people and they individually estimating would lead to reliable results.
- People who would do the work are making estimates thus making valid estimates.
- Anonymity maintained throughout makes it possible for everyone to express their results confidently.
- A very simple technique.
- Assumptions are documented, discussed and agreed.

Disadvantages

- Management support is required.
- The estimation results may not be what the management wants to hear.

4.7 Work break-down structure and Process break-down structure

Work Breakdown Structure (WBS), in Project Management and Systems Engineering, is a deliverable-oriented decomposition of a project into smaller components. WBS is a key project deliverable that organizes the team's work into manageable sections. The Project Management Body of Knowledge (PMBOK) defines WBS as a "deliverable oriented hierarchical decomposition of the work to be executed by the project team."

WBS element may be a product, data, service, or any combination thereof. WBS also provides the necessary framework for detailed cost estimation and control along with providing guidance for schedule development and control.

Representation of WBS

WBS is represented as a hierarchical list of project's work activities. There are two formats of WBS −

- Outline View (Indented Format)
- Tree Structure View (Organizational Chart)

Let us first discuss how to use the outline view for preparing a WBS.

Outline View

The outline view is a very user-friendly layout. It presents a good view of the entire project and allows easy modifications as well. It uses numbers to record the various stages of a project. It looks somewhat similar to the following −

- **Software Development**

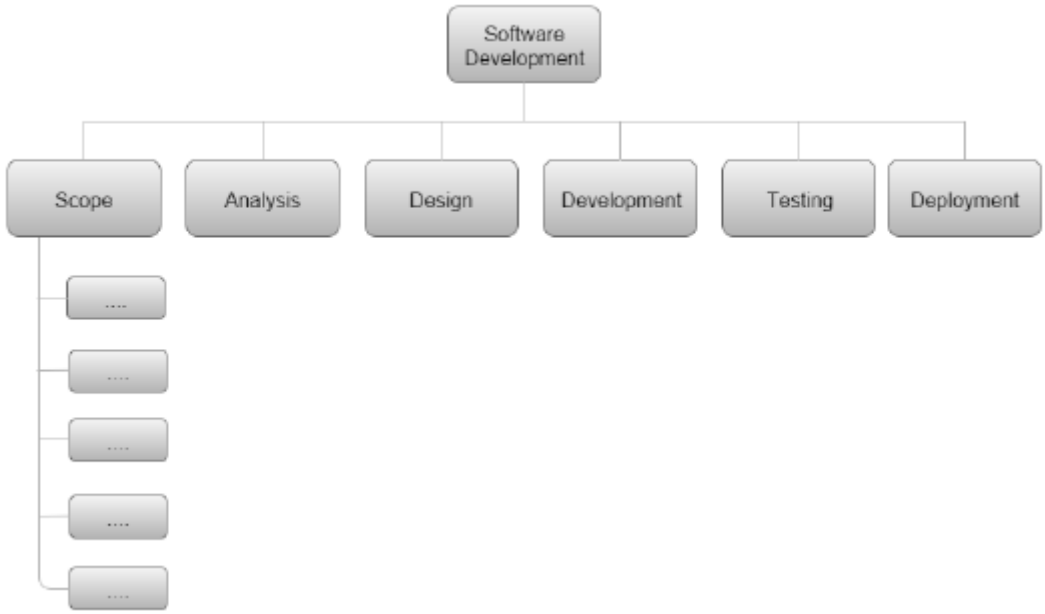
- **Scope**
  - Determine project scope
  - Secure project sponsorship
  - Define preliminary resources
  - Secure core resources
  - Scope complete
- **Analysis/Software Requirements**
  - Conduct needs analysis
  - Draft preliminary software specifications
  - Develop preliminary budget
  - Review software specifications/budget with the team
  - Incorporate feedback on software specifications
  - Develop delivery timeline
  - Obtain approvals to proceed (concept, timeline, and budget)
  - Secure required resources
  - Analysis complete
- **Design**
  - Review preliminary software specifications
  - Develop functional specifications
  - Obtain approval to proceed
  - Design complete
- **Development**
  - Review functional specifications
  - Identify modular/tiered design parameters
  - Develop code
  - Developer testing (primary debugging)
  - Development complete
- **Testing**
  - Develop unit test plans using product specifications
  - Develop integration test plans using product specifications
- **Training**
  - Develop training specifications for end-users
  - Identify training delivery methodology (online, classroom, etc.)
  - Develop training materials
  - Finalize training materials
  - Develop training delivery mechanism
  - Training materials complete
- **Deployment**

- Determine final deployment strategy
- Develop deployment methodology
- Secure deployment resources
- Train support staff
- Deploy software
- Deployment complete

Let us now take a look at the tree structure view.

Tree Structure View

The Tree Structure View presents a very easy-to-understand view of the entire project. The following illustration shows how a tree structure view looks like. This type of organizational chart structure can be easily drawn with the features available in MS-Word.



Types of WBS

There are two types of WBS −

- **Functional WBS** − In functional WBS, the system is broken based on the functions in the application to be developed. This is useful in estimating the size of the system.
- **Activity WBS** − In activity WBS, the system is broken based on the activities in the system. The activities are further broken into tasks. This is useful in estimating effort and schedule in the system.

Estimate Size

Step 1 − Start with functional WBS.

Step 2 − Consider the leaf nodes.

Step 3 − Use either Analogy or Wideband Delphi to arrive at the size estimates.

Estimate Effort

Step 1 − Use Wideband Delphi Technique to construct WBS. We suggest that the tasks should not be more than 8 hrs. If a task is of larger duration, split it.

Step 2 − Use Wideband Delphi Technique or Three-point Estimation to arrive at the Effort Estimates for the Tasks.

Scheduling

Once the WBS is ready and the size and effort estimates are known, you are ready for scheduling the tasks.

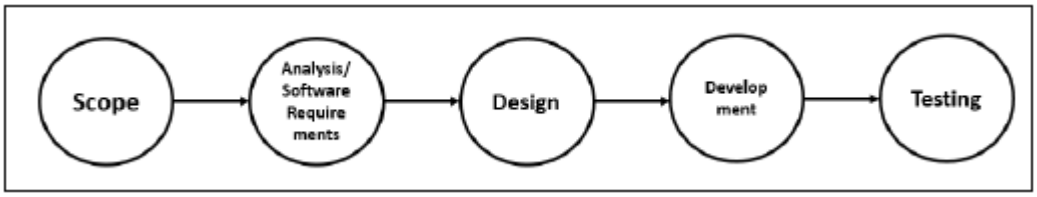
While scheduling the tasks, certain things should be taken into account −

- **Precedence** − A task that must occur before another is said to have precedence of the other.
- **Concurrence** − Concurrent tasks are those that can occur at the same time (in parallel).
- **Critical Path** − Specific set of sequential tasks upon which the project completion date depends.
    - All projects have a critical path.
    - Accelerating non-critical tasks do not directly shorten the schedule.

Critical Path Method

Critical Path Method (CPM) is the process for determining and optimizing the critical path. Non-critical path tasks can start earlier or later without impacting the completion date.

Please note that critical path may change to another as you shorten the current one. For example, for WBS in the previous figure, the critical path would be as follows −



As the project completion date is based on a set of sequential tasks, these tasks are called critical tasks.

The project completion date is not based on the training, documentation and deployment. Such tasks are called non-critical tasks.

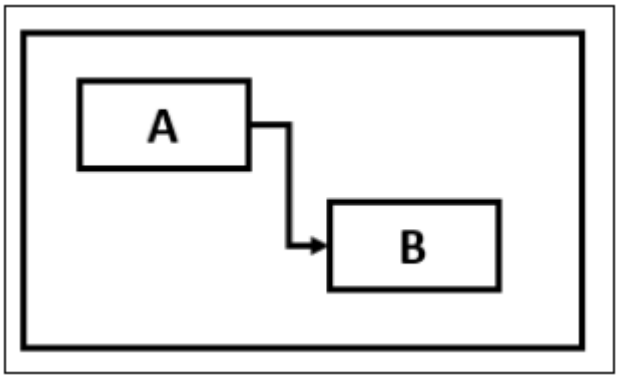
Task Dependency Relationships

Certain times, while scheduling, you may have to consider task dependency relationships. The important Task Dependency Relationships are −

- Finish-to-Start (FS)
- Finish-to-Finish (FF)

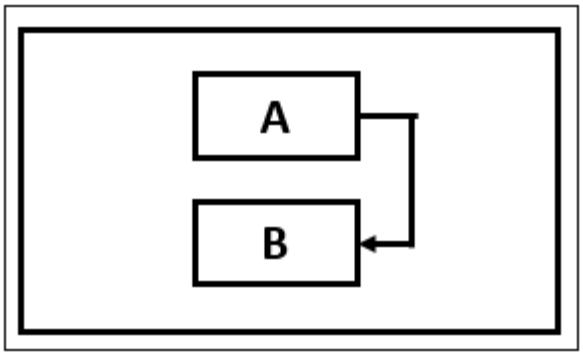
Finish-to-Start (FS)

In Finish-to-Start (FS) task dependency relationship, Task B cannot start till Task A is completed.



Finish-to-Finish (FF)

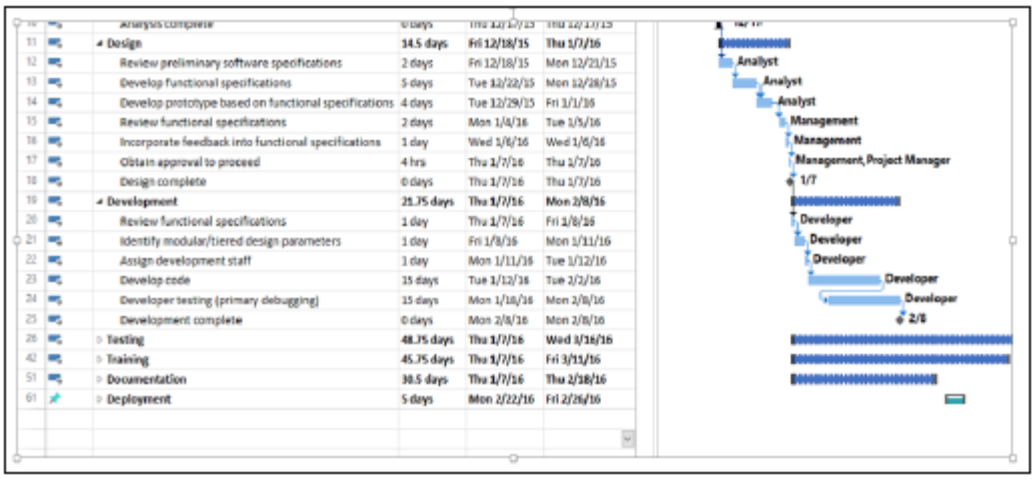
In Finish-to-Finish (FF) task dependency relationship, Task B cannot finish till Task A is completed.



Gantt Chart

A Gantt chart is a type of bar chart, adapted by Karol Adamiecki in 1896 and independently by Henry Gantt in the 1910s, that illustrates a project schedule. Gantt charts illustrate the start and finish dates of the terminal elements and summary elements of a project.

You can take the Outline Format in Figure 2 into Microsoft Project to obtain a Gantt Chart View.



Milestones

Milestones are the critical stages in your schedule. They will have a duration of zero and are used to flag that you have completed certain set of tasks. Milestones are usually shown as a diamond.

For example, in the above Gantt Chart, Design Complete and Development Complete are shown as milestones, represented with a diamond shape.

Milestones can be tied to Contract Terms.

Advantages of Estimation using WBS

WBS simplifies the process of project estimation to a great extent. It offers the following advantages over other estimation techniques −

- In WBS, the entire work to be done by the project is identified. Hence, by reviewing the WBS with project stakeholders, you will be less likely to omit any work needed to deliver the desired project deliverables.
- WBS results in more accurate cost and schedule estimates.
- The project manager obtains team participation to finalize the WBS. This involvement of the team generates enthusiasm and responsibility in the project.
- WBS provides a basis for task assignments. As a precise task is allocated to a particular team member who would be accountable for its accomplishment.
- WBS enables monitoring and controlling at task level. This allows you to measure progress and ensure that your project will be delivered on time.

4.8 COCOMO and COCOMO-II

Barry Boehm introduced a hierarchy of software estimation models bearing the name COCOMO stands for **COnstructive COst MOdel**. The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, called COCOMO II . Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the following areas:

Application composition model. Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.
Early design stage model. Used once requirements have been stabilized and basic software architecture has been established.
Post-architecture-stage model. Used during the construction of the software.

- Like all estimation models for software, the COCOMO II models require sizing information.
- Three different sizing options are available as part of the model hierarchy: object points, function points, and lines of source code.

The COCOMO II application composition model uses object points and is illustrated in the following paragraphs. It should be noted that other, more sophisticated estimation models (using FP and KLOC) are also available as part of COCOMO II.

Like function points, the object point is an indirect software measure that is computed using counts of the number of (1) screens (at the user interface), (2)reports, and (3) components likely to be required to build the application. Each object instance (e.g., a screen or report) is classified into one of three complexity levels (i.e., simple, medium, or difficult) using criteria suggested by Boehm [BOE96]. In essence, complexity is a function of the number and source of the client and server data tables that are required to generate the screen or report and the number of views or sections presented as part of the screen or report.

Once complexity is determined, the number of screens, reports, and components are weighted according to Table 5.1. The object point count is then determined by multiplying the original number of object instances by the weighting factor in Table 5.1 and summing to obtain a total object point count. When component-based development or general software reuse is to be applied, the percent of reuse (%reuse) is estimated and the object point count is adjusted:

NOP = (object points) x [(100 _ %reuse)/100]

where NOP is defined as new object points.

To derive an estimate of effort based on the computed NOP value, a "productivity rate" must be derived. Table 5.2 presents the productivity rate

- PROD = NOP/person-month

TABLE 5.1 Complexity weighting for object types [BOE96]

Object type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

TABLE 5.2 Productivity rates for object points [BOE96]

PROD = NOP/person-month

Developer's experience/capability	Very low	Low	Nominal	High	Very high
Environment maturity/capability	Very low	Low	Nominal	High	Very high
PROD	4	7	13	25	50

for different levels of developer experience and development environment maturity. Once the productivity rate has been determined, an estimate of project effort can be derived as

estimated effort = NOP/PROD

In more advanced COCOMO II models, a variety of scale factors, cost drivers, and adjustment procedures are required.

UNIT V

ADVANCED TOPICS

Formal Methods in Software Engineering

- Formal methods allow a software engineer to create a specification that is more complete, consistent, and unambiguous than those produced using conventional or object oriented methods.
- Set theory and logic notation are used to create a clear statement of facts (requirements).
- This mathematical specification can then be analyzed to prove correctness and consistency.
- Because the specification is created using mathematical notation, it is inherently less ambiguous than informal modes of representation.

Importance of Formal Methods

- In safety-critical or mission critical systems, failure can have a high price.
- Lives may be lost or severe economic consequences can arise when computer software fails.
- In such situations, it is essential that errors are uncovered before software is put into operation.
- Formal methods reduce specification errors dramatically and, as a consequence, serve as the basis for software that has very few errors once the customer begins using it.

Steps in the application of formal methods:

- The first step in the application of formal methods is to define the data invariant, state, and operations for a system function.
- The data invariant is a condition that is true throughout the execution of a function that contains a collection of data,
- The state is the stored data that a function accesses and alters
- Operations are actions that take place in a system as it reads or writes data to a state.
- An operation is associated with two conditions: a precondition and a postcondition.
- The notation and heuristics of sets and constructive specification—set operators, logic operators, and sequences—form the basis of formal methods.

Formal methods used in developing computer systems are mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop, and verify systems in a systematic, rather than ad hoc manner. A method is formal if it has a sound mathematical basis, typically given by a formal specification language. This basis provides a means of precisely defining notions like consistency and completeness, and more relevantly, specification, implementation and correctness. The desired

properties of a formal specification—consistency, completeness, and lack of ambiguity—are the objectives of all specification methods. However, the use of formal methods results in a much higher likelihood of achieving these ideals. The formal syntax of a specification language enables requirements or design to be interpreted in only one way, eliminating ambiguity that often occurs when a natural language (e.g., English) or a graphical notation must be interpreted by a reader. The descriptive facilities of set theory and logic notation enable clear statement of facts (requirements). To be consistent, facts stated in one place in a specification should not be contradicted in another place. Consistency is ensured by mathematically proving that initial facts can be formally mapped (using inference rules) into later statements within the specification. Completeness is difficult to achieve, even when formal methods are used. Some aspects of a system may be left undefined as the specification is being created; other characteristics may be purposely omitted to allow designers some freedom in choosing an implementation approach; and finally, it is impossible to consider every operational scenario in a large, complex system. Things may simply be omitted by mistake.

Deficiencies of Less Formal Approaches

Contradictions are sets of statements that are at variance with each other. For example, one part of a system specification may state that the system must monitor all the temperatures in a chemical reactor while another part, perhaps written by another member of staff, may state that only temperatures occurring within a certain range are to be monitored. Normally, contradictions that occur on the same page of a system specification can be detected easily. However, contradictions are often separated by a large number of pages.

Ambiguities are statements that can be interpreted in a number of ways. For example, the following statement is ambiguous: The operator identity consists of the operator name and password; the password consists of six digits. It should be displayed on the security VDU and deposited in the login file when an operator logs into the system. In this extract, does the word it refer to the password or the operator identity?

Vagueness often occurs because a system specification is a very bulky document. Achieving a high level of precision consistently is an almost impossible task. It can lead to statements such as “The interface to the system used by radar operators should be user-friendly” or “The virtual interface shall be based on simple overall concepts that are straightforward to understand and use and few in number.” A casual perusal of these statements might not detect the underlying lack of any useful information. Incompleteness is probably one of the most frequently occurring problems with system specifications.

Mathematics in Software Development

Mathematics has many useful properties for the developers of large systems. One of its most useful properties is that it is capable of succinctly and exactly describing a physical situation, an object, or the outcome of an action. Ideally, the software engineer should be in the same position as the applied mathematician. A mathematical specification of a system should be presented, and a solution developed in terms of a software architecture that implements the specification should be produced.

Example : A Symbol Table

- A program is used to maintain a symbol table.

- Such a table is used frequently in many different types of applications.
- It consists of a collection of items without any duplication.
- It represents the table used by an operating system to hold the names of the users of the system

Data invariant

- A data invariant is a condition that is true throughout the execution of the system that contains a collection of data.
- The data invariant that holds for the symbol table just discussed has two components: (1) that the table will contain no more than MaxIds names and (2) that there will be no duplicate names in the table.

State

- Another important concept is that of a state.
- In the context of formal methods, a state is the stored data that a system accesses and alters.
- In the example of the symbol table program, the state is the symbol table

Operations

- The final concept is that of an operation.
- This is an action that takes place in a system and reads or writes data to a state.
- If the symbol table program is concerned with adding and removing staff names from the symbol table, then it will be associated with two operations: an operation to add a specified name to the symbol table and an operation to remove an existing name from the table.
- If the program provides the facility to check whether a specific name is contained in the table, then there would be an operation that would return some indication of whether the name is in the table.
- An operation is associated with two conditions: a precondition and a postcondition.
- A precondition defines the circumstances in which a particular operation is valid.
- For example, the precondition for an operation that adds a name to the staff identifier symbol table is valid only if the name that is to be added is not contained in the table and also if there are fewer than MaxIds staff identifiers in the table.
- The postcondition of an operation defines what happens when an operation has completed its action.
- In the example of an operation that adds an identifier to the staff identifier symbol table, the postcondition would specify mathematically that the table has been augmented with the new identifier.

MATHEMATICAL PRELIMINARIES

- To apply formal methods effectively, a software engineer must have a working knowledge of the mathematical notation associated with
- sets
- sequences
- logical notation used in predicate calculus

Sets and Constructive Specification

- A set is a collection of objects or elements and is used as a cornerstone of formal methods.
- The elements contained within a set are unique (i.e., no duplicates are allowed).
- Sets with a small number of elements are written within curly brackets (braces) with the elements separated by commas. For example, the set {C++, Pascal, Ada, COBOL, Java} contains the names of five programming languages.
- The order in which the elements appear within a set is immaterial.
- The number of items in a set is known as its cardinality.
- The # operator returns a set's cardinality.
- For example, the expression #{A, B, C, D} = 4

A constructive set specification

- Consider the following constructive specification example: {n : N| n < 3 . n}
- This specification has three components,
- a signature, n : N,
- a predicate n < 3,
- and a term, n.
- The signature specifies the range of values that will be considered when forming the set, the predicate (a Boolean expression) defines how the set is to be constricted, and, finally, the term gives the general form of the item of the set
- Therefore, this specification defines the set {0, 1, 2}

Set Operators

- A specialized set of symbology is used to represent set and logic operations.
- These symbols must be understood by the software engineer who intends to apply formal methods.
- The operator is used to indicate membership of a set.

- For example, the expression x X has the value true if x is a member of the set X and the value false otherwise. For example, the predicate 12 {6, 1, 12, 22} has the value true since 12 is a member of the set.

The opposite of the $\in$ operator is the $\notin$ operator. The expression

$x \notin X$

has the value *true* if *x* is not a member of the set *X* and *false* otherwise. For example, the predicate

$13 \notin \{13, 1, 124, 22\}$

has the value *false*.

The operators $\subset$ and $\subseteq$ take sets as their operands. The predicate

$A \subset B$

has the value *true* if the members of the set *A* are contained in the set *B* and has the value *false* otherwise. Thus, the predicate

$\{1, 2\} \subset \{4, 3, 1, 2\}$

has the value *true*. However, the predicate

- UNION operator
- INTERSECTION operator
- SET DIFFERENCE operator

Logic Operators

- AND
- OR
- NOT

Sequences

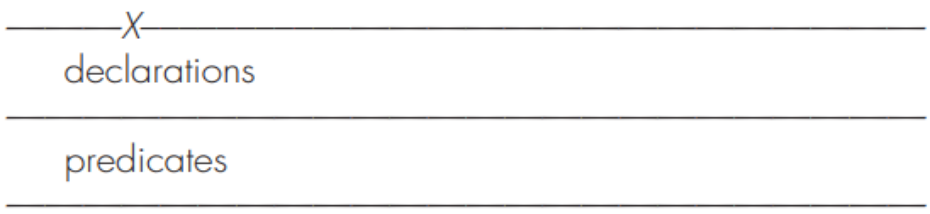
- A sequence is a mathematical structure that models the fact that its elements are ordered.
- A sequence s is a set of pairs whose elements range from 1 to the highest number element.
- For example, {(1, Jones), (2, Wilson), (3, Shapiro), (4, Estavez)}
- Unlike sets, duplication in a sequence is allowed and the ordering of a sequence is important.

Z notation

- Z specifications are structured as a set of schemas
- A boxlike structure that introduces variables and specifies the relationship between these variables.
- A schema is essentially the formal specification analog of the programming language subroutine or procedure.

- In the same way that procedures and subroutines are used to structure a system, schemas are used to structure a formal specification.
- Z notation is based on typed set theory and first-order logic.
- Z provides a construct, called a schema, to describe a specification's state space and operations.
- A schema groups variable declarations with a list of predicates that constrain the possible value of a variable.

Schema



Global functions and constants are defined by the form

Sets in z notation

Sets:

$S : \mathbb{P}\, X$	S is declared as a set of Xs.
$x \in S$	x is a member of S.
$x \notin S$	x is not a member of S.
$S \subseteq T$	S is a subset of T: Every member of S is also in T.
$S \cup T$	The union of S and T: It contains every member of S or T or both.
$S \cap T$	The intersection of S and T: It contains every member of both S and T.
$S \setminus T$	The difference of S and T: It contains every member of S except those also in T.
$\varnothing$	Empty set: It contains no members.
$\{x\}$	Singleton set: It contains just x.
$\mathbb{N}$	The set of natural numbers 0, 1, 2, ...
$S : \mathbb{F}\, X$	S is declared as a finite set of Xs.
max (S)	The maximum of the nonempty set of numbers S.

Functions in z notation

Functions:

$f : X \rightarrowtail Y$	f is declared as a partial injection from X to Y
dom f	The domain of f: the set of values x for which $f(x)$ is defined.
ran f	The range of f: the set of values taken by $f(x)$ as x varies over the domain of f.
$f \oplus \{x \mapsto y\}$	A function that agrees with f except that x is mapped to y.
$\{x\} \trianglelefteq f$	A function like f, except that x is removed from its domain.

Hoare logic

Proof of Correctness

- **Program verification** attempts to ensure that a computer program is correct. And a program is correct if it behaves in accordance with its **specifications**.
- This does not necessarily mean that the program solves the problem that it was intended to solve; the **program's specifications** may be at odds with or not address all aspects of a **client's requirements**.
- **Program validation** attempts to ensure that the program indeed meets the **client's original requirements**.
- **Program testing** seeks to show that particular input values produce acceptable output values.
- **Proof of correctness** uses the techniques of a formal logic system to prove that if the input variables satisfy certain specified predicates or properties, the output variables produced by executing the program satisfy other specified properties.

Assertions

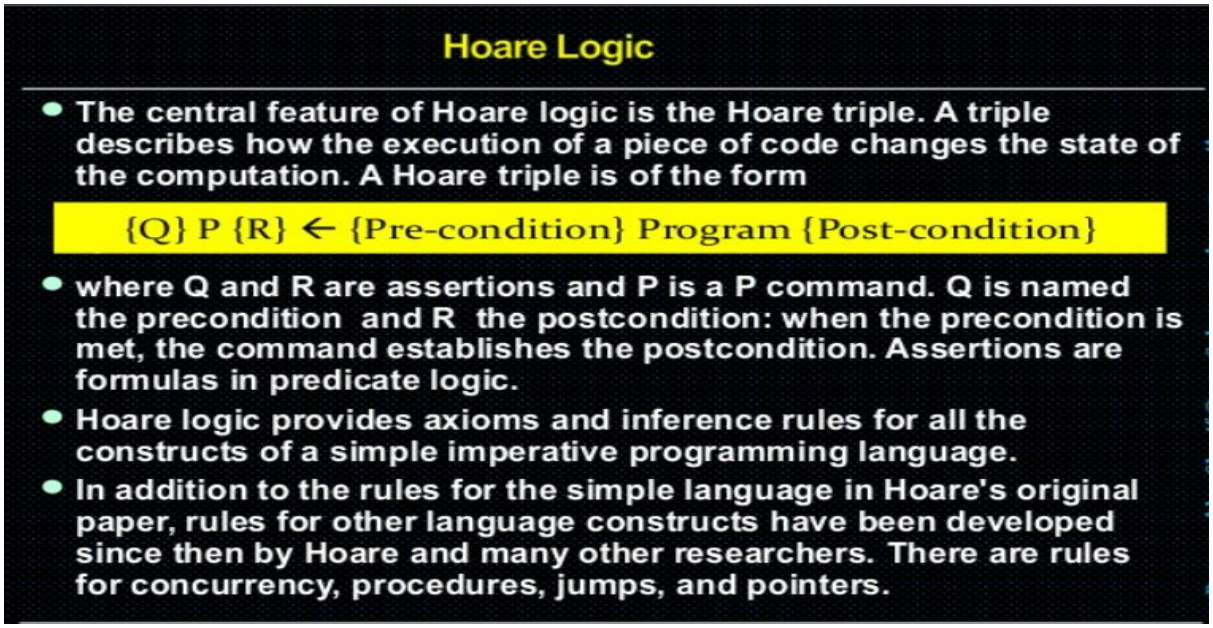
What is assertions in computer programming

- an assertion is a predicate (for example a true–false statement) placed in a program to indicate that the developer thinks that the **predicate is always true at that place**.
- For example, the following code contains two assertions:

```
x := 5;
{x > 0}
x := x + 1
{x > 1}
```

x > 0 and x > 1, and they are indeed true at the indicated points during execution.

- Programmers can use assertions to help specify programs and to reason about program correctness.
- For example, a **precondition** — an assertion placed at the beginning of a section of code — determines the set of states under which the programmer expects the code to execute. A **postcondition** — placed at the end — describes the expected state at the end of execution.



THE TEN COMMANDMENTS OF FORMAL METHODS

- 1.Thou shalt choose the appropriate notation.
- 2. Thou shalt formalize but not overformalize
- 3. Thou shalt estimate costs.
- 4. Thou shalt have a formal methods guru on call
- 5. Thou shalt not abandon thy traditional development methods
- 6. Thou shalt document sufficiently.
- 7. Thou shalt not compromise thy quality standards. “There is nothing magical about formal methods”
- 8. Thou shalt not be dogmatic. A software engineer must recognize that formal methods are not a guarantee of correctness.
- 9. Thou shalt test, test, and test again.
- 10. Thou shalt reuse.

Support environment for Development of Software Products

- A software system that provides support for the development, repair, and enhancement of software, and for the management and control of these activities.
- A typical system contains a central database and a set of software tools.
- The central database acts as a repository for all information related to a project throughout the lifetime of that project.

- The software tools offer support for the various activities, both technical and managerial, that must be performed on the project.
- Different environments vary in the general nature of their databases and in the coverage provided by the set of tools.
- In particular, some encourage (or even enforce) one specific software engineering methodology, while others provide only general support and therefore allow any of a variety of methodologies to be adopted.
- All environments, however, reflect concern for the entire software life cycle (rather than just the program development phase) and offer support for project management (rather than just technical activities).

Programming support environment

- A software system that provides support for the programming aspects of software development, repair, and enhancement.
- A typical system contains a central database and a set of software tools.
- The central database acts as a repository for all the information related to the programming activities.
- PSEs vary in the general nature of their databases and in the coverage provided by, and the degree of cooperative interaction of, the set of tools and the programming languages supported.
- A programming support environment might be considered as a more technologically advanced form of program development system.

project support environment

- A software system that provides support for the full life cycle of software development and also the project control and management aspects of a software-intensive project.
- The project support environment will have all the features of a programming support environment plus software tools to support the earlier phases of software development (CASE (upper)) and tools associated with the management and control of the project

Representative tools: Editors

- Textual or graphical
- Can follow a formal syntax, or can be used for informal text or free-form pictures
- Monolingual (e.g., Java editor) or multilingual

Representative tools: Linkers

- Combine object-code fragments into a larger program
    - can be monolingual or polylingual

- In a broader sense, tools for linking specification modules, able to perform checking and binding across various specification modules

Representative tools: Interpreters

- Traditionally at the programming language level
- Also at the requirements specification level
    - requirements animation
- Can be numeric or symbolic

Representative tools: Code generators

- In a general sense, transform a high level description into a lower-level description
    - a *specification* into an *implementation*
- Practical example
    - 4th Generation Languages

Representative tools: Debuggers

- May be viewed as special kinds of interpreters where
    - execution state inspectable
    - execution mode definable
    - animation to support program understanding

Representative tools: Software testing

- Test documentation tools
- support bookkeeping of test cases
- forms for test case definition, storage, retrieval

Project Name:	Date of test:
Tested function:	
Tested module:	
Test case description:	
Description of results:	
Comments:	

- Tools for test data derivation

        - e.g., synthesizing data from path condition
    - Tools for test evaluation
        - e.g., various coverage metrics
    - Tools for testing other software qualities

Representative tools: Static analyzers

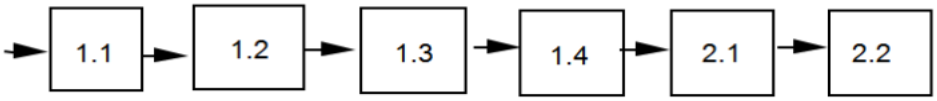
- Data and control flow analyzers
    - can point out possible flaws or suspicious-looking statements
        - e.g., detecting uninitialized variables

User-Interface Management Systems

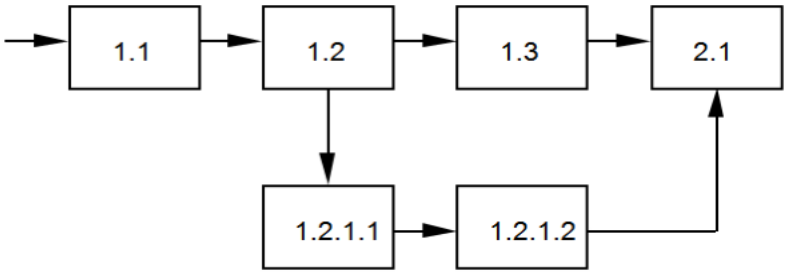
- Provide a set of basic abstractions (windows, menus, scroll bars, etc.) that may be used to customize a variety of interfaces
- Provide a library of run-time routines to be linked to the developed application in order to support input and output
    - UIMS fall both under the category of development tools and under the category of end-product components

Representative tools: Configuration Management

- Repository
    - shared database of artifacts
- Version management
    - versions stored, change history maintained
- Work-space control
    - private work-space
    - shared work-space
- Product modeling and building
    - facilities to (re)build products



sequence of revisions



a branch and a later join

Representative tools: Reverse and reengineering

- Program understanding systems
    - synthesize suitable abstractions from code
        - e.g., control and data flow graphs or use graphs
    - extract cross-references and other kinds of documentation material on the product
- Reverse engineering tools also support the process of making the code and other artifacts consistent with each other

Representative tools: project management

- Project management is one of the high-responsibility tasks in modern organizations. Project management is used in many types of projects ranging from software development to developing the next generation fighter aircrafts.
- In order to execute a project successfully, the project manager or the project management team should be supported by a set of tools.
- These tools can be specifically designed tools or regular productivity tools that can be adopted for project management work.
- The use of such tools usually makes the project managers work easy as well as it standardizes the work and the routine of a project manager.
- Milestone Checklist

- -This is one of the best tools the project manager can use to determine whether he or she is on track in terms of the project progress.
- Gantt Chart
- - Gantt chart illustrates the project schedule and shows the project manager the interdependencies of each activity. Gantt charts are universally used for any type of project from construction to software development.
- Although deriving a Gantt chart looks quite easy, it is one of the most complex tasks when the project is involved in hundreds of activities.
- Tools for Gantt and PERT charts
    - graphical interface
    - support to analysis
- Cost estimation tools
    - based on models, such as COCOMO

Tools for Decision Support and Synthesis

- Project Management has emerged as a discipline of high level decision making with the help of tools which would help augment the intuition of a Project Manager and his team for taking decisions in favour of the future of the project.
- These decision making tools are general, they are based on common sense and are used in all the trades for backing up the decisions taken by the decision making authorities.
- Time, cost and scope are the triple constraints of any project.
- The process decision program chart (PDPC) is defined as a new management planning tool that systematically identifies what might go wrong in a plan under development.
- Countermeasures are developed to prevent or offset those problems.
- By using PDPC, you can either revise the plan to avoid the problems or be ready with the best response when a problem occurs.
- Failure Mode and Effects Analysis (FMEA) is a structured approach to discovering potential failures that may exist within the design of a product or process.
- Failure modes are the ways in which a process can fail.
- Effects are the ways that these failures can lead to waste, defects or harmful outcomes for the customer.
- A decision tree analysis exercise will allow project leaders to easily compare different courses of action against each other and evaluate the risks, probabilities of success, and potential benefits associated with each.
- Risk management is an important function in organizations today. Companies undertake increasingly complex and ambitious projects, and those projects must be executed successfully, in an uncertain and often risky environment.

- The Risk Impact/Probability Chart provides a useful framework that helps you decide which risks need your attention

Tools for engineering databases

- Basic Requirements for Visual Tools of Data Design
- The first and foremost thing that a database design software must ensure are forward and reverse engineering.
- One way to achieve it is visual design.
- It generates code when changes are introduced into the data model, supporting forward engineering.
- And, it builds a graphical model from code, supporting reverse engineering.
- Therefore, the visual solution for database design must provide graphical forward and reverse engineering in the following capabilities:

1. Visualize database structure for further analysis.
2. Cluster logically related objects using containers.
3. Track logical correlations between tables.
4. Open large SQL database diagrams.
5. Create and edit database objects in a diagram.

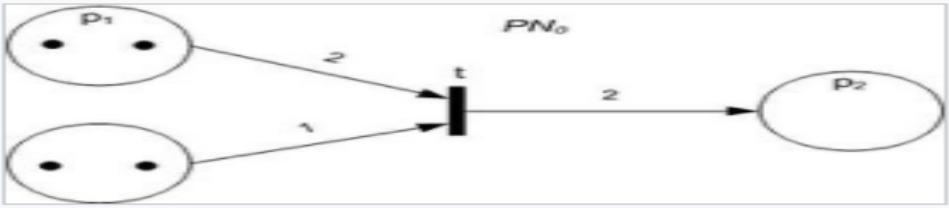
- Also, there should be the following instruments for visualization:

1. The list of notations on data modeling syntax.
2. Objects' commenting.
3. The documented plan, showing the logical structure of an SQL diagram at present. You can use it when navigating through tables and their dependencies in a database diagram.
4. The diagram overview
5. The level of detail for the displayed data (whether to show constraints, indexes, etc.).
6. The list of designs (optimization for black and white printing, color printing, etc.).

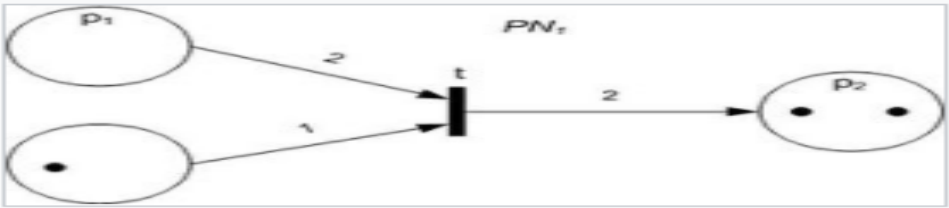
Petri nets

- A Petri net, also known as a place/transition (PT) net, is one of several mathematical modeling languages for the description of distributed systems.
- It is a class of discrete event dynamic system.
- A Petri net is a directed bipartite graph that has two types of elements, places and transitions, depicted as white circles and rectangles, respectively.
- A place can contain any number of tokens, depicted as black circles.
- A transition is enabled if all places connected to it as inputs contain at least one token.

- Like industry standards such as UML activity diagrams, Business Process Model and Notation and event-driven process chains, Petri nets offer a graphical notation for stepwise processes that include choice, iteration, and concurrent execution.
- Unlike these standards, Petri nets have an exact mathematical definition of their execution semantics, with a well-developed mathematical theory for process analysis
- A Petri net consists of *places*, *transitions*, and *arcs*.
- Arcs run from a place to a transition or vice versa, never between places or between transitions.
- The places from which an arc runs to a transition are called the *input places* of the transition; the places to which arcs run from a transition are called the *output places* of the transition.
- Graphically, places in a Petri net may contain a discrete number of marks called *tokens*.
- Any distribution of tokens over the places will represent a configuration of the net called a *marking*.
- In an abstract sense relating to a Petri net diagram, a transition of a Petri net may *fire* if it is *enabled*, i.e. there are sufficient tokens in all of its input places;
- when the transition fires, it consumes the required input tokens, and creates tokens in its output places.
- A firing is atomic, i.e. a single non-interruptible step.
- Unless an *execution policy* is defined, the execution of Petri nets is nondeterministic: when multiple transitions are enabled at the same time, they will fire in any order.
- Since firing is nondeterministic, and multiple tokens may be present anywhere in the net (even in the same place), Petri nets are well suited for modeling the concurrent behavior of distributed systems.



A Petri net with an enabled transition.



The Petri net that follows after the transition fires (Initial Petri net in the figure above).

Properties of Petri Nets

- Sequential Execution
  Transition t_2 can fire only after the firing of t_1. This impose the precedence of constraints "t_2 after t_1."
- Synchronization
  Transition t_1 will be enabled only when there are at least one token at each of its input places.
- Merging
  Happens when tokens from several places arrive for service at the same transition.
- Conflict
  t_1 and t_2 are both ready to fire but the firing of any leads to the disabling of the other transitions.
- Concurrency
  t_1 and t_2 are concurrent.
  - with this property, Petri net is able to model systems of distributed control with multiple processes executing concurrently in time.

Aspect oriented programming

- An aspect is a common feature that's typically scattered across methods, classes, object hierarchies, or even entire object models.
- It is behavior that looks and smells like it should have structure, but you can't find a way to express this structure in code with traditional object-oriented techniques.
- For example, metrics is one common aspect.

- To generate useful logs from your application, you have to sprinkle informative messages throughout your code.
- However, metrics is something that your class or object model really shouldn't be concerned about.
- After all, metrics is irrelevant to your actual application: it doesn't represent a customer or an account, and it doesn't realize a business rule.
- It's simply orthogonal.
- In AOP, a feature like metrics is called a crosscutting concern, as it's a behavior that "cuts" across multiple points in your object models, yet is distinctly different. As a development methodology, AOP recommends that you abstract and encapsulate crosscutting concerns.
- For example, let's say you wanted to add code to an application to measure the amount of time it would take to invoke a particular method.