SOFTWARE ENGINEERING [CS T55]

UNIT- I

Introduction to Software Engineering: The Software Engineering Discipline – Evolution and Impact – Software Development projects – Emergence of Software Engineering – Computer System Engineering – Software Life Cycle Models – classic Waterfall model – Iterative Lifecycle model – prototyping model – Evolutionary model – spiral model – Comparison of Life cycle models.

SOFTWARE ENGINEERING DISCIPLINE

SOFTWARE ENGINEERING:

- Software engineering is an engineering discipline that 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 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.

DIFFERENCE BETWEEN SOFTWARE ENGINEERING AND COMPUTER SCIENCE:

- Computer science is concerned with **theory and fundamentals**; software engineering is concerned with the practicalities of **developing and delivering useful software**.
- Computer science theories are still insufficient to act as a complete underpinning for software engineering (unlike e.g. physics and electrical engineering).

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 process** concerned with **developing the software infrastructure, control, applications and databases in the system**.
- System engineers are involved in system specification, architectural design, integration and deployment.

SOFTWARE PRODUCT:

Software Product may be developed for a particular customer or may be developed for a general market. Software product may be,

- **Generic:** developed to be sold to it range of different customer.
- **Custom:** developed for a single customer according to their specification.

SOFTWARE PROCESS:

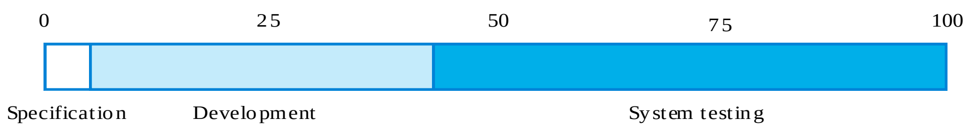
- A set of activities whose goal is the development or evolution of software.
- Generic activities in all software processes are:
    - **Specification** - what the system should do and its development constraints.
    - **Development** - production of the software system.
    - **Validation** - checking that the software is what the customer wants
    - **Evolution** - changing the software in response to changing demands.

SOFTWARE MODEL:

- ✓ A simplified representation of software process, presented from specific perspective.
- ✓ Example: Workflow perspective, data flow perspective.

COSTS OF SOFTWARE ENGINEERING:

- Roughly **60% of costs are development costs**, **40% are testing costs**. For custom software, evolution costs often exceed development costs.
- Costs vary depending on the type of system being developed and the requirements of system attributes such as performance and system reliability.
- Distribution of **costs depends on the development model** that is used.
- Example : Product development costs



SOFTWARE ENGINEERING METHODS:

Structured approaches to software development which include system models, notations, rules, design advice and process guidance.

- **Model descriptions**
    - Descriptions of graphical models which should be produced.
- **Rules**

    - Constraints applied to system models.
- **Recommendations**
    - Advice on good design practice.
- **Process guidance**
    - What activities to follow.

ATTRIBUTES OF GOOD SOFTWARE:

- **Maintainability**
    - Software must evolve to meet changing needs.
- **Dependability**
    - Software must be trustworthy.
- **Efficiency**
    - Software should not make wasteful use of system resources.
- **Acceptability**
    - Software must accepted by the users for which it was designed.

KEY CHALLENGES FACING SOFTWARE ENGINEERING:

- **Heterogeneity**
    - Building software that can cope with heterogeneous platforms and execution environments.
- **Delivery**
    - Developing techniques that lead to faster delivery of software.
- **Trust**
    - Developing techniques that demonstrate that software can be trusted by its users.

EVOLUTION AND IMPACT

Software evolution is the term used in software engineering (specifically software maintenance) to refer to the process of developing software initially, then repeatedly updating it for various reasons.

SOFTWARE CHANGE:

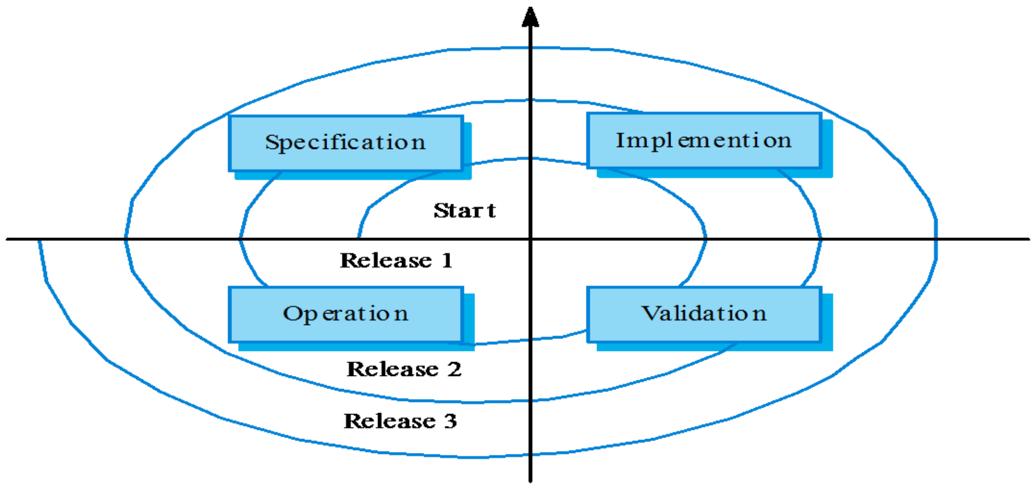
- ✓ Software change is inevitable
    - New requirements emerge when the software is used.
    - The business environment changes,

  - Errors must be repaired.
  - New computers and equipment is added to the system.
  - The performance or reliability of the system may have to be improved.
- ✓ A key problem for organisations is implementing and managing change to their existing software systems.

IMPORTANCE OF EVOLUTION:

- ✓ Organizations have huge investments in their software systems - they are critical business assets.
- ✓ To maintain the value of these assets to the business, they must be changed and updated.
- ✓ The majority of the software budget in large companies is devoted to evolving existing software rather than developing new software.

Spiral model of Evolution:



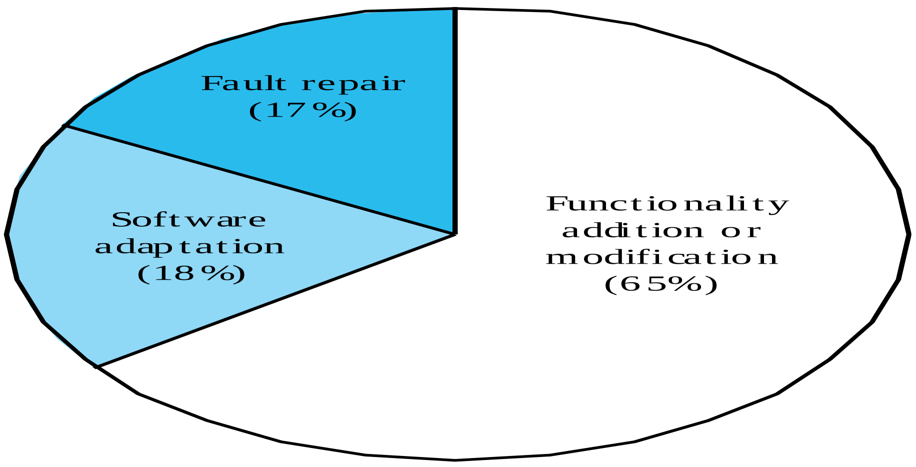
SOFTWARE MAINTENANCE:

Software maintenance in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes.

<u>Types of maintenance:</u>

- ✓ **Maintenance to repair software faults**
  - Changing a system to correct deficiencies in the way meets its requirements.
- ✓ **Maintenance to adapt software to a different operating environment**
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- ✓ **Maintenance to add to or modify the system's functionality**
  - Modifying the system to satisfy new requirements.

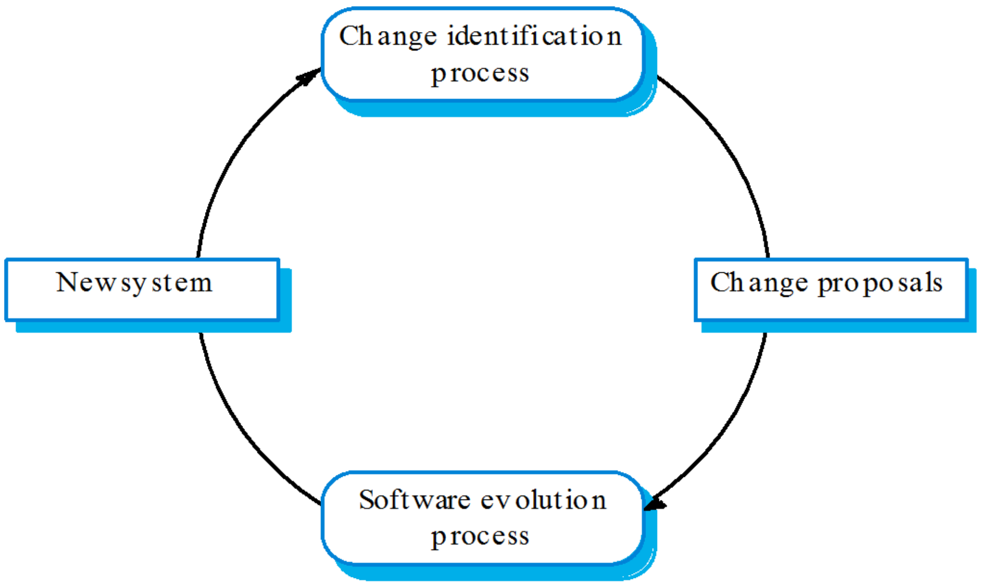
Distribution of maintenance effort:



EVOLUTION PROCESSES:

- Evolution processes depend on
  - The type of software being maintained;
  - The development processes used;
  - The skills and experience of the people involved.
- Proposals for change are the driver for system evolution. Change identification and evolution continue throughout the system lifetime.

Change identification and evolution:

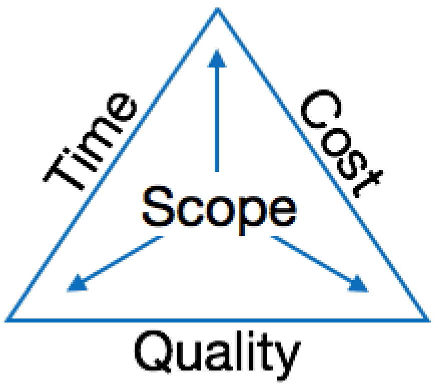


SOFTWARE DEVELOPMENT PROJECTS

Software project management is the art and science of planning and leading software projects. It is a sub-discipline of project in which software projects are planned, implemented, monitored and controlled.

SOFTWARE PROJECT:

A Software Project is the complete procedure of software development from requirement gathering to testing and maintenance, carried out according to the execution methodologies, in a specified period of time to achieve intended software product.



SOFTWARE PROJECT MANAGER:

- A software project manager is a person who undertakes the responsibility of executing the software project.
- Software project manager is thoroughly aware of all the phases of SDLC that the software would go through.
- Project manager may never directly involve in producing the end product but he controls and manages the activities involved in production.

FEW RESPONSIBILITIES FOR PROJECT MANAGER:

Managing People

- Act as project leader
- Connection with stakeholders
- Managing human resources
- Setting up reporting hierarchy etc.

Managing Project

- Defining and setting up project scope
- Managing project management activities
- Monitoring progress and performance

- Risk analysis at every phase
- Take necessary step to avoid or come out of problems
- Act as project spokesperson

PROJECT MANAGEMENT ACTIVITIES:

Software project management comprises of a number of activities, which contains planning of project, deciding scope of software product, estimation of cost in various terms, scheduling of tasks and events, and resource management. Project management activities may include:

- **Project Planning**
- **Scope Management**
- **Project Estimation**

Project Planning:

- Software project planning is task, which is performed before the production of software actually starts.
- It is there for the software production but involves no concrete activity that has any direction connection with software production.

Scope Management:

- It defines the scope of project; this includes all the activities, process need to be done in order to make a deliverable software product.
- During Project Scope management, it is necessary to -

✓ Define the scope
✓ Decide its verification and control
✓ Divide the project into various smaller parts for ease of management.
✓ Verify the scope
✓ Control the scope by incorporating changes to the scope

Project Estimation:

For an effective management accurate estimation of various measures is a must. With correct estimation managers can manage and control the project more efficiently and effectively. Project estimation may involve the following:

- Software size estimation
- Cost estimation
- Time estimation
- Effort estimation

PROJECT RISK MANAGEMENT:

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

Risk Management Process:

There are following activities involved in risk management process:

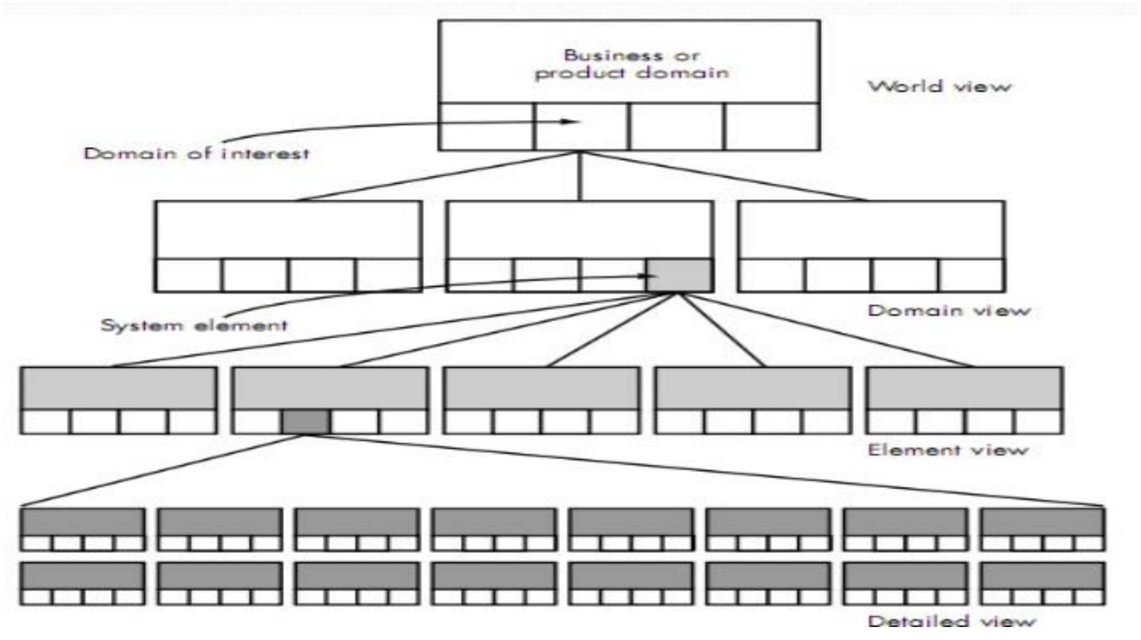
- **Identification -** Make note of all possible risks, which may occur in the project.
- **Categorize -** Categorize known risks into high, medium and low risk intensity as per their possible impact on the project.
- **Manage -** Analyze the probability of occurrence of risks at various phases. Make plan to avoid or face risks. Attempt to minimize their side-effects.
- **Monitor -** Closely monitor the potential risks and their early symptoms. Also monitor the effects of steps taken to mitigate or avoid them.

COMPUTER SYSTEM ENGINEERING

- Software Engineering occurs as a **consequence of process** is called **system engineering**.
- Systems engineering is an interdisciplinary field of engineering that focuses on **how to design and manage complex engineering systems over their life cycles**.
- Issues such as requirements engineering, reliability, logistics, coordination of different teams, testing and evaluation, maintainability and many other disciplines necessary for successful system development, design, implementation, and ultimate decommission become more difficult when dealing with large or complex projects.

SYSTEM ENGINEERING PROCESS:

- The system engineering process begins with a world view; the business or product domain is examined to ensure that the proper business or technology context can be established
- The world view is refined to focus on a specific domain of interest
- Within a specific domain, the need for targeted system elements is analyzed
- Finally, the analysis, design, and construction of a targeted system element are initiated
- At the world view level, a very broad context is established
- At the bottom level, detailed technical activities are conducted by the relevant engineering discipline (e.g., software engineering).



COMPUTER-BASED SYSTEM:

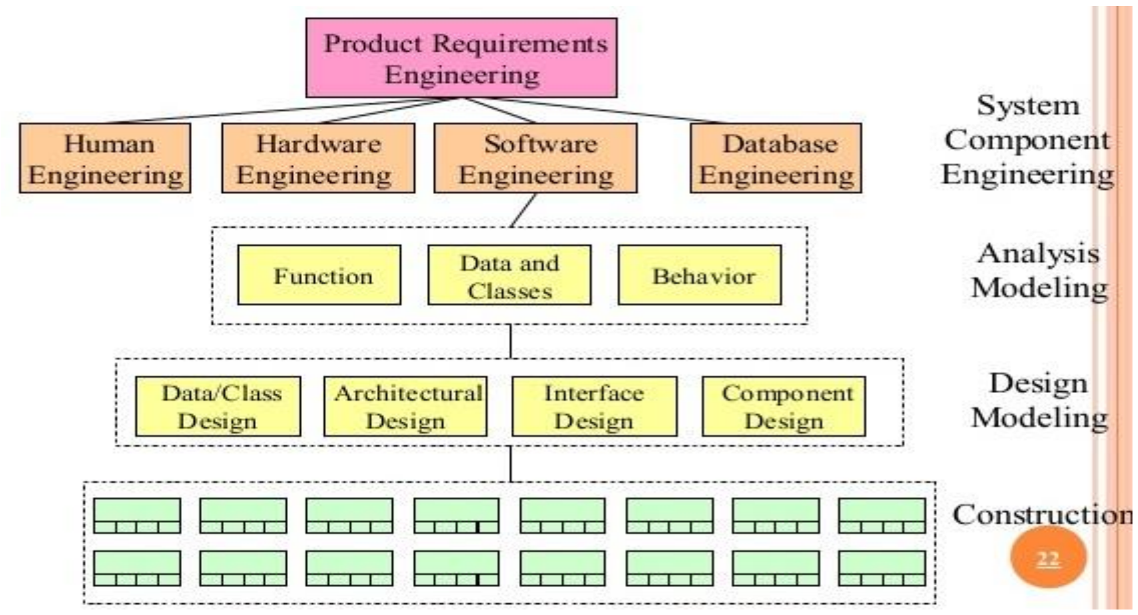
A set or arrangement of elements that are organized to accomplish some **predefined goal** by processing information.

A computer-based system makes use of the following **four system elements** that combine in a variety of ways to transform information:

- **Software**: computer programs, data structures, and related work products that serve to effect the logical method, procedure, or control that is required
- **Hardware**: electronic devices that provide computing capability, interconnectivity devices that enable flow of data, and electromechanical devices that provide external functions
- **People**: Users and operators of hardware and software
- **Database**: A large, organized collection of information that is accessed via software and persists over time

PRODUCT ENGINEERING:

- Product engineering translates the customer's desire for a set of defined capabilities into a working product
- It achieves this goal by establishing **product architecture** and a **support infrastructure.**



EMERGENCE OF SOFTWARE ENGINEERING

Software engineering discipline is the result of advancement in the field of technology. The various innovations and technologies that led to the emergence of software engineering discipline.

Early Computer Programming (1950's):

- In the early 1950s, computers were slow and expensive.
- Though the programs at that time were very small in size, these computers took considerable time to process them.
- They relied on **assembly language** which was specific to computer architecture.
- Programs were limited to about a **few hundreds of lines of assembly code**.
- Every programmer used his own style to develop the programs.

High Level Language Programming (1960's):

- High level programming languages such as **COBOL, ALGOL** and **FORTRAN** were introduced.
- This reduces software development efforts greatly.
- Typical program sizes were limited to a **few thousands of lines of assembly code**.

Control Flow Based Design (Late 1960's):

- Size and complexity of the programs increased further: Exploratory programming style proved to be insufficient.
- Programs found very difficult to write cost-effective and correct programs.
- Programs written by others, very difficult to understand and maintain.

- To help the programmer to design programs having good control flow structure, **flowcharting technique** was developed.
- In flowcharting technique, the algorithm is represented using flowcharts.
- Note that having more **GOTO** constructs in the flowchart makes the control flow messy, which makes it difficult to understand and debug.
- In order to provide clarity of control flow, the use of GOTO constructs in flowcharts should be avoided and **structured constructs-decision,** sequence, and loop-should be used to develop **structured flowcharts.**

Structures Programming:

- Decision structure used for conditional **execution of statement (if statement).**
- Sequence Structure for sequentially execution statements.
- **Loop Structure** is used for performing some repetitive task in the program.
- Structured Program becomes powerful tool that allowed programmers to write moderately complex program easily.

Data Flow Oriented Design (1970's):

- It is important to pay more attention to the design of data structure of the program than to the design of the control structure.
- In this technique, the flow of data through processes is represented using **DFD (Data Flow Diagram).**
- Data flow diagram also known as **Bubble chart** and **work flow diagram**.
- **IEEE** defines a data-flow diagram as 'a diagram that depicts data sources, data sinks, data storage, and processes performed on data as nodes, and logical flow of data as links between the nodes.

Object Oriented Design:

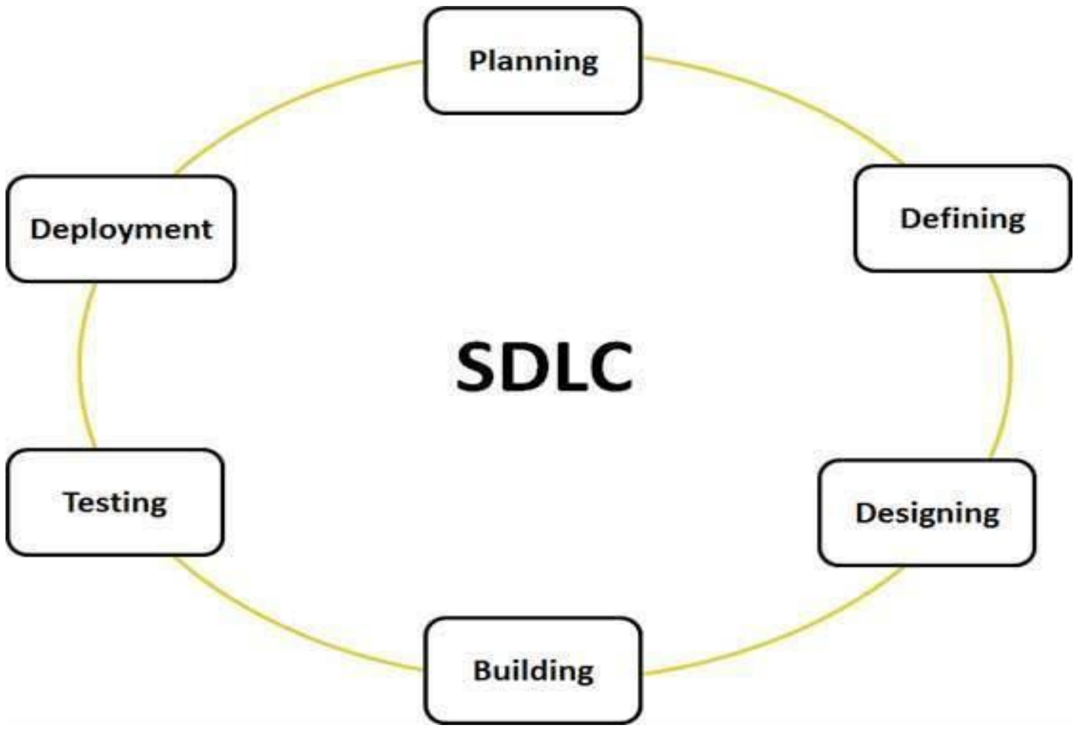
- Object-oriented design technique has revolutionized the process of software development.
- It not only includes the best features of structured programming but also some new and powerful features such as **encapsulation, abstraction, inheritance, and polymorphism**.
- These new features have tremendously helped in the development of well-designed and high-quality software.
- Object-oriented techniques are widely used these days as they allow **reusability of the code.**
- They lead to faster software development and high-quality programs.

- Moreover, they are easier to adapt and scale, that is, large systems can be created by assembling reusable subsystems.

SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC) AND ITS STAGES

- SDLC is the acronym of Software Development Life Cycle.
- It is also called as **Software development process**.
- The software development life cycle (SDLC) is a framework defining **tasks performed at each step in the software development process.**

Stages/Phases of SDLC:



A typical Software Development life cycle consists of the following stages:

Stage 1: Planning and Requirement Analysis

- Requirement analysis is the most important and fundamental stage in SDLC.
- Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage.

Stage 2: Defining Requirements

- Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysis.
- This is done through Software Requirement Specification (SRS).

- It consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: Designing the product architecture

- A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules.
- The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS (Design Document Specification).

Stage 4: Building or Developing the Product

- In this stage of SDLC the actual development starts and the product is built.
- The programming code is generated as per DDS during this stage.
- Developers have to follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers etc are used to generate the code.
- Different high level programming languages such as C, C++, Pascal, Java, and PHP are used for coding.

Stage 5: Testing the Product

- The testing activities are mostly involved in all the stages of SDLC. Here functional and non functional testing is done.
- However this stage refers to the testing only stage of the product where products defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 6: Deployment in the Market and Maintenance

- Once the product is tested and ready to be deployed it is released formally in the appropriate market.
- The product may release in a limited segment and tested in the real business environment.
- After the product is released in the market, its maintenance is done for the existing customer base.

SOFTWARE LIFE CYCLE MODELS

- There are various software development life cycle models defined and designed which are followed during software development process.
- These models are also referred as **"Software Development Process Models".**
- Following are the most important and popular SDLC models:

1. **Waterfall Model**

2. **Iterative Model**

3. **Spiral Model**

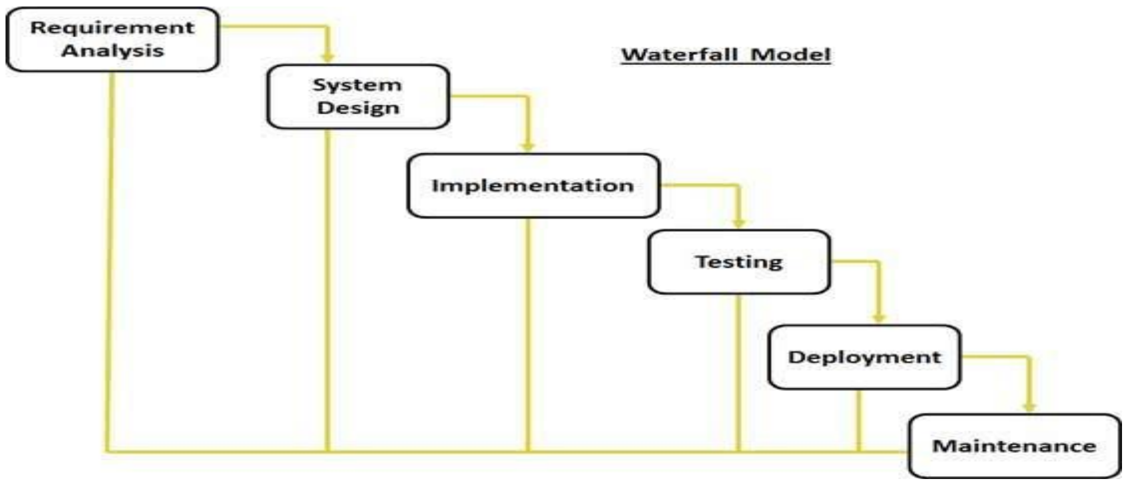
4. **V-Model**

5. **Big Bang Model**

The other related methodologies are **Agile Model, RAD Model, Rapid Application Development and Prototyping Models**.

CLASSIC WATERFALL MODEL

- It is also known as **Linear Sequential life cycle model**.
- It is very simple to understand and use.
- In waterfall model, **each phase must be completed fully before next phase can begin**.
- The main concept of this model is that only when one development level is completed will the next one be initiated. At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project.
- Following is a diagrammatic representation of different phases of waterfall model.



The sequential phases in Waterfall model are:

Requirement Gathering and analysis:

- All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.

System Design:

- The requirement specifications from first phase are studied in this phase and system design is prepared.
- System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture.

Implementation:

- With inputs from system design, the system is first developed in small programs called units, which are integrated in the next phase.
- Each unit is developed and tested for its functionality which is referred to as Unit Testing.

Integration and Testing:

- All the units developed in the implementation phase are integrated into a system after testing of each unit.
- Post integration the entire system is tested for any faults and failures.

Deployment of system:

- Once the functional and non functional testing is done, the product is deployed in the customer environment or released into the market.

Maintenance:

- There are some issues which come up in the client environment.
- To fix those issues patches are released and to enhance the product some better versions are released.
- Maintenance is done to deliver these changes in the customer environment.

Advantages:

- ✓ Simple and easy to understand.
- ✓ It works well for smaller projects where requirements are very well understood.
- ✓ Clearly defined stages.
- ✓ Easy to arrange tasks.

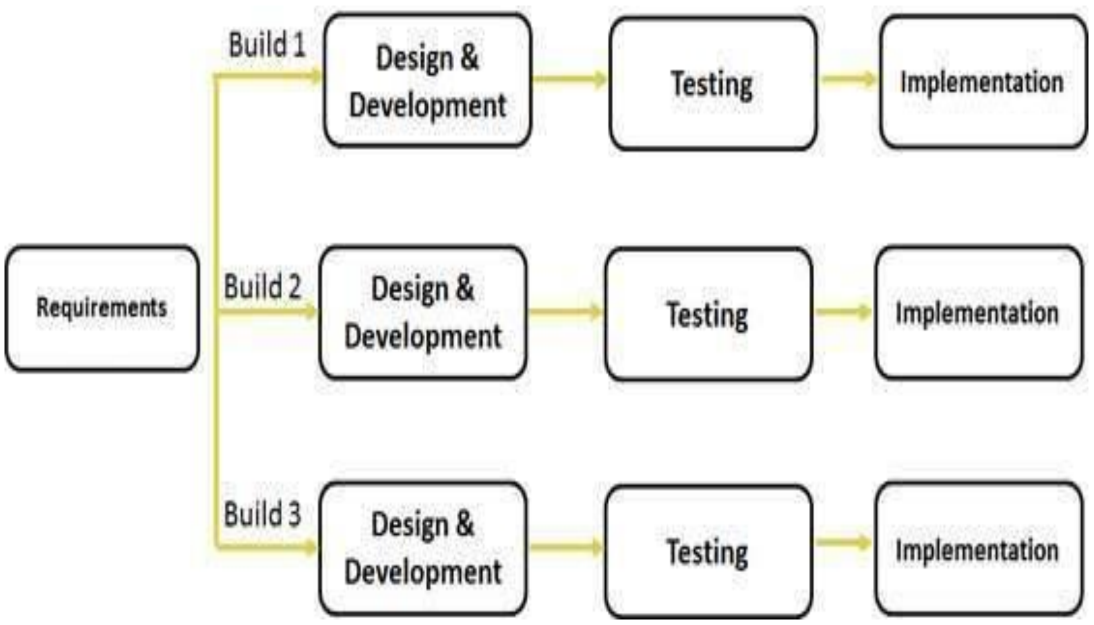
Disadvantages:

- ✓ It cannot accommodate changing requirements.
- ✓ Poor model for long projects.
- ✓ High amounts of risk and uncertainty.

ITERATIVE LIFECYCLE MODEL

- It is also called **Incremental Process Model**.
- This model leads the software development process in iterations.

- The Iterative model can be thought of as a **“multi-waterfall” cycle**. Cycles are divided into smaller and easily managed iterations. Each iteration passes through a series of phases, so after each cycle you will get working software.
- Following is the pictorial representation of Iterative and Incremental model:



An iterative life cycle model which consists of repeating the following four phases in sequence:

- ✓ **A Requirements phase,** in which the requirements for the software are gathered and analyzed. Iteration should eventually result in a requirements phase that produces a complete and final specification of requirements.
- ✓ **A Design phase,** in which a software solution to meet the requirements is designed. This may be a new design, or an extension of an earlier design.
- ✓ **An Implementation and Test phase,** when the software is coded, integrated and tested.
- ✓ **A Review phase,** in which the software is evaluated, the current requirements are reviewed, and changes and additions to requirements proposed.

Advantages:

- Testing and debugging during smaller iteration is easy.
- It supports changing requirements.
- Better suits for large projects.

- Each iteration is an easily managed milestone.

Disadvantages:

- End of project may not be known risk.
- Complexity is more.

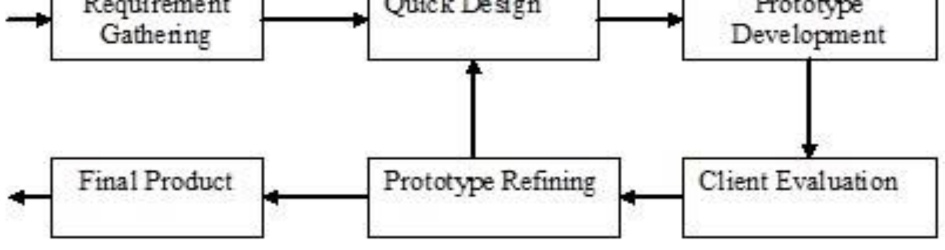
DIFFERENCE BETWEEN WATERFALL AND ITERATIVE MODEL

S.NO	WATERFALL MODEL	ITERATIVE MODEL
1	Once requirement is freezed, changes cannot be done during any phase.	User can change requirement at any phase.
2	Used for smaller projects	Used for larger projects

PROTOTYPE MODEL

- The prototype model is used to **overcome the limitations of waterfall model**. In this model, instead of freezing the requirements before coding or design, a prototype is built to clearly understand the requirements. This prototype is built based on the current requirements. Through examining this prototype, the client gets a better understanding of the features of the final product.
- **Software prototyping** refers to building software application prototypes which display the functionality of product under development, but it does not hold exact logic of original software.
- Prototype is a working model of software with some limited functionality.

DESIGN:



Following is the stepwise approach to design a software prototype:

- **Requirements Gathering:**
    - ✓ Developer and customer meet.
    - ✓ Define overall objective.
- **Quick Design:**
    - ✓ Objects visible to user.
- **Building Prototype:**
    - ✓ Leads to construction of a prototype.
- **Customer Evaluation of Prototype:**
    - ✓ Prototype is evaluated by customer.
- **Refining Prototyping:**
    - ✓ Refine the requirements to develop the software.
- **Engineer Product:**
    - ✓ Prototype serves as a mechanism for identify software requirements.

Software Prototyping Types:

There are different types of software prototypes used in the industry. Following are the major software prototyping types used widely:

- ❖ **Throwaway/Rapid Prototyping:** Derive End system requirements.
- ❖ **Evolutionary Prototyping:** Deliver a working system + requirements
- ❖ **Operational Prototyping:** A trained prototyper keeps track of user.

Advantages:

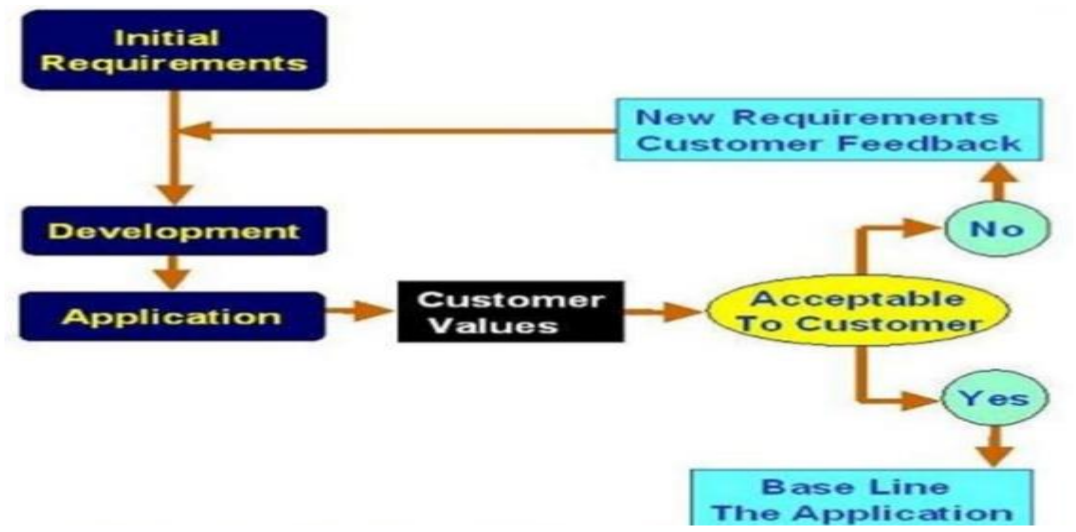
- Increased user involvement in the product even before implementation.
- Reduces time and cost as the defects can be detected much earlier.
- Missing functionality can be identified easily.
- Misunderstanding between developers and customers can be identified.

Disadvantages:

- Users may get confused in the prototypes and actual systems.
- The development cost of prototype may be high depending upon the complexity.

EVOLUTIONARY MODEL

- Evolutionary model is designed to accommodate iterative development of software product.
- Requirements may change in subsequent iterations.
- The below figure represents evolutionary model in software development.



Steps Occur in Evolutionary Model:

- ✓ Listening to the customer to understand the requirements for the software product.
- ✓ Engineering, building and modifying the product based upon the customer requirements.
- ✓ Allowing the customer to evaluate the project.

Advantages:

- ✓ Risk analysis is better.
- ✓ It supports changing requirements.
- ✓ Initial operating time is less.
- ✓ Better suited for critical and mission- critical projects.
- ✓ During lifecycle software is produced early which facilitates customer evaluation and feedback.

Disadvantages:

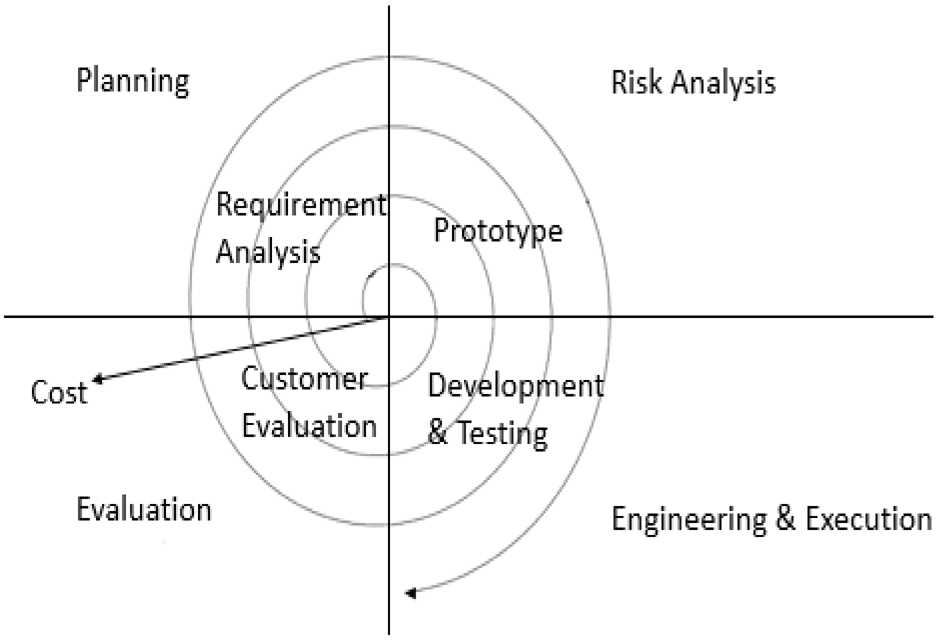
- ✓ Not suitable for smaller projects.
- ✓ Management complexity is more.
- ✓ End of project may not be known which a risk is.
- ✓ Can be costly to use.
- ✓ Highly skilled resources are required for risk analysis.

SPIRAL MODEL

- Spiral model is a **combination of iterative development process model and sequential linear development model** i.e. waterfall model with very high emphasis on risk analysis.
- Spiral model is a sequence of activities with **backtracking**.
- It allows for incremental releases of the product, or incremental refinement through each iteration around the spiral.

Spiral Model Design:

The spiral model has four phases. A software project repeatedly passes through these phases in iterations called Spirals.



Planning Phase:

Requirements are gathered during the planning phase. Requirements like 'BRS' that is 'Business Requirement Specifications' and 'SRS' that is 'System Requirement specifications'.

Risk Analysis:

In the **risk analysis phase**, a process is undertaken to identify risk and alternate solutions. A prototype is produced at the end of the risk analysis phase. If any risk is found during the risk analysis then alternate solutions are suggested and implemented.

Engineering Phase:

In this phase software is **developed**, along with testing at the end of the phase. Hence in this phase the development and testing is done.

Evaluation phase:

This phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral.

Advantages:

- Changing requirements can be accommodated.
- Allows for extensive use of prototypes.
- Requirements can be captured more accurately.
- Important issues are discovered earlier (cost estimation and schedule).
- Changes made in one module will not have any impact on another module.

Disadvantages:

- Limited reusability.
- Requirements and design are not tested.
- Different for each application.
- Management is more complex.
- End of project may not be known early.

DIFFERENCE BETWEEN WATERFALL AND SPIRAL MODEL

S.NO	WATERFALL MODEL	SPIRAL MODEL
1	waterfall model is simple and sequential	Spiral model is complex.
2	waterfall works in sequential flow	Spiral works in loop.
3	In waterfall model the customer should have patience.	In spiral model there is a better communication between developer and customer
4	Risk factor is not considered in waterfall model	Risk factor is considered in the Spiral Model

COMPARE WATERFALL VS INCREMENTAL VS SPIRAL MODEL

Properties of Model	Water-Fall Model	Incremental Model	Spiral Model
Planning in early stage	Yes	Yes	Yes
Returning to an earlier phase	No	Yes	Yes
Handle Large-Project	Not Appropriate	Not Appropriate	Appropriate
Detailed Documentation	Necessary	Yes but not much	Yes
Cost	Low	Low	Expensive
Requirement Specifications	Beginning	Beginning	Beginning
Flexibility to change	Difficult	Easy	Easy
User Involvement	Only at beginning	Intermediate	High
Maintenance	Least	Promotes Maintainability	Typical
Risk Involvement	High	Low	Medium to high risk
Framework Type	Linear	Linear + Iterative	Linear + Iterative
Testing	After completion of coding phase	After every iteration	At the end of the engineering phase
Overlapping Phases	No	Yes (As parallel development is there)	No
Maintenance	Least Maintainable	Maintainable	Yes
Re-usability	Least possible	To some extent	To some extent

Time-Frame	Very Long	Long	Long	
Working software availability	At the end of the life-cycle	At the end of every iteration	At the end of every iteration	
Objective	High Assurance	Rapid Development	High Assurance	
Team size	Large Team	Not Large Team	Large Team	
Customer control over administrator	Very Low	Yes	Yes	

TWO MARKS

1. What is software engineering?

Software engineering is a discipline in which theories, methods and tools are applied to develop professional software.

2. What is Software?

Software is nothing but a collection of computer programs that are related documents that are indented to provide desired features, functionalities and better performance.

3. What are the characteristics of the software?

- Software is engineered, not manufactured.
- Software does not wear out.
- Most software is custom built rather than being assembled from components.

4. What are the various categories of software?

- System software
- Application software
- Engineering/Scientific software
- Embedded software

5. What are the challenges in software Engineering?

- Copying with legacy systems.
- Heterogeneity challenge
- Delivery times challenge.

6. Define software process.

Software process is defined as the structured set of activities that are required to develop the software system.

7. What are the fundamental activities of a software process?

- Specification
- Design and implementation
- Validation
- Evolution

8. Differentiate between Software Engineering and Computer Science.

Software Engineering	Computer Science
The software engineering is concerned with the practical problems of producing software.	The computer science deals with the theories and methods used by the computers and software systems.
Some knowledge of computer science is necessary for the software engineers to develop the software.	Elegant theories cannot be completely applicable to the software engineering when software solution to complex or real world problem has to be developed.

9. What is the difference between software engineering and system engineering?

Software Engineering	System Engineering
Software engineering deals with designing and developing software of the highest quality.	System Engineering is the sub discipline of engineering which deals with the overall management of engineering projects during their life cycle (focusing more on physical aspects).
Software Engineering focuses more on n implementing quality software.	System Engineers focus more on users and domains

10. What is software process model?

The software process model can be defined as an abstract representation of process. The software process model consists of various process activities, role of people involved in process development and software product.

11. How to determine the costs in the software engineering.

The costs in the software engineering system depends upon- the type of software being developed and the software process being used.

12. How to create good software (Attributes of good software).

Satisfying the user requirement is not the only goal of good software. There some essential attributes of good software and those are,

i) Maintainability
ii) Usability
iii) Dependability
iv) Efficiency

13. What are goals/ objectives of software engineering?

- Satisfy user requirements.
- High reliability.
- Low maintenance costs
- Delivery on time
- Low production costs.
- High performance.
- Ease of reuse.

14. What are Generic and custom Product?

- Generic Product: Developed to be sold to a range of different customer.
- Custom Product: Developed for single customer according to their specification.

15. What are the umbrella activities of a software process?

- Software project tracking and control.
- Risk management.
- Software Quality Assurance.
- Formal Technical Reviews.
- Software Configuration Management.
- Work product preparation and production.
- Reusability management.
- Measurement.

16. What is the Key challenges facing software engineering?

- Heterogeneity – Developing techniques for building software that can cope with heterogeneous platform and execution environments.
- Trust – Software must be trustworthy. Software can be trusted by its user.
- Delivery – Developing techniques that can make the delivery fast.

17. List the task regions in the Spiral model.

- Customer communication - it is suggested to establish customer communication.

- Planning – All planning activities are carried out
- Risk analysis – The tasks required to calculate technical and management risks.
- Engineering – tasks required to build one or more representations of applications
- Construct and release – tasks required to construct, test, install the applications
- Customer evaluation - tasks are performed and implemented at installation stage based on the customer evaluation.

18. What are the drawbacks of spiral model?

- ✓ It is based on customer communication. If the communication is not proper then the software product that gets developed will not be the up to the mark.
- ✓ It demands considerable risk assessment. If the risk assessment is done properly then only the successful product can be obtained.

19. What is System Engineering?

System Engineering means designing, implementing, deploying and operating systems which include hardware, software and people. Various phases of System engineering process:

- Requirement analysis
- System design
- Sub_ System development
- System Integration
- System deployment
- System evolution

20. Define the computer based system.

The computer based system can be defined as “a set or an arrangement of elements that are organized to accomplish some predefined goal by processing information”.

21. What is the difference between verification and validation?

Verification	Validation
1. Are we building the product right?	1. Are we building the right product?
2. Verification shows conformance with specification.	2. Validation shows that the program meets the customer’s needs

22. What are the steps followed in testing?

1) Unit testing - The individual components are tested in this type of testing.

2) Module testing – Related collection of independent components are tested.

3) Sub-system testing –Various modules are integrated into a subsystem and the whole subsystem is tested.

4) System testing – The whole system is tested in this system.

5) Acceptance testing – This type of testing involves testing of the system with customer data.

23. Name the Evolutionary process Models.

- Incremental model
- Spiral model
- WIN-WIN spiral model
- Concurrent Development

24. What is meant by Software engineering paradigm?

The development strategy that encompasses the process, methods and tools and generic phases is often referred to as a process model or software engineering paradigm.

25. What are the various elements for computer based system?

- ✓ Software
- ✓ Hardware
- ✓ People
- ✓ Database
- ✓ Documentation
- ✓ Procedures

26. Define software life cycle.

Software life cycle is the period of time beginning with a concept for a software product and ending whenever the software is no longer available for use.

27. Types of lifecycle model.

- Prescriptive process model
- Water fall model
- Incremental process Model
    1. Incremental Model
    2. RAD Model
- Evolutionary process Model
    1. Prototyping model
    2. Spiral Mode
    3. The concurrent development Model
- Specialized Process Model
- Unified process model

28. Define software prototyping.

Software prototyping is defined as a rapid software development for validating the requirements.

29. What are the prototyping approaches in software process?

- **Evolutionary prototyping** – the initial prototype is prepared and it is then refined through number of stages to final stage.
- **Throw-away prototyping** – a rough practical implementation of the system is produced. The requirement problems can be identified from this implementation.

30. Define Product engineering.

The goal of product engineering is to translate the customer's desire for a set of defined functionalities into a working product. Four components are software, hardware, data, and people.

31. Difference between Program and Software product?

Program	**Software Product**
Programs are developed by individual user for their personal use.	Software product are the developed by developers for their commercial uses.
Programs are small in size	It is large in size
Programs are developed by single user	Software product are developed by Multiple users
Little documentation is expected	Large documentation is expected

SOFTWARE ENGINEERING [CS T55]

UNIT- II

Software Project Management and Requirements Analysis: Responsibilities of a Software Project Manager – Project Planning – Metrics for Project Size Estimation – Empirical Estimation Techniques – COCOMO – Halstead's Software Science – Staffing Level Estimation – Scheduling – Organization and Team structures – Staffing – Risk Management – Software Configuration Management – Requirements Gathering and Analysis – Software Requirements specification – Formal System Specification – Axiomatic Specification - Algebraic Specification – 4GL.

RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER

A project manager is a person who has the overall responsibility for the successful initiation, planning, design, execution, monitoring, controlling and closure of a project. Construction, petrochemical, architecture, information technology and many different industries that produce products and services use this job title.

The project manager must have a combination of skills including an ability to ask penetrating questions, detect unstated assumptions and resolve conflicts, as well as more general management skills.

Roles and Responsibilities:

- Project proposal writing,
- Project cost estimation,

- Scheduling,
- Project staffing,
- Project monitoring and control,
- Software configuration management,
- Risk management,
- Managerial report writing and presentations, etc.

Software Management Activities:

Project management activities may include:

- **Project Planning:** Software project planning is task, which is performed before the production of software actually starts.
- **Scope Management:** It defines the scope of project; this includes all the activities, process need to be done in order to make a deliverable software product.
- **Project Estimation:** With correct estimation managers can manage and control the project more efficiently and effectively.

PROJECT PLANNING

- Software project planning is task, which is performed before the production of software actually starts.
- It is there for the software production but involves no concrete activity that has any direction connection with software production;
- Rather it is a set of multiple processes, which facilitates software production. Project planning may include the following:

1. Introduction
2. Project Organization
3. Risk analysis
4. Work breakdown – Mile stone and deliverables.
5. Project schedule
6. Hardware and software resource requirements.
7. Monitoring and reporting mechanism.

METRICS FOR PROJECT SIZE ESTIMATION

Metric:

- Metric is a quantitative measure of the degree to which a system, component or process possesses an attribute.
- The main goal of metrics is to improve product quality and development team productivity.

Software Measurements:

Software measurements are of two categories, namely, **direct measures** and **indirect measures**.

Direct Measures:

Direct measures include software processes like **cost and effort** applied and products like **lines of code** produced, **execution speed**, and **other defects** that have been reported.

Indirect Measures:

Indirect measures include products like **functionality**, **quality**, **complexity**, **reliability**, **maintainability**, and many more.

Metrics to Estimate Size:

- Two metrics are popularly being used widely to estimate size:
- ✓ Lines of code (LOC)
- ✓ Function point (FP)
- ✓ Test Point (TP)
- The usage of each of these metrics in project size estimation has its own advantages and disadvantages.
- Software sizing is an activity in software engineering that is used to estimate the size of a software application or component in order to be able to implement other software project management activities.
- Sizing estimates the probable size of a piece of software while effort estimation predicts the effort needed to build it.
- Size can be estimated in Lines of code or function point.
- Lines of code cannot be estimated correctly before software completion because it varies due to language complexities of different language.
- While Function points are technologically independent, consistent, repeatable, help normalize data, enable comparisons and set project scope and client expectation.

Lines Of Code (LOC):

- "A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program header, declarations, and executable and non-executable statements."
- LOC is the simplest among all metrics available to estimate project size. This metric is very popular because it is the simplest to use.

- Using this metric, the project size is estimated by counting the number of source instructions in the developed program.
- Obviously, while counting the number of source instructions, lines used for commenting the code and the header lines should be ignored.

For Example the below C Program of calculating the Average of three numbers. This C program is written in 9 lines including comments that is begins with “/*”. Remember that C comments are not executed.

Line no	
1	Main()
2	{
3	int a, b, c, avg;
4	/*enter value of a,b,c*/
5	scanf(“%d %d %d”, &a, &b, &c
6	6 /*calculate average*/
7	avg = (a+b+c)/3;
8	printf(“avg = %d”, avg);
9	}

According to the definition of the LOC, above Program has 7 **Lines Of Code**.

Advantage of LOC:

- Counting LOC of any Program text is a very simplest method.
- Comment and blank lines should not be count.

Disadvantage of LOC:

- Size can vary with coding style.
- Focuses on coding activity alone.
- Difficult to estimate LOC from problem description and therefore it is not very useful for project planning.
- LOC is language dependent. For example a line of assembler is not the same as a line of COBOL.

Function Point (FP):

- This metric overcomes many of the shortcomings of the LOC metric.
- The function point metric is that it can be used to easily estimate the size of a software product directly from the problem specification.

- This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed.
- Function point measures functionality from the user's point of view, that is, on the basis of what the user requests and receives in return from the system.

The Function point is obtained by using the following relationship:

FP=UFP*CAF

Where CAF is complexity adjustment factor and UFP is Unadjusted Function Point. Where CAF is complexity adjustment factor and is equal to [0.65+0.01*∑fi]. The fi (i=1 to 14) are the degrees of influence.

UFP = (no of inputs)*4 + (no of outputs)*5 + (no of inquires)*4 + (no of files)*10 + (no of interfaces)*10

For example, consider a project with the following functional units:

Number of user inputs = 50

Number of user outputs = 40

Number of user enquiries = 35

Number of user files = 06

Number of external interfaces = 04

Assume all complexity adjustment factors and weighting factors are average. Compute the function points for the project.

UFP = (no of inputs)*4 + (no of outputs)*5 + (no of inquires)*4 + (no of files)*10 + (no of interfaces)*10

UFP= 50*4+40*5+35*4+6*10+4*7

UFP = 628

CAF= (0.65+0.01(∑ fi))

= (0.65+0.01(14*3))

CAF = 1.07

FP= UFP*CAF

=628*1.07

FP=672

Advantages of FP:

- Function Point helps in software development cost estimation.
- It's also helps in improving product quality.
- Function point work well with Use Cases.

Disadvantages of FP:

- Function point suffers from a major drawback that the size of a function is considered to be independent of its complexity.

Test Point (TP):

Test point used for test point analysis (TPA), to estimate test effort for system and acceptance tests. However, it is important to note that TPA itself only covers black-box testing.

Test effort refers to the expenses for (still to come) tests. There is a relation with test costs and failure costs (direct, indirect, costs for fault correction). Some factors which influence test effort are: maturity of the software development process, quality and testability of the test object, test infrastructure, skills of staff members, quality goals and test strategy.

EMPIRICAL ESTIMATION TECHNIQUES

Estimation is the process of finding an estimate, or approximation, which is a value that can be used for some purpose even if input data may be incomplete, uncertain, or unstable.

Estimation determines how much money, effort, resources, and time it will take to build a specific system or product. Estimation is based on:

- Past Data/Past Experience
- Available Documents/Knowledge
- Assumptions
- Identified Risks

The four basic steps in Software Project Estimation are :

- Estimate the size of the development product.
- Estimate the effort in person-months or person-hours.
- Estimate the schedule in calendar months.
- Estimate the project cost in agreed currency.

There are two broad categories of empirical estimation techniques:

- ✓ Expert Judgment Technique
- ✓ Delphi cost estimation

Expert Judgment Technique:

- Expert judgment is one of the most widely used estimation techniques.

- In this approach, an expert makes an educated guess of the problem size after analyzing the problem thoroughly.
- Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) of the system and then combines them to arrive at the overall estimate.
- However, this technique is subject to human errors and individual bias. Also, it is possible that the expert may overlook some factors inadvertently.
- Further, an expert making an estimate may not have experience and knowledge of all aspects of a project.
- For example, he may be conversant with the database and user interface parts but may not be very knowledgeable about the computer communication part.
- A more refined form of expert judgment is the estimation made by group of experts.
- Estimation by a group of experts minimizes factors such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates.

Delphi Cost Estimation:

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

COCOMO (CONSTRUCTIVE COST MODEL) (OR) HEURISTIC TECHNIQUES

- The Constructive Cost Model (COCOMO) is an algorithmic software cost estimation model developed by Barry W. Boehm.
- A method for **evaluating and/or estimating the cost of software development**.
- The COCOMO model predicts the effort and duration of a project based on inputs relating to the size of the resulting systems and a number of "cost drives" that affect productivity.
- The COCOMO cost estimation model is used by thousands of software project managers, and is based on a study of hundreds of software projects.

Effort:

The Effort Equation is calculated by,

$$\mathbf{PM = C * (KDSI)^n} \text{ (person-months)}$$

Where **PM** = number of person-month (=152 working hours), **C** = a constant, **KDSI** = thousands of "delivered source instructions" (DSI) and n = a constant.

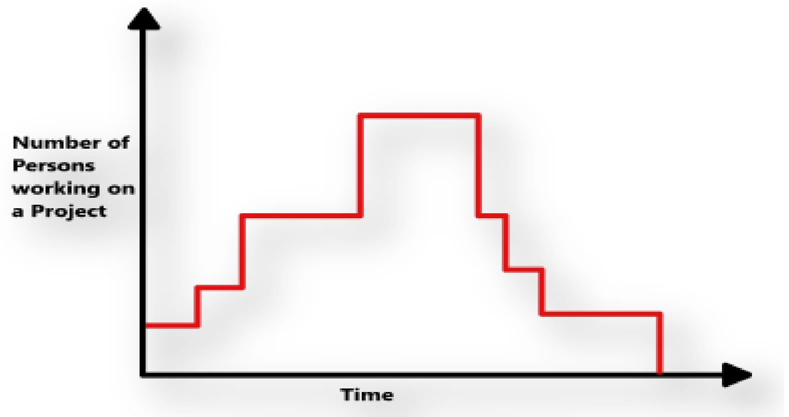


Fig. Person Month Curve

Productivity:

The Productivity equation is,

$$\mathbf{Productivity = (DSI) / (PM)}$$

Where **PM** = number of person-month (=152 working hours), **DSI** = "delivered source instructions".

Schedule:

The Schedule equation is,

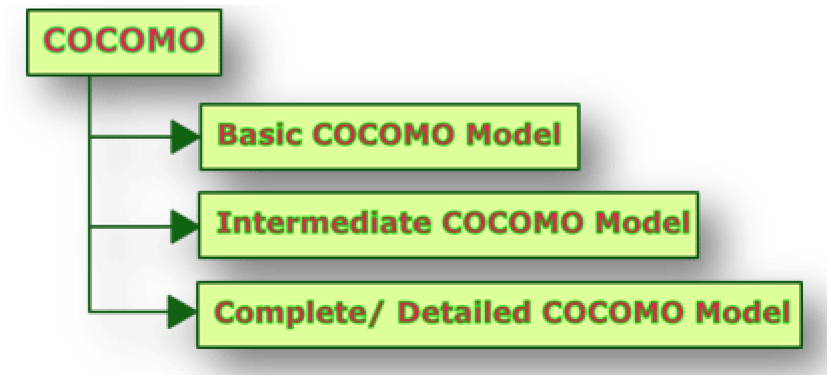
$$\mathbf{TDEV = C * (PM)^n} \text{ (months)}$$

Where TDEV = number of months estimated for software development.

COCOMO Model Stages:

According to software cost estimation should be done through three stages:

1. Basic COCOMO Model
2. Intermediate COCOMO Model
3. Complete/Detailed COCOMO Model



Basic COCOMO Model:

- ✓ It is used for relatively small project.
- ✓ Only a few cost drivers are associated.
- ✓ Cost drivers depend upon project size mainly.
- ✓ Useful when the team size is small, i.e. small staff.

The basic cocomo gives the magnitude of cost of the project. It varies depending upon size of the project.

Intermediate COCOMO Model:

- ✓ It is used for medium sized projects.
- ✓ The cost drivers are intermediate to basic and advanced cocomo.
- ✓ Cost drivers depend upon product reliability, database size, execution and storage.
- ✓ Team size is medium.

Detailed COCOMO Model:

- ✓ It is used for large sized projects.
- ✓ The cost drivers depend upon requirements, analysis, design, testing and maintenance.
- ✓ Team size is large.

DEVELOPMENT TYPES OF MODE:

There are three modes of development:

1. **Organic Mode:**
   - o Relatively Small, Simple Software projects.

- Small teams with good application experience work to a set of less than rigid requirements.
- Similar to previously developed projects.
- Relatively small and require little innovation.

2. **Semidetached Mode:**
   - Intermediate (in size and complexity) software projects in which teams with mixed experience levels must meet a mix of rigid and less than rigid requirements.
3. **Embedded Mode:**

Software projects that must be developed within set of tight hardware, software and operational Constraints.

Development Mode Project Characteristics:

	Size	**Innovation**	**Deadline**	**Dev. Environment**
ORGANIC	Small	Little	Not Tight	Stable
SEMI-DITACHED	Medium	Medium	Medium	Medium
EMBEDDED	Large	Greater	Tight	Complex Hardware

Advantages of COCOMO Model:

- COCOMO is factual and easy to interpret. One can clearly understand how it works.
- Accounts for various factors that affect cost of the project.
- It Works on historical data and hence is more predictable and accurate.
- The drivers are very helpful to understand the impact on the different factors that affect the project costs.

Disadvantages of COCOMO Model:

- COCOMO model ignores requirements and all documentation.
- It ignores customer skills, cooperation, knowledge and other parameters.
- It ignores hardware issues.
- It is dependent on the amount of time spent in each phase.

HALSTEAD'S SOFTWARE SCIENCE (OR) ANALYTICAL ESTIMATION TECHNIQUE

- Halstead introduced metrics to measure software complexity.
- According to Halstead, **"A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operands".**
- Halstead's metrics depends upon the actual implementation of program and its measures, which are computed directly from the operators and operands from source code, in static manner.
- It allows evaluating testing time, vocabulary, size, difficulty, errors, and efforts for C/C++/Java source code.

Halstead's defines various indicators to check complexity of module.

Parameter	Meaning
n1	Number of unique operators
n2	Number of unique operands
N1	Number of total occurrence of operators
N2	Number of total occurrence of operands

Metrics:

1. Token Count:

The size of the vocabulary of a program, which consists of the number of unique tokens used to build a program, is defined as:

$$\eta = \eta 1 + \eta 2$$

ŋ : vocabulary of a program

ŋ 1 : number of unique operators

ŋ 2 : number of unique operands

2. Length of the program:

The length of the program in the terms of the total number of tokens used is,

$$N = N1+N2$$

N : program length

N1 : total occurrences of operators

N2 : total occurrences of operands

3. Program volume (v):

The program volume (V) is the information contents of the program, measured in mathematical bits. It is calculated as the Program length times the 2-base logarithm of the vocabulary size

$$V = N * \log2 (ŋ)$$

V: Program Volume

4. Difficult level:

The difficulty level or error proneness (D) of the program is proportional to the number of unique operators in the program.

$$D = (ŋ1 / 2) * (N2 / ŋ2)$$

5. Program level (L):

The program level (L) is the inverse of the error proneness of the program. I.e. a low level program is more prone to errors than a high level program.

$$L = 1 / D$$

6. Effort to implement (E):

The effort to implement (E) or understand a program is proportional to the volume and to the difficulty level of the program.

$$E = V * D$$

7. Errors:

An error (or fault) is a design flaw or a deviation from a desired or intended state. An error won't yield a failure without the conditions that trigger it.

Error = Volume / 3000

8. Testing Time:

Testing is evaluation of the software against requirements gathered from users and system specifications.

Time = Efforts / S, where S=18 seconds.

EXAMPLE:

Let us consider the following C program:

```
main()
{
    int a, b, c, avg;
    scanf("%d %d %d", &a, &b, &c);
    avg = (a + b + c) / 3;
    printf("av4g = %d", avg);
}
```

The unique operators are: main, (), {}, int, scanf, &, =, +, /, printf

The unique operands are: a, b, c, avg, "%d %d %d", 3, "avg = %d"

Operators (n1)	**Operands (n2)**	**Number of total occurrence of operators (N1)**	**Number of total occurrence of operators (N2)**
main	a	1	2
( )	b	2	2
{ }	c	2	2
int	avg	1	4
scanf	"%d %d %d"	1	4
&	3	3	1
=	"avg = %d"	2	
+		2	
/		1	
printf		1	

Here,

n1 = 10, n2 = 7

N1 = 16, N2 = 15

1. Token Count:

ŋ = ŋ 1+ ŋ 2

= 10 + 7

n = 17

2. Size of Program:

N = N1+N2

= 16 + 15

N = 31

3. Program Length:

V = N * log2 (ŋ)

= 31 *log2 (17)

V = 126.7

4. Difficult level:

D = (ŋ1 / 2) * (N2 / ŋ2)

= (10/2) * (15 /7)

D = 10.7

5. Program level (L):

L = 1 / D

= 1 / 10.7 = **0.093**

6. Effort:

E = V * D

= 126.7 * 10.7

E = 1355.69

7. Errors:

Error = Volume / 3000

= 126.7 / 3000

Error = 0.042

8. Testing Time:

Time = Efforts / S

= 1355.69 / 18

Time = 75.3 seconds

STAFFING LEVEL ESTIMATION

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

Norden's Work:

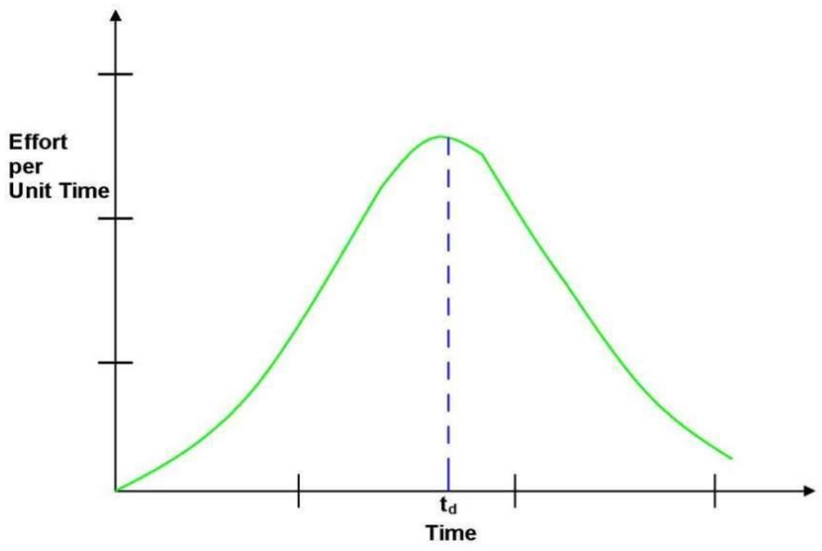
- Norden studied the staffing patterns of several R & D projects. He found that the staffing pattern can be approximated by the Rayleigh distribution curve.
- Norden represented the Rayleigh curve by the following equation:

$$\mathbf{E = K/t^2d * t * e\text{-}t^2 / 2\ t^2\ d}$$

Where E is the effort required at time t. E is an indication of the number of engineers (or the staffing level) at any particular time during the duration of the project, K is the area under the curve, and td is the time at which the curve attains its maximum value.

- It must be remembered that the results of Norden are applicable to general R & D projects and were not meant to model the staffing pattern of software development projects.

(Rayleigh curve)



Putnam's Work:

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

L = Ck K1/3td 4/3

The various terms of this expression are as follows:

• K is the total effort expended (in PM) in the product development and L is the product size in KLOC.

• td corresponds to the time of system and integration testing. Therefore, td can be approximately considered as the time required developing the software.

• Ck is the state of technology constant and reflects constraints that impede the progress of the programmer. Typical values of Ck = 2 for poor development environment (no methodology, poor documentation, and review, etc.), Ck = 8 for good software development environment (software engineering principles are adhered to), Ck = 11 for an excellent environment.

- Putnam suggested that optimal staff build-up on a project should follow the Rayleigh curve. Only a small number of engineers are needed at the beginning of a project to carry out planning and specification tasks.
- As the project progresses and more detailed work is required, the number of engineers reaches a peak.
- After implementation and unit testing, the number of project staff falls. However, the staff build-up should not be carried out in large installments.

- The team size should either be increased or decreased slowly whenever required to match the Rayleigh-Norden curve.
- Experience shows that a very rapid buildup of project staff any time during the project development correlates with schedule slippage.

PROJECT SCHEDULING

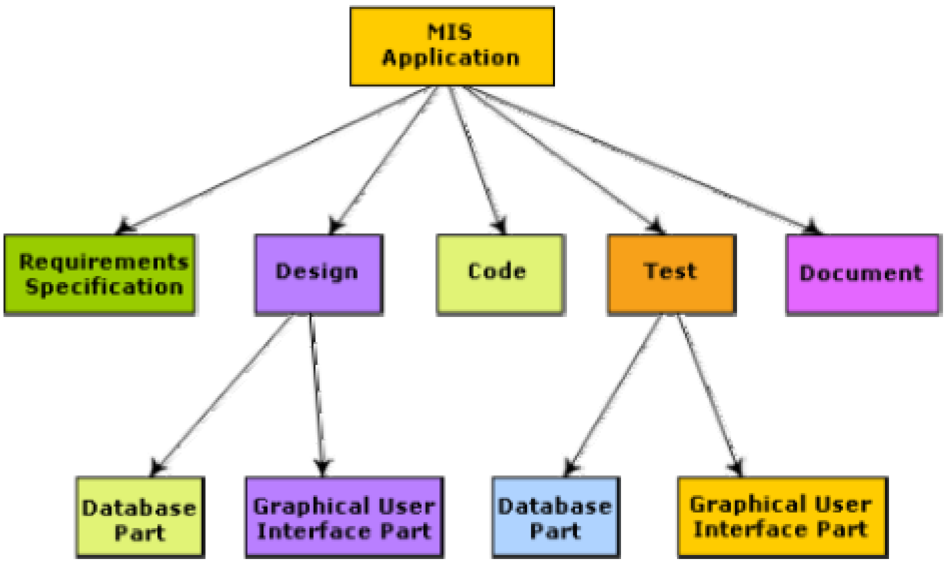
Project-task scheduling is an important project planning activity. It involves deciding which tasks would be taken up when.

In order to schedule the project activities, a software project manager needs to do the following:

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

Work breakdown structure:

- Work Breakdown Structure (WBS) is used to decompose a given task set recursively into small activities.
- WBS provides a notation for representing the major tasks need to be carried out in order to solve a problem.
- The root of the tree is labeled by the problem name. Each node of the tree is broken down into smaller activities that are made the children of the node.
- Figure represents the WBS of MIS (Management Information System) software.

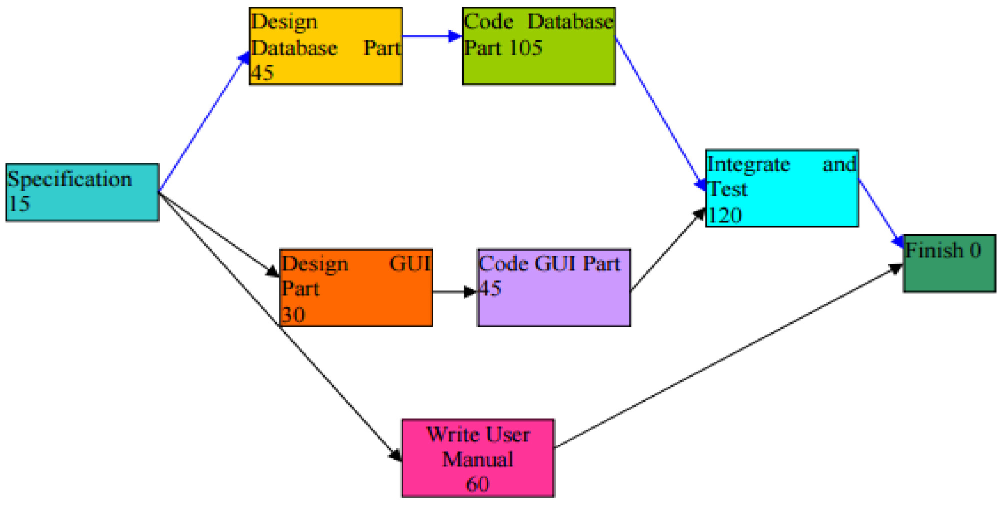

MIS Application
Requirements Specification
Design
Code
Test
Document
Database Part
Graphical User Interface Part
Database Part
Graphical User Interface Part

- While breaking down a task into smaller tasks, the manager has to make some hard decisions. If a task is broken down into large number of very small activities, these can be carried out independently. Thus, it becomes possible to develop the product faster.

Activity networks and critical path method:

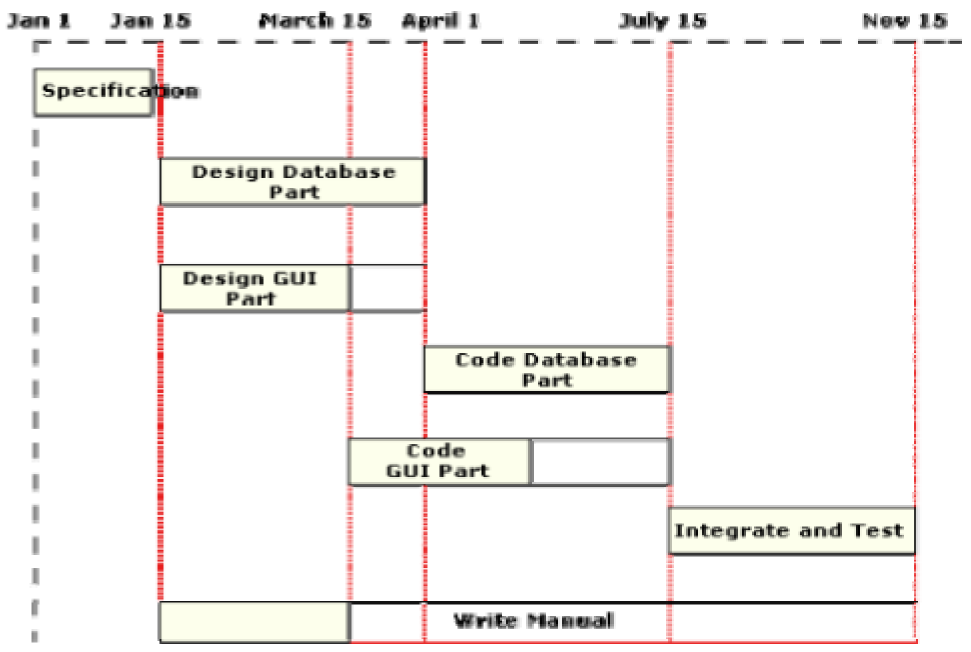
- WBS representation of a project is transformed into an activity network by representing activities identified in WBS along with their interdependencies. An activity network shows the different activities making up a project, their estimated durations, and interdependencies.
- Each activity is represented by a rectangular node and the duration of the activity is shown alongside each task.

Activity network representation of the MIS problem



Gantt chart:

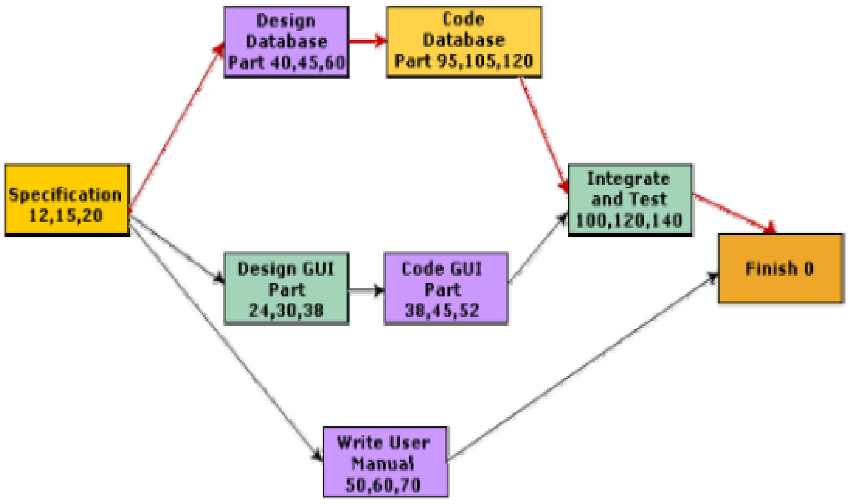
- Gantt charts are mainly used to allocate resources to activities. The resources allocated to activities include staff, hardware, and software. Gantt charts (named after its developer Henry Gantt) are useful for resource planning.
- A Gantt chart is a special type of bar chart where each bar represents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



- Gantt charts are used in software project management are actually an enhanced version of the standard Gantt charts.

PERT Chart:

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


Design Database Part 40,45,60
Code Database Part 95,105,120
Specification 12,15,20
Integrate and Test 100,120,140
Finish 0
Design GUI Part 24,30,38
Code GUI Part 38,45,52
Write User Manual 50,60,70

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

ORGANIZATION AND TEAM STRUCTURES

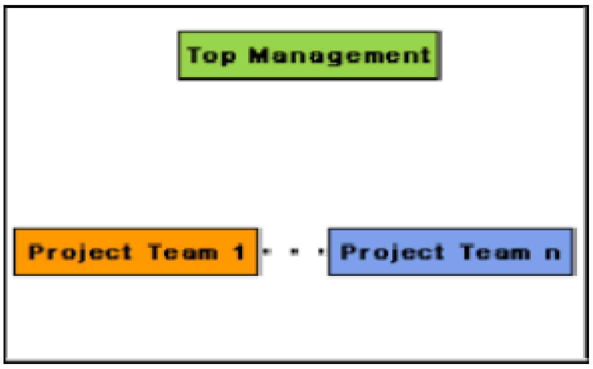
Organization Structure:

- Usually every software development organization handles several projects at any time. Software organizations assign different teams of engineers to handle different software projects.
- Each type of organization structure has its own advantages and disadvantages so the issue "how is the organization as a whole structured?" must be taken into consideration so that each software project can be finished before its deadline.

Software Development Organization:

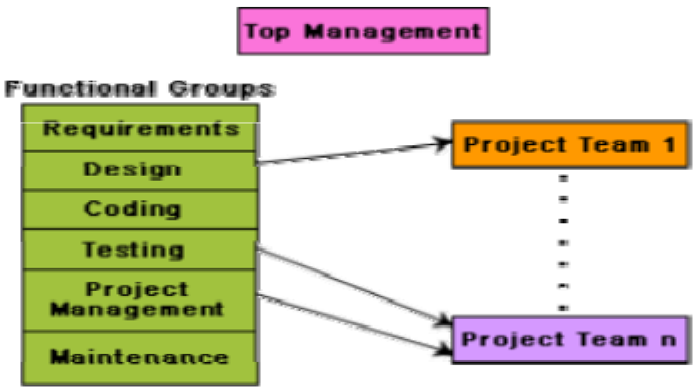
- There are essentially two broad ways in which a software development organization can be structured: functional format and project format.
- In the **project format**, the project development staffs are divided based on the project for which they work.

Project Organization



- In the project format, a set of engineers is assigned to the project at the start of the project and they remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities.
- In the **functional format**, the development staffs are divided based on the functional group to which they belong.

Functional Organization



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

Advantages of Functional Organization over Project Organization:

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

• Ease of staffing

• Production of good quality documents

• Job specialization

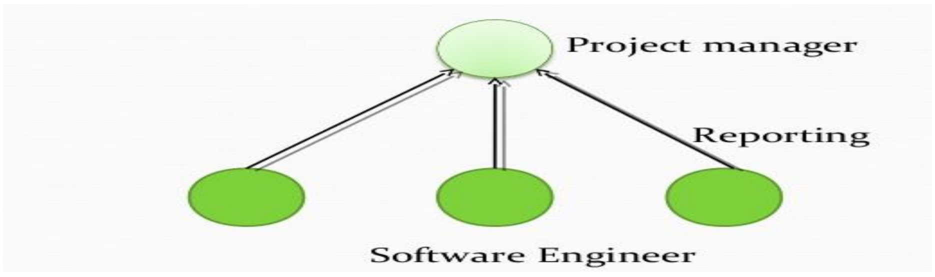
• Efficient handling of the problems associated with manpower turnover.

<u>Team structures:</u>

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

Chief Programmer Team:

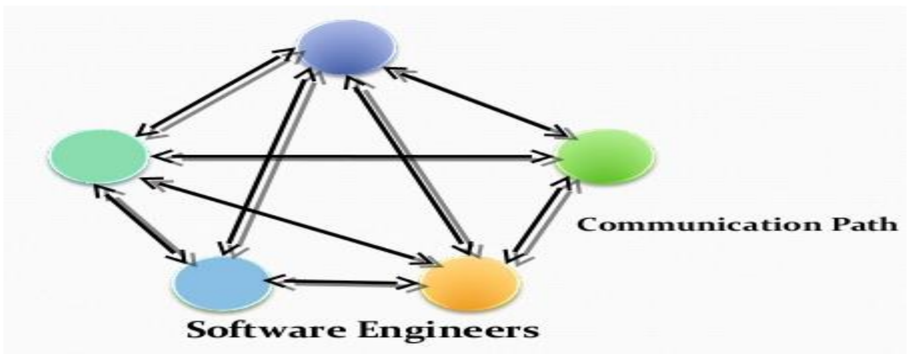
- In this team organization, a senior engineer provides the technical leadership and is designated as the chief programmer.
- The chief programmer partitions the task into small activities and assigns them to the team members. He also verifies and integrates the products developed by different team members.



- The chief programmer provides an authority, and this structure is arguably more efficient than the democratic team for well-understood problems.
- However, the chief programmer team leads to lower team morale, since team-members work under the constant supervision of the chief programmer.

Democratic Team:

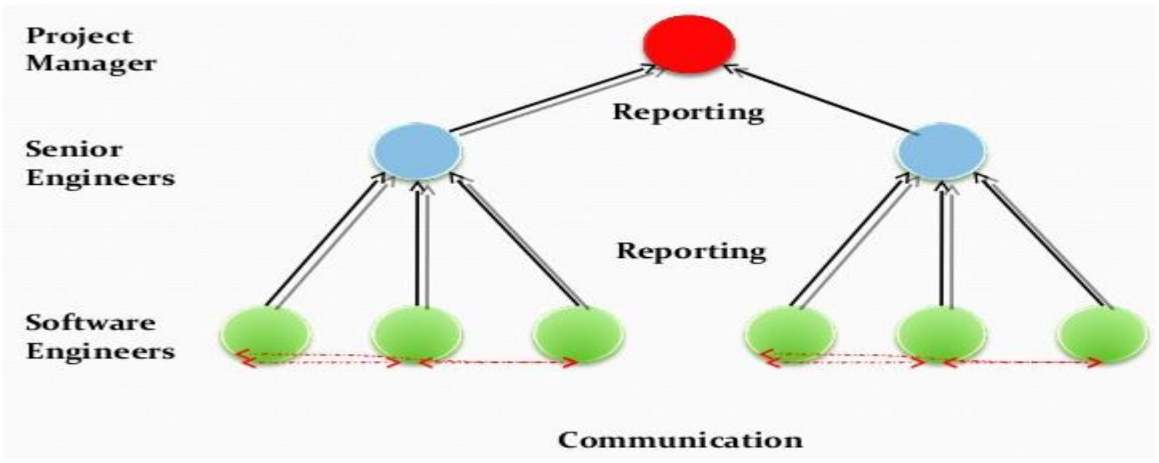
- The democratic team structure, as the name implies, does not enforce any formal team hierarchy.



- Typically, a manager provides the administrative leadership. At different times, different members of the group provide technical leadership.
- A democratic team structure is suitable for projects requiring less than five or six engineers and for research-oriented projects. For large sized projects, a pure democratic organization tends to become chaotic. The democratic team organization encourages egoless programming as programmers can share and review one another's work.

Mixed Control Team Organization:

- The mixed team organization, as the name implies, draws upon the ideas from both the democratic organization and the chief-programmer organization.



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

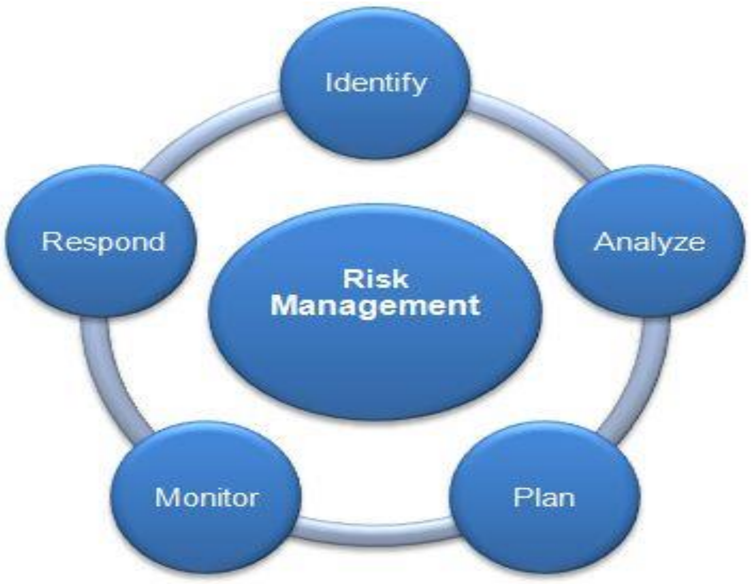
RISK MANAGEMENT

- A risk is any anticipated unfavorable event or circumstance that can occur while a project is underway.
- Risk management aims at reducing the impact of all kinds of risks that might affect a project.
- A software project can be affected by a large variety of risks. In order to be able to systematically identify the important risks which might affect a software project, it is necessary to categorize risks into different classes. The project manager can then examine which risks from each class are relevant to the project.
- Risk management consists of three essential activities:

✓ Risk identification

✓ Risk assessment

✓ Risk containment



<u>Risk Identification and its Types:-</u>

✓ Risks in the project must be minimized by making effective risk management plans.

✓ Risks that are likely to affect a project must be identified and listed.

Types:

- Project risks
- Technical risks
- Business risks

Project risks:

Project risks concern varies forms of budgetary, schedule, personnel, resource, and customer-related problems. An important project risk is schedule slippage.

Technical risks:

Technical risks concern potential design, implementation, interfacing, testing, and maintenance problems.

Business risks:

This type of risks include risks of building an excellent product that no one wants, losing budgetary or personnel commitments, etc.

Risk assessment:

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

• The likelihood of a risk coming true (denoted as r).

• The consequence of the problems associated with that risk (denoted as s).

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

$$p = r * s$$

Where, p is the **priority,** with which the risk must be handled,

r is the probability of the **risk becoming true**, and

s is the **severity of damage caused** due to the risk becoming true.

Risk containment:

After all the identified risks of a project are assessed, plans must be made to contain the most damaging and the most likely risks. There are three main strategies to plan for risk containment:

- ✓ **Avoid the risk:** This may take several forms such as discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the engineers to avoid the risk of manpower turnover, etc.
- ✓ **Transfer the risk:** This strategy involves getting the risky component developed by a third party, buying insurance cover, etc.
- ✓ **Risk reduction:** This involves planning ways to contain the damage due to a risk. For example, if there is risk that some key personnel might leave, new recruitment may be planned.

Risk leverage:

- To choose between the different strategies of handling a risk, the project manager must consider the cost of handling the risk and the corresponding reduction of risk. For this the risk leverage of the different risks can be computed.
- Risk leverage is the difference in risk exposure divided by the cost of reducing the risk. More formally,

$$\text{Risk leverage} = \frac{\text{Risk exposure before reduction} - \text{Risk exposure after reduction}}{\text{Cost of reduction}}$$

SOFTWARE CONFIGURATION MANAGEMENT

- Software Configuration Management (SCM) deals with effectively tracking and controlling the configuration of a software product during its life cycle.
- The state of each deliverable object changes as development progresses and also as bugs is detected and fixed.

SCM Terminology:

- Configuration Item (CI)
- Version, Variant, and Revision
- Configuration
- Baseline
- Workspace

CONFIGURATION ITEM (CI):

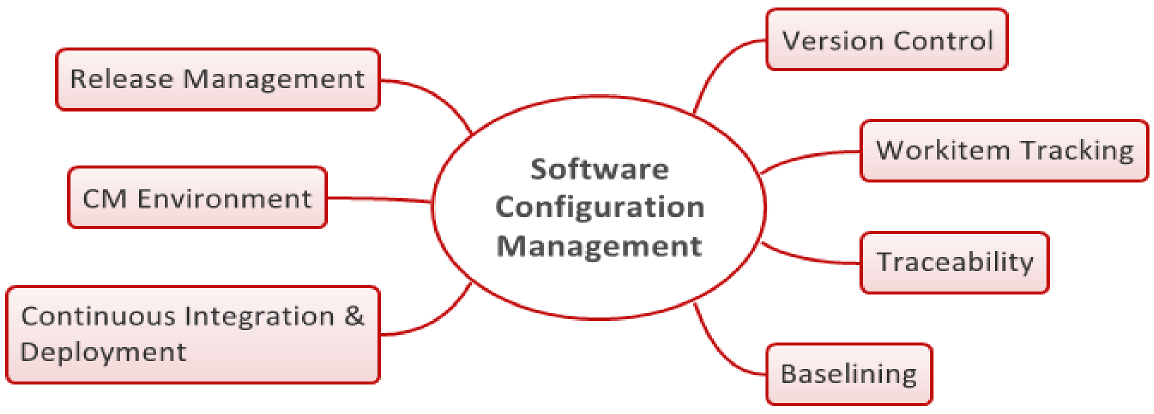
An approved and accepted deliverable, changes have to be made through formal procedure.

Examples:

- Management plan
- Requirement
- Design specification
- Source code and executable code
- Test specification, data, and records
- Log information
- User documentation
- Library and supporting software
- Bug reports, etc.

Version, Variant, and Revision:

- Version: a CI at one point in its development, includes revision and variant
- Revision: a CI linked to another via revision-of relationship, and ordered in time
- Variant: functionally equivalent versions, but designed for different settings, e.g. hardware and software
- Branch: a sequence of versions in the time line.



Configuration:

- An arrangement of functional CIs according to their nature, version and other characteristics
- Guaranteed to recreate configurations with quality and functional assurance
- Sometimes, configuration needs to record environment details, e.g. compiler version, library version, hardware platform, etc.

Baseline:

- A collection of item versions that have been formally reviewed and agreed on, a version of configuration.
- Marks milestones and serves as basis for further development
- Can only be changed via formal change management process
- Baseline + change sets to create new baselines

Workspace:

- An isolated environment where a developer can work (edit, change, compile, test) without interfering other developers.
- Examples
    - Local directory under version control
    - Private workspace on the server

Software Configuration Management Elements:

- **Component elements** – set of tools coupled within a file management system to enable access to and management of each SCI
- **Process elements** – collection of procedures and tasks that define and effective approach to change management for all stakeholders
- **Construction elements** – set of tolls that automate the construction of software by ensure a set of validated components is assembled
- **Human elements** – team uses a set of tools and process features encompassing other CM elements.

SCM Processes:

- Change control process
- Status accounting
- Configuration audit
- Release management
- CM planning

REQUIREMENTS GATHERING AND ANALYSIS

Requirements Engineering:

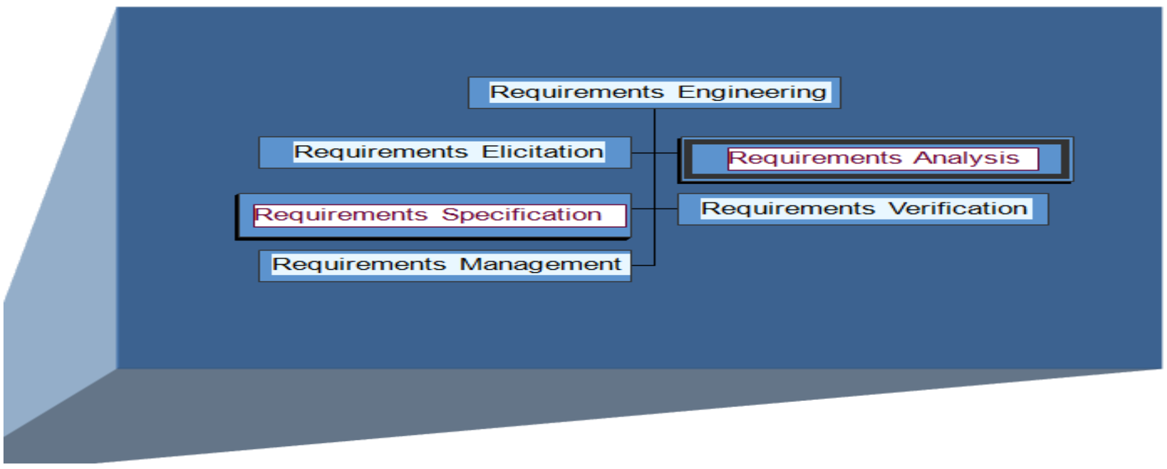
- The process to gather the software requirements from client, analyze and document them is known as requirement engineering.
- The goal of requirement engineering is to develop and maintain sophisticated and descriptive 'System Requirements Specification (SRS)' document.

Requirement Engineering Process:

It is a four step process, which includes –

- Requirement Elicitation (or) Requirement Gathering
- Requirement Analysis
- Requirement Specification
- Requirement Verification
- Requirement Management

(Requirement Engineering)



Requirement Elicitation:

In requirements engineering, requirements elicitation is the practice of collecting the requirements of a system from users, customers and other stakeholders. The practice is also sometimes referred to as **"requirement gathering"**.

Requirement Specification:

SRS is a document created by system analyst after the requirements are collected from various stakeholders.

The requirements received from client are written in natural language. It is the responsibility of system analyst to document the requirements in technical language so that they can be comprehended and useful by the software development team.

Requirement Analysis:

Requirements analysis in systems engineering and software engineering, encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, such as beneficiaries or users.

Requirement Verification:

After requirement specifications are developed, the requirements mentioned in this document are verified and validated. User might ask for illegal, impractical solution or experts may interpret the requirements incorrectly.

Requirement Management:

Requirements management is the process of documenting, analyzing, tracing, prioritizing and agreeing on requirements and then controlling change and communicating to relevant stakeholders.

SOFTWARE REQUIREMENTS SPECIFICATION

A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.

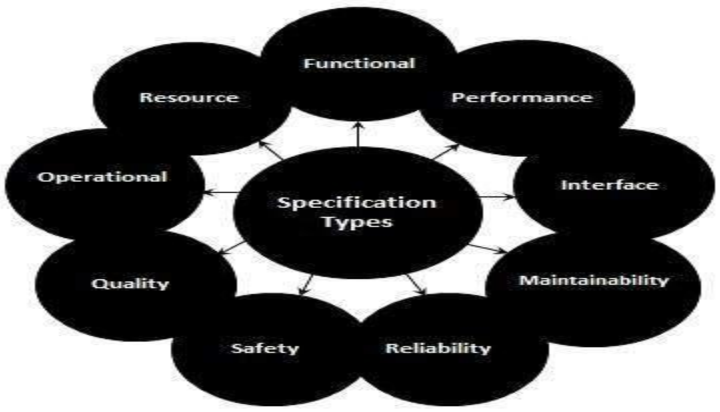
Characteristics of a Good SRS:

- **Correct:** Every requirement given in SRS is a requirement of the software.
- **Unambiguous:** Every requirement has exactly one interpretation.
- **Complete:** Includes all functional, performance, design, external interface requirements; definition of the response of the software to all inputs.
- **Consistent:** Internal consistency.
- **Ranked importance**: Essential vs. desirable.

- **Verifiable:** A requirement is verifiable if and only if there exists some finite cost effective process with which a person or machine can check that the SW meets the requirement.
- **Modifiable:** SRS must be structured to permit effective modifications (e.g. don't be redundant, keep requirements separate)
- **Traceable:** Origin of each requirement is clear.

Types of Requirements:

The below diagram depicts the various types of requirements that are captured during SRS.

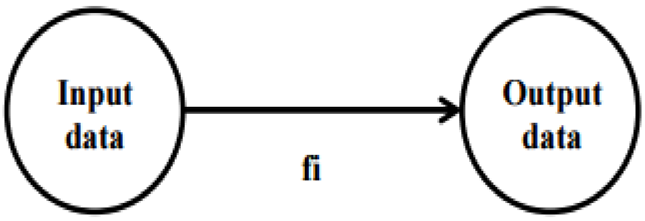


The important parts of SRS document are:

- ✓ Functional requirements of the system
- ✓ Non-functional requirements of the system

Functional Requirements:

The functional requirements part discusses the functionalities required from the system. Here we list all high-level functions {fi} that the system performs. Each high-level function fi, as shown in below figure, is considered as a transformation of a set of input data to some corresponding output data. The user can get some meaningful piece of work done using a high-level function.



Non-Functional Requirements:

Non-functional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

Non-functional requirements may include:

• Reliability issues

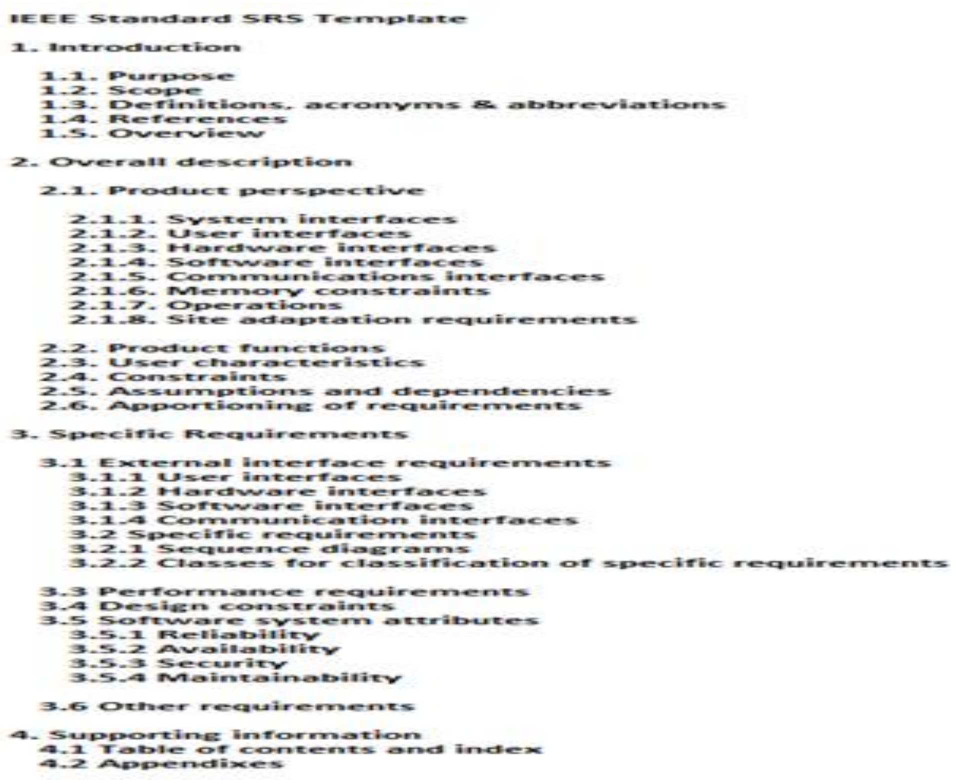
• Accuracy of results

• Human-computer interface issues

• Constraints on the system implementation, etc.

Structure of the Requirements Document:

- A number of large organizations, such as the US Department of Defense and the IEEE, have defined standards for requirements documents.
- The most widely known standard is IEEE/ANSI 830-1998 (IEEE, 1998). This IEEE standard suggests the following structure for requirements documents:



IEEE Standard SRS Template

1. Introduction
    - 1.1. Purpose
    - 1.2. Scope
    - 1.3. Definitions, acronyms & abbreviations
    - 1.4. References
    - 1.5. Overview
2. Overall description
    - 2.1. Product perspective
        - 2.1.1. System interfaces
        - 2.1.2. User interfaces
        - 2.1.3. Hardware interfaces
        - 2.1.4. Software interfaces
        - 2.1.5. Communications interfaces
        - 2.1.6. Memory constraints
        - 2.1.7. Operations
        - 2.1.8. Site adaptation requirements
    - 2.2. Product functions
    - 2.3. User characteristics
    - 2.4. Constraints
    - 2.5. Assumptions and dependencies
    - 2.6. Apportioning of requirements
3. Specific Requirements
    - 3.1 External interface requirements
        - 3.1.1 User interfaces
        - 3.1.2 Hardware interfaces
        - 3.1.3 Software interfaces
        - 3.1.4 Communication interfaces
        - 3.2 Specific requirements
        - 3.2.1 Sequence diagrams
        - 3.2.2 Classes for classification of specific requirements
    - 3.3 Performance requirements
    - 3.4 Design constraints
    - 3.5 Software system attributes
        - 3.5.1 Reliability
        - 3.5.2 Availability
        - 3.5.3 Security
        - 3.5.4 Maintainability
    - 3.6 Other requirements
4. Supporting information
    - 4.1 Table of contents and index
    - 4.2 Appendixes

Purpose of an SRS:

- The SRS precisely defines the software product that will be built.
- SRS used to know all the requirements for the software development and thus that will help in designing the software.
- It provides feedback to the customer.

<u>FORMAL SYSTEM SPECIFICATION</u>

Formal Technique:

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

Formal Specification Language:

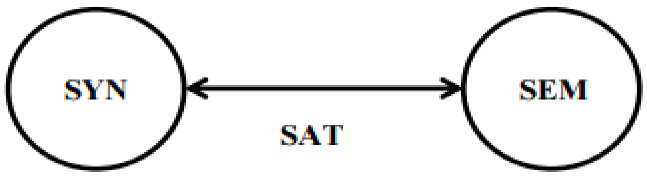
- A formal specification language consists of two sets syn and sem, and a relation sat between them.
- The set syn is called the syntactic domain, the set sem is called the semantic domain, and the relation sat is called the satisfaction relation.

Syntactic Domains:

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

Semantic Domains:

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



Satisfaction Relation:

- Given the model of a system, it is important to determine whether an element of the semantic domain satisfies the specifications.
- This satisfaction is determined by using a homomorphism known as semantic abstraction function.
- The semantic abstraction function maps the elements of the semantic domain into equivalent classes. There can be different specifications describing different aspects of a system model, possibly using different specification languages.

- Some of these specifications describe the system's behavior and the others describe the system's structure.

Limitations of Formal Requirements Specification:

- Formal methods are difficult to learn and use.
- The basic incompleteness results of first-order logic suggest that it is impossible to check absolute correctness of systems using theorem proving techniques.
- Formal techniques are not able to handle complex problems

AXIOMATIC SPECIFICATION

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

Steps to Develop an Axiomatic Specification:

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

- Establish the range of input values over which the function should behave correctly. Also find out other constraints on the input parameters and write them in the form of a predicate.
- Specify a predicate defining the conditions which must hold on the output of the function if it behaved properly.
- Establish the changes made to the function's input parameters after execution of the function. Pure mathematical functions do not change their input and therefore this type of assertion is not necessary for pure functions.
- Combine all of the above into pre and post conditions of the function.

Example:

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

$$f\ (x : \text{real}) : \text{real}$$

$$\text{Pre: } x \in R$$

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

ALGEBRAIC SPECIFICATION

In the algebraic specification technique an object class or type is specified in terms of relationships existing between the operations defined on that type.

Representation of Algebraic Specification:

An algebraic specification is usually presented in four sections.

1. Types section

In this section, the sorts (or the data types) being used are specified.

2. Exceptions section

This section gives the names of the exceptional conditions that might occur when different operations are carried out. These exception conditions are used in the later sections of an algebraic specification.

3. Syntax section

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

$$\text{Stack x element} \rightarrow \text{stack}$$

4. Equations section

This section gives a set of rewrite rules (or equations) defining the meaning of the interface procedures in terms of each other. In general, this section is allowed to contain conditional expressions.

Properties of Algebraic Specifications:

Three important properties that every good algebraic specification should possess are:

Completeness:

This property ensures that using the equations, it should be possible to reduce any arbitrary sequence of operations on the interface procedures. There is no simple procedure to ensure that an algebraic specification is complete.

Finite termination property:

This property essentially addresses the following question: Do applications of the rewrite rules to arbitrary expressions involving the interface procedures always terminate? For arbitrary algebraic equations, convergence (finite termination) is undecidable. But, if the right hand side of each rewrite rule has fewer terms than the left, then the rewrite process must terminate.

Unique termination property:

This property indicates whether application of rewrite rules in different orders always result in the same answer.

Advantages:

Algebraic specifications have a strong mathematical basis and can be viewed as heterogeneous algebra. Therefore, they are unambiguous and precise.

Disadvantages:

Algebraic specifications are difficult to interchange with typical programming languages. Also, algebraic specifications are hard to understand.

4GL (Fourth Generation Language)

- A fourth-generation programming language (1970s-1990) (abbreviated 4GL) is a programming language or programming environment designed with a specific purpose in mind, such as the development of commercial business software.
- In the history of computer science, the 4GL followed the 3GL in an upward trend toward higher abstraction and statement power. The 4GL was followed by efforts to define and use a 5GL.
- All 4GLs are designed to reduce programming effort, the time it takes to develop software, and the cost of software development.
- They are not always successful in this task, sometimes resulting in inelegant and unmaintainable code.
- However, given the right problem, the use of an appropriate 4GL can be successful.
- 3GL development methods can be slow and error-prone.
- Some applications could be developed more rapidly by adding a higher-level programming language and methodology which would generate the equivalent of very complicated 3GL instructions with fewer errors.
- All 4GLs are designed to reduce :
    - Programming effort,
    - The time it takes to develop software
    - The cost of software development.

Types of 4 GL:

- **Table-driven** (codeless) programming, usually running with runtime framework and libraries. Instead of using code.
- **Report generators** take a description of the data format and the report to generate and from that they either generate the required report directly or they generate a program to generate the report.
- **Data management 4GLs** such as SAS, SPSS and Stata provide sophisticated coding commands for data manipulation, file reshaping, case selection and data documentation in the preparation of data for statistical analysis and reporting.

Some Fourth-Generation Languages:

- FoxPro
- PowerBuilder
- SQL
- Report Builder
- Oracle Reports
- Graph Talk
- MATLAB
- CSS

TWO MARKS

1. What are the responsibilities of a software project manager?

- Project proposal writing,
- Project cost estimation,
- Scheduling,
- Project staffing,
- Project monitoring and control,
- Software configuration management,
- Risk management,
- Managerial report writing and presentations, etc.

2. List activities involved in project planning?

- Estimation
  - ✓ Effort, cost, resource, and project duration
- Project scheduling
- Staff organization
  - ✓ staffing plans

- Risk handling
    - ✓ identification, analysis, and abatement procedures

3. List metrics for project size estimation?

- Lines of code (LOC)
- Function point (FP)

4. List project estimation techniques?

Three main approaches to estimation:

- Empirical
- Heuristic
- Analytical

5. Define heuristic estimation technique or COCOMO?

Heuristic techniques assume that the characteristics to be estimated can be expressed in terms of some mathematical expression.

6. Define Hallstead's software science or analytical technique?

Analytical techniques derive the required results starting from certain simple assumptions.

7. List staffing level estimation works?

- Norden's Work
- Putnam's Work:
- Jensen Model

8. Write difference between Gantt chart and PERT chart?

GANTT CHART	PERT CHART
A Gantt chart is a **special type of bar chart** where each bar represents an activity. The bars are drawn along a time line. The **length of each bar is proportional to the duration of time planned** for the corresponding activity.	The boxes of PERT charts are usually annotated with the **pessimistic, likely, and optimistic estimates** for every task.

9. Write differences between function format and project format in organization structure?

FUNCTIONAL ORGANIZATION	PROJECT ORGANIZATION
Engineers are organized into functional groups, e.g. specification, design, coding, testing, maintenance, etc.	**Engineers get assigned to a project for the entire duration of the project** Same set of engineers carry out all the phases

Engineers from functional groups get assigned to different projects	

10. What is Risk management and list activities in it?

- A risk is any anticipated unfavorable event or circumstance that can occur while a project is underway.
- Risk management aims at reducing the impact of all kinds of risks that might affect a project.
- It involves risk identification, risk assessment and risk mitigation.

11. How to compute risk leverage of the different risks?

$$\text{Risk leverage} = \frac{\text{Risk exposure before reduction} - \text{Risk exposure after reduction}}{\text{Cost of reduction}}$$

12. What is software configuration management (SCM)?

Software configuration management deals with effectively tracking and controlling the configuration of a software product during its life cycle.

13. List necessity of Software configuration management (SCM)?

- Inconsistency problem when the objects are replicated.
- Problems associated with concurrent access.
- Providing a stable development environment
- System accounting and maintaining status information
- Handling variants**.** Existence of variants of a software product causes some peculiar problems.

14. What is the goal of the requirements analysis and specification phase?

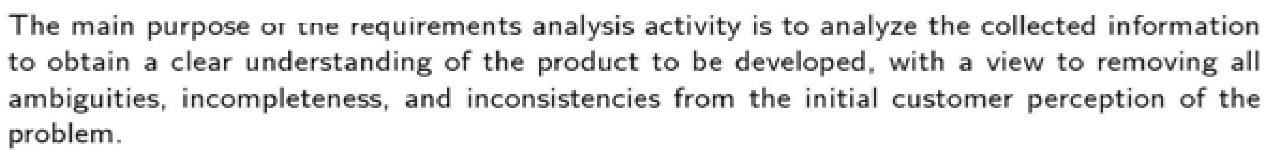
The goal of the requirements analysis and specification phase is to clearly understand the customer requirements and to systematically organize the requirements into a specification document.

15. What are the ways the analyst gathers requirements?

Analyst gathers requirements through:

- observation of existing systems,
- studying existing procedures,
- discussion with the customer and end-users,
- Analysis of what needs to be done, etc.

16. What is the goal of requirements analysis?


The main purpose of the requirements analysis activity is to analyze the collected information to obtain a clear understanding of the product to be developed, with a view to removing all ambiguities, incompleteness, and inconsistencies from the initial customer perception of the problem.

17. Give difference between Model and Property-oriented methods?

In a model-oriented style, one defines **a system's behavior directly** by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, sequences, etc.

In the property-oriented style, the **system's behavior is defined indirectly** by stating its properties, usually in the form of a set of axioms that the system must satisfy.

18. Write Pros and Cons of Algebraic Specification?

Advantage: Unambiguous and precise

Disadvantage: Difficult to integrate with typical programming language and hard to understand.

19. Define 4GL

- 4GLs (Fourth Generation Languages) are examples of **executable specification languages.**
- 4GLs are successful because there is a lot of commonality across data processing applications.
- 4GLs rely on **software reuse** where common abstractions have been identified and parameterized

20. What is formal technique?

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

21. List advantages of formal techniques?

- Formal specifications encourage rigor.
- Formal methods usually have a well-founded mathematical basis.
- Formal methods have well-defined semantics.
- The mathematical basis of the formal methods facilitates automating the analysis of specifications.
- Formal specifications to obtain immediate feedback

UNIT 3

SOFTWARE DESIGN

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

The design document produced at the end of the design phase should be implementable using a programming language in the subsequent (coding) phase.

1.OUTCOME OF A DESIGN PROCESS

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

- Different modules required
  The different modules in the solution should be clearly indentified. Each module is a collection of the functions and the data shared by the functions of the module. Each module should accomplish some well-defined task out of the overall responsibility of the software. Each module should be named according to the task it performs.
- Control relationships among modules
  A control relationship between two modules essentially arises due to function calls between the two modules, the control relationships existing among various modules should be identified in the design document.
- Interfaces among different modules
  The interfaces between two modules identify the exact data items exchanged between the two modules when a one module invokes a function of the other module.
- Data structures of the individual modules
  Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module. Suitable data structures for the data to be stored in a module need to be properly designed and documented.
- Algorithms required to implement the individual modules
  Each function in a module usually performs some processing activity. The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due considerations given to the accuracy of the results, space and time complexities.

1.1 Classification of design activities

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

• Preliminary (or high-level) design and

• Detailed design.

The outcome of high-level design is called the program structure or software architecture.During detailed design, the data structure and the algorithms of the different modules are designed. The outcome of the detailed design stage is usually known as the module-specification document.

1.2 Classification of design methodologies

The design activities vary considerably based on the specific design methodology being used. A large number of software design methodologies are available.

We roughly classify these methodologies into procedural and object-oriented approaches.

Does a design technique ensure a unique solution?

Even while using the same design methodology, different designers usually arrive at very different design solutions.

The reason is that a design technique often requires the designer to make subjective decisions and work out compromises to contradictory objectives.

As a result, it is possible that even the same designer can work out many different solutions to the same problem.

1.3 Analysis versus design

The aim of analysis is to understand the problem with a view to eliminate any deficiencies in the requirement specification such as incompleteness, inconsistencies, etc. The model which we are trying to build may be or may not be ready.

The aim of design is to produce a model that will provide a seamless transition to the coding phase, i.e. once the requirements are analyzed and found to be satisfactory, a design model is created which can be easily implemented.

2. CHARACTERISTICS OF A GOOD SOFTWARE DESIGN

The definition of "a good software design" can vary depending on the application being designed. For example, the memory size used by a program may be an important issue to characterize a good solution for embedded software development – since embedded applications are often required to be implemented using memory of limited size due to cost, space, or power consumption considerations.

For embedded applications, one may sacrifice design comprehensibility to achieve code compactness. For embedded applications, factors like design comprehensibility may take a back seat while judging the goodness of design. Therefore, the criteria used to judge how good a given design solution is can vary widely depending upon the application.

Not only is the goodness of design dependent on the targeted application, but also the notion of goodness of a design itself varies widely across software engineers and academicians. However, most researchers and software engineers agree on a few desirable characteristics that every good software design for general application must possess. The characteristics are listed below:

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

Understandability of a design – a major concern

While performing the design of a certain problem, assume that we have arrived at a large number of design solutions and need to choose the best one.

Given that we are choosing from only correct solutions, understandability of a design solution is possibly the most important issue to be considered while judging the goodness of a design.

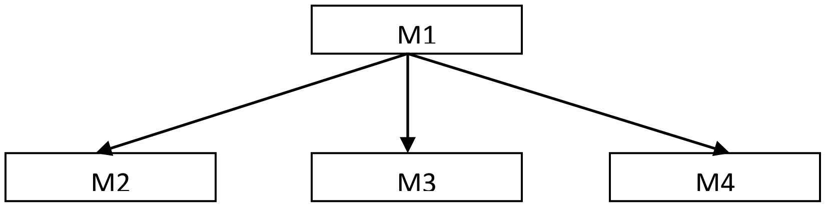
An understandable design is modular and layered

A design solution should have the following features to be easily understandable:

- It should assign consistent and meaningful names to various design components.
- It should make use of the principles of decomposition and abstraction in good measures to simplify the design.

A design solution is understandable, if it is modular and the modules are arranged in layers.

1. **Modularity:**
   A modular design achieves effective decomposition of a problem. It is a basic characteristic of any good design solution. A modular design, in simple words, implies that the problem has been decomposed into a set of modules.
   Decomposition of a problem into modules facilities asking advantage of the divide and conquer principle. If the different modules have either no interaction with each other, then each module can be understood separately.



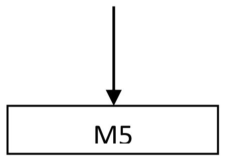


Fig. A modular and hierarchical design

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

2. **Layered design**:
    - A layered design is one that a call relations among different modules are represented graphically and result in a tree-like diagram with clear layering
    - The modules are arranged in a hierarchy of layers.
    - A module can invoke functions of the modules in the layer immediately below it, and it is consider to be implementing control abstractions.
    - It makes design solutions easily understandable and the source of error is detected easily.

3.COHESION AND COUPLING

Good module decomposition is indicated to high cohesion of the individual modules and low coupling of the modules with each other.

Cohesion is a measure of functional strength of a module, whereas the coupling between to modules is the measure of the degree of interactions between the two modules.

3.1 COHESION

When the functions of the module cooperate with each other for performing the single objective, when the module has good cohesion.

If the functions of the module do very different things and do not cooperate with each other to perform the single piece of work, then the module has poor cohesion

3.2 COUPLING

Two situations arises to be highly coupled,

- If the functions call between two modules involve passing large chunks of shared data, the modules are tightly coupled.
- If the interactions occur through some shared data, then also we say that they are highly coupled.

3.3 FUNCTONAL INDEPENDENCE

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

ADVANTAGES OF FUNCTIONAL INDEPENDENCE

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

- **Error isolation:** Functional independence reduces error propagation. The reason behind this is that if a module is functionally independent, its degree of interaction with the other modules is less. Therefore, any error existing in a module would not directly effect the other modules.
- **Scope of reuse:** Reuse of a module becomes possible. Because each module does some well-defined and precise function, and the interaction of the module with the other modules is simple and minimal. Therefore, a cohesive module can be easily taken out and reused in a different program.
- **Understandability:** Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.

3.4 CLASSIFICATION OF COHESIVENESS

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

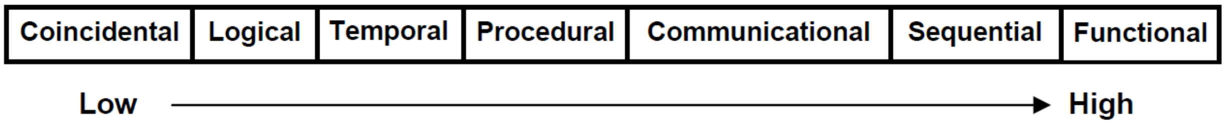


Fig. Classification of cohesion

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

Logical cohesion: A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc. An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.

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

Procedural cohesion: A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message.
Communicational cohesion: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack.

Sequential cohesion: A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next. For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.

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

3.5 CLASSIFICATION OF COUPLING

The degree of coupling between two modules depends on their interface complexity. Even if there are no techniques to precisely and quantitatively estimate the coupling between two modules, classification of the different types of coupling will help to quantitatively estimate the degree of coupling between two modules. Five types of coupling can occur between any two modules. This is shown in fig.

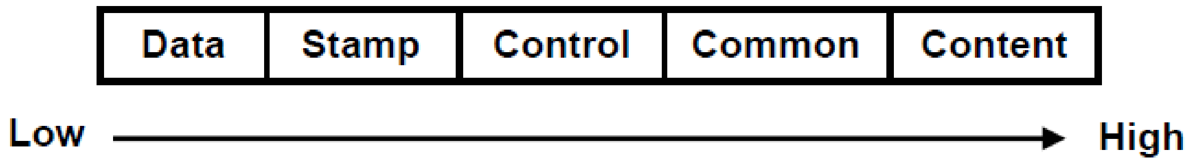


Fig. Classification of coupling

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

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

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

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

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

4. APPROACHES TO SOFWARE DESIGN

There are fundamentally two approaches. They are Function oriented design and object oriented design.

For the development of large programs, the OO approach is increasingly used and function oriented design is a mature technology.

4.1FUNCTION-ORIENTED DESIGN

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

1. **Top down decomposition:**
    - A system is viewed as something that performs a set of functions.
    - Starting at this high-level view of the system, each function is successively refined into more detailed functions.
    - For example, consider a function create-new-library-member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge.
    - This function may consist of the following sub-functions:
        - assign-membership-number
        - create-member-record
        - print-bill

    Each of these sub-functions may be split into more detailed subfunctions and so on.
2. **Centralized system state:**

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

4.2 OBJECT ORIENTED DESIGN

- In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities).
- The state is decentralized among the objects and each object manages its own state information.
- For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data.
- In fact, the functions defined for one object cannot refer or change data of other objects. Objects have their own internal data which define their state.
- Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from super class. Conceptually, objects communicate by message passing.
- Objects can also be considered as instances of abstract dta types(ADTs).
- The extensively used in the ADA programming language introduced in 1970s.
- ADT is a important concept that forms an important pillar of object-orientation..
- Let us discuss some important concepts behind ADT.

Data abstraction

- The principles of data abstraction implies that how data is exactly stored is abstracted away. This means that any entity external to the object would have no knowledge about how data is exactly stored, organized, and manipulated inside the object.
- The entities external to the object can access the data internal to an object only by calling certain well-defined methods supported by the object.
- Consider an ADT such as a stack.
- The data of a stack object may internally be stored in an array, a linearly linked list, or a bidirectional linked list, the external entities have no knowledge of this an d can access data of a stack object only through the supported operations

Data structure

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

Data type

- A type is a programming language terminology that refers to anything that can be instantiated.

- For eg., int, char,float,etc., are the basic datatypes supported by the c programming language and we can say that ADTs are user defined datatypes.
- In Object Orientation, the class are ADTs
- Three advantages of ADTs are,
    - The data of objects are encapsulated within the methods. The encapsulation principles are also known as data hiding.
    - ADT based design displays high cohesion and low coupling. therefore, object oriented designs are highly modular.
    - Principle of abstraction makes a design solution easily understandable and helps to manage complexity.

5.FUNCTION-ORIENTED VS. OBJECT-ORIENTED DESIGN APPROACH

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

1. Unlike function-oriented design methods, in OOD, the basic abstraction are not real-world functions such as sort, display, track, etc, but real-world entities such as employee, picture, machine, radar system, etc. For example in OOD, an employee pay-roll software is not developed by designing functions such as update-employee-record, get-employee-address, etc. but by designing objects such as employees, departments, etc. Grady Booch sums up this difference as "identify verbs if you are after procedural design and nouns if you are after object-oriented design"
2. In OOD, state information is not represented in a centralized shared memory but is distributed among the objects of the system. For example, while developing an employee pay-roll system, the employee data such as the names of the employees, their code numbers, basic salaries, etc. are usually implemented as global data in a traditional programming system; whereas in an object-oriented system these data are distributed among different employee objects of the system. Objects communicate by message passing. Therefore, one object may discover the state information of another object by interrogating it. Of course, somewhere or other the real-world functions must be implemented. In OOD, the functions are usually associated with specific real-world entities (objects); they directly access only part of the system state information.
3. Function-oriented techniques such as SA/SD group functions together if, as a group, they constitute a higher-level function. On the other hand, object-oriented techniques group functions together on the basis of the data they operate on.

To illustrate the differences between the object-oriented and the function-oriented design approaches, an example can be considered.

Example: Fire-Alarm System

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

5.1Function-Oriented Approach:

/* **Global data (system state) accessible by various functions** */
BOOL detector_status[MAX_ROOMS]; int detector_locs[MAX_ROOMS]; BOOL alarm_status[MAX_ROOMS];
/* alarm activated when status is set */ int alarm_locs[MAX_ROOMS];
/* room number where alarm is located */
int neighbor-alarm[MAX_ROOMS][10];
/* each detector has at most 10 neighboring locations */

The functions which operate on the system state are:
interrogate_detectors();
get_detector_location();
determine_neighbor();
ring_alarm();
reset_alarm();
report_fire_location();

5.2 **Object-Oriented Approach:**

class detector

attributes**:** status, location, neighbors

operations**:** create, sense_status, get_location, find_neighbors

class alarm

attributes**:** location, status

operations**:** create, ring_alarm, get_location, reset_alarm

1. In the object oriented program, an appropriate number of instances of the class detector and alarm should be created.

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

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

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

FUNCTION-ORIENTED SOFTWARE DESIGN

During design process, the high level functions are successively decomposed into more detailed functions.

The term top-down decomposition is often used to denote the successive decomposition of the set of the high level functions into detailed functions.

1. OVERVIEW OF SA/SD METHODOLOGY

It carries out two distinct activities.

- structured analysis
- structured design.

During structured analysis, the SRS document is transformed into a DFD model. During structured design DFD model id transformed into structure chart.

It is important to understand that the purpose of structure analysis is to capture the detailed structure of the system as perceived by the user, whereas, the purpose of structure design is to define the structure of the solution that is for implementation in some programming language.

2. STRUCTURED ANALYSIS

Structured analysis is used to carry out the top-down decomposition of a set of high-level functions depicted in the problem description and to represent them graphically.

During structured analysis, functional decomposition of the system is achieved. That is, each function that the system performs is analyzed and hierarchically decomposed into more detailed functions.

Structured analysis technique is based on the following essential underlying principles:

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

The goal of structured analysis is to perform functional decomposition and to represent using Data Flow Diagrams (DFDs).

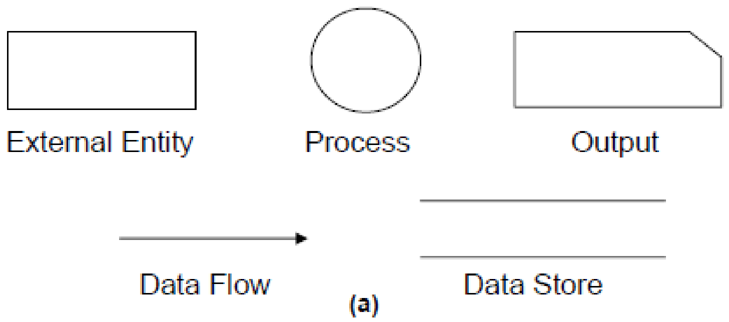
DFDs are a hierarchical model of a system that shows different processing activities or functions that the system performs and the data interchange among those functions.

3. DATA FLOW DIAGRAMS(DFDS)

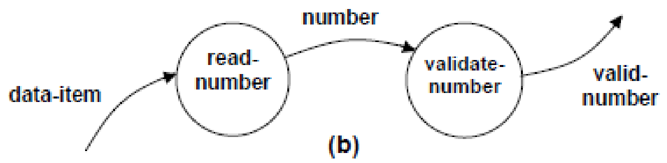
The DFD (also known as a bubble chart) is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among these functions.

Each function is considered as a processing station (or process) that consumes some input data and produces some output data. The system is represented in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system.

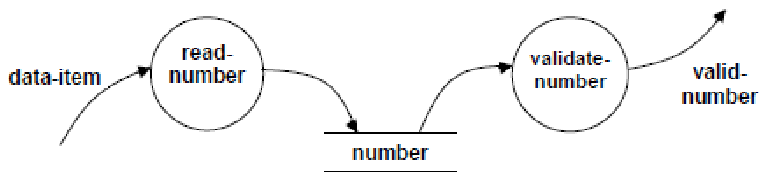
A DFD model uses a very limited number of primitive symbols (as shown in fig) to represent the functions performed by a system and the data flow among these functions.



(a)



(b)



(c)

Fig (a) Symbols used for defining DFDs

Fig (b), (c) Synchronous and asynchronous data flow

3.1 PRIMITIVE SYMBOLS USED FOR CONSTRUCTING DFDs

1. **Function symbol:**
   Function is represented using a circle. This symbol is called a process or a bubble. Bubbles are annotated with the names of the corresponding functions.
2. **External Entity Symbol**
   These are physical entities external to the software systems which interact with the system by inputting data to the system or by consuming the dat produced by the system.
3. **Data Flow Symbol**
   The directed arc used as data flow symbol and it represents the data flow occurring between two process or between an external entity and a process in the directions of the data flow arrow.
4. **Data Store Symbol**
   It is represented using two parallel lines and it represents a logical file. It can represent either the data structure or a physical file on disk. Each data store is connected to a process by means of the data flow symbol
5. **Output Symbol**

It is used when a hard copy is produced.

3.2 SOME IMPORTANT CONCEPTS ASSOCIATED WITH CONSTRUCTING DFD MODELS

1.Sysnchronous and Asysnchronous operations

Here, two examples of data flow that describe input and validation of data are considered. In Fig.(b), the two processes are directly connected by a data flow. This means that the 'validate-number' process can start only after the 'read-number' process had supplied data to it. However in Fig (c), the two processes are connected through a data store. Hence, the operations of the two bubbles are independent. The first one is termed 'synchronous' and the second one 'asynchronous'.

2.Data Dictionary

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

grossPay = regularPay + overtimePay

For the smallest units of data items, the data dictionary lists their name and their type. Composite data items can be defined in terms of primitive data items using the following data definition operators:

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

3.Developing the DFD model of the system

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

Context diagram

It establishes the context in which the system operates that is, who are the users, what data do they input to the system, and what data they received by the system.

Level 1 DFD

It usually contains between 3 and 7 bubbles i.e) the system is represented as performing 3 to 7 important functions

To develop the level 1 DFD, examine the high level functional requirents in SRS document.

Decomposition

The bubble are decomposed into sub functions at the successive levels of the DFD models.

Decomposition of the bubbles are also known as factoring or exploding the bubble. Each bubble at any level of DFD is decomposed to anything between 3 and 7 bubbles.

Systematic development of DFD model,

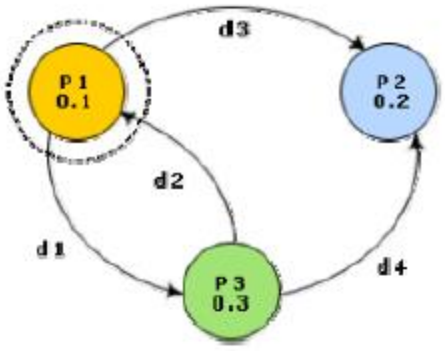
1. **Construction of context diagram:** Examine the SRS document to determine
   a) Different high level function that the system needs to perform
   b) Dat input to every high level function
   c) Data output from every high level function
   d) Interactions among the identified high level functions.
   This would form the top level DFD, usually called the DFD 0.
2. **Construction of level 1 diagram:**Examine the high level functions described in SRS document. If there are between 3 and 5 high level requirements in the SRS document, then represent each of the high level function in the form of the bubble.If there are more than the 7 bubble then some of them have been combined.If there are less than 3 bubbles then some of these have to be spilit
3. **Construction of lower level diagram**:
   a) **Decompose** each high level function into its constituent sub functions through the following set of activities.
      - Identify the different sub functions of the high level function.
      - Identify data input to each of these dub functions
      - Identify the data output from each of these sub functions
      - Identify the interactions among these sub functions.
      Represent these aspects in diagrammatic form using DFD.
   b) Repeat step 3(a) for each sub function util a sub function can be represented by usin a simple algorithm.

Numbering of Bubbles

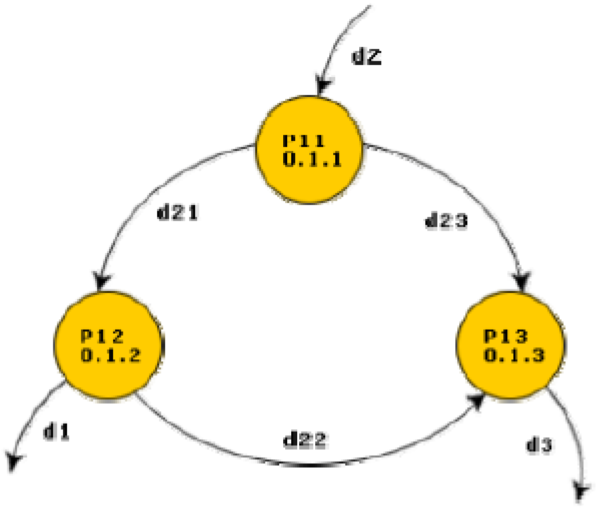
- It is necessary to number the different bubbles occurring in the DFD. These numbers help in uniquely identifying any bubble in the DFD from its bubble number.
- The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD.
- Bubbles at level 1 are numbered, 0.1,0.2,0.3,etc.

Balancing DFDs

- The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as balancing a DFD.
- The concept of balancing a DFD has been illustrated in fig. 5.3. In the level 1 of the DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1. In the next level, bubble 0.1 is decomposed.
- The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in.



(a) Level 1 DFD



(b) Level 2 DFD

Fig. An example showing balanced decomposition

How far to decompose?

- A bubble should not be decomposed any further once a bubble is found to represent a simple set of instructions.
- For simple problems, decomposition up to level 1 should suffice. However,large industry standard problems may need decomposition up to level 3 or level 4.
- Rarely if ever, decomposition beyond level 4 is needed

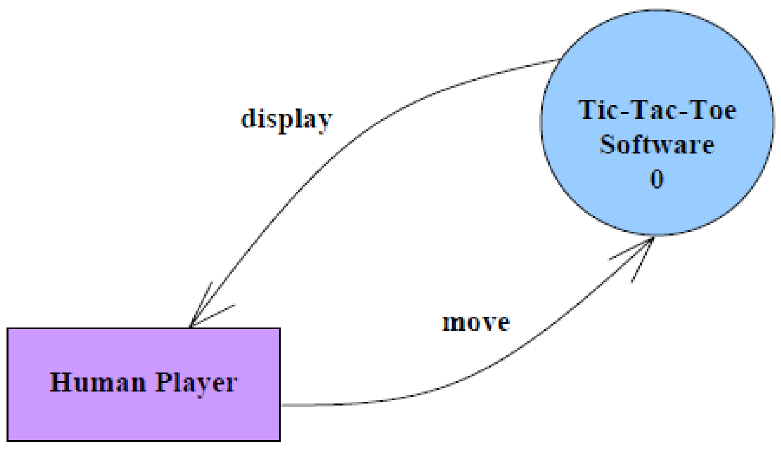
Commonly made errors while constructing a DFD models.

DFDs are simple to understand and draw, but still while performing those DFDs. The errors are as follows:

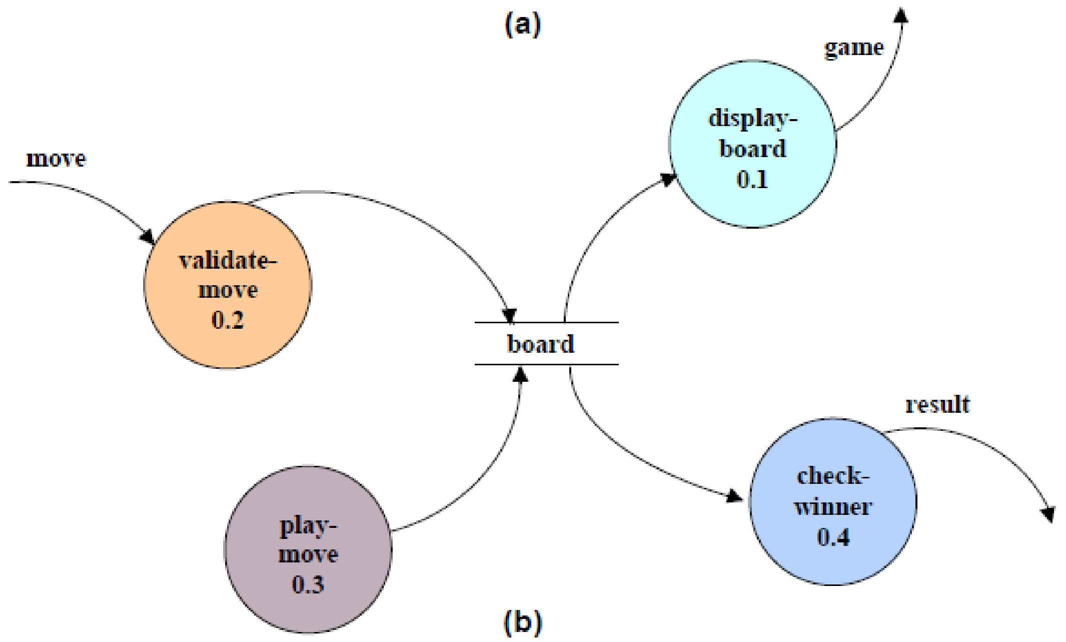
- Many beginners commit the mistake of drawing more than one bubble in the context diagram. Context diagram should depict the system as a single bubble.
- Many beginners create DFD models in which external entities appearing at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear in the DFDs at any other level
- It is a common oversight to have either too few or too many bubbles in a DFD. Only 3 to 7 bubbles per diagram should be allowed. This also means that each bubble in a DFD should be decomposed to between 3 to 7 bubbles in the next level.
- Many beginners leave the DFDs at the different levels of a DFD model unbalanced.
- A common mistake committed by many beginners while developing a DFD model is attempting to represent control information in a DFD.

Example 1: Tic-Tac-Toe Computer Game

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



(a)

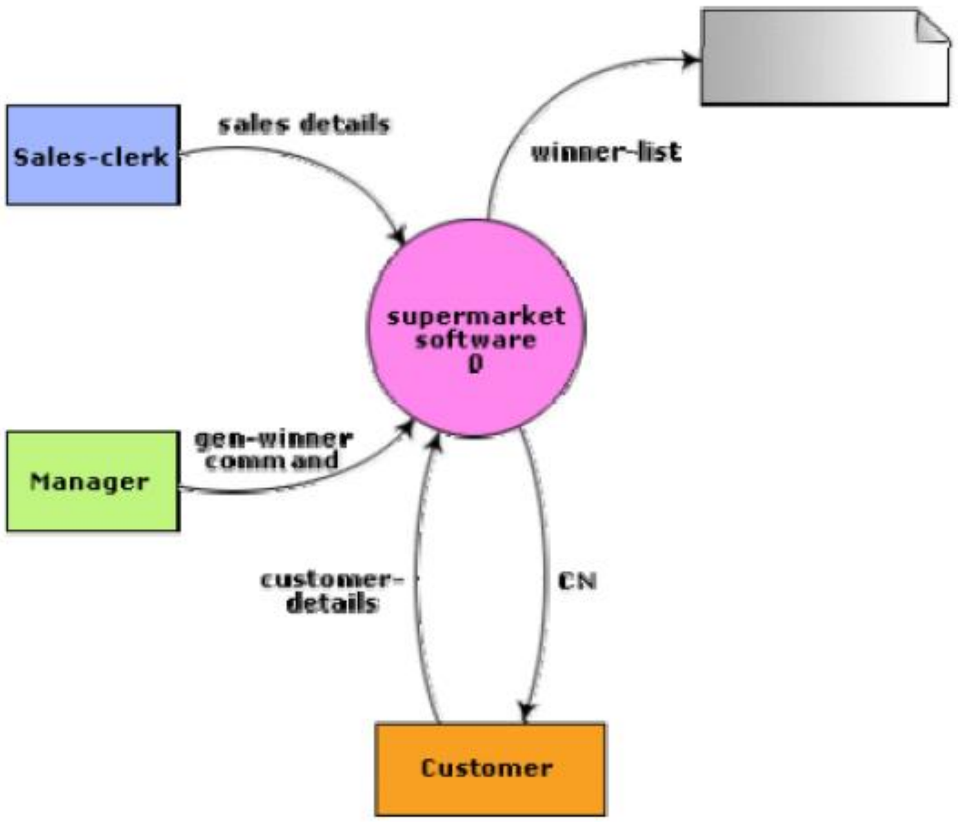


(b)

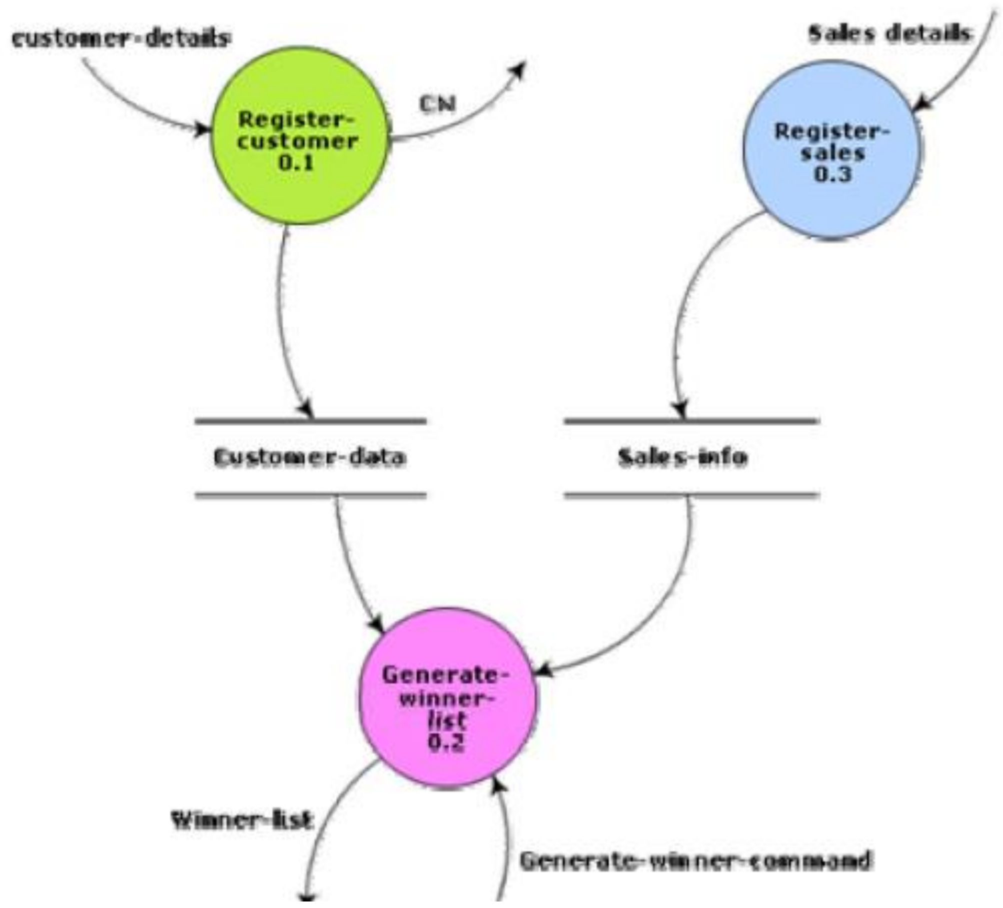
Fig (a), (b) validate-move 0.2 play-move 0.3 check-winner 0.4 display-board 0.1 boardmove result game Tic-Tac-Toe Software 0 Human Player displaymove

Example 2:-

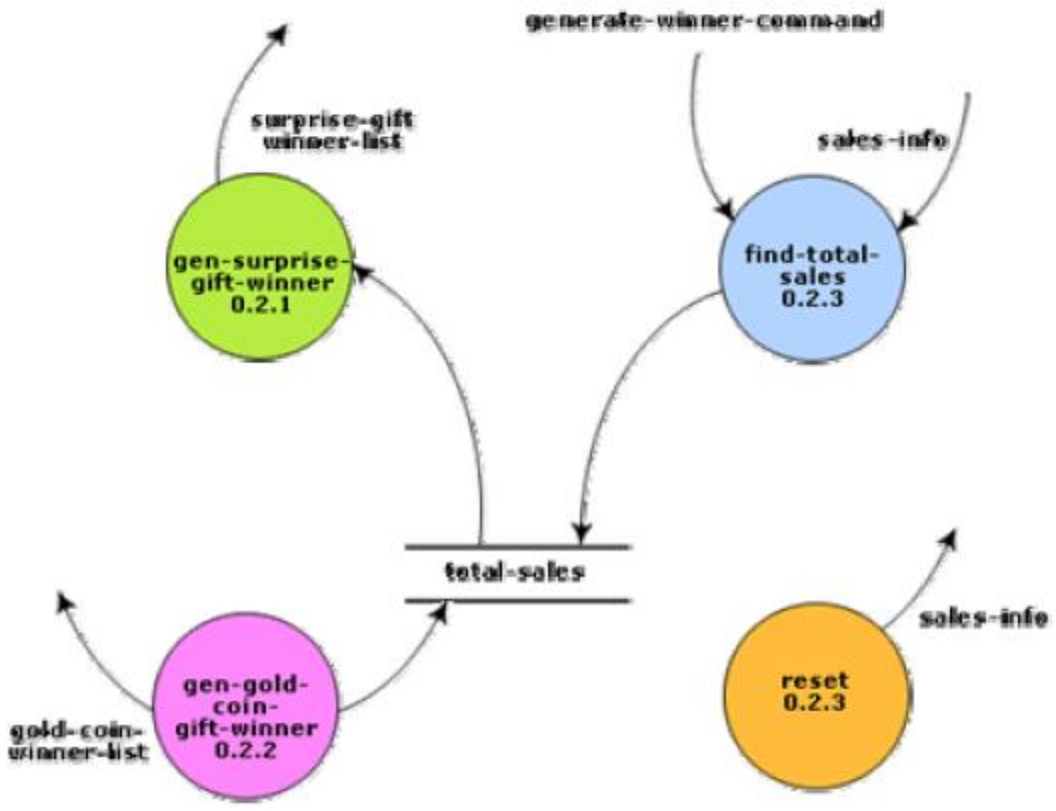
A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer.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



Context diagram for supermarket problem



Level 1 diagram for supermarket problem



Level 2 diagram for supermarket problem

Example 3 : Trading-House Automation System (TAS).
The trading house wants us to develop a computerized system that would automate various book-keeping activities associated with its business. The following are the salient features of the system to be developed:

- The trading house has a set of regular customers. The customers place orders with it for various kinds of commodities. The trading house maintains the names and addresses of its regular customers. Each of these regular customers should be assigned a unique customer identification number (CIN) by the computer. The customers quote their CIN on every order they place.
- Once order is placed, as per current practice, the accounts department of the trading house first checks the credit-worthiness of the customer. The credit-worthiness of the customer is determined by analyzing the history of his payments to different bills sent to him in the past. After automation, this task has to be done by the computer.
- If the customer is not credit-worthy, his orders are not processed any further and an appropriate order rejection message is generated for the customer.
- If a customer is credit-worthy, the items that have been ordered are checked against a list of items that the trading house deals with. The items in the order

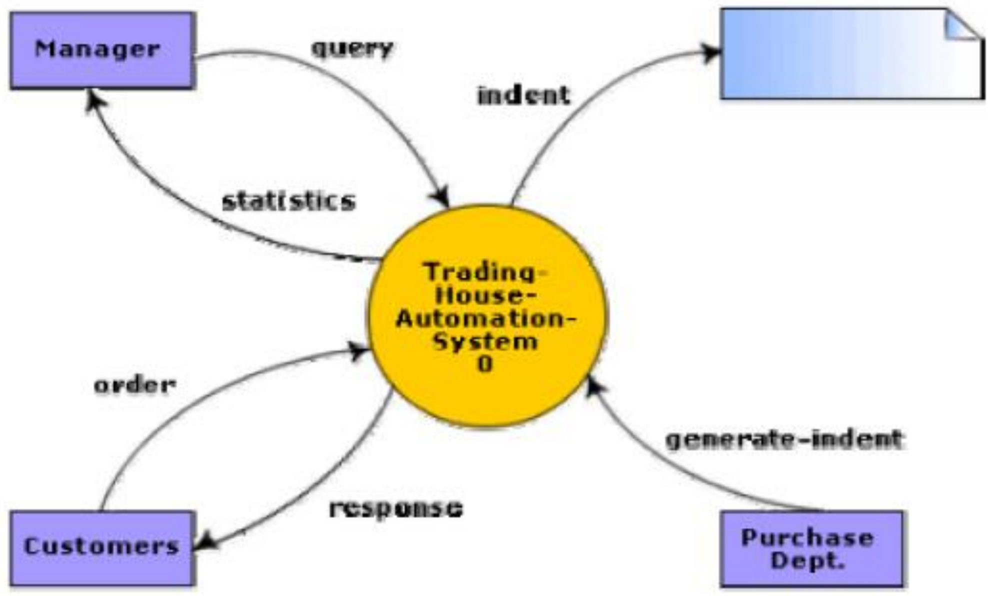
which the trading house does not deal with, are not processed any further and an appropriate apology message for the customer for these items is generated.

- The items in the customer's order that the trading house deals with are checked for availability in the inventory. If the items are available in the inventory in the desired quantity, then
  - o A bill with the forwarding address of the customer is printed.
  - o A material issue slip is printed. The customer can produce this material issue slip at the store house and take delivery of the items.
  - o Inventory data is adjusted to reflect the sale to the customer.

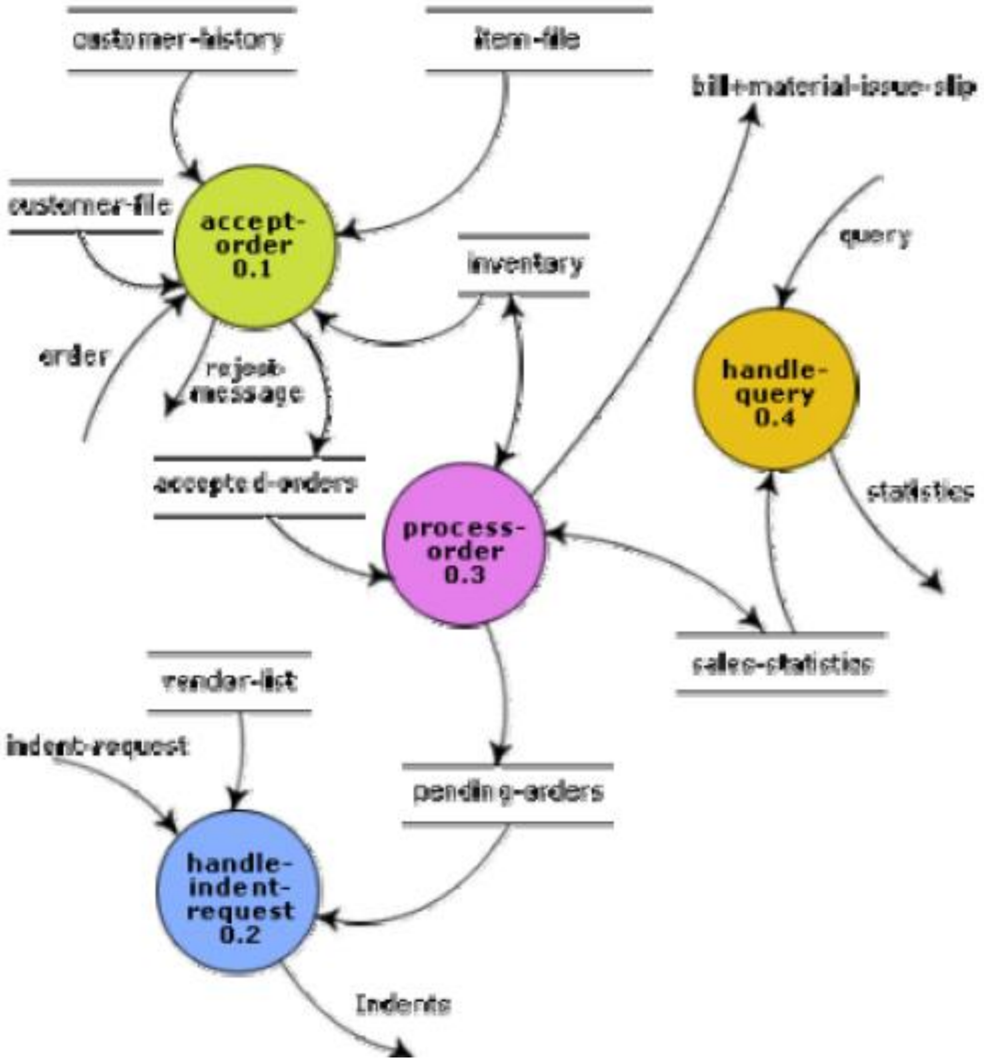
• If any of the ordered items are not available in the inventory in sufficient quantity to satisfy the order, then these out-of-stock items along with the quantity ordered by the customer and the CIN are stored in a "pending-order" file for the further processing to be carried out when the purchase department issues the "generate indent" command.

• The purchase department should be allowed to periodically issue commands to generate indents. When a command to generate indents is issued, the system should examine the "pending-order" file to determine the orders that are pending and determine the total quantity required for each of the items. It should find out the addresses of the vendors who supply these items by examining a file containing vendor details and then should print out indents to these vendors.

• The system should also answer managerial queries regarding the statistics of different items sold over any given period of time and the corresponding quantity sold and the price realized.



Context diagram for TAS



Level 1 DFD for TAS

Shortcomings of a DFD model

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

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

problem, several alternative DFD representations are possible. Further, many times it is not possible to say which DFD representation is superior or preferable to another one.

4.STRUCTURED DESIGN

The aim of structured design

transform the results of structured analysis (i.e., a DFD representation) into a structure chart.

A structure chart represents the software architecture:

- various modules making up the system,
- module dependency (i.e. which module calls which other modules),
- parameters passed among different modules.

1. **Rectangular box:**
- A rectangular box represents a module. Annotated with the name of the module it represents.

2. **Arrows:**
- An arrow between two modules implies:
- during execution control is passed from one module to the other in the direction of the arrow.

3. **Data flow arrows:**
- Data passing from one module to another in the direction of the arrow.

4. **Library modules:**
- A rectangle with double side edges.
- Simplifies drawing when a module is called by several modules.

5. **Selection:**
- The diamond symbol represents selection
- one module of several modules connected to the diamond symbol is invoked depending on some condition.

6. **Repetition:**
- A loop around control flow arrows denotes that the concerned modules are invoked repeatedly.

4.1 FLOW CHART VERSUS STRUCTURE CHART

A structure chart differs from a flow chart in three principal ways:

- It is difficult to identify modules of a software from its flow chart representation.
- Data interchange among the modules is not represented in a flow chart.
- Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

4.2 TRANSFORMATION OF A DFD MODEL INTO STRUCTURE CHART

Two strategies exist to guide transformation of a DFD into a structure chart:

1. Transform Analysis
2. Transaction Analysis

Normally, one would start with the level 1 DFD, transform it into module representation using either the transform or transaction analysis and the proceed toward the lower level DFDs.

4.3 TRANSFORM ANALYSIS

The first step in transform analysis: divide the DFD into 3 types of parts:

1. input,
2. logical processing,
3. output.

1. Input portion in the DFD:
    - processes which convert input data from physical to logical form.
    - e.g. read characters from the terminal and store in internal tables or lists.
    - Each input portion:
        - called an afferent branch.
        - Possible to have more than one afferent branch in a DFD.
2. Output portion of a DFD:
    - transforms output data from logical form to physical form.
    - e.g., from list or array into output characters.
    - Each output portion:
        - called an efferent branch.
3. Logical Processing
    - The remaining portions of a DFD called central transform
    - Derive structure chart by drawing one functional component for:
        - the central transform,
        - each afferent branch,
        - each efferent branch.
    - Identifying the highest level input and output transforms:
        - requires experience and skill.
    - Some guidelines:
        - trace the inputs until a bubble is found whose output cannot be deduced from the inputs alone.
        - Processes which validate input are not central transforms.
        - Processes which sort input or filter data from it are.

- First level of structure chart:
    - draw a box for each input and output units
    - a box for the central transform.
- Next, refine the structure chart:
    - Add subfunctions required by each high-level module.
    - Many levels of modules may required to be added.
- The process of breaking functional components into subcomponents is called Factoring.
- Factoring includes adding:
    - read and write modules,
    - error-handling modules,
    - initialization and termination modules, etc.
- Finally check:
    - whether all bubbles have been mapped to modules.

4.4 TRANSACTION ANALYSIS

Useful for designing transaction processing programs.

1. Transform-centered systems:
    - characterized by similar processing steps for every data item processed by input, process, and output bubbles.
2. Transaction-driven systems,
    - one of several possible paths through the DFD is traversed depending upon the input data value.

Transaction:

- any input data value that triggers an action:
- For example, selected menu options might trigger different functions.
- Represented by a tag identifying its type.

Transaction analysis uses this tag to divide the system into:

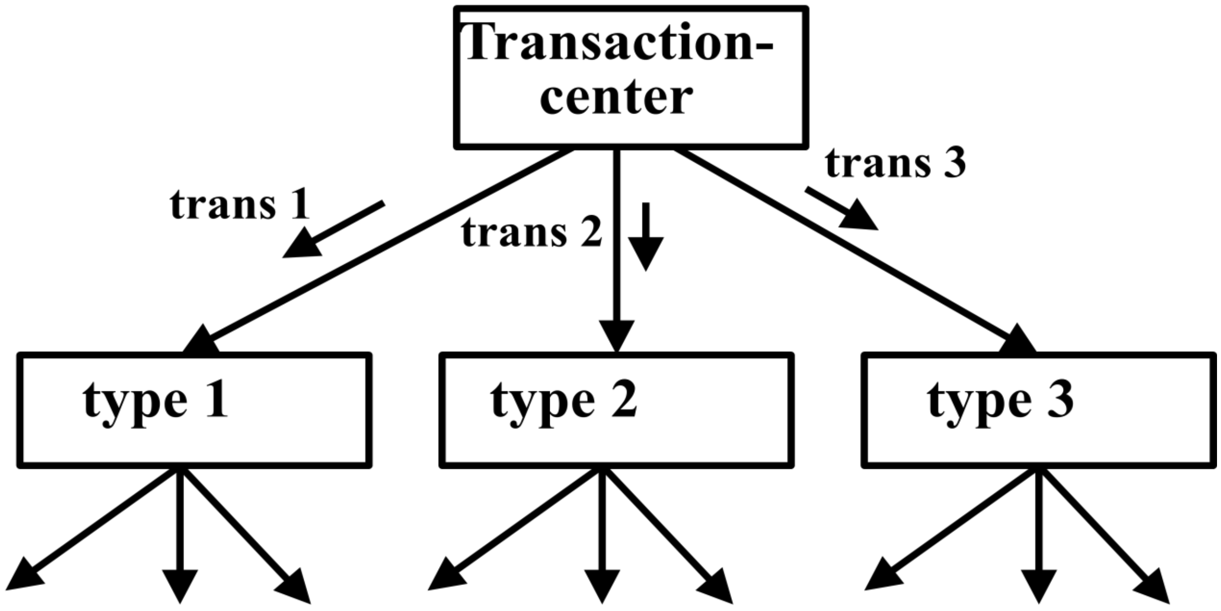
- several transaction modules
- one transaction-center module.

5.DETAILED DESIGN

During Detailed design the pseudo code description of the processing and the different data structure are designed for the different modules of the structure chart.

These are usually described in the form of module specification(MSPEC). MSPEC is usually written using structured English. The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower-level modules.

The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegates to the lower-level modules.



UNIT 4

1.OBJECT MODELLING USING UML

- The object-oriented software development Style has become extremely popular, and is being widely used in industry as well as in academic circles.
- In the context of model construction we need to carefully understand the basic distinction between a modeling language and a design process.
- A modeling language specifies a set of notations using which a design or analysis result can be documented.
- A design process address the starting from a given problem description that is how exactly can one work out the design solution to the problem? In other words, a design process recommends a step by step procedure using a problem description can be converted to a design solution a design process is referred as design methodology.

1.1 OVERVIEW OF BASIC OBJECT-ORIENTATION CONCEPTS

The principles of object-orientation are founded on few important concepts. The important concepts and mechanisms are based on which the principles of object-orientation have been founded.

1.1.1BASIC MECHANISMS

The following are the important mechanisms used in the object-oriented paradigm.

Objects:

It is convenient to think of each object has representing a real world entity such as library member, employee or a book etc.,

A key advantage of considering a system as a set of objects is the following,

When the system is analyzed, developed, and implemented in terms of natural objects, it becomes easy to understand the design and the implementation of the system.

Each object stores some data and supports certain operations on the stored data.

For example, Consider library member object can be:

- Name of the Member
- Membership number
- Address
- Phone number
- Email

The operations supported by a library member object can be:

- Issue-book
- Find-books-outstanding
- Find-books-overdue
- Return-book
- Find-membership-details

The data stored internally in an object are called its attributes and the functions supported by an object are called in methods.

Class:

Similar objects constitute a class. This means that objects possessing similar attributes and having similar methods would constitute a class. For example, the set of all the library members would constitute a class in a library automation application. In this case, each library member object has attributes such as member name, membership number,member address,etc. And also methods such as issue-book, return-book, etc.

A class can be considered as an abstract data type (ADT). A class being an ADT has the following implications:

- The data of an object can be accessed only through its methods. In other words, the way data is stored internally (stack, array, queue, etc.) in the object is abstracted out(not known to other objects).
- We can instantiate a class into objects(i.e. a class is a type).

Methods

The operations(such as create, issue, return, etc.) supported by an object are implemented in the form of methods. The terms operation and 'method' are often used interchangeably. However, there is a technical difference between these two terms which we explain in the following section.

A method is an implementation of the responsibility. It would be sometimes useful to implement a responsibility through more than one method.

When an operation is implemented through multiple methods, we say that the method name is overloaded.

The implementation of a class responsibility through multiple methods is known as method overloading.

Class relationships

Classes can be related to each other in the following four ways

- Inheritance
- Association
- Aggregation and composition
- Dependency

In the following, we discuss these different types of relationships that may exist among classes.

Inheritance

The inheritance feature allows us to define a new class by extending the features of modifying an existing class.

The original class is called the base class and the new class obtained through inheritance is called derived class.

An example of inheritance is the classes Faculty, Students, and staff as having been derived from the base class LibraryMember through an inheritance relationship.

Each derived class can be considered as a specialization of its base class because it modifies or extends the basic properties of the base class in certain ways.

A derived class may even give a new definition to a method which already exists in the base class. Redefinition of a method which already exists in its base class is termed as method overriding.

When a new definition of a method that existed in the base class is provided in a derived class, the method is said to be overridden in the derived class.

Inheritance is a basic mechanism that almost every object-oriented programming language supports in fact, languages that support ADTs, but do not support inheritance are called object based languages.

The important advantage of using the inheritance mechanism in programming include code reuse and simplicity of program design.

Multiple Inheritance

Multiple inheritance is a mechanism by which a subclass can inherit attributes and methods from more than one base class.

Association and Link
If one class is associated with another, then the corresponding objects of two classes know each other.
Each object of one class in the association relationships knows the address of the corresponding object of the other class.
As a result, it becomes possible for the object of one class to invoke the methods of the corresponding object of the other class.
When two classes are associated the relationship between two objects of the two corresponding class is called a link.
In other words, an association describes a group of similar links. A link can be considered has an instance of an association relation.
In case of recursive association, two or more different objects of the same class are linked by association relationship
Composition and Aggregation
Composition and Aggregation represent whole or part relationships among different objects. Objects which contains other objects are called composite objects.
For example, Suppose a student can register in five courses in a semester in this case a studentSemesterRegistration object is composed of five course object.
For this reason, the composition/aggregation relationship is also known as has relationship. Composition can occur in a hierarchy of levels.
The composition and aggregation relationship are not reflexive. In other words, association relationship cannot be circularly defined. An object cannot contain the object of same type.
Dependency
A dependency relation between two classes shows that any change made to the independent class would require the corresponding change to be made to the independent class.
Independencies among classes may arise due to various causes. There are two important reasons as follows:

- A class invokes the method provided by another class
- A class implements an interface class. If the properties of the interface class are changed, then a change becomes necessary to the class implementing the interface class as well.

Abstract Class
Classes that are not intended to produce instances of themselves are called abstract classes. In other words, abstract classes cannot be instantiated.
By using abstract classes, code reuse can be enhanced and the effort required to develop software brought down.
Abstract classes usually supports generic methods, and the subclasses of the abstract classes are expected to provide specific implementation of these methods.
1.1.2 KEY CONCEPTS
Abstraction
Abstraction is the selective examination of certain aspects of a problem while ignoring all the remaining aspects of a problem.
The abstraction mechanism allows us to represent a problem in a simpler way by considering only those aspects that are relevant to some purpose and omitting all other details that are irrelevant.
Feature Abstraction

A class hierarchy can be viewed as defining several levels of abstraction, where each class is an abstraction of its subclasses.

Data Abstraction

An object itself can be considered as a data abstraction entity, because it abstracts out the exact way in which it stores its various private data items, and it merely provides a set of methods to other objects to access and manipulate these data items.

An important advantage of the principle of data abstraction is that it reduces coupling among various objects, Therefore, it leads to a reduction of the overall complexity of a design, and helps in easy maintenance and code reuse.

Encapsulation

The data of an object is encapsulated within its methods. To access the data internal to an object, other objects have to invoke its methods, and cannot directly access the data.

Encapsulation offers three important advantages as follows:

- Protection from unauthorized data access
- Data hiding
- Weak coupling

Polymorphism

Polymorphism literally means poly(many) morphism(forms). Polymorphism can be implemented using the following two mechanisms:

1.Static polymorphism:

In this type of polymorphism, the same method call results in different actions. This type of polymorphism is also referred to as static binding.

Static polymorphism occurs when multiple methods with same operation name exist.

2.Dynamic polymorphism:

In dynamic binding, address of an invoked method can be known only at run time. It is important to remember that dynamic binding occurs when we have some methods of the base class overridden by the derived classes and the instances of the derived classes are stored in instances of the base class.

Even when the method of an object of the base class is invoked, an appropriate overridden method of a derived class would be invoked depending on the exact object that may have been assigned at the run-time to the object of the base class.

The main advantage of dynamic binding is that it leads to elegant programming and facilitates code reuse and maintenance.

1.1.3 RELATED TECHNICAL TERMS

Persistence

Objects usually get destroyed once a program finishes execution. Persistent objects are stored permanently.

Agents

A passive object is one that performs some action only when requested through invocation of some its methods. An agent on the other hand, monitors events occurring in the application and takes actions autonomously.

Widgets

The term widget stands for window object. A widget is a primitive object used for graphical user interface(GUI) design.

1.1.4 ADVANTAGES OF OOD

The main reason for the popularity of OOD is that it holds the following promises:

- Code and design reuse
- Increased productivity
- Ease of testing and maintenance
- Better code and design understandability

The chief advantage of OOD is improved productivity which comes about due to a variety of factors, such as

- Code reuse by the use of pre developed class libraries
- Code reuse due to inheritance
- Simpler and more intuitive abstraction, i.e. better organization of inherent complexity
- Better problem decomposition

2. UNIFIED MODELING LANGUAGE (UML)

UML, as the name implies, is a modeling language. It may be used to visualize, specify, construct, and document the artifacts of a software system. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create a visual model of the system. Like any other language, UML has its own syntax (symbols and sentence formation rules) and semantics (meanings of symbols and sentences). Also, we should clearly understand that UML is not a system design or development methodology, but can be used to document object-oriented and analysis results obtained using some methodology.

Origin of UML

In the late 1980s and early 1990s, there was a proliferation of object-oriented design techniques and notations. Different software development houses were using different notations to document their object-oriented designs. These diverse notations used to give rise to a lot of confusion.

UML was developed to standardize the large number of object-oriented modeling notations that existed and were used extensively in the early 1990s. The principles ones in use were:

- Object Management Technology [Rumbaugh 1991]
- Booch's methodology [Booch 1991]
- Object-Oriented Software Engineering [Jacobson 1992]
- Odell's methodology [Odell 1992]
- Shaler and Mellor methodology [Shaler 1992]

It is needless to say that UML has borrowed many concepts from these modeling techniques. Especially, concepts from the first three methodologies have been heavily drawn upon. UML was adopted by Object Management Group (OMG) as a *de facto* standard in 1997. OMG is an association of industries which tries to facilitate early formation of standards.

We shall see that UML contains an extensive set of notations and suggests construction of many types of diagrams. It has successfully been used to model both large and small problems. The elegance of UML, its adoption by OMG, and a strong industry backing have helped UML find widespread acceptance. UML is now being used in a large number of software development projects worldwide.

What is a model?

A model is an abstraction of a real problem (or situation), and is constructed by leaving out unnecessary details. This reduces the problem complexity and makes it easy to understand the problem(or situation).

Why construct a model?

An important reason behind constructing a model is that it helps to manage the complexity in a problem and facilities arriving at good solutions and at the same time helps to reduce the design cost.

Once models of a system have been constructed, these can be used for a variety of purposes during software development. Including the following:

- Analysis
- Specification
- Design
- Coding
- Visualization and understanding of an implementation
- Testing, etc.,

3.UML DIAGRAMS

UML can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

The UML diagrams can capture the following five views of a system**:**

- User’s view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

Fig. shows the UML diagrams responsible for providing the different views.

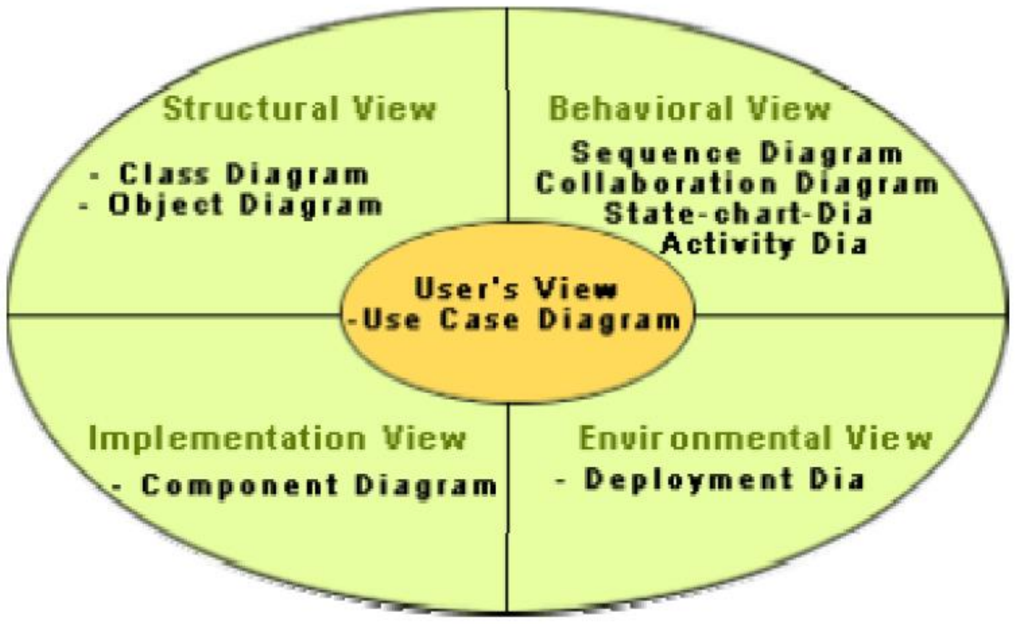


Fig. Different types of diagrams and views supported in UML

1. **User's view:** This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users' view is very different from all other views in the sense that it is a functional model compared to the object model of all other views. The users' view can be considered as the central view and all other views are expected to conform to this view. This thinking is in fact the crux of any user centric development style.
2. **Structural view:** The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.
3. **Behavioral view:** The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.
4. **Implementation view:** This view captures the important components of the system and their dependencies.
5. **Environmental view:** This view models how the different components are implemented on different pieces of hardware.

4.USE CASE MODEL

The use case model for any system consists of a set of "use cases".

Intuitively, use cases represent the different ways in which a system can be used by the users.

A simple way to find all the use cases of a system is to ask the question: “What the users can do using the system?” Thus for the Library Information System (LIS), the use cases could be**:**

- issue-book
- query-book
- return-book
- create-member
- add-book, etc

Use cases correspond to the high-level functional requirements. The use cases partition the system behavior into transactions, such that each transaction performs some useful action from the user’s point of view. To complete each transaction may involve either a single message or multiple message exchanges between the user and the system to complete.
Purpose of use cases

The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system. The use cases do not mention any specific algorithm to be used or the internal data representation, internal structure of the software, etc. A use case typically represents a sequence of interactions between the user and the system. These interactions consist of one mainline sequence.

The mainline sequence represents the normal interaction between a user and the system. The mainline sequence is the most occurring sequence of interaction. For example, the mainline sequence of the withdraw cash use case supported by a bank ATM drawn, complete the transaction, and get the amount. Several variations to the main line sequence may also exist.

Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the amount balance. The variations are also called alternative paths.

A use case can be viewed as a set of related scenarios tied together by a common goal. The mainline sequence and each of the variations are called scenarios or instances of the use case. Each scenario is a single path of user events and system activity through the use case.

4.1REPRESENTATION OF USE CASES

Use cases can be represented by drawing a use case diagram and writing an accompanying text elaborating the drawing. In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse. All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary. The name of the system being modeled (such as Library Information System) appears inside the rectangle.

The different users of the system are represented by using the stick person icon. Each stick person icon is normally referred to as an actor. An actor is a role played by a user with respect to the system use. It is possible that the same user may play the role of multiple actors. Each actor can participate in one or more use cases. The line connecting the actor and the use case is called the communication relationship. It indicates that the actor makes use of the functionality provided by the use case. Both the human users and the external systems can be represented by stick person icons. When a stick person icon represents an external system, it is annotated by the stereotype <<external system>>.

Example 1:

The use case model for the Tic-tac-toe problem is shown in fig. This software has only one use case "play move". Note that the use case "get-user-move" is not used here. The name "get-user-move" would be inappropriate because the use cases should be named from the users' perspective.

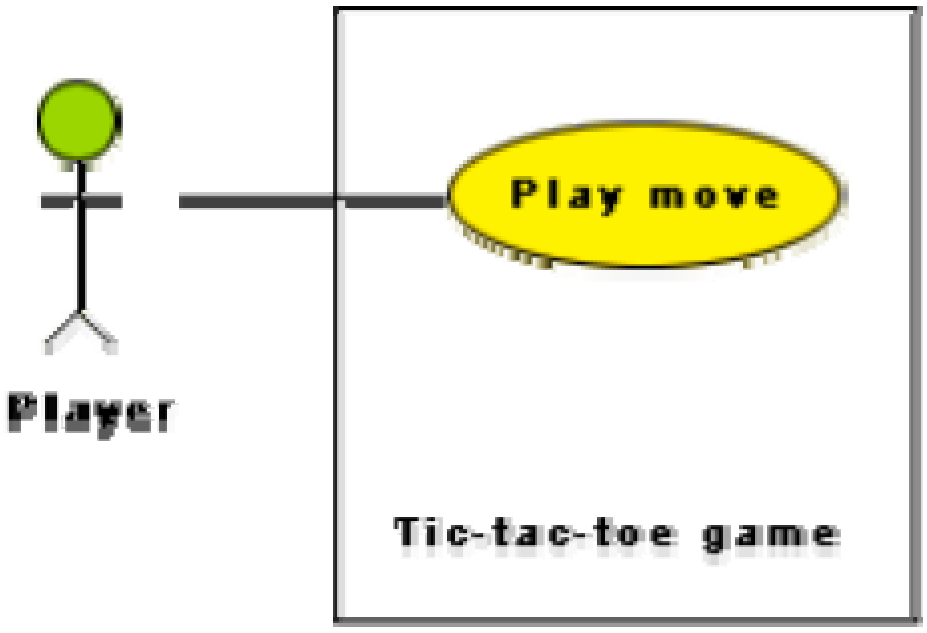


Fig.: Use case model for tic-tac-toe game **Text Description**
Each ellipse on the use case diagram should be accompanied by a text description. The text description should define the details of the interaction between the user and the computer and other aspects of the use case. It should include all the behavior associated with the use case in terms of the mainline sequence, different variations to the normal behavior, the system responses associated with the use case, the exceptional conditions that may occur in the behavior, etc. The behavior description is often written in a conversational style describing the interactions between the actor and the system. The text description may be informal, but some structuring is recommended. The following are some of the information which may be included in a use case text description in addition to the mainline sequence, and the alternative scenarios.

1. **Contact persons:** This section lists the personnel of the client organization with whom the use case was discussed, date and time of the meeting, etc.

2. **Actors:** In addition to identifying the actors, some information about actors using this use case which may help the implementation of the use case may be recorded.
3. **Pre-condition:** The preconditions would describe the state of the system before the use case execution starts.
4. **Post-condition:** This captures the state of the system after the use case has successfully completed.
5. **Non-functional requirements:** This could contain the important constraints for the design and implementation, such as platform and environment conditions, qualitative statements, response time requirements, etc.
6. **Exceptions, error situations:** This contains only the domain-related errors such as lack of user's access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.
7. **Sample dialogs:** These serve as examples illustrating the use case.
8. **Specific user interface requirements:** These contain specific requirements for the user interface of the use case. For example, it may contain forms to be used, screen shots, interaction style, etc.
9. **Document references:** This part contains references to specific domain-related documents which may be useful to understand the system operation.

Example 2:

The use case model for the Supermarket Prize Scheme described in Lesson 5.2 is shown in fig. As discussed earlier, the use cases correspond to the high-level functional requirements. From the problem description and the context diagram in fig., we can identify three use cases: "register-customer", "register-sales", and "select-winners". As a sample, the text description for the use case "register-customer" is shown.

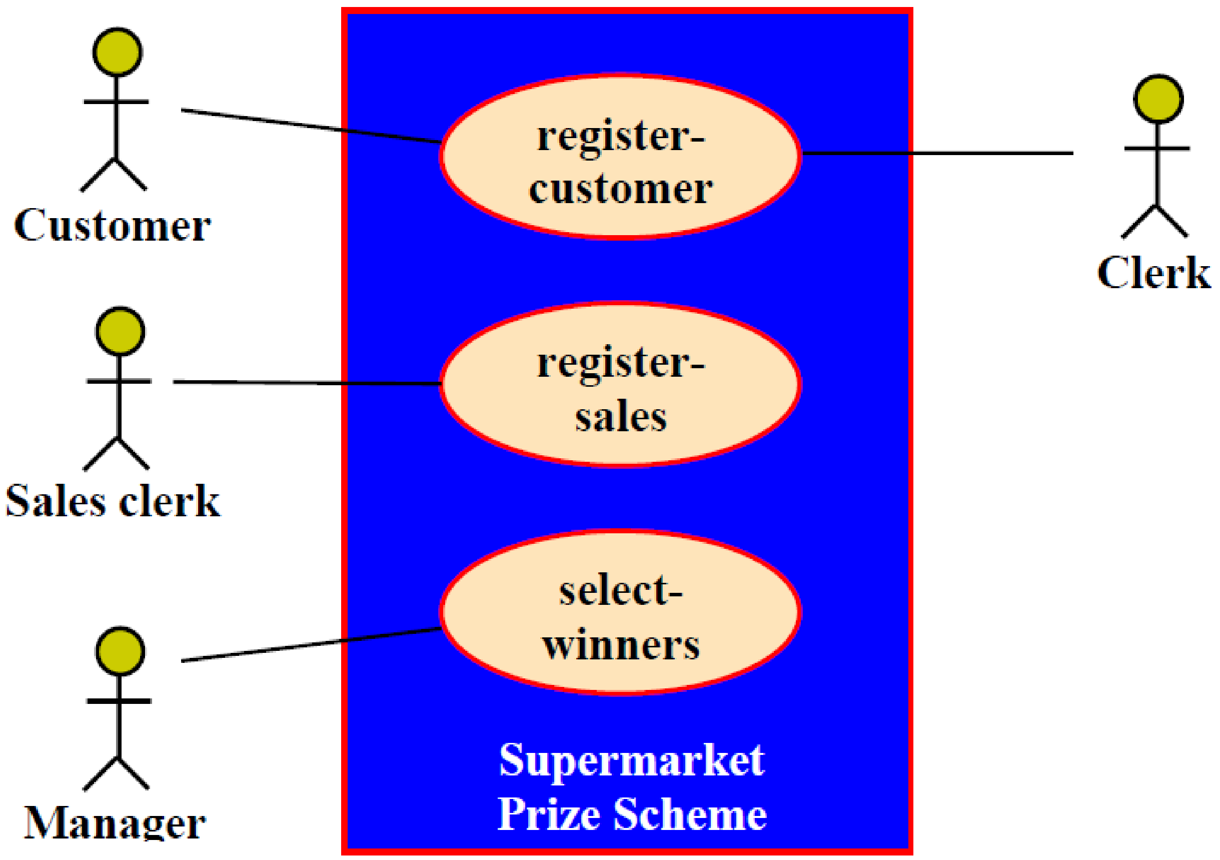


Fig. Use case model for Supermarket Prize Scheme **register-customerregister-salesselect-winnersSupermarket PrizeSchemeCustomer Sales clerk Manager Clerk**

Text description

U1: register-customer: Using this use case, the customer can register himself by providing the necessary details.

Scenario 1: Mainline sequence

1. **Customer: select register customer option.**
2. **System: display prompt to enter name, address, and telephone number.**
3. **Customer: enter the necessary values.**
4. **System: display the generated id and the message that the customer has been successfully registered.**

Scenario 2: at step 4 of mainline sequence

1. **System: displays the message that the customer has already registered.**

Scenario 2: at step 4 of mainline sequence

1. System: displays the message that some input information has not been entered. The system display a prompt to enter the missing value.

The description for other use cases is written in a similar fashion.

Utility of use case diagrams

From use case diagram, it is obvious that the utility of the use cases are represented by ellipses. They along with the accompanying text description serve as a type of requirements specification of the system and form the core model to which all other models must conform. But, what about the actors (stick person icons)? One possible use of identifying the different types of users (actors) is in identifying and implementing a security mechanism through a login system, so that each actor can involve only those functionalities to which he is entitled to. Another possible use is in preparing the documentation (e.g. users' manual) targeted at each category of user. Further, actors help in identifying the use cases and understanding the exact functioning of the system.

Factoring of use cases

It is often desirable to factor use cases into component use cases. Actually, factoring of use cases are required under two situations. First, complex use cases need to be factored into simpler use cases. This would not only make the behavior associated with the use case much more comprehensible, but also make the corresponding interaction diagrams more tractable. Without decomposition, the interaction diagrams for complex use cases may become too large to be accommodated on a single sized (A4) paper.

Secondly, use cases need to be factored whenever there is common behavior across different use cases. Factoring would make it possible to define such behavior only once and reuse it whenever required. It is desirable to factor out common usage such as error handling from a set of use cases. This makes analysis of the class design much simpler and elegant. However, a word of caution here. Factoring of use cases should not be done except for achieving the above two objectives. From the design point of view, it is not advantageous to break up a use case into many smaller parts just for the shake of it.

UML offers three mechanisms for factoring of use cases as follows:

1. Generalization

Use case generalization can be used when one use case that is similar to another, but does something slightly differently or something more. Generalization works the same way with use cases as it does with classes. The child use case inherits the behavior and meaning of the parent use case. The notation is the same too (as shown in fig.). It is important to remember that the base and the derived use cases are separate use cases and should have separate text descriptions.

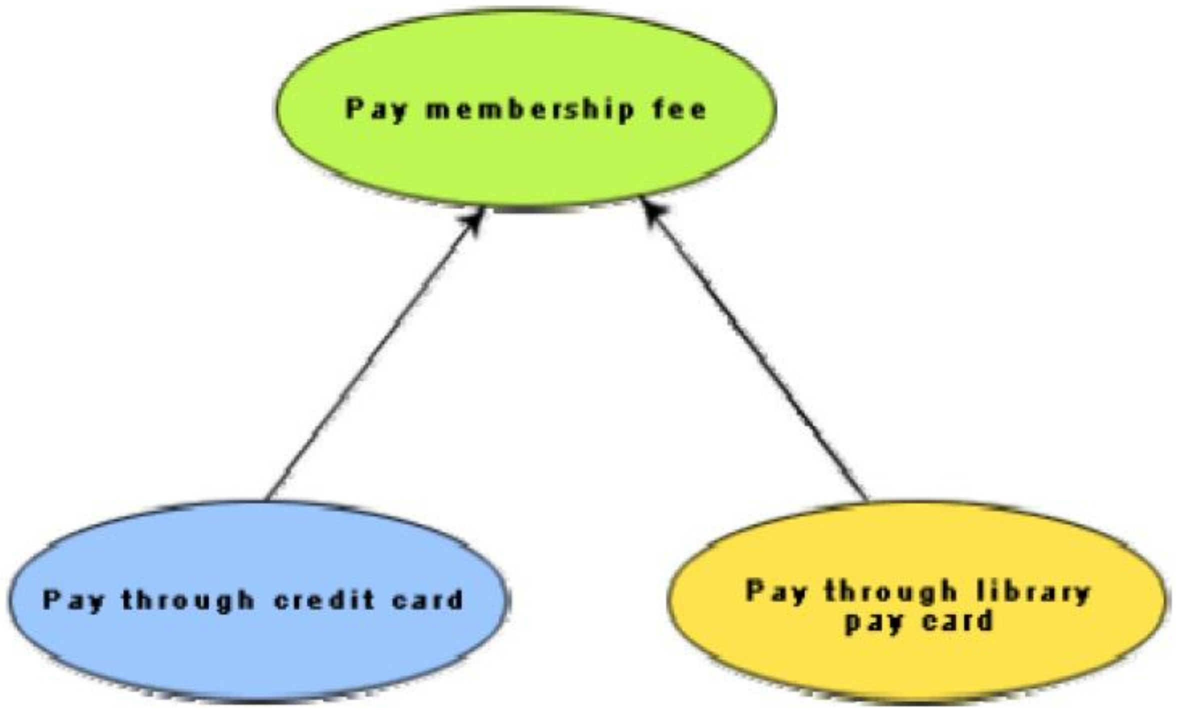


Fig.: Representation of use case generalization

2. Includes

The includes relationship in the older versions of UML (prior to UML 1.1) was known as the uses relationship. The includes relationship involves one use case including the behavior of another use case in its sequence of events and actions.

The includes relationship occurs when a chunk of behavior that is similar across a number of use cases. The factoring of such behavior will help in not repeating the specification and implementation across different use cases. Thus, the includes relationship explores the issue of reuse by factoring out the commonality across use cases. It can also be gainfully employed to decompose a large and complex use cases into more manageable parts. As shown in fig, the includes relationship is represented using a predefined stereotype <<include>>. In the includes relationship, a base use case compulsorily and automatically includes the behavior of the common use cases. As shown in example fig., issue-book and renew-book both include check-reservation use case. The base use case may include several use cases. In such cases, it may interleave their associated common use cases together. The common use case becomes a separate use case and the independent text description should be provided for it.

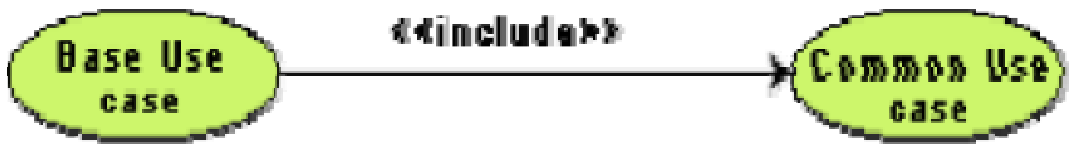


Fig. 7.5: Representation of use case inclusion

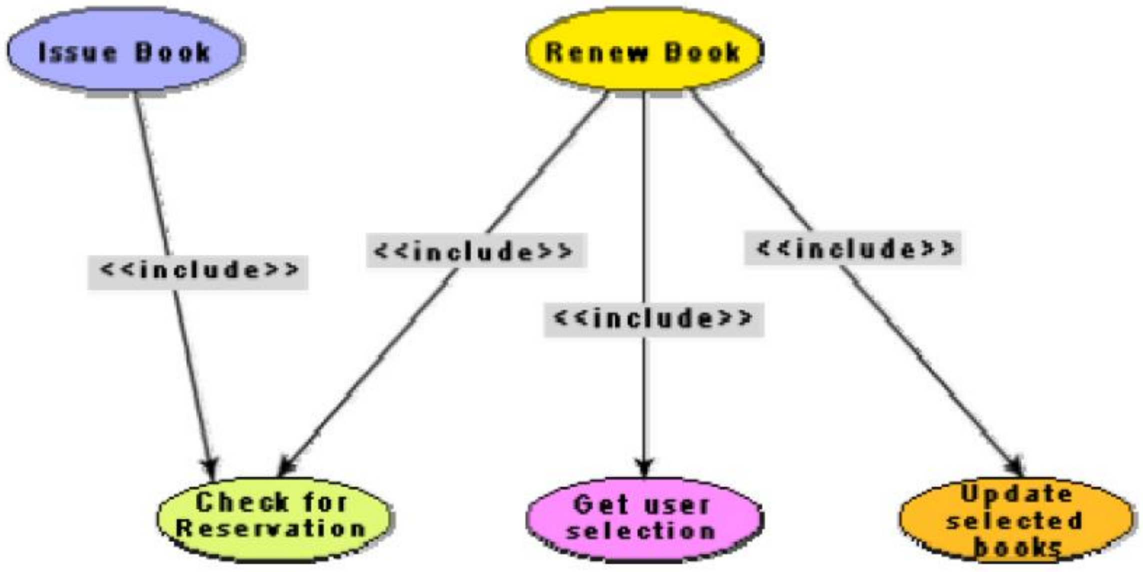


Fig. 7.6: Example use case inclusion

3. Extends

The main idea behind the extends relationship among the use cases is that it allows you to show optional system behavior. An optional system behavior is extended only under certain conditions. This relationship among use cases is also predefined as a stereotype as shown in fig. 7.7. The extends relationship is similar to generalization. But unlike generalization, the extending use case can add additional behavior only at an extension point only when certain conditions are satisfied. The extension points are points within the use case where variation to the mainline (normal) action sequence may occur. The extends relationship is normally used to capture alternate paths or scenarios.

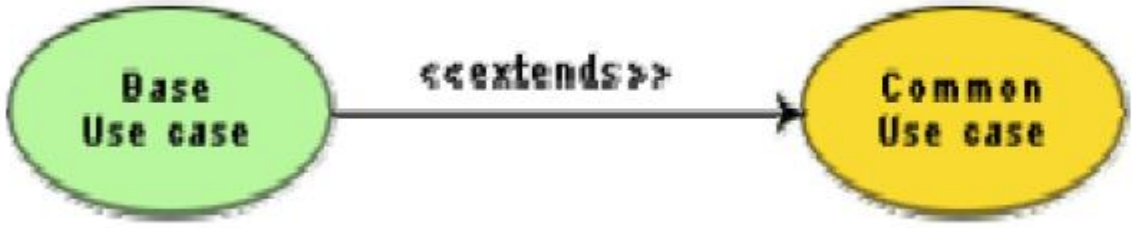


Fig. 7.7: Example use case extension

4. Organization of use cases

When the use cases are factored, they are organized hierarchically. The high-level use cases are refined into a set of smaller and more refined use cases as shown in fig. 7.8. Top-level use cases are super-ordinate to the refined use cases. The refined use cases are sub-ordinate to the top-level use cases. Note that only the complex use cases should be decomposed and organized in a hierarchy. It is not necessary to decompose simple use cases.

The functionality of the super-ordinate use cases is traceable to their sub-ordinate use cases. Thus, the functionality provided by the super-ordinate use cases is composite of the functionality of the sub-ordinate use cases. In the highest level of the use case model, only the fundamental use cases are shown. The focus is on the application context.

Therefore, this level is also referred to as the context diagram. In the context diagram, the system limits are emphasized. In the top-level diagram, only those use cases with which external users of the system. The subsystem-level use cases specify the services offered by the subsystems. Any number of levels involving the subsystems may be utilized. In the lowest level of the use case hierarchy, the class-level use cases specify the functional fragments or operations offered by the classes.

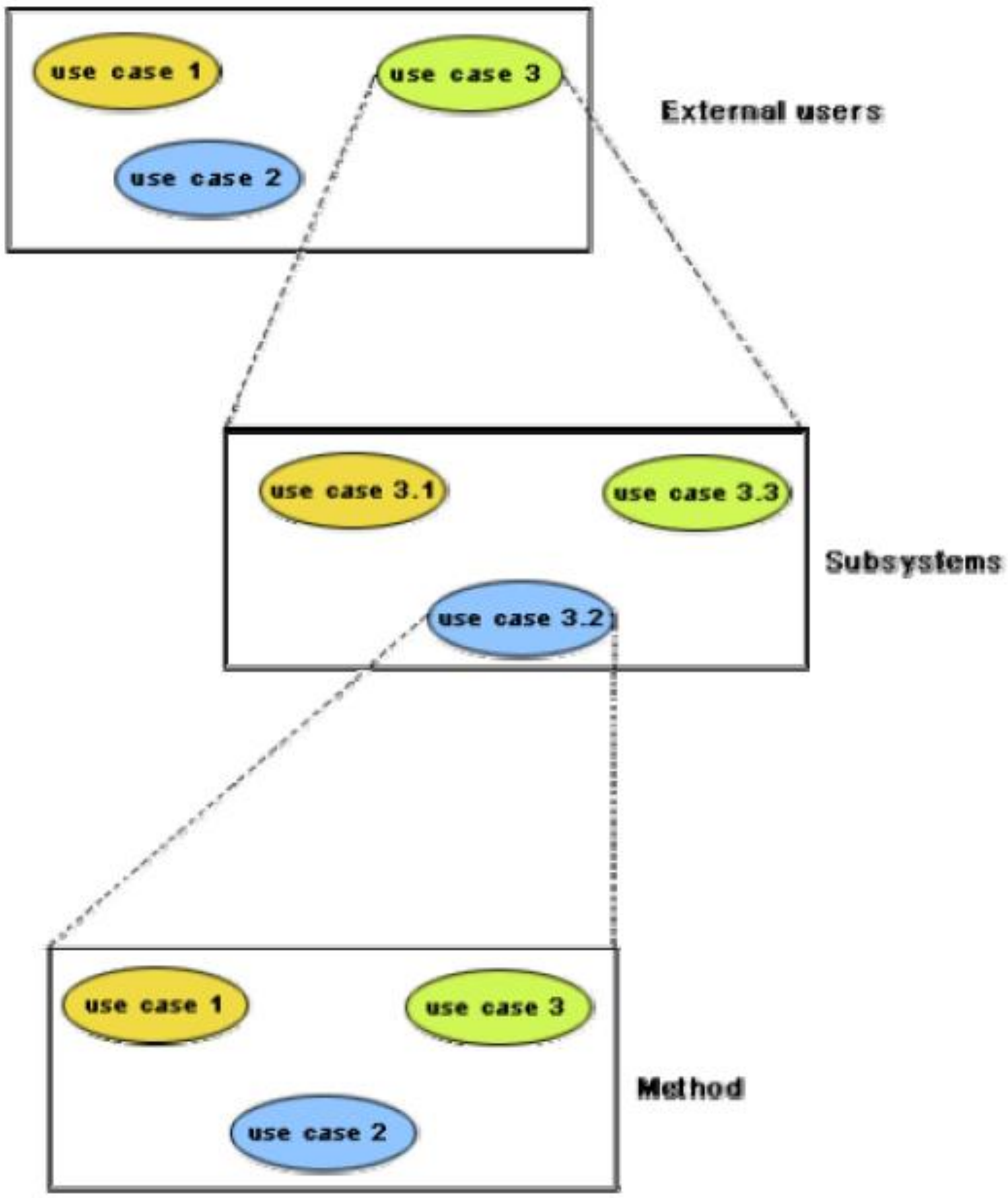


Fig.: Hierarchical organization of use cases

4.CLASS DIAGRAMS
A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system comprises of a number of class diagrams and their dependencies. The main constituents of a class diagram are classes and their relationships: generalization, aggregation, association, and various kinds of dependencies.

Classes

The classes represent entities with common features, i.e. attributes and operations. Classes are represented as solid outline rectangles with compartments. Classes have a mandatory name compartment where the name is written centered in boldface. The class name is usually written

using mixed case convention and begins with an uppercase. The class names are usually chosen to be singular nouns. An example of a class is shown in fig. 6.2 (Lesson 6.1).
Classes have optional attributes and operations compartments. A class may appear on several diagrams. Its attributes and operations are suppressed on all but one diagram.

Attributes

An attribute is a named property of a class. It represents the kind of data that an object might contain. Attributes are listed with their names, and may optionally contain specification of their type, an initial value, and constraints. The type of the attribute is written by appending a colon and the type name after the attribute name. Typically, the first letter of a class name is a small letter. An example for an attribute is given.

bookName : String

Operation

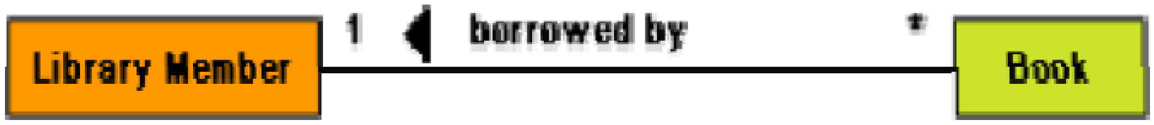
Operation is the implementation of a service that can be requested from any object of the class to affect behaviour. An object's data or state can be changed by invoking an operation of the object. A class may have any number of operations or no operation at all. Typically, the first letter of an operation name is a small letter. Abstract operations are written in italics. The parameters of an operation (if any), may have a kind specified, which may be 'in', 'out' or 'inout'. An operation may have a return type consisting of a single return type expression. An example for an operation is given.

issueBook(in bookName):Boolean

Association

Associations are needed to enable objects to communicate with each other. An association describes a connection between classes. The association relation between two objects is called object connection or link. Links are instances of associations. A link is a physical or conceptual connection between object instances. For example, suppose Amit has borrowed the book Graph Theory. Here, borrowed is the connection between the objects Amit and Graph Theory book. Mathematically, a link can be considered to be a tuple, i.e. an ordered list of object instances. An association describes a group of links with a common structure and common semantics. For example, consider the statement that Library Member borrows Books. Here, borrows is the association between the class LibraryMember and the class Book. Usually, an association is a binary relation (between two classes). However, three or more different classes can be involved in an association. A class can have an association relationship with itself (called recursive association). In this case, it is usually assumed that two different objects of the class are linked by the association relationship.
Association between two classes is represented by drawing a straight line between the concerned classes. Fig. 7.9 illustrates the graphical representation of the association relation. The name of the association is written along side the association line. An arrowhead may be placed on the association line to indicate the reading direction of the association. The arrowhead should not be misunderstood to be indicating the direction of a pointer implementing an association. On each side of the association relation, the multiplicity is noted as an individual number or as a value range. The multiplicity indicates how many instances of one class are associated with each other. Value ranges of multiplicity are noted by specifying the minimum and maximum value, separated by two dots, e.g. 1.5. An asterisk is a wild card and means many (zero or more). The

association of fig. 7.9 should be read as “Many books may be borrowed by a Library Member”. Observe that associations (and links) appear as verbs in the problem statement.



Association between two classes

Associations are usually realized by assigning appropriate reference attributes to the classes involved. Thus, associations can be implemented using pointers from one object class to another. Links and associations can also be implemented by using a separate class that stores which objects of a class are linked to which objects of another class. Some CASE tools use the role names of the association relation for the corresponding automatically generated attribute.

Aggregation

Aggregation is a special type of association where the involved classes represent a whole-part relationship. The aggregate takes the responsibility of forwarding messages to the appropriate parts. Thus, the aggregate takes the responsibility of delegation and leadership. When an instance of one object contains instances of some other objects, then aggregation (or composition) relationship exists between the composite object and the component object. Aggregation is represented by the diamond symbol at the composite end of a relationship. The number of instances of the component class aggregated can also be shown as in fig.

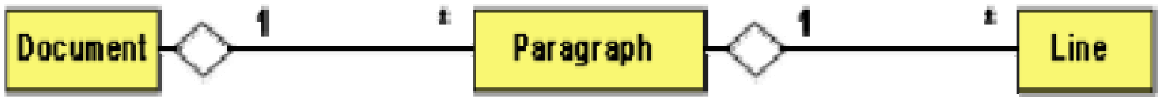


Fig.: Representation of aggregation

Aggregation relationship cannot be reflexive (i.e. recursive). That is, an object cannot contain objects of the same class as itself. Also, the aggregation relation is not symmetric. That is, two classes A and B cannot contain instances of each other. However, the aggregation relationship can be transitive. In this case, aggregation may consist of an arbitrary number of levels.

Composition

Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole. This means that the life of the parts closely ties to the life of the whole. When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed. A typical example of composition is an invoice object with invoice items. As soon as the invoice object is created, all the invoice items in it are created and as soon as the invoice object is destroyed, all invoice items in it are also destroyed. The composition relationship is represented as a filled diamond drawn at the composite-end. An example of the composition relationship is shown in fig.

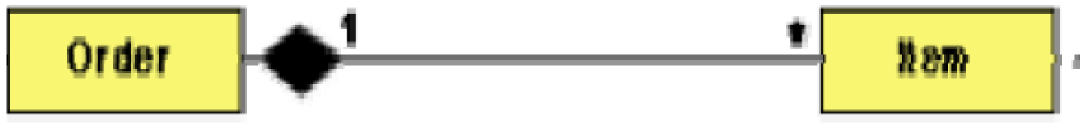


Fig.: Representation of composition

Association vs. Aggregation vs. Composition

- Association is the most general (m:n) relationship. Aggregation is a stronger relationship where one is a part of the other. Composition is even stronger than aggregation, ties the lifecycle of the part and the whole together.
- Association relationship can be reflexive (objects can have relation to itself), but aggregation cannot be reflexive. Moreover, aggregation is anti-symmetric (If B is a part of A, A can not be a part of B).
- Composition has the property of exclusive aggregation i.e. an object can be a part of only one composite at a time. For example, a **Frame** belongs to exactly one **Window** whereas in simple aggregation, a part may be shared by several objects. For example, a **Wall** may be a part of one or more **Room** objects.
- In addition, in composition, the whole has the responsibility for the disposition of all its parts, i.e. for their creation and destruction.

  o in general, the lifetime of parts and composite coincides
  o parts with non-fixed multiplicity may be created after composite itself
  o parts might be explicitly removed before the death of the composite

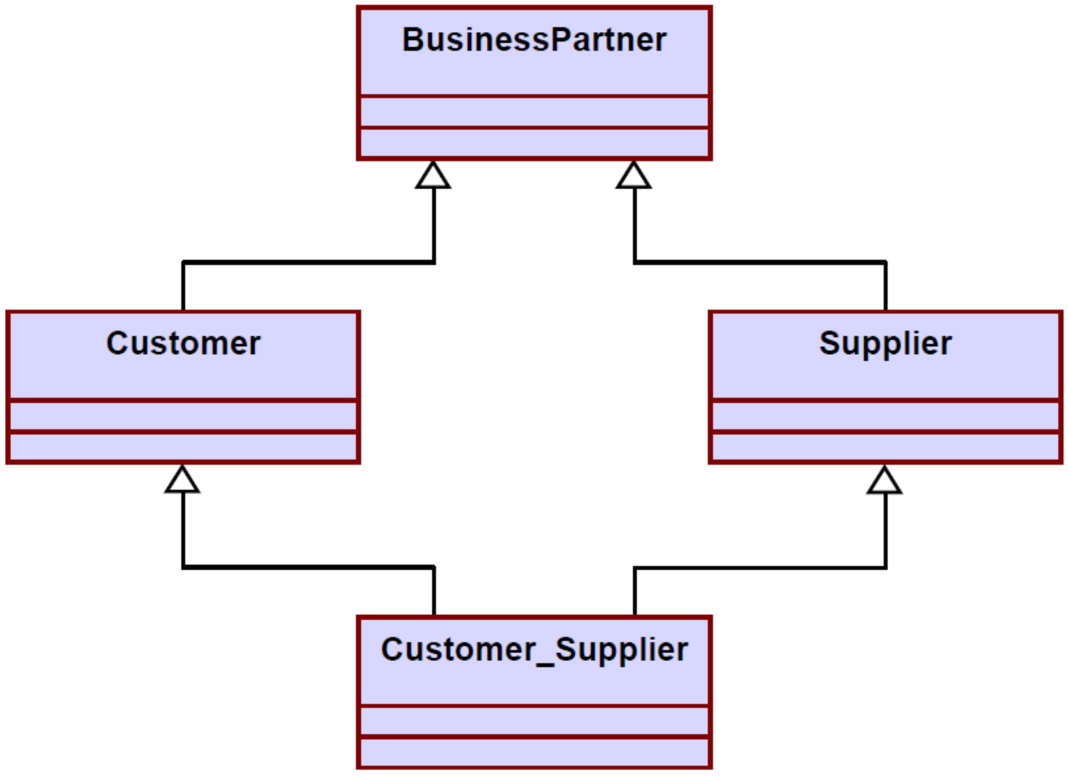
For example, when a **Frame** is created, it has to be attached to an enclosing **Window**. Similarly, when the **Window** is destroyed, it must in turn destroy its **Frame** parts.

Inheritance vs. Aggregation/Composition

- Inheritance describes *'is a' / 'is a kind of'* relationship between classes (base class - derived class) whereas aggregation describes *'has a'* relationship between classes. Inheritance means that the object of the derived class inherits the properties of the base class; aggregation means that the object of the whole has objects of the part. For example, the relation "cash payment *is a kind of* payment" is modeled using inheritance; "purchase order has a few items" is modeled using aggregation.
  Inheritance is used to model a "generic-specific" relationship between classes whereas aggregation/composition is used to model a "whole-part" relationship between classes.
- Inheritance means that the objects of the subclass can be used anywhere the super class may appear, but not the reverse; i.e. wherever we could use instances of 'payment' in the system, we could substitute it with instances of 'cash payment', but the reverse can not be done.
- Inheritance is defined statically. It can not be changed at run-time. Aggregation is defined dynamically and can be changed at run-time. Aggregation is used when the type of the object can change over time.

For example, consider this situation in a business system. A **BusinessPartner** might be a **Customer** or a **Supplier** or both. Initially we might be tempted to model it as in Fig 7.12(a). But in fact, during its lifetime, a business partner might become a customer as well as a supplier, or it might change from one to the other. In such cases, we prefer aggregation instead (see Fig 7.12(b). Here, a business partner is a **Customer** if it has an aggregated **Customer** object, a **Supplier** if it has an aggregated **Supplier** object and a "C**ustomer_Supplier**" if it has both. Here, we have only two types. Hence, we are able to model it as inheritance. But what if there were several different types and combinations there of? The inheritance tree would be absolutely incomprehensible.
Also, the aggregation model allows the possibility for a business partner to be neither - i.e. has neither a customer nor a supplier object aggregated with it.

- The advantage of aggregation is the integrity of encapsulation. The operations of an object are the interfaces of other objects which imply low implementation dependencies. The significant disadvantage of aggregation is the increase in the number of objects and their relationships. On the other hand, inheritance allows for an easy way to modify implementation for reusability. But the significant disadvantage is that it breaks encapsulation, which implies implementation dependence.



(a)

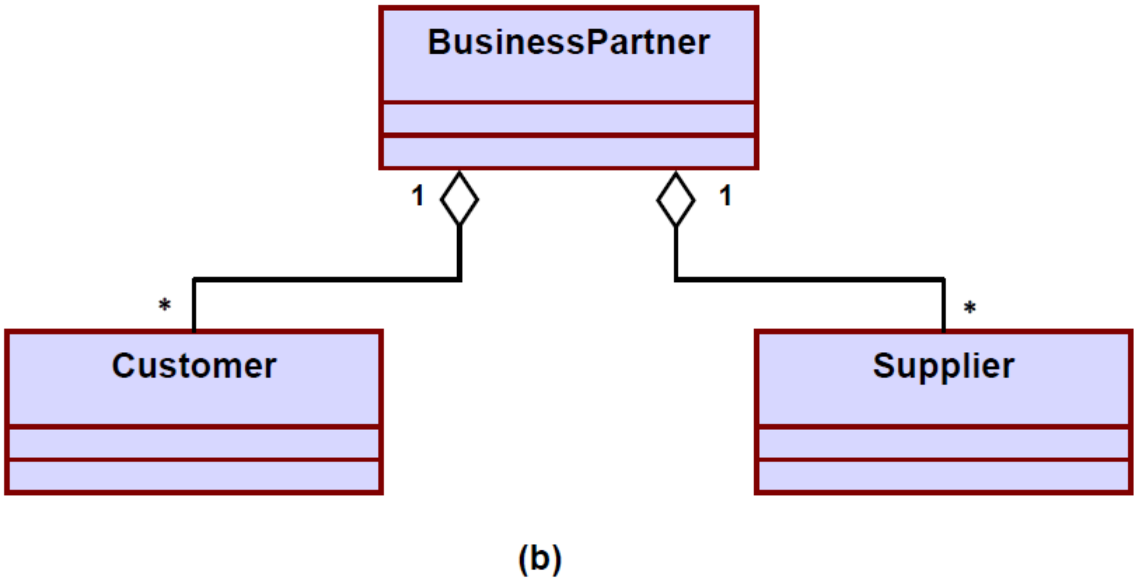


Fig. Representation of **BusinessPartner, Customer, Supplier** relationship **(a)** using inheritance **(b)** using aggregation

5.INTERACTION DIAGRAMS

Interaction diagrams are models that describe how group of objects collaborate to realize some behavior. Typically, each interaction diagram realizes the behavior of a single use case. An interaction diagram shows a number of example objects and the messages that are passed between the objects within the use case.

There are two kinds of interaction diagrams: sequence diagrams and collaboration diagrams. These two diagrams are equivalent in the sense that any one diagram can be derived automatically from the other. However, they are both useful. These two actually portray different perspectives of behavior of the system and different types of inferences can be drawn from them. The interaction diagrams can be considered as a major tool in the design methodology.

Sequence Diagram

A sequence diagram shows interaction among objects as a two dimensional chart. The chart is read from top to bottom. The objects participating in the interaction are shown at the top of the chart as boxes attached to a vertical dashed line. Inside the box the name of the object is written with a colon separating it from the name of the class and both the name of the object and the class are underlined. The objects appearing at the top signify that the object already existed when the use case execution was initiated. However, if some object is created during the execution of the use case and participates in the interaction (e.g. a method call), then the object should be shown at the appropriate place on the diagram where it is created. The vertical dashed line is called the object's lifeline. The lifeline indicates the existence of the object at any particular point of time. The rectangle drawn on the lifetime is called the activation symbol and indicates that the object is active as long as the rectangle exists. Each message is indicated as an arrow between the lifeline of two objects. The messages are shown in chronological order from the top to the bottom. That is, reading the diagram from the top to the bottom would show the sequence in which the messages occur. Each message is labeled with the message name. Some control information can also be included. Two types of control information are particularly valuable.

- A condition (e.g. [invalid]) indicates that a message is sent, only if the condition is true.
- An iteration marker shows the message is sent many times to multiple receiver objects as would happen when a collection or the elements of an array are being iterated. The basis of the iteration can also be indicated e.g. [for every book object].

The sequence diagram for the book renewal use case for the Library Automation Software is shown in fig. The development of the sequence diagram in the development methodology would help us in determining the responsibilities of the different classes; i.e. what methods should be supported by each class

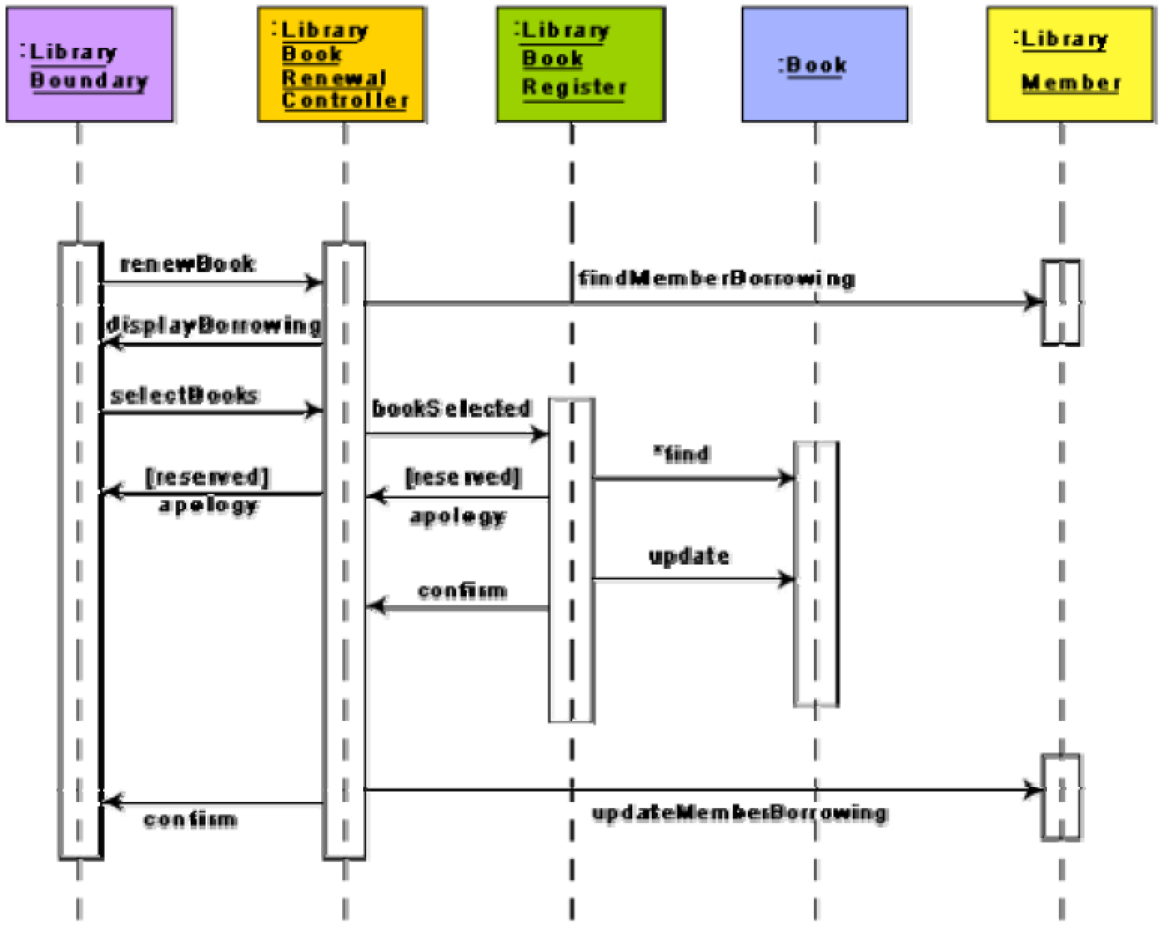


Fig.: Sequence diagram for the renew book use case

Collaboration Diagram

A collaboration diagram shows both structural and behavioral aspects explicitly. This is unlike a sequence diagram which shows only the behavioral aspects. The structural aspect of a collaboration diagram consists of objects and the links existing between them. In this diagram, an object is also called a collaborator. The behavioral aspect is described by the set of messages exchanged among the different collaborators. The link between objects is shown as a solid line and can be used to send messages between two objects. The message is shown as a labeled arrow placed near the link. Messages are prefixed with sequence numbers because they are only way to describe the relative sequencing of the messages in this diagram. The collaboration diagram for the example of fig. 7.13 is shown in fig. 7.14. The use of the collaboration diagrams in our development process would be to help us to determine which classes are associated with which other classes.

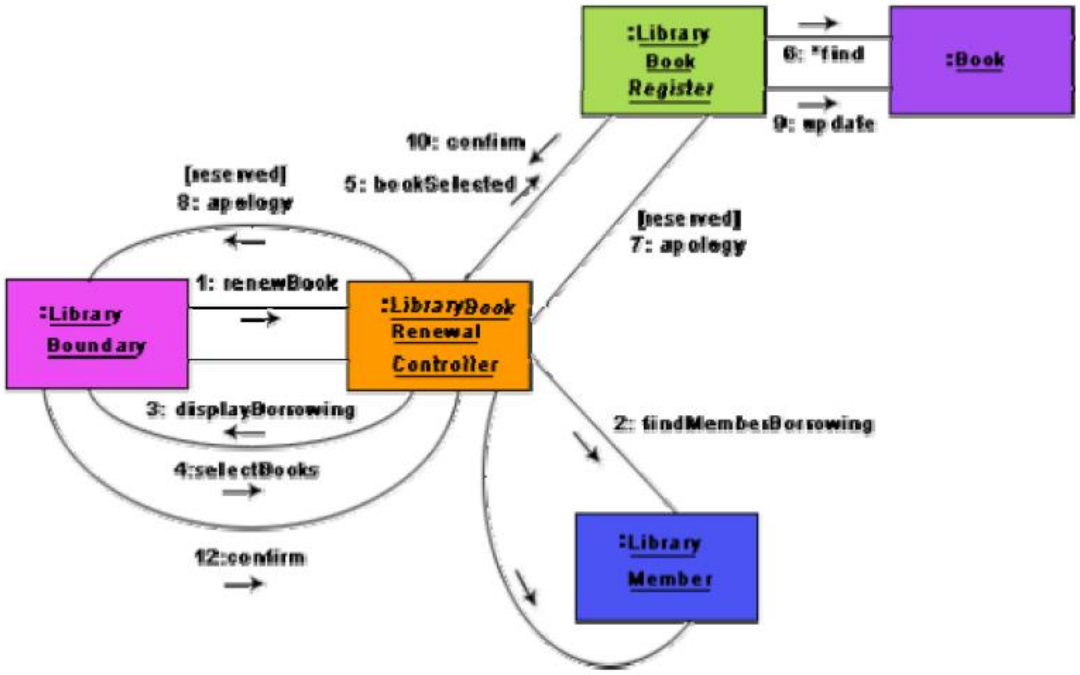


Fig.: Collaboration diagram for the renew book use case s

6.ACTIVITY DIAGRAMS

The activity diagram is possibly one modeling element which was not present in any of the predecessors of UML. No such diagrams were present either in the works of Booch, Jacobson, or Rumbaugh. It is possibly based on the event diagram of Odell [1992] through the notation is very different from that used by Odell. The activity diagram focuses on representing activities or chunks of processing which may or may not correspond to the methods of classes. An activity is a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity. If an activity has more than one outgoing transitions, then these must be identified through conditions. An interesting feature of the activity diagrams is the swim lanes. Swim lanes enable you to group activities based on who is performing them, e.g. academic department vs. hostel office. Thus swim lanes subdivide activities based on the responsibilities of some components. The activities in a swim lane can be assigned to some model elements, e.g. classes or some component, etc.

Activity diagrams are normally employed in business process modeling. This is carried out during the initial stages of requirements analysis and specification. Activity diagrams can be very useful to understand complex processing activities involving many components. Later these diagrams can be used to develop interaction diagrams which help to allocate activities (responsibilities) to classes.

The student admission process in IIT is shown as an activity diagram in fig. 7.15. This shows the part played by different components of the Institute in the admission procedure. After the fees are received at the account section, parallel activities start at the hostel office, hospital, and the Department. After all these activities complete (this synchronization is represented as a horizontal line), the identity card can be issued to a student by the Academic section.

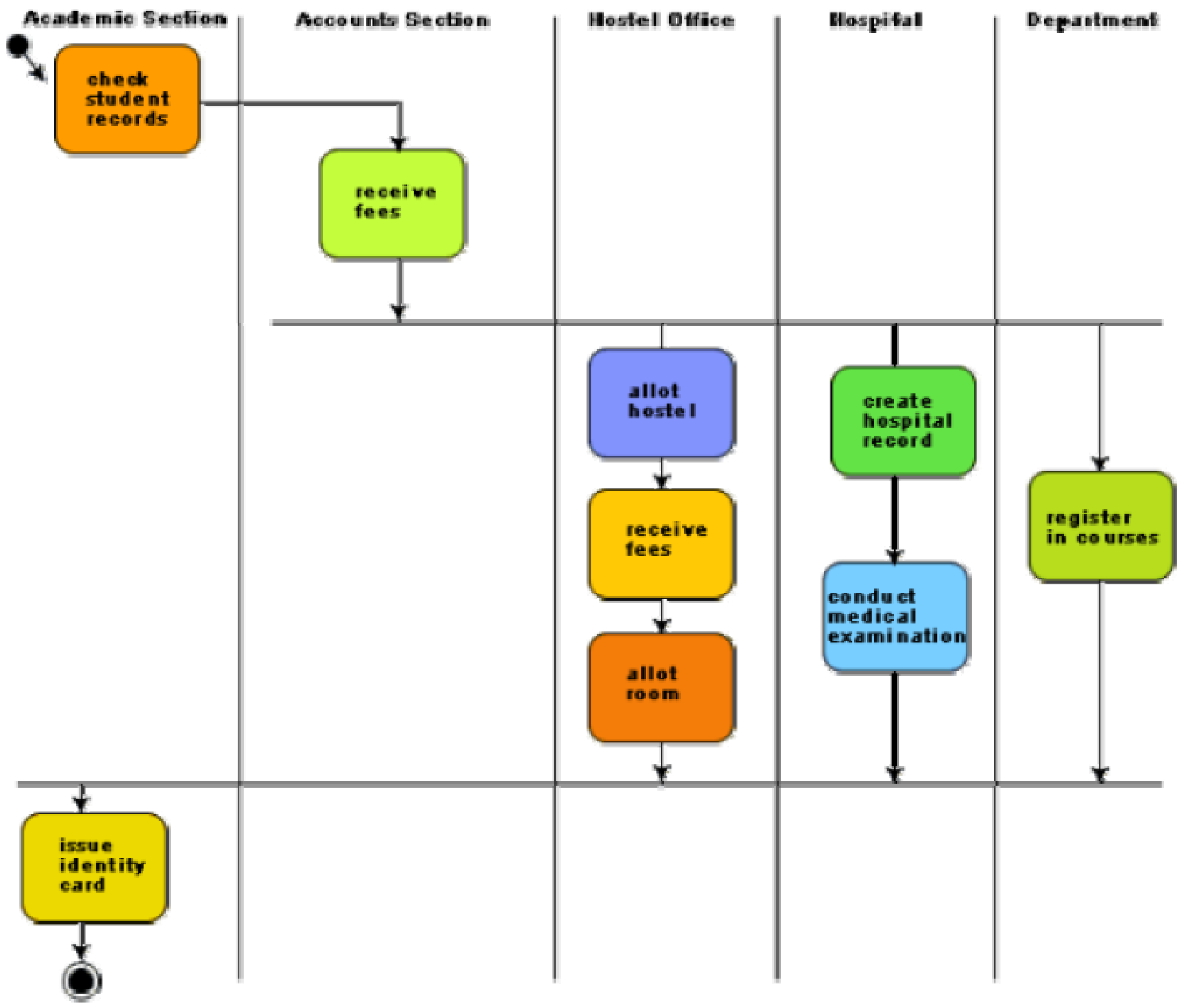


Fig.: Activity diagram for student admission procedure at IIT

Activity diagrams vs. procedural flow charts

Activity diagrams are similar to the procedural flow charts. The difference is that activity diagrams support description of parallel activities and synchronization aspects involved in different activities.

7. STATE CHART DIAGRAM

A state chart diagram is normally used to model how the state of an object changes in its lifetime. State chart diagrams are good at describing how the behavior of an object changes across several use case executions. However, if we are interested in modeling some behavior that involves several objects collaborating with each other, state chart diagram is not appropriate. State chart diagrams are based on the finite state machine (FSM) formalism.

An FSM consists of a finite number of states corresponding to those of the object being modeled. The object undergoes state changes when specific events occur. The FSM formalism existed long before the object-oriented technology and has been used for a wide variety of applications. Apart from modeling, it has even been used in theoretical computer science as a generator for regular languages.

A major disadvantage of the FSM formalism is the state explosion problem. The number of states becomes too many and the model too complex when used to model practical systems. This problem is overcome in UML by using state charts. The state chart formalism was proposed by David Harel [1990]. A state chart is a hierarchical model of a system and introduces the concept of a composite state (also called nested state).

Actions are associated with transitions and are considered to be processes that occur quickly and are not interruptible. Activities are associated with states and can take longer. An activity can be interrupted by an event.

The basic elements of the state chart diagram are as follows:

- **Initial state.** This is represented as a filled circle.
- **Final state.** This is represented by a filled circle inside a larger circle.
- **State.** These are represented by rectangles with rounded corners.
- **Transition.** A transition is shown as an arrow between two states. Normally, the name of the event which causes the transition is places along side the arrow. A guard to the transition can also be assigned. A guard is a Boolean logic condition. The transition can take place only if the grade evaluates to true. The syntax for the label of the transition is shown in 3 parts: event[guard]/action.

An example state chart for the order object of the Trade House Automation software is shown in fig.

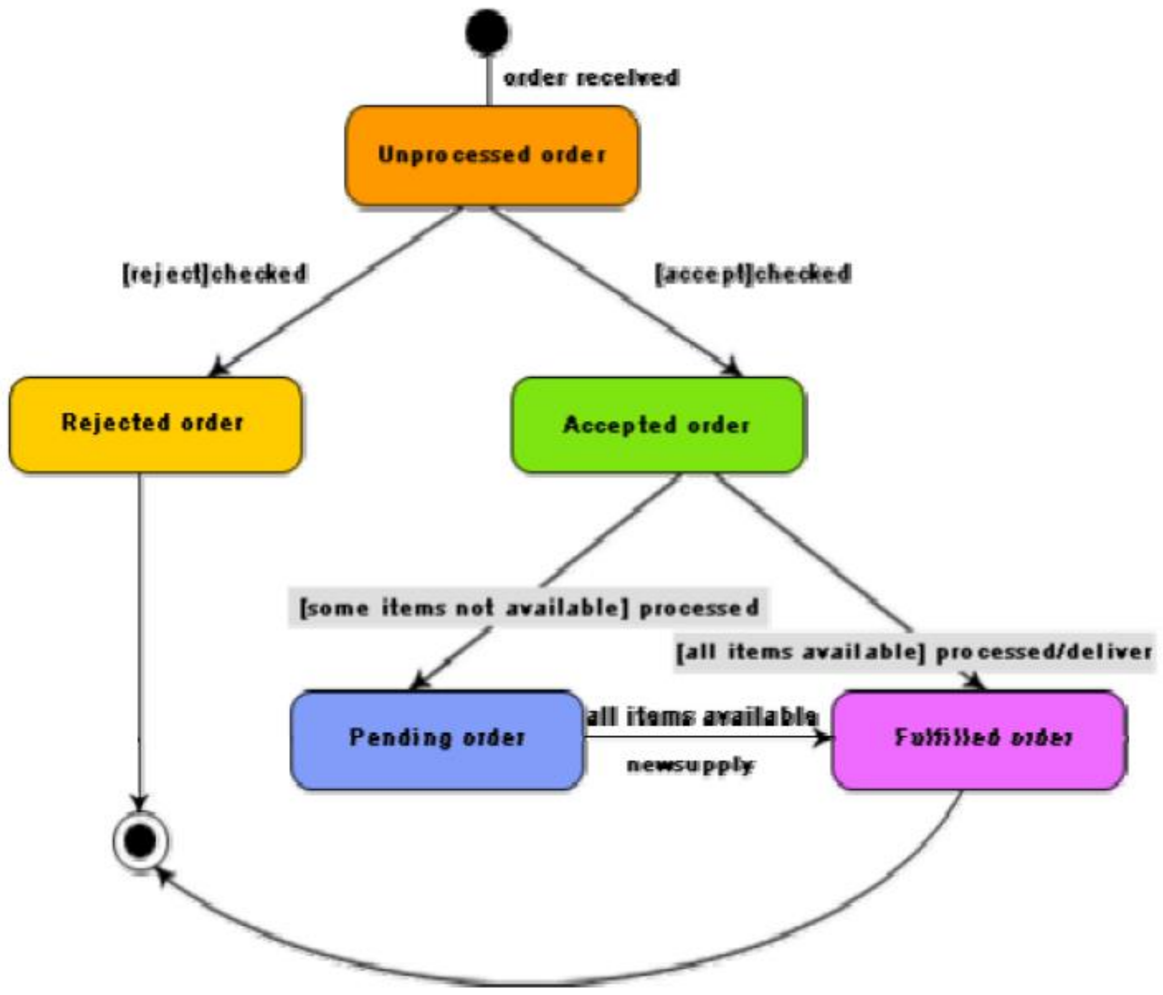


Fig.: State chart diagram for an order object

Activity diagram vs. State chart diagram

- Both activity and state chart diagrams model the dynamic behavior of the system Activity diagram is essentially a flowchart showing flow of control from activity to activity. A state chart diagram shows a state machine emphasizing the flow of control from state to state.
- An activity diagram is a special case of a state chart diagram in which all or most of the states are activity states and all or most of the transitions are triggered by completion of activities in the source state (An activity is an ongoing non-atomic execution within a state machine).
- Activity diagrams may stand alone to visualize, specify, and document the dynamics of a society of objects or they may be used to model the flow of control of an operation. State chart diagrams may be attached to classes, use cases, or entire systems in order to visualize, specify, and document the dynamics of an individual object.

8. DESIGN PATTERNS

Design patterns are reusable solutions to problems that recur in many applications. A pattern serves as a guide for creating a “good” design. Patterns are based on sound common sense and the application of fundamental design principles. These are created by people who spot repeating themes across designs. The pattern solutions are typically described in terms of class and interaction diagrams. Examples of design patterns are expert pattern, creator pattern, controller pattern etc.

8.1 BASIC PATTERN CONCEPTS:

Design patterns are very useful in creating good software design solutions. In addition to providing the model of a good solution, design patterns include a clear specification of the problem, and also explain the circumstances in which the solution would and would not work. Thus, a design pattern has four important parts:

- The problem.
- The context in which the problem occurs.
- The solution.
- The context within which the solution works.

8.2 TYPES OF PATTERNS

Starting from use in very high-level designs termed as architectural designs, pattern solutions have been defined for use in concrete designs and even in code. Some basic concepts about these three types of patterns as follows:

- **Architectural patterns**
  The architectural patterns identify and provide solutions to problems that are identifiable while carrying out architectural designs.
  Architectural designs concern the overall structure of software systems.
- **Design patterns**
  A design pattern usually suggests a scheme for structuring the classes in a design solution and defines the interaction required among those classes.
  In other words, a design pattern describes some commonly recurring structure of communicating classes that can be used to solve some general design problems.
  Design pattern solutions are typically described in terms of classes, their instances,their roles and collaborations.
- **Idioms**
  Idioms are low-level patterns that are programming language-specific.
  An idiom describes how to implement a solution to a particular problem using the features of a given programming language.

MORE PATTERN CONCEPTS

Few other important pattern concepts in the following.

Patterns versus algorithms

In contrast of algorithms, patterns are more concerned with aspects such as maintainability and ease of development rather than space and time efficiency.

Prons and cons of design patterns

Some advantages of design patterns:

- Design patterns providen a common vocabulary that helps to improve communication among the developers.
- Design patterns help capture and disseminate expert knowledge.

- Use of design patterns help designers to produce designs that are flexible, efficient, and easily maintainable.
- Design patterns guide developers to arrive at correct design decisions and help improve the quality of the design.
- Design patterns reduce the number of design iterations, and help improve the designer productivity.

Antipattern

The following are two types of antipatterns that are popular:

- Those that describe bad solutions to problems which leads to bad situations.
- Those that describe how to avoid bad solutions to problems.

We mention here only a few interesting antipatterns without discussing them in detail,

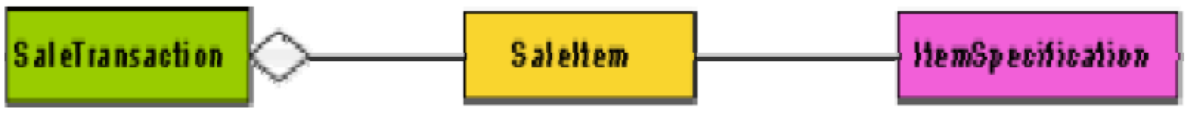
1. **Input kludge**: This concerns failing to specify and implement a mechanism for handling invalid inputs.
2. **Magic pushbutton:** This concerns coding implementation logic directly within the code of the user interface, rather than performing them in separate classes.
3. **Race hazard:** This concerns failing to see the consequence of all the different orders in which events might take place in practice.

8.3 SOME COMMON DESIGN PATTERNS:

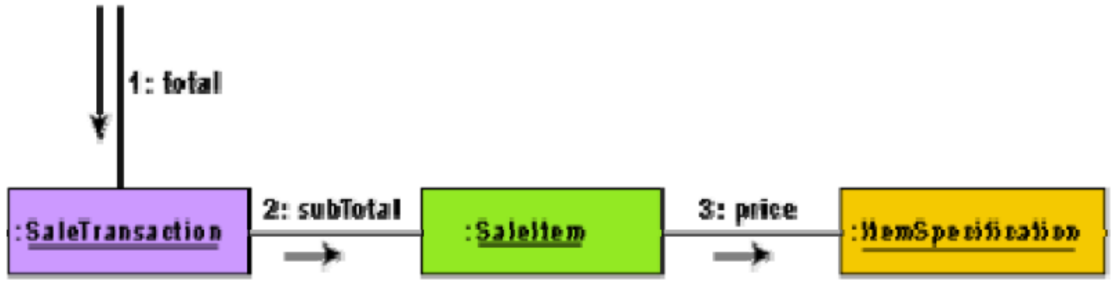
Expert Pattern

Problem: Which class should be responsible for doing certain things?

Solution: Assign responsibility to the information expert – the class that has the information necessary to fulfill the required responsibility. The expert pattern expresses the common intuition that objects do things related to the information they have. The class diagram and collaboration diagrams for this solution to the problem of which class should compute the total sales is shown in the fig.



(a)



(b)

Fig.: Expert pattern: (a) Class diagram (b) Collaboration diagram

Creator Pattern
Problem: Which class should be responsible for creating a new instance of some class?
Solution: Assign a class C1 the responsibility to create an instance of class C2, if one or more of the following are true**:**

- C1 is an aggregation of objects of type C2.
- C1 contains objects of type C2.
- C1 closely uses objects of type C2.
- C1 has the data that would be required to initialize the objects of type C2, when they are created.

Controller Pattern:
Problem: Who should be responsible for handling the actor requests?

Solution: For every use case, there should be a separate controller object which would be responsible for handling requests from the actor. Also, the same controller should be used for all the actor requests pertaining to one use case so that it becomes possible to maintain the necessary information about the state of the use case. The state information maintained by a controller can be used to identify the out-of-sequence actor requests, e.g. whether voucher request is received before arrange payment request.

Façade Pattern:
Problem: How should the services be requested from a service package?
Context in which the problem occurs: A package as already discussed is a cohesive set of classes – the classes have strongly related responsibilities. For example, an RDBMS interface package may contain classes that allow one to perform various operations on the RDBMS.

Solution: A class (such as DBfacade) can be created which provides a common interface to the services of the package.

Model View Separation Pattern:
Problem: How should the non-GUI classes communicate with the GUI classes?

Context in which the problem occurs: This is a very commonly occurring pattern which occurs in almost every problem. Here, model is a synonym for the domain layer objects, view is a synonym for the presentation layer objects such as the GUI objects.
Solution: The model view separation pattern states that model objects should not have direct knowledge (or be directly coupled) to the view objects. This means that there should not be any direct calls from other objects to the GUI objects. This results in a good solution, because the GUI classes are related to a particular application whereas the other classes may be reused.

There are actually two solutions to this problem which work in different circumstances as follows:

Solution 1: Polling or Pull from above

It is the responsibility of a GUI object to ask for the relevant information from the other objects, i.e. the GUI objects pull the necessary information from the other objects whenever required.

This model is frequently used. However, it is inefficient for certain applications. For example, simulation applications which require visualization, the GUI objects would not know when the necessary information becomes available. Other examples are, monitoring applications such as network monitoring, stock market quotes, and so on. In these situations, a "push-from-below" model of display update is required. Since "push-from-below" is not an acceptable solution, an indirect mode of communication from the other objects to the GUI objects is required.

Solution 2: Publish- subscribe pattern

An event notification system is implemented through which the publisher can indirectly notify the subscribers as soon as the necessary information becomes available. An event manager class can be defined which keeps track of the subscribers and the types of events they are interested in. An event is published by the publisher by sending a message to the event manager object. The event manager notifies all registered subscribers usually via a parameterized message (called a callback). Some languages specifically support event manager classes. For example, Java provides the EventListener interface for such purposes.

Intermediary Pattern or Proxy
Problem: How should the client and server objects interact with each other?
Context in the problem occurs: The client and server terms as used here refer to software components existing across a network. The clients are consumers of services provided by the servers.

Solution: A proxy object at the client side can be defined which is a local sit-in for the remote server object. The proxy hides the details of the network transmission. The proxy is responsible for determining the server address, communicating the client request to the server, obtaining the server response and seamlessly passing that to the client. The proxy can also augment (or filter) information that is exchanged between the client and the server. The proxy could have the same interface as the remote server object so that the client feels as if it is interacting directly with the remote server object and the complexities of network transmissions are abstracted out.

9. OBJECT ORIENTED ANALYSIS AND DESIGN METHODOLOGY

9.1 THE UNIFIED PROCESS

The two main characteristics of the unified process are: use case-driven and iterative. The use case model is the central model. All models that are constructed in the subsequent design activities must conform to the use case model.
The unified process involves iterating over the following four distinct phases as follows.
1.Inception: During this phase, the scope of the project is defined and prototype may be developed to form a clear idea about the project.
2.Elaboration: In this phase, the functional and the non-functional requirements are captured.
3.Construction: During this phase, analysis, design, and implementation activities are carried out. Full text descriptions of use cases are written during the construction phase and each use case is taken up for the start of a new iteration. System features are implemented in a series of short iterations. Each iteration results in an executable release of the software.
4.Transition: During this phase the product is installed in the user's environment and maintained. The design process that we discuss in this section can be undertaken during the construction phase of the unified process.

9.2 OVERVIEW OF THE OOAD METHODOLOGY

- The Object-Oriented Analysis and Design(OOAD) methodology that we are going to discuss has shown in Figure
- The use case model is developed first in any user-centric development process such as the unified process, all developed models must conformed to the use case model.
- Therefore, the use case model has a crucial role in the design process, and needs to be developed first.
- The domain model is constructed next through an analysis of the use case model and the SRS document. The domain model is refined into a class diagram through a number of iterations involving the interaction diagrams. Once the class diagram has been constructed, it can be easily be translated to code.
- Throughout the analysis and design process, a glossary is continually and consciously created and maintained.
- A glossary is a dictionary of terms which can help in understanding the various terms(or concepts) used in the constructed model.
- The terms listed in the glossary are essentially concept names.
- The glossary (or model dictionary) lists and all the terms that require explanation in order to improve communication and to reduce the risk of misunderstanding. Maintaining the glossary is an ongoing activity throughout the project.

9.3 USE CASE MODEL DEVELOPMENT

An overriding principle while identifying and packaging use cases is that there should be a strong correlation between the GUI prototype, the contents of the users' manual and the use case model of the system.

Each of the menu options in the top-level menu of the GUI would usually correspond to a package in the use case diagram.

Common mistakes committed in use case model development

The following are some common mistakes that beginners commit during use case model development. List of mistakes are:

1. **Clutter:**
2. **Too detailed:**
3. **Omitting text description:**
4. **Overlooking some alternate scenarios:**

9.4 DOMAIN METHODOLOGY

Domain modeling is known as conceptual modeling. A domain model is a representation of the concepts or objects appearing in the problem domain. It also captures the obvious relationships among these objects. Examples of such conceptual objects are the Book, BookRegister, MemeberRegister, LibraryMember, etc. The recommended strategy is to quickly create a rough conceptual model where the emphasis is in finding the obvious concepts expressed in the requirements while deferring a detailed investigation. Later during the development process, the conceptual model is incrementally refined and extended.

The objects identified during domain analysis can be classified into three types:

• Boundary objects

• Controller objects

• Entity objects

Boundary objects: The boundary objects are those with which the actors interact. These include screens, menus, forms, dialogs, etc. The boundary objects are mainly responsible for user interaction. Therefore, they normally do not include any processing logic. However, they may be responsible for validating inputs, formatting, outputs, etc. The boundary objects were earlier being called as the interface objects. However, the term interface class is being used for Java, COM/DCOM, and UML with different meaning. A recommendation for the initial identification of the boundary classes is to define one boundary class per actor/use case pair.

Entity objects: These normally hold information such as data tables and files that need to outlive use case execution, e.g. Book, BookRegister, LibraryMember, etc. Many of the entity objects are "dumb servers". They are normally responsible for storing data, fetching data, and doing some fundamental kinds of operation that do not change often.

Controller objects: The controller objects coordinate the activities of a set of entity objects and interface with the boundary objects to provide the overall behavior of the system. The responsibilities assigned to a controller object are closely related to the realization of a specific use case. The controller objects effectively decouple the boundary and entity objects from one another making the system tolerant to changes of the user interface and processing logic. The controller objects embody most of the logic involved with the use case realization (this logic may change time to time). A typical interaction of a controller object with boundary and entity objects is shown in fig. Normally, each use case is realized using one controller object.

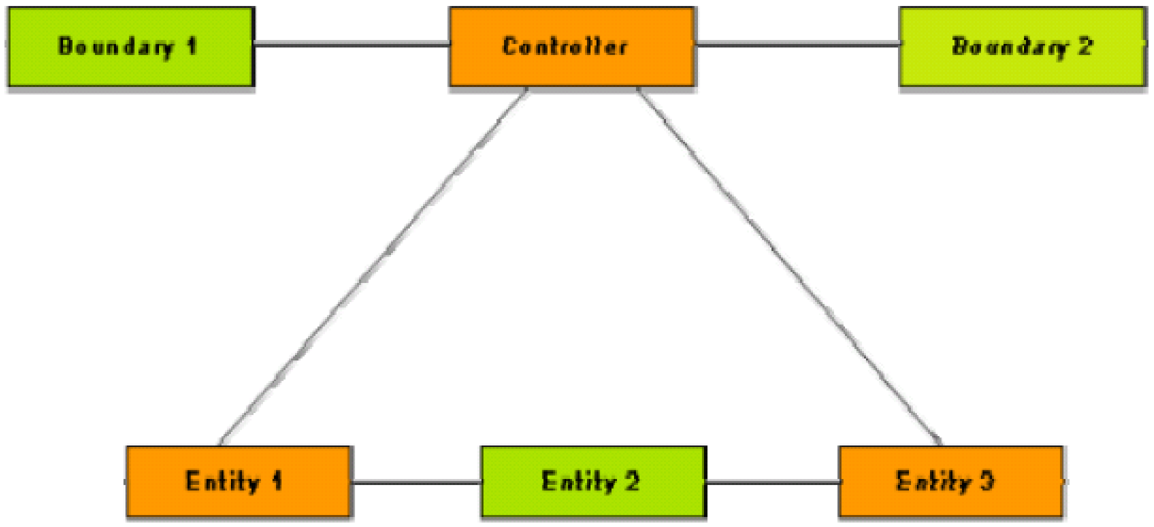


Fig. A typical realization of a use case through the collaboration of boundary, controller, and entity objects

9.5 IDENTIFICATION OF ENTITY OBJECTS

One of the most important steps in any object-oriented design methodology is the identification of objects. In fact, the quality of the final design depends to a great extent on the appropriateness of the objects identified. However, to date no formal methodology exists for identification of objects. Several semi-formal and informal approaches have been proposed for object identification. These can be classified into the following broad classes**:**

- Grammatical analysis of the problem description.
- Derivation from data flow.
- Derivation from the entity relationship (E-R) diagram.

A widely accepted object identification approach is the grammatical analysis approach. Grady Booch originated the grammatical analysis approach [1991]. In Booch's approach, the nouns occurring in the extended problem description statement (processing narrative) are mapped to objects and the verbs are mapped to methods. The identification approaches based on derivation from the data flow diagram and the entity-relationship model are still evolving and therefore will not be discussed in this text.

9.6 BOOCH'S OBJECT IDENTIFICATION METHOD

Booch's object identification approach requires a processing narrative of the given problem to be first developed. The processing narrative describes the problem and discusses how it can be solved. The objects are identified by noting down the nouns in the processing narrative. Synonym of a noun must be eliminated. If an object is required to implement a solution, then it is said to be part of the solution space. Otherwise, if an object is necessary only to describe the problem, then it is said to be a part of the problem space. However, several of the nouns may not be objects. An imperative procedure name, i.e., noun form of a verb actually represents an action and should not be considered as an object. A potential object found after lexical analysis is usually considered legitimate, only if it satisfies the following criteria**:**

Retained information. Some information about the object should be remembered for the system to function. If an object does not contain any private data, it can not be expected to play any important role in the system.

Multiple attributes. Usually objects have multiple attributes and support multiple methods. It is very rare to find useful objects which store only a single data element or support only a single method, because an object having only a single data element or method is usually implemented as a part of another object.

Common operations. A set of operations can be defined for potential objects. If these operations apply to all occurrences of the object, then a class can be defined. An attribute or operation defined for a class must apply to each instance of the class. If some of the attributes or operations apply only to some specific instances of the class, then one or more subclasses can be needed for these special objects.

Normally, the actors themselves and the interactions among themselves should be excluded from the entity identification exercise. However, some times there is a need to maintain information about an actor within the system. This is not the same as modeling the actor. These classes are sometimes called surrogates. For example, in the Library Information System (LIS) we would need to store information about each library member. This is independent of the fact that the library member also plays the role of an actor of the system description. Useful abstractions usually result from clever factoring of the problem description into independent and intuitively correct elements.

Although the grammatical approach is simple and intuitively appealing, yet through a naive use of the approach, it is very difficult to achieve high quality results. In particular, it is very difficult to come up with useful abstractions simply by doing grammatical analysis of the problem

10. INTERACTION MODELING

The primary goal of interaction modeling are the following**:**

- To allocate the responsibility of a use case realization among the boundary, entity, and controller objects. The responsibilities for each class is reflected as an operation to be supported by that class.
- To show the detailed interaction that occur over time among the objects associated with each use case.

CRC cards

The interactions diagrams for only simple use cases that involve collaboration among a limited number of classes can be drawn from an inspection of the use case description. More complex use cases require the use of CRC cards where a number of team members participate to determine the responsibility of the classes involved in the use case realization.

CRC (Class-Responsibility-Collaborator) technology was pioneered by Ward Cunningham and Kent Becka at the research laboratory of Tektronix at Portland, Oregon, USA. CRC cards are index cards that are prepared one per each class. On each of these cards, the responsibility of each class is written briefly. The objects with which this object needs to collaborate its responsibility are also written.

CRC cards are usually developed in small group sessions where people role play being various classes. Each person holds the CRC card of the classes he is playing the role of. The cards are deliberately made small (4 inch ´ 6 inch) so that each class can have only limited number of responsibilities. A responsibility is the high level description of the part that a class needs to play in the realization of a use case. An example CRC card for the BookRegister class of the Library Automation System is shown in fig

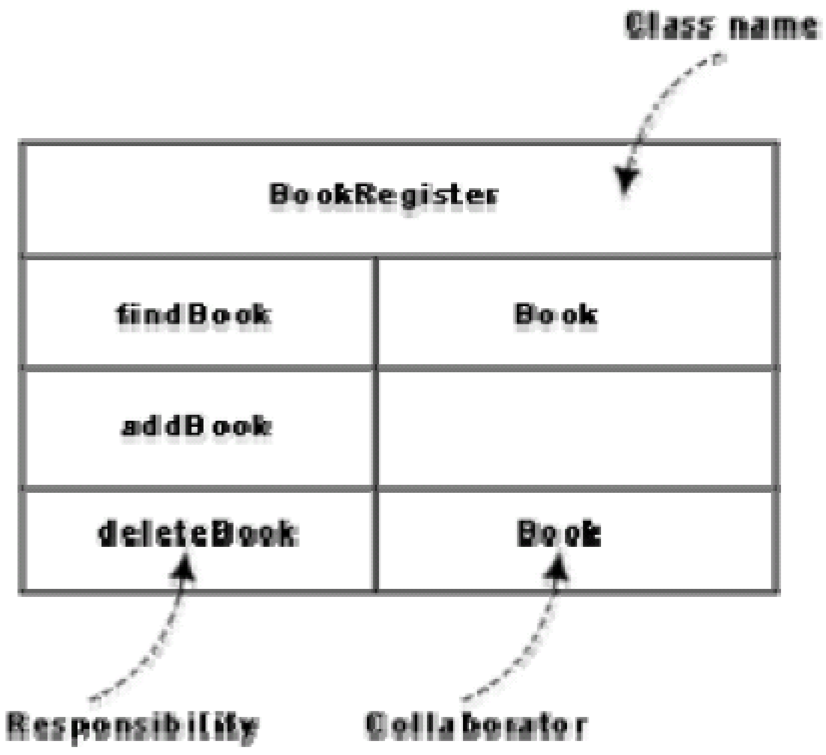


Fig. CRC card for the BookRegister class

11. OOD GOODNESS CRITERIA

there are several subjective judgments involved in arriving at a good object-oriented design. Therefore, several alternative design solutions to the same problem are possible. In order to be able to determine which of any two designs is better, some criteria for judging the goodness of a design must be identified. The following are some of the accepted criteria for judging the goodness of a design.

1. **Coupling guidelines.** The number of messages between two objects or among a group of objects should be minimum. Excessive coupling between objects is determined to modular design and prevents reuse.
2. **Cohesion guideline.** In OOD, cohesion is about three levels**:**

   - **Cohesiveness of the individual methods.** Cohesiveness of each of the individual method is desirable, since it assumes that each method does only a well-defined function.
   - **Cohesiveness of the data and methods within a class.** This is desirable since it assures that the methods of an object do actions for which the object is naturally responsible, i.e. it assures that no action has been improperly mapped to an object
   - **Cohesiveness of an entire class hierarchy.** Cohesiveness of methods within a class is desirable since it promotes encapsulation of the objects.
3. **Hierarchy and factoring guidelines.** A base class should not have too many subclasses. If too many subclasses are derived from a single base class, then it becomes difficult to understand the design. In fact, there should approximately be no more than 7±2 classes derived from a base class at any level.
4. **Keeping message protocols simple.** Complex message protocols are an indication of excessive coupling among objects. If a message requires more than 3 parameters, then it is an indication of bad design.
5. • **Number of Methods.** Objects with a large number of methods are likely to be more application-specific and also difficult to comprehend – limiting the possibility of their reuse. Therefore, objects should not have too many methods. This is a measure of the complexity of a class. It is likely that the classes having more than about seven methods would have problems.
6. • **Depth of the inheritance tree.** The deeper a class is in the class inheritance hierarchy, the greater is the number of methods it is likely to inherit, making it more complex. Therefore, the height of the inheritance tree should not be very large.
7. • **Number of messages per use case.** If methods of a large number of objects are invoked in a chain action in response to a single message, testing and debugging of the objects becomes complicated. Therefore, a single message should not result in excessive message generation and transmission in a system.
8. • **Response for a class.** This is a measure of the maximum number of methods that an instance of this class would call. If the same method is called more than once, then it is counted only once. A class which calls more than about seven different methods is susceptible to errors.

UNIT- V

User Interface Design and Testing: Characteristics of a good User Interface – Types– Fundamentals of Component based GUI Development – A User Interface Design methodology – Coding – Software Documentation – Testing – Unit Testing – Black Box testing – White Box testing – Debugging – Program Analysis tools – Integration testing – Testing Object Oriented programs – System Testing – Issues

1. CHARACTERISTICS OF A GOOD USER INTERFACE

1.1 Good User Interface (GUI):

- A Good User Interface has high conversion rates and is easy to use. In other words, it's nice to both the business side as well as the people using it.
- It is a type of user interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation, instead of text-based user interfaces, typed command labels or text navigation.

1.2 Characteristics of User Interface:

It is very important to identify the characteristics desired of a good user interface. Because unless we are aware of these, it is very much difficult to design a good user interface. A few important characteristics of a good user interface are the following:

Speed of learning:

- ➢ A good user interface should be easy to learn.
- ➢ Speed of learning is hampered by complex syntax and semantics of the command issue procedures.
- ➢ A good user interface should not require its users to memorize commands.
- ➢ Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface.
- ➢ Besides, the following three issues are crucial to enhance the speed of learning:
    - Use of Metaphors and intuitive command names
    - Consistency
    - Component-based interface

Speed of use:

- ➢ Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands.
- ➢ It indicates how fast the users can perform their intended tasks.
- ➢ The time and user effort necessary to initiate and execute different commands should be minimal.

- This can be achieved through careful design of the interface.

Speed of recall:

- Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximized.
- This characteristic is very important for intermittent users.

Error prevention:

- A good user interface should minimize the scope of committing errors while initiating different commands.
- The error rate of an interface can be easily determined by monitoring the errors committed by average users while using the interface.

Attractiveness:

- A good user interface should be attractive to use.
- An attractive user interface catches user attention and fancy.
- In this respect, graphics-based user interfaces have a definite advantage over text-based interfaces.

Consistency:

- The commands supported by a user interface should be consistent.
- The basic purpose of consistency is to allow users to generalize the knowledge about aspects of the interface from one part to another.
- Thus, consistency facilitates speed of learning, speed of recall, and also helps in reduction of error rate.

Feedback:

- A good user interface must provide feedback to various user actions.
- Especially, if any user request takes more than few seconds to process, the user should be informed about the state of the processing of his request.
- In the absence of any response from the computer for a long time, a novice user might even start recovery/shutdown procedures in panic.
- If required, the user should be periodically informed about the progress made in processing his command.

Support for multiple skill levels:

- A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users.
- This is necessary because users with different levels of experience in using an application prefer different types of user interfaces.

Error recovery (undo facility):

- While issuing commands, even the expert users can commit errors. Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface.
- Users are put to inconvenience, if they cannot recover from the errors they commit while using the software.

User guidance and on-line help:

- Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software.
- Whenever users need guidance or seek help from the system, they should be provided with the appropriate guidance and help.

2. TYPES OF USER INTERFACES

User interfaces can be classified into the following three categories:

a. Command language based interfaces
b. Menu-based interfaces
c. Direct manipulation interfaces

Command Language-based Interface:

- A command language-based interface – as the name itself suggests, is based on designing a command language which the user can use to issue the commands.
- The user is expected to frame the appropriate commands in the language and type them in appropriately whenever required.
- A simple command language-based interface might simply assign unique names to the different commands.
- **Advantages:**

1. Easy to develop
2. Can be implemented even on cheap alphanumeric terminals
3. Much more efficient

- **Disadvantages:**

1. Difficult to learn
2. Require the user to type in commands
3. Also, most users make errors while formulating commands in the command language and also while typing them in.

Menu-based Interface:

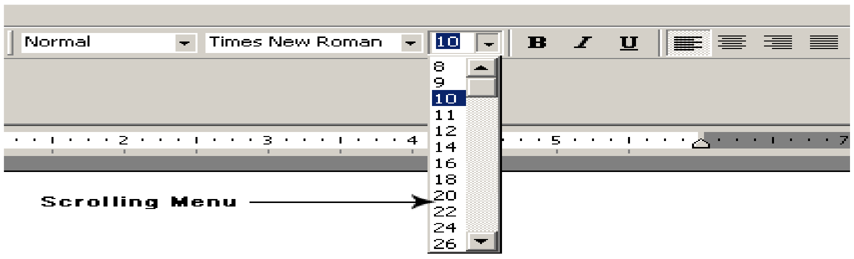
- An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands.

- A menu-based interface is based on recognition of the command names, rather than recollection.
- When the menu choices are large, they can be structured as the following way:

Types of Menu (or) Menu Based Interfaces:

1. **Scrolling menu:**

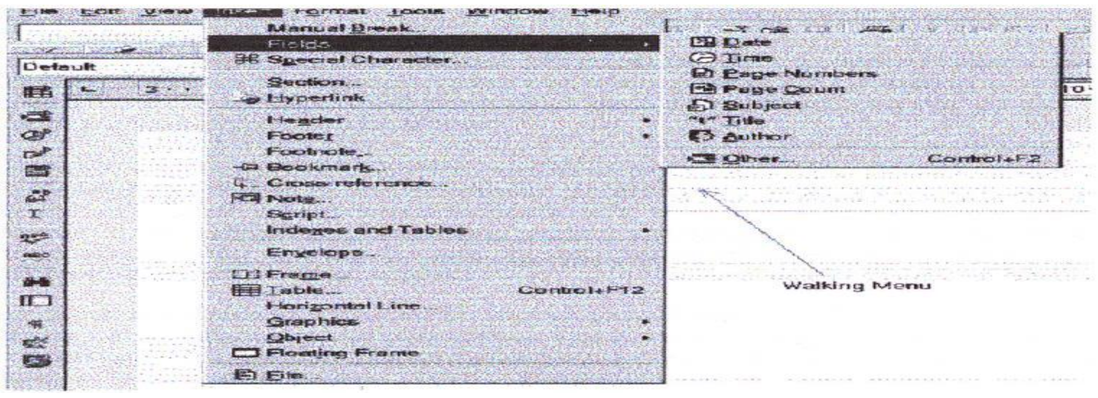
- When a full choice list cannot be displayed within the menu area, scrolling of the menu items is required.
- This would enable the user to view and select the menu items that cannot be accommodated on the screen.
- However, in a scrolling menu all the commands should be highly correlated, so that the user can easily locate a command that he needs.



(Fig: Scrolling Menu)

2. **Walking menu:**

- Walking menu is very commonly used to structure a large collection of menu items.
- In this technique, when a menu item is selected, it causes further menu items to be displayed adjacent to it in a sub-menu.



(Fig: Walking Menu)

3. **Hierarchical menu:**

- In this technique, the menu items are organized in a hierarchy or tree structure.
- Selecting a menu item causes the current menu display to be replaced by an appropriate sub-menu.
- Thus in this case, one can consider the menu and its various sub-menus to form a hierarchical tree-like structure.

- Hierarchical menu can be used to manage large number of choices, but the users are likely to face navigational problems because they might lose track of where they are in the menu tree.

Advantages:

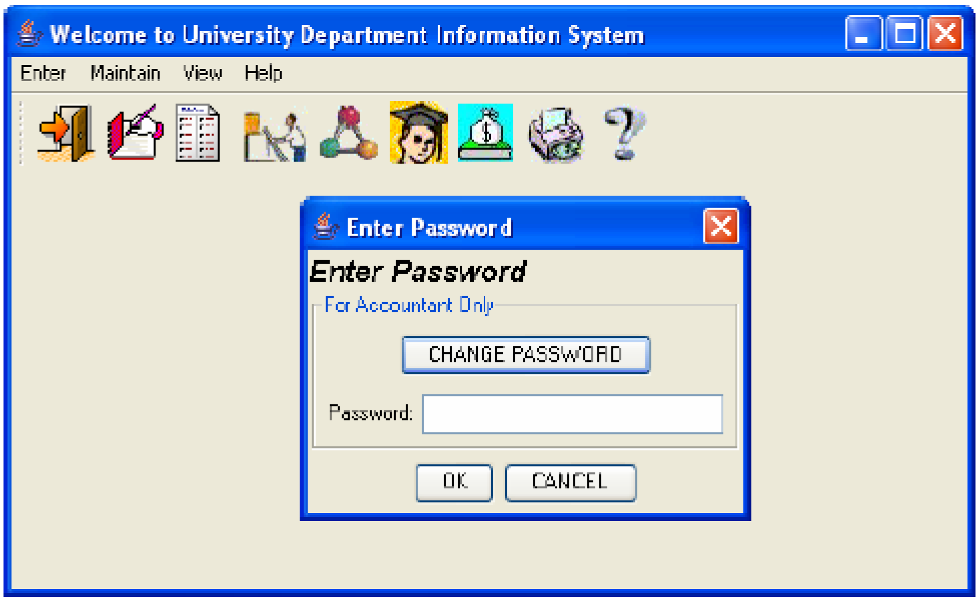
- Users are not required to remember exact command names.
- typing effort is minimal:
  - Menu selections using a pointing device.
  - This factor becomes very important for the occasional users who cannot type fast.
- For experienced users:
  - menu-based interfaces is slower than command language interfaces
  - experienced users can type fast
  - Also get speed advantage by composing simple commands into complex commands.

Disadvantages:

- It is difficult to design a menu-based interface.
- Even moderate sized software needs hundreds or thousands of menu choices.
- Structuring large number of menu choices into manageable forms.

<u>Direct Manipulation Interfaces:</u>

- Direct manipulation interfaces present the interface to the user in the form of visual models (i.e. icons or objects).
- For this reason, direct manipulation interfaces are sometimes called as iconic interface.



(Fig: Example of an iconic interface)

- In this type of interface, the user issues commands by performing actions on the visual representations of the objects. e.g. pull an icon representing a file into an icon representing a trash box, for deleting the file.

Advantages:

- Icons can be recognized by users very easily
- Icons are language-independent

Disadvantages:

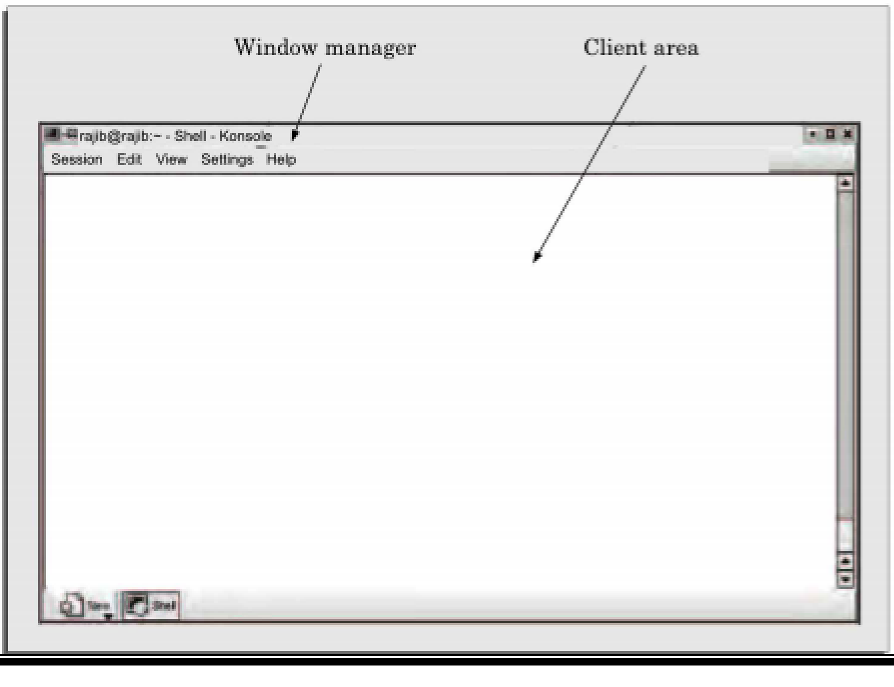
- However, experienced users consider direct manipulation interfaces too slowly.
- It is difficult to form complex commands using a direct manipulation interface.

- For example, if one has to drag an icon representing the file to a trash box icon for deleting a file, then in order to delete all the files in the directory one has to perform this operation individually for all files – which could be very easily done by issuing a command like delete *.*.

3. FUNDAMENTALS OF COMPONENT BASED GUI DEVELOPMENT

3.1 Window:

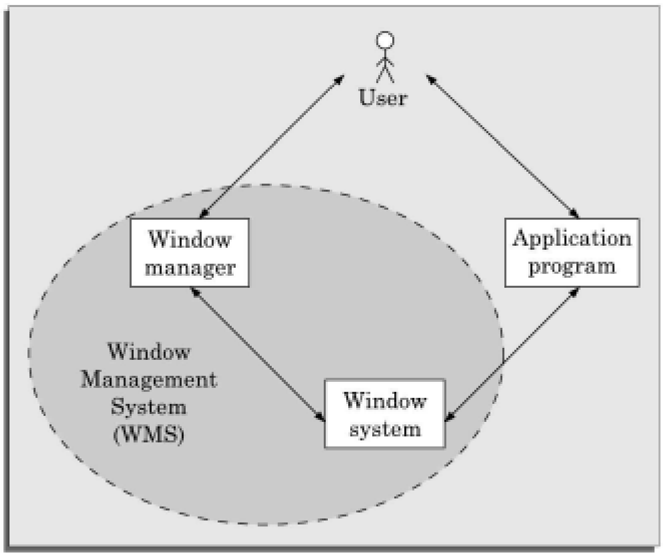
- A window is a rectangular area on the screen.
- A window can be considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent activities, e.g. one window can be used for editing a program and another for drawing pictures, etc.
- A window can be divided into two parts: client part, and non-client part.
- The client area makes up the whole of the window, except for the borders and scroll bars.
- The client area is the area available to a client application for display.
- The non-client part of the window determines the look and feel of the window.
- The look and feel defines a basic behavior for all windows, such as creating, moving, resizing, and iconifying the windows. A basic window with its different parts is shown in fig.


Window manager
Client area
rajib@rajib:~ - Shell - Konsole
Session Edit View Settings Help
New
Shell

(Fig: Window with client and user areas marked)

3.2 Window Management System (WMS):

- A graphical user interface typically consists of a large number of windows. Therefore, it is necessary to have some systematic way to manage these windows.
- Most graphical user interface development environments do this through a window management system (WMS).
- A window management system is primarily a resource manager. It keeps track of the screen area resource and allocates it to the different windows that seek to use the screen.
- A WMS simplifies the task of a GUI designer to a great extent by providing the basic behavior to the various windows such as move, resize, iconify, etc. as soon as they are created and by providing the basic routines to manipulate the windows from the application such as creating, destroying, changing different attributes of the windows, and drawing text, lines, etc.
- A WMS consists of two parts:
  1. Window manager
  2. Window system



(Fig: Window Management System)

3.2.1 Window Manager and Window System:

- Window manager is the component of WMS with which the end user interacts to do various window-related operations such as window repositioning, window resizing, iconification, etc.
- The window manager is built on the top of the window system in the sense that it makes use of various services provided by the window system.
- The window manager and not the window system determines how the windows look and behave.

- In fact, several kinds of window managers can be developed based on the same window system.
- The window manager can be considered as a special kind of client that makes use of the services (function calls) supported by the window system.
- The application programmer can also directly invoke the services of the window system to develop the user interface.
- The relationship between the window manager, window system, and the application program is shown in fig. This figure shows that the end-user can either interact with the application itself or with the window manager (resize, move, etc.) and both the application and the window manager invoke services.
- The window manager is responsible for managing and maintaining the non-client area of a window.
- Window manager manages the real-estate policy, provides look and feel of each individual window.

3.2.2 Types of widgets (window objects):

Different interface programming packages support different widget sets. However, a surprising number of them contain similar kinds of widgets, so that one can think of a generic widget set which is applicable to most interfaces. The following widgets are representatives of this generic class.

Label widget: This is probably one of the simplest widgets. A label widget does nothing except to display a label, i.e. it does not have any other interaction capabilities and is not sensitive to mouse clicks. A label widget is often used as a part of other widgets.

Container widget: These widgets do not stand by themselves, but exist merely to contain other widgets. Other widgets are created as children of the container widget. When the container widget is moved or resized, its children widget also get moved or resized. A container widget has no callback routines associated with it.

Pop-up menu: These are transient and task specific. A pop-up menu appears upon pressing the mouse button, irrespective of the mouse position.

Pull-down menu: These are more permanent and general. You have to move the cursor to a specific location and pull down this type of menu.

Dialog boxes: A dialog box can include areas for entering text as well as values. If an apply command is supported in a dialog box, the newly entered values can be tried without dismissing the box.

Push button: A push button contains key words or pictures that describe the action that is triggered when you activate the button. Usually, the action related to a push button occurs immediately when you click a push button unless it contains an ellipsis (…). A push button with an ellipsis generally indicates that another dialog box will appear.

Radio buttons: A set of radio buttons is used when only one option has to be selected out of many options. A radio button is a hollow circle followed by text describing the option it stands for. When a radio button is selected, it appears filled and the previously selected radio button from the group is unselected.

Combo boxes: A combo box looks like a button until the user interacts with it. When the user presses or clicks it, the combo box displays a menu of items to choose from. Normally a combo box is used to display either one-of-many choices when space is limited, the number of choices is large, or when the menu items are computed at run-time.

3.3 X-Window:

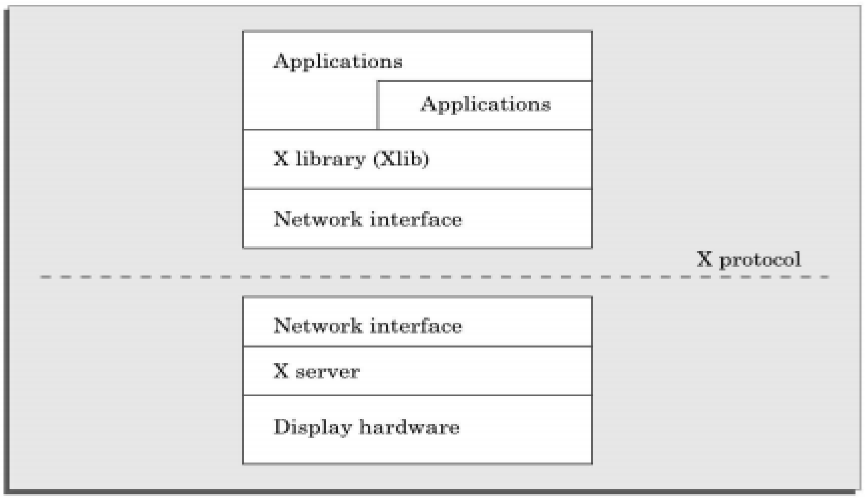
- The X-window functions are low-level functions written in C language which can be called from application programs. But only the very serious application designer would program directly using the X-windows library routines.
- One of the most widely used widget sets is X/Motif. Digital Equipment Corporation (DEC) used the basic X-window functions to develop its own look and feel for interface designs called DEC Windows.

3.3.1 Architecture of an X-System:

The X-architecture is pictorially depicted in fig.. The different terms used in this diagram are explained below:

X-server: The X server runs on the hardware to which the display and keyboard attached. The X server performs low-level graphics, manages window, and user input functions. The X server controls accesses to a bit-mapped graphics display resource and manages it.

X-protocol: The X protocol defines the format of the requests between client applications and display servers over the network. The X protocol is designed to be independent of hardware, operating systems, underlying network protocol, and the programming language used.



(Fig: Architecture of the X-System)

X-library (Xlib): The Xlib provides a set of about 300 utility routines for applications to call. These routines convert procedure calls into requests that are transmitted to the server.

Xtoolkit (Xt): The Xtoolkit consists of two parts: the intrinsics and the widgets. We have already seen that widgets are predefined user interface components such as scroll bars, push buttons, etc. for designing GUIs. Intrinsics are a set of about a dozen library routines that allow a programmer to combine a set of widgets into a user interface.

3.4 Visual Programming:

- Visual programming is the drag and drop style of program development. In this style of user interface development, a number of visual objects (icons) representing the GUI components are provided by the programming environment.
- The application programmer can easily develop the user interface by dragging the required component types (e.g. menu, forms, etc.) from the displayed icons and placing them wherever required.
- Thus, visual programming can be considered as program development through manipulation of several visual objects.

3.5 Implications of human cognition capabilities on user interface design:

An area of human-computer interaction where extensive research has been conducted is how human cognitive capabilities and limitations influence the way an interface should be designed. The following are some of the prominent issues extensively discussed in the literature.

(i) Limited memory. Humans can remember at most seven unrelated items of information for short periods of time. Therefore, the GUI designer should not require the user to remember too many items of information at a time.

(ii) Frequent task closure: Doing a task (except for very trivial tasks) requires doing several subtasks. When the system gives a clear feedback to the user that a task has been successfully completed, the user gets a sense of achievement and relief.

(iii) Recognition rather than recall: Information recall incurs a larger memory burden on the users and is to be avoided as far as possible. On the other hand, recognition of information from the alternatives shown to him is more acceptable.

(iv) Procedural versus object-oriented: Procedural designs focus on tasks, prompting the user in each step of the task, giving them few options for anything else. This approach is the best applied in situations where the tasks are narrow and well-defined or where the users are inexperienced, such as an ATM. An object-oriented interface on the other hand focuses on objects. This allows the users a wide range of options.

4. A USER INTERFACE DESIGN METHODOLOGY

GUI design methodology consists of the following important steps:

1. Examine the use case model of the software. Interview, discuss, and review the GUI issues with the end-users.
2. Task and object modeling
3. Metaphor selection
4. Interaction design and rough layout
5. Detailed presentation and graphics design
6. GUI construction
7. Usability evaluation

- ➢ The starting point for GUI design is the use case model.
- ➢ This captures the important tasks the users need to perform using the software.
- ➢ As far as possible, a user interface should be developed using one or more metaphors.
- ➢ Metaphors help in interface development at lower effort and reduced costs for training the users. Over time, people have developed efficient methods of dealing with some commonly occurring situations.
- ➢ These solutions are the themes of the metaphors. Metaphors can also be based on physical objects such as a visitor's book, a catalog, a pen, a brush, a scissor, etc.
- ➢ A solution based on metaphors is easily understood by the users, reducing learning time and training costs.
- ➢ Some commonly used metaphors are the following:

- White board
- Shopping cart
- Desktop
- Editors work bench
- White page
- Yellow page
- Office cabinet
- Post box
- Bulletin board
- Visitors book

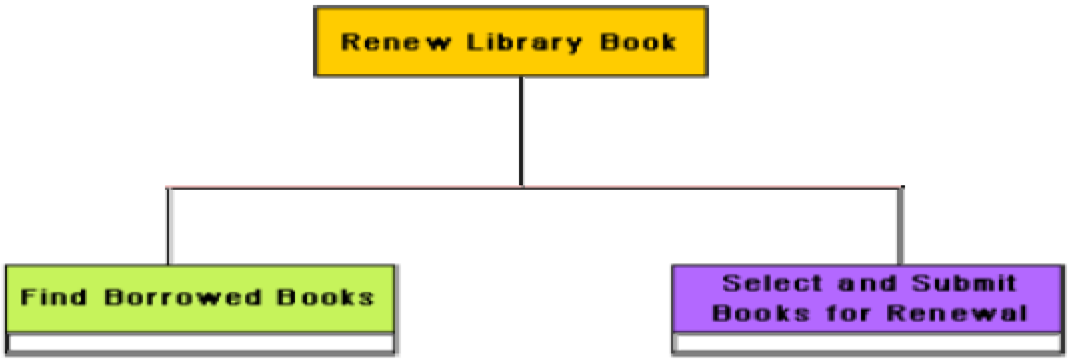
Task and Object Modeling:

- ➢ A task is a human activity intended to achieve some goals. Example of task goals can be:

- reserve an airline seat
- buy an item
- transfer money from one account to another

• Book a cargo for transmission to an address

- ➢ A task model is an abstract model of the structure of a task. A task model should show the structure of the subtasks that the user needs to perform to achieve the overall task goal.
- ➢ Each task can be modeled as a hierarchy of subtasks. A task model can be drawn using a graphical notation similar to the activity network model.
- ➢ Tasks can be drawn as boxes with lines showing how a task is broken down into subtasks. An underlined task box would mean that no further decomposition of the task is required.
- ➢ An example decomposition of a task into subtasks is shown in fig.



(Fig: Decomposition of a task into subtasks)

Selecting a metaphor:

- ➢ The first place one should look for while trying to identify the candidate metaphors is the set of parallels to objects, tasks, and terminologies of the use cases.
- ➢ If no obvious metaphors can be found, then the designer can fall back on the metaphors of the physical world of concrete objects.

Example:

We need to develop the interface for the automation shop, where the users can examine the contents of the shop through a web interface and can order them.

Several metaphors are possible for different parts of this problem.

• Different items can be picked up from **racks** and examined. The user can request for the **catalog** associated with the items by clicking on the item.

• Related items can be picked from the **drawers** of an item cabinet.

• The items can be organized in the form of a book, similar to the way information about electronic components is organized in a semiconductor hand book.

Once the users make up their mind about an item they wish to buy, they can put them into a **shopping cart**.

User interface inspection:

Nielson [Niel94] studied common usability problems and built a check list of points which can be easily checked for an interface. The following check list is based on the work of Nielson [Niel94].

Visibility of the system status: The system should as far as possible keep the user informed about the status of the system and what is going on.

Match between the system and the real world: The system should speak the user's language words, phrases, and concepts familiar to that used by the user, rather than using system-oriented terms.

Undoing mistakes: The user should feel that he is in control rather than feeling helpless or to be at the control of the system. An important step toward this is that the users should be able to undo and redo operations.

Consistency: The user should not have to wonder whether different words, concepts, and operations mean the same thing in different situations.

Recognition rather than recall: The user should not have to recall information which was presented in another screen. All data and instructions should be visible on the screen for selection by the user.

Support for multiple skill levels: Provision of accelerations for experienced users allows them to efficiently carry out the actions they frequently require to perform.

Aesthetic and minimalist design: Dialogs should not contain information which are irrelevant and are rarely needed. Every extra unit of information in a dialog competes with the relevant units and diminishes their visibility.

Help and error messages: These should be expressed in plain language (no codes), precisely indicating the problem, and constructively suggesting a solution.

Error prevention: Error possibilities should be minimized. A key principle in this regard is to prevent the user from entering wrong values.

5. CODING

- Good software development organizations normally require their programmers to adhere to some well-defined and standard style of coding called coding standards.
- The purpose of requiring all engineers of an organization to adhere to a standard style of coding is the following:
    - A coding standard gives a uniform appearance to the codes written by different engineers.
    - It enhances code understanding.
    - It encourages good programming practices.

5.1 Coding standards and guidelines:

- ✓ **Rules for limiting the use of global:** These rules list what types of data can be declared global and what cannot.
- ✓ **Contents of the headers preceding codes for different modules:** The information contained in the headers of different modules should be standard for an organization. The exact format in which the header information is organized in the header can also be specified. The following are some standard header data:
    - Name of the module.
    - Date on which the module was created.
    - Author's name.
    - Modification history.
    - Synopsis of the module.
    - Different functions supported, along with their input/output parameters.
    - Global variables accessed/modified by the module.
- ✓ **Naming conventions for global variables, local variables, and constant identifiers:** A possible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
- ✓ **Error return conventions and exception handling mechanisms:** The way error conditions are reported by different functions in a program are handled should be standard within an organization.
- ✓ **Do not use a coding style that is too clever or too difficult to understand:** Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can obscure meaning of the code and hamper understanding. It also makes maintenance difficult.
- ✓ **Avoid obscure side effects:** The side effects of a function call include modification of parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code.
- ✓ **Do not use an identifier for multiple purposes:** Programmers often use the same identifier to denote several temporary entities.
- ✓ **The code should be well-documented:** As a rule of thumb, there must be at least one comment line on the average for every three-source line.

- ✓ **The length of any function should not exceed 10 source lines:** A function that is very lengthy is usually very difficult to understand as it probably carries out many different functions.
- ✓ **Do not use goto statements:** Use of goto statements makes a program unstructured and makes it very difficult to understand.

5.2 Code review:

- Code review for a model is carried out after the module is successfully compiled and the all the syntax errors have been eliminated.
- Code reviews are extremely cost-effective strategies for reduction in coding errors and to produce high quality code.
- Normally, two types of reviews are carried out on the code of a module. These two types' code review techniques are code inspection and code walk through.

Code Walk Through:

- ✓ Code walk through is an informal code analysis technique. In this technique, after a module has been coded, successfully compiled and all syntax errors eliminated.
- ✓ A few members of the development team are given the code few days before the walk through meeting to read and understand code.
- ✓ The main objectives of the walk through are to discover the algorithmic and logical errors in the code. Some of these guidelines are the following:
  - o The team performing code walk through should not be either too big or too small. Ideally, it should consist of between three to seven members.
  - o Discussion should focus on discovery of errors and not on how to fix the discovered errors.
  - o In order to foster cooperation and to avoid the feeling among engineers that they are being evaluated in the code walk through meeting, managers should not attend the walk through meetings.

Code Inspection:

- ✓ In contrast to code walk through, the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming.
- ✓ In other words, during code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk through.
- ✓ Following is a list of some classical programming errors which can be checked:
- During code inspection:
- Use of uninitialized variables.
- Jumps into loops.

- Non terminating loops.
- Incompatible assignments.
- Array indices out of bounds
- Improper storage allocation and deallocation.
- Use of incorrect logical operators or incorrect precedence among operators.
- Improper modification of loop variables.
- Comparison of equally of floating point variables, etc.

5.3 Clean room testing:

- ✓ Clean room testing was pioneered by IBM. This type of testing relies heavily on walk through, inspection, and formal verification. The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler.
- ✓ The clean room approach to software development is based on five characteristics:
- o **Formal specification:** The software to be developed is formally specified. A state-transition model which shows system responses to stimuli is used to express the specification.
- o **Incremental development:** The software is partitioned into increments which are developed and validated separately using the clean room process. These increments are specified, with customer input, at an early stage in the process.
- o **Structured programming:** Only a limited number of control and data abstraction constructs are used. The program development process is process of stepwise refinement of the specification.
- o **Static verification:** The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.
- o **Statistical testing of the system:** The integrated software increment is tested statistically to determine its reliability.

6. SOFTWARE DOCUMENTATION

- ➢ When various kinds of software products are developed then not only the executable files and the source code are developed but also various kinds of documents such as users' manual, software requirements specification (SRS) documents, design documents, test documents, installation manual, etc are also developed as part of any software engineering process.
- ➢ Good documents are very useful and server the following purposes:
  - o Good documents enhance understandability and maintainability of a software product. They reduce the effort and time required for maintenance.
  - o Use documents help the users in effectively using the system.

- Good documents help in effectively handling the manpower turnover problem. Even when an engineer leaves the organization, and a new engineer comes in, he can build up the required knowledge easily.
- Production of good documents helps the manager in effectively tracking the progress of the project. The project manager knows that measurable progress is achieved if a piece of work is done and the required documents have been produced and reviewed.

6.1 Types of software documents:

Different types of software documents can broadly be classified into the following:

- Internal documentation
- External documentation

Internal documentation:

- ✓ It is the code comprehension features provided as part of the source code itself.
- ✓ Internal documentation is provided through appropriate module headers and comments embedded in the source code.
- ✓ It is also provided through the useful variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant identifiers, use of user-defined data types, etc.

External documentation:

- ✓ External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test documents, etc.
- ✓ A systematic software development style ensures that all these documents are produced in an orderly fashion.

7. TESTING

- ✓ Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected.
- ✓ The aim of the testing process is to identify all defects existing in a software product.

Testing in the large vs. testing in the small:

Software products are normally tested first at the individual component (or unit) level. This is referred to as testing in the small. After testing all the components individually, the components are slowly integrated and tested at each level of integration (integration testing). Finally, the fully integrated system is tested (called system testing). Integration and system testing are known as testing in the large.

7.1 Unit testing:

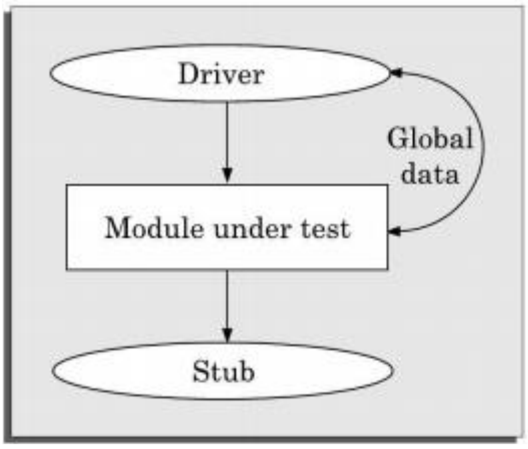
Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation.

In order to test a single module, a complete environment is needed to provide all that is necessary for execution of the module. That is, besides the module under test itself, the following steps are needed in order to be able to test the module:

• The procedures belonging to other modules that the module under test calls.

• Nonlocal data structures that the module accesses.

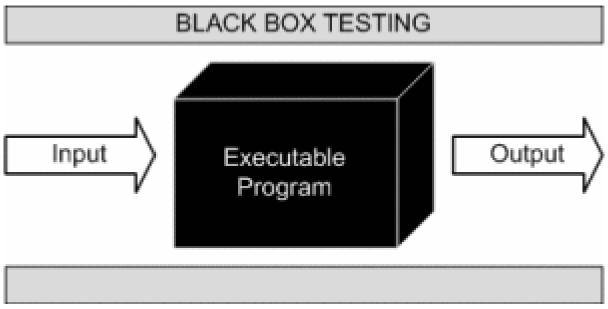
• A procedure to call the functions of the module under test with appropriate parameters.



- ✓ Modules required providing the necessary environment (which either call or are called by the module under test) is usually not available until they too have been unit tested; stubs and drivers are designed to provide the complete environment for a module.
- ✓ The role of stub and driver modules is pictorially shown in fig...
- ✓ A stub procedure is a dummy procedure that has the same I/O parameters as the given procedure but has a highly simplified behavior.
- ✓ For example, a stub procedure may produce the expected behavior using a simple table lookup mechanism. A driver module contains the nonlocal data structures accessed by the module under test, and would also have the code to call the different functions of the module with appropriate parameter values.

<u>7.2 Black Box Testing:</u>

Black Box Testing, also known as Behavioral Testing, is a software testing method which is used to test the software without knowing the internal structure of code or program.



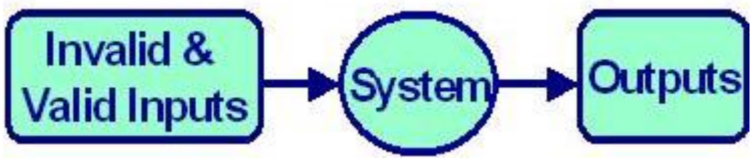
This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see. This method attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors

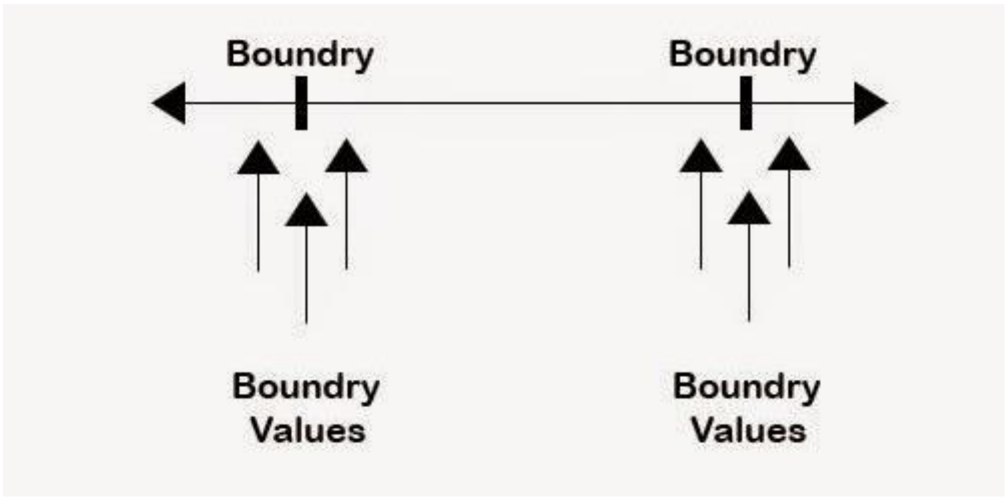
Black box testing techniques:

Following are some techniques that can be used for designing black box tests:

- **Equivalence partitioning**: it is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.

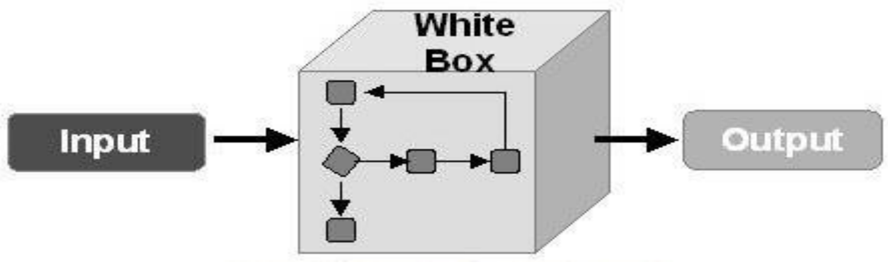


- **Boundary value analysis**: it is a software test design technique that involves determination of boundaries for input values and selecting values that are at the boundaries and just inside/ outside of the boundaries as test data.



7.3 White Box Testing:

White Box Testing (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.



Techniques used in White box Testing:

1. Statement Coverage - This technique is aimed at exercising all programming statements with minimal tests. Hence “Statement Coverage”, as the name suggests, is the method of validating that each and every line of code is executed at least once.

2. Branch Coverage - This technique is running a series of tests to ensure that all branches are tested at least once. “Branch” in programming language is like the “IF statements”. If statement has two branches: true and false. So in Branch coverage (also called Decision coverage), we validate that each branch is executed at least once.

In case of a “IF statement”, there will be two test conditions:

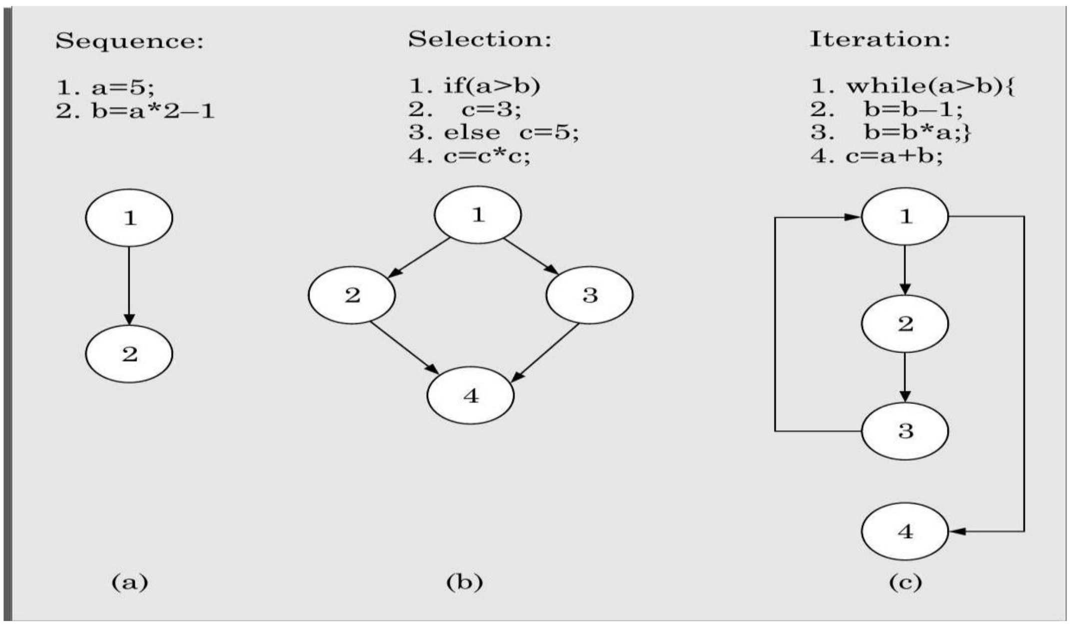
- ✓ One to validate the true branch and
- ✓ Other to validate the false branch

Hence in theory, Branch Coverage is a testing method which when executed ensures that each branch from each decision point is executed.

3. Path Coverage - This technique corresponds to testing all possible paths which means that each statement and branch is covered. Path coverage tests all the paths of the program. This technique is useful for testing the complex programs.

Control Flow Graph:

Control flow graph describes how the sequence in which the different instructions of a program get executed.



Linearly independent set of path:

A set of paths is linearly independent set of path, if each path in the set introduces at least one new edge that is not included in any other path in the set.

Cyclomatic complexity:

- ✓ Cyclomatic complexity is a source code complexity measurement that is being correlated to a number of coding errors. It is calculated by developing a Control Flow Graph of the code that measures the number of linearly-independent paths through a program module.
- ✓ Lower the Program's cyclomatic complexity, lower the risk to modify and easier to understand. It can be represented using the below formula:

Cyclomatic complexity = E - N + P

Where,

E = number of edges in the flow graph.

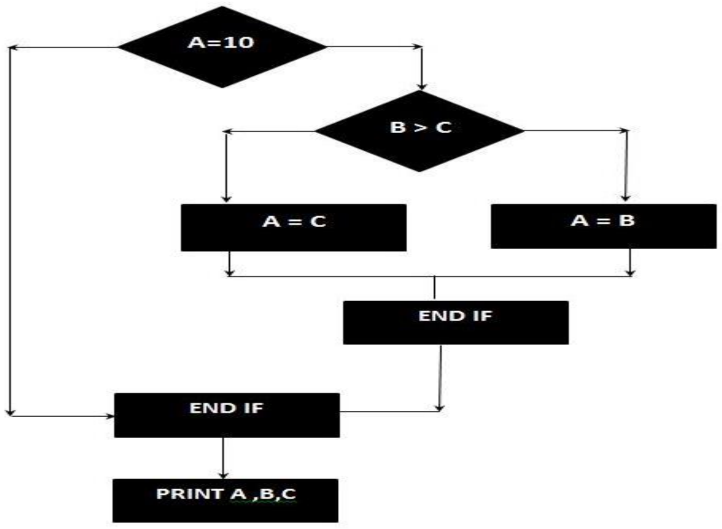
N = number of nodes in the flow graph.

P = number of nodes that have exit points

Example:

```
IF A = 10 THEN
IF B > C THEN
A = B
ELSE
A = C
ENDIF
ENDIF
Print A
Print B
```

Print C



- ✓ The Cyclomatic complexity is calculated using the above control flow diagram that shows seven nodes(shapes) and eight edges (lines), hence the cyclomatic complexity is 8 - 7 + 2 = 3

4. Path Testing:

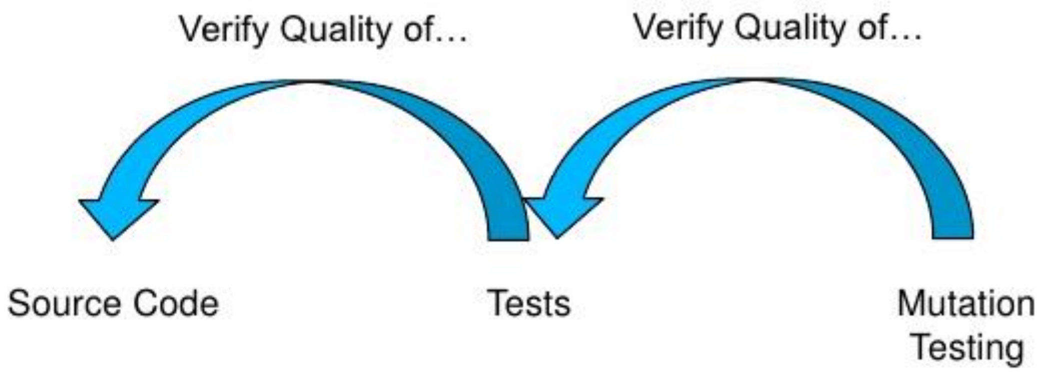
Path Testing is a structural testing method based on the source code or algorithm and NOT based on the specifications.

Path Testing Techniques:

- ✓ **Control Flow Graph (CFG)** - The Program is converted into Flow graphs by representing the code into nodes, regions and edges.
- ✓ **Decision to Decision path (D-D)** - The CFG can be broken into various Decision to Decision paths and then collapsed into individual nodes.
- ✓ **Independent (basis) paths** - Independent path is a path through a DD-path graph which cannot be reproduced from other paths by other methods.

5. Mutation Testing:

Mutation testing (or Mutation analysis or Program mutation) is used to design new software tests and evaluate the quality of existing software tests. Mutation testing involves modifying a program in small ways.



Mutation Testing Types:

- **Value Mutations:** An attempt to change the values to detect errors in the programs. We usually change one value to a much larger value or one value to a much smaller value. The most common strategy is to change the constants.
- **Decision Mutations:** The decisions/conditions are changed to check for the design errors. Typically, one changes the arithmetic operators to locate the defects and also we can consider mutating all relational operators and logical operators (AND, OR , NOT)
- **Statement Mutations:** Changes done to the statements by deleting or duplicating the line which might arise when a developer is copy pasting the code from somewhere else.

8. DEBUGGING

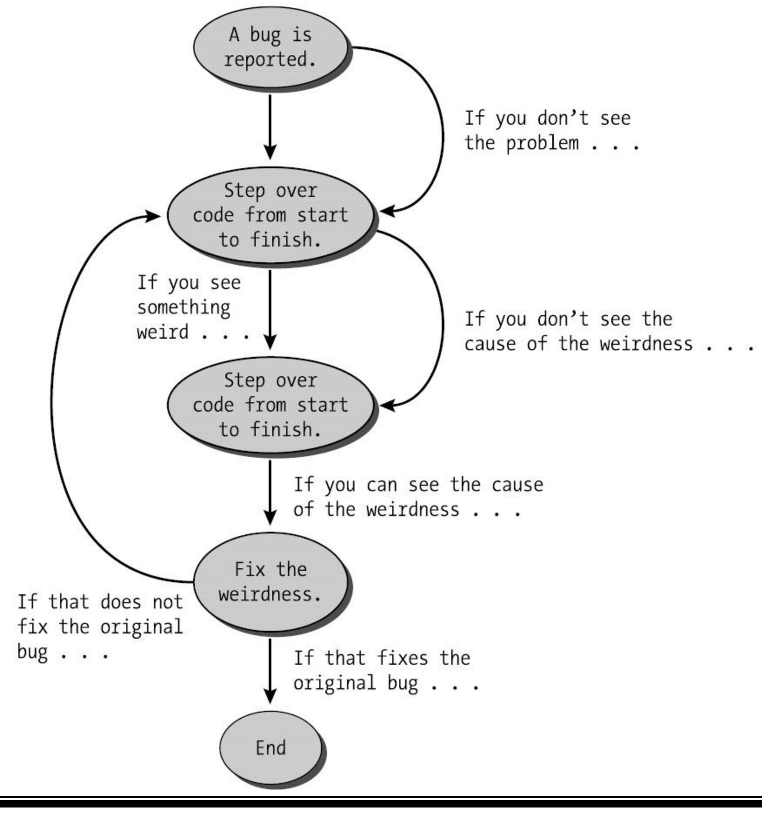
Once errors are identified in a program code, it is necessary to first identify the precise program statements responsible for the errors and then to fix them. Identifying errors in a program code and then fix those up are known as debugging.

Debugging approaches:

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

(i) Brute Force Method:

- This is the most common method of debugging but is the least efficient method.
- In this approach, the program is loaded with print statements to print the intermediate values with the hope that some of the printed values will help to identify the statement in error.
- This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger).


A bug is reported.
If you don't see the problem . . .
Step over code from start to finish.
If you see something weird . . .
If you don't see the cause of the weirdness . . .
Step over code from start to finish.
If you can see the cause of the weirdness . . .
Fix the weirdness.
If that does not fix the original bug . . .
If that fixes the original bug . . .
End

(ii) Backtracking:

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

(iii) Cause Elimination Method:

- In this approach, a list of causes which could possibly have contributed to the error symptom is developed and tests are conducted to eliminate each.
- A related technique of identification of the error from the error symptom is the software fault tree analysis.

Debugging guidelines:

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

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

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

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

9. PROGRAM ANALYSIS TOOLS

A program analysis tool means an automated tool that takes the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, etc.

To classify these into two broad categories of program analysis tools:

• Static Analysis tools

• Dynamic Analysis tools

Static program analysis tools:

Static analysis tool is also a program analysis tool. It assesses and computes various characteristics of a software product without executing it. Typically, static analysis tools analyze some structural representation of a program to arrive at certain analytical conclusions, e.g. that some structural properties hold. The structural properties that are usually analyzed are:

  - Whether the coding standards have been adhered to?
  - Certain programming errors such as uninitialized variables and mismatch between actual and formal parameters, variables that are declared but never used are also checked.

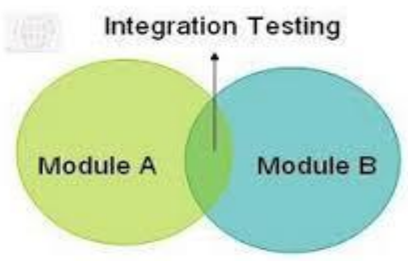
Code walk through and code inspections might be considered as static analysis methods. But, the term static program analysis is used to denote automated analysis tools. So, a compiler can be considered to be a static program analysis tool.

Dynamic program analysis tools:

- Dynamic program analysis techniques require the program to be executed and its actual behavior recorded.
- A dynamic analyzer usually instruments the code (i.e. adds additional statements in the source code to collect program execution traces).
- The instrumented code when executed allows us to record the behavior of the software for different test cases.
- After the software has been tested with its full test suite and its behavior recorded, the dynamic analysis tool caries out a post execution analysis and produces reports which describe the structural coverage that has been achieved by the complete test suite for the program.
- For example, the post execution dynamic analysis report might provide data on extent statement, branch and path coverage achieved.
- Dynamic analysis results can help to eliminate redundant test cases from the test suite.

10. INTEGRATION TESTING

- The primary objective of integration testing is to test the module interfaces, i.e. there are no errors in the parameter passing, when one module invokes another module.
- During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realize the full system.



10.1 Integration test approaches:

There are four types of integration testing approaches. Any one (or a mixture) of the following approaches can be used to develop the integration test plan. Those approaches are the following:

- ✓ Big bang approach
- ✓ Top-down approach
- ✓ Bottom-up approach
- ✓ Mixed-approach

Big-Bang Integration Testing:

- It is the simplest integration testing approach, where all the modules making up a system are integrated in a single step.
- In simple words, all the modules of the system are simply put together and tested. However, this technique is practicable only for very small systems.
- The main problem with this approach is that once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated.
- Therefore, debugging errors reported during big bang integration testing are very expensive to fix.

Bottom-Up Integration Testing:

- In bottom-up testing, each subsystem is tested separately and then the full system is tested.
- A subsystem might consist of many modules which communicate among each other through well-defined interfaces.

Top-Down Integration Testing:

- Top-down integration testing starts with the main routine and one or two subordinate routines in the system.
- After the top-level 'skeleton' has been tested, the immediately subroutines of the 'skeleton' are combined with it and tested.
- Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test.

Mixed Integration Testing:

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

- In the mixed testing approaches, testing can start as and when modules become available. Therefore, this is one of the most commonly used integration testing approaches.

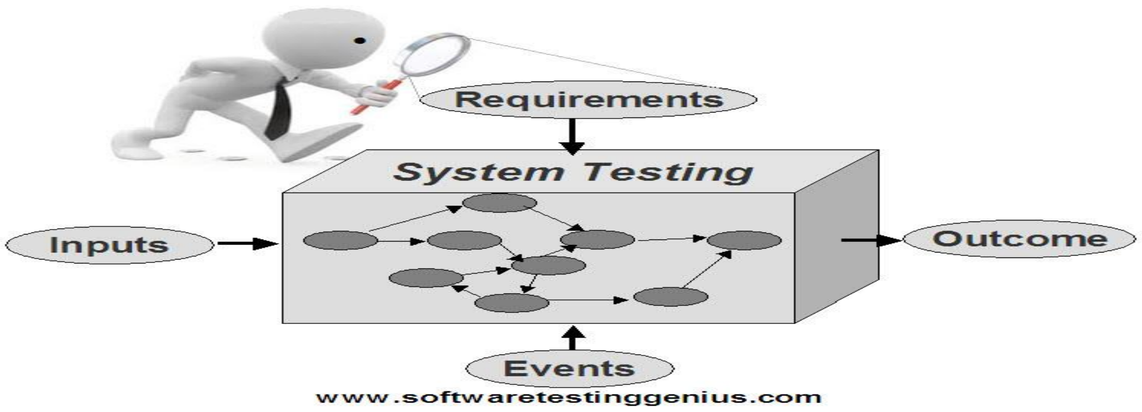
10.2 Strategies of Integration Testing:

The different integration testing strategies are either phased or incremental. A comparison of these two strategies is as follows:

- ❖ In incremental integration testing, only one new module is added to the partial system each time.
- ❖ In phased integration, a group of related modules are added to the partial system each time.

11. SYSTEM TESTING

System tests are designed to validate a fully developed system to assure that it meets its requirements.



There are essentially three main kinds of system testing:

• **Alpha Testing.** Alpha testing refers to the system testing carried out by the test team within the developing organization.

• **Beta testing.** Beta testing is the system testing performed by a select group of friendly customers.

• **Acceptance Testing.** Acceptance testing is the system testing performed by the customer to determine whether he should accept the delivery of the system.

11.1 Performance testing:

Performance testing is carried out to check whether the system needs the nonfunctional requirements identified in the SRS document. There are several types of performance testing. Among of them nine types are discussed below.

• Stress testing

• Volume testing

• Configuration testing

• Compatibility testing

- Regression testing
- Recovery testing
- Maintenance testing
- Documentation testing
- Usability testing

Stress Testing:

Stress testing is also known as endurance testing. Stress testing evaluates system performance when it is stressed for short periods of time. Stress tests are black box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software. Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity.

Volume Testing:

It is especially important to check whether the data structures (arrays, queues, stacks, etc.) have been designed to successfully extraordinary situations.

Compatibility Testing:

This type of testing is required when the system interfaces with other types of systems. Compatibility aims to check whether the interface functions perform as required.

Configuration Testing:

This is used to analyze system behavior in various hardware and software configurations specified in the requirements. Sometimes systems are built in variable configurations for different users.

Regression Testing:

This type of testing is required when the system being tested is an upgradation of an already existing system to fix some bugs or enhance functionality, performance, etc.

Recovery Testing:

Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc. The system is subjected to the loss of the mentioned resources (as applicable and discussed in the SRS document) and it is checked if the system recovers satisfactorily.

Maintenance Testing:

This testing addresses the diagnostic programs, and other procedures that are required to be developed to help maintenance of the system. It is verified that the artifacts exist and they perform properly.

Documentation Testing:

It is checked that the required user manual, maintenance manuals, and technical manuals exist and are consistent. If the requirements specify the types of audience for which a specific manual should be designed, then the manual is checked for compliance.

Usability Testing:

Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface. During usability testing, the display screens, report formats, and other aspects relating to the user interface requirements are tested.

12. TESTING OBJECT-ORIENTED PROGRAMS

- Testing object-oriented programs is much more difficult and required much more cost and effort as compared to testing similar procedural programs.
- Additional test cases are needed to be designed to detect bugs.
- Basic issues in testing object-oriented programs
    - What is a Suitable Unit for Testing Object-oriented Programs?
    - Do various Object-orientation features make testing easy?
    - Why are traditional techniques considered not satisfactory for testing object-oriented programs?
    - Grey-Box testing of objects oriented programs?
- What is a Suitable Unit for Testing Object-oriented Programs?
    - Do various Object-orientation features make testing easy?
    - Why are traditional techniques considered not satisfactory for testing object-oriented programs?
    - Grey-Box testing of objects oriented programs?

Grey-Box Testing of Object-oriented programs:

- For object-oriented programs, several types of test cases can be designed based on the design models of object-oriented programs. These are called the grey-box test cases.
- Types of grey-box testing based on UML models:

✓ State-model-based testing

✓ Use case-based testing

✓ Class diagram-based testing

✓ Sequence diagram-based testing

State-model based testing:

✓ State coverage : Each method of an object are tested at each state of the object

✓ State transition coverage: It is tested whether all transitions depicted in the state model work satisfactorily

✓ State transition path coverage: All transition paths in the state model are tested.

Use case based testing:

- ✓ Scenario coverage: Each use case consists of a mainline scenario and several alternate scenarios.
- ✓ For each use case, the mainline and all alternate sequences are tested to check if any errors show up

Class diagram-based testing:

- ✓ Testing derived classes: All derived classes of the base class have to be instantiated and tested.
- ✓ Association testing: All association relations are tested
- ✓ Aggregation testing: Various aggregate objects are created and tested.

Sequence diagram-based testing:

- ✓ Method coverage: All methods depicted in the sequence diagrams are covered.
- ✓ Message path coverage: All message paths that can be constructed form the sequence diagrams are covered.

Integration Testing of Object-oriented Programs:

Two main approaches:

- ❖ Thread-based
- ❖ Use based

Thread-based approach:

- ✓ In this approach, all classes that need to collaborate to realize the behavior of a single use case are integrated and tested.
- ✓ After all the required classes for a use case are integrated and tested, another use case is taken up and other classes necessary for execution of the second use case to run are integrated and tested.
- ✓ This is continued till all use cases have been considered.

Use-based approach:

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

13. ISSUES IN TESTING

- **Test planning and scheduling problems** often occur when there is no separate test plan, but rather highly incomplete and superficial summaries in other planning documents. Test plans are often ignored once they are written, and test case descriptions are often mistaken for overall test plans. The schedule of testing is often inadequate for the amount of testing that should be performed, especially when testing is primarily manual.

- **Stakeholder involvement and commitment problems** include having the wrong testing mindset (that the purpose of testing is to show that the software works instead of finding defects), having unrealistic testing expectations (that testing will find all of the significant defects), and having stakeholders who are inadequate committed to and supporting of the testing effort.
- **Management-related testing problems** involve the impact of inadequate management. For example, management can fail to supply adequate test resources or place inappropriate external pressures one testing. There may be inadequate test-related risk management or test metrics. Testing lessons learned are far too often ignored, so the same problems are repeated project after project.
- **Test organizational and professionalism problems** include a lack of independence, unclear testing responsibilities, and inadequate testing expertise.
- **Test process problems** often occur when testing and engineering processes are poorly integrated. Organizations sometimes take a "one-size-fits-all" approach taken to testing, regardless of the specific needs of the project. Testing may not be adequately prioritized so that functional testing, black-box system testing, or white-box unit and integration testing may be overemphasized. Testing of components, subsystems, or the system may begin before they are sufficiently mature for testing. Other problems include inadequate test evaluations and inadequate test maintenance.
- **Test tools and environments problems** include an over-reliance on manual testing or COTS testing tools. Often, there are an insufficient number of test environments. Some of the test environments may also have poor quality (excessive defects) or insufficient fidelity to the actual system being tested. Moreover, the system and software under test may behave differently during testing than during operation. Other common problems are that tests were not delivered or the test software, test data, and test environments were not under sufficient configuration control.
- **Test communication problems** primarily involve inadequate test documentation. These types of problems often occur when test documents are not maintained or inadequate communication concerning testing is taking place.
- **Requirements-related testing problems** are related to the requirements that should be driving testing. Often, the requirements are ambiguous, missing, incomplete, incorrect, or unstable. Lower-level requirements may be improperly derived from their higher-level

sources. Likewise, verification methods may be unspecified and the tracing between requirements and tests may be lacking.

2 MARKS

1. Identify five desirable characteristics of a user interface.

- Speed of learning.
- Speed of use
- Speed of recall
- Error prevention
- Attractiveness
- Consistency
- Feedback
- Support for multiple skill levels
- Error recovery (undo facility)
- User guidance and on-line help

2. Differentiate between user guidance and online help system.

- ✓ Online help system help users to know about the operation of the software any time while using the software
- ✓ The guidance messages prompt the user regarding the options he has regarding the next command, and the status of the last command, etc.

3. Differentiate between a mode-based interface and the modeless interface.

- ✓ A modeless interface has only a single mode and all the commands are available all the time during the operation of software
- ✓ In a mode-based interface, different sets of commands can be invoked depending on the mode in which the system is.

4. What are the User interface Types?

- Command language based interfaces
- Menu-based interfaces
- Direct manipulation interfaces

5. What are the types of menus?

- ✓ Scrolling menu - enable the user to view and select the menu items that cannot be accommodated on the screen
- ✓ Walking menu -structure a large collection of menu items.
- ✓ Hierarchical menu - menu items are organized in a hierarchy or tree structure

6. Compare Graphical User Interface (GUI) with text-based user interface?

Graphical User Interface	Text-based User Interface
A. Multiple windows with different information can be simultaneously be displayed on the screen	Multiple windows with different information cannot be simultaneously be displayed on the screen
B. Iconic information representation and symbolic information manipulation is possible	Iconic information representation and symbolic information manipulation is not possible
C. Supports command selection using an attractive and user friendly menu selection system.	It is not attractive one
D. Graphics terminal with graphics capability is **Expensive**	A **cheap** alphanumeric display terminal

7. Define iconic interface

A Direct manipulation interfaces present the interface to the user in the form of visual models.

8. **What do you mean by component-based GUI development style?**

- A development style based on widgets (window objects) is called component-based (or widget-based) GUI development style.
- Use widgets as building blocks are because they help users learn an interface fast.

9. Give the necessity of component-based GUI development.

Component-based GUI development recognizes that every user interface can easily be built from a handful of predefined components such as menus, dialog boxes, forms, etc.

10. What is window in terms of GUI?

- A window is a rectangular area on the screen. A window can be considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent activities, e.g. one window can be used for editing a program and another for drawing pictures, etc
- A window can be divided into two parts: client part, and non-client part.

11. Identify the responsibilities of a window manager.

The window manager is responsible for managing and maintaining the non-client area of a window. Window manager manages the real-estate policy, provides look and feel of each individual window.

12. Identify at least eight primary types of window objects.

Label widget, Container widget, Pop-up menu, Pull-down menu, Dialog boxes, Push button, Radio buttons, and Combo boxes.

13. What is meant by window management system?

- A window management system is a resource manager. It keeps track of the screen area resource and allocates it to the different windows that seek to use the screen. It simplifies the task of a GUI designer.
- A WMS consists of two parts a window manager, and a window system.

14. What is X Window?

- A windowing system, which is used to provide the basic framework for a GUI environment: drawing and moving windows on the display device and interacting with a mouse and keyboard.
- The X-window functions are low-level functions written in C language which can be called from application programs.

15. Differentiate between a user-centered design and a design by users.

- ✓ User-centered design is the **theme of almost all modern user interface design techniques**. However, user-centered design does not mean design by users. One should not get the users to design the interface, nor should one assume that the user's opinion of which design alternative is superior is always right.
- ✓ Users have good knowledge of the tasks they have to perform, they also **know whether they find an interface easy to learn** and use but they have less understanding and experience in GUI design than the GUI developers.

16. Explain implications of human cognition capabilities on user interface design.

- ✓ Limited memory
- ✓ Frequent task closure.
- ✓ Recognition rather than recall
- ✓ Procedural versus object-oriented

17. List out seven important steps needed to design a GUI methodology.

 Examine the use case model of the software. Interview, discuss, and review the GUI issues with the end-users.

 Task and object modeling

 Metaphor selection

 Interaction design and rough layout

 Detailed presentation and graphics design

 GUI construction

 Usability evaluation

18. Write check list for user interface inspection.

 Visibility of the system status

 Match between the system and the real world

 Undoing mistakes

 Consistency.

 Recognition rather than recall

 Support for multiple skill levels

 Aesthetic and minimalist design.

 Help and error messages

 Error prevention

19. What is code review?

- ✓ A cost-effective strategy, which is used for reduction in coding errors and to produce high quality code.
- ✓ Types:

  a. code inspection

  b. code walk through

20. Define Code Inspection

A technique, which is used to discover some common types of errors caused due to oversight and improper programming.

21. Define Code walk through

An informal code analysis technique, which is used to eliminate successfully compiled and all syntax errors.

22. List out some classical programming errors during code inspection.

- Use of uninitialized variables.
- Jumps into loops.
- Non-terminating loops.

- Incompatible assignments.
- Array indices out of bounds.
- Improper storage allocation and de-allocation
- Mismatches between actual and formal parameter in procedure calls.
- Use of incorrect logical operators or incorrect precedence among operators.
- Improper modification of loop variables.
- Comparison of equally of floating point variables, etc

23. What is clean room testing?

A rigorous inspection process to test the code by other person (not programmer) to achieve zero-defect software.

24. What are the characteristic of clean room testing?

- ✓ Formal specification
- ✓ Incremental development
- ✓ Structured programming
- ✓ Statistical testing of the system
- ✓ Static verification

25. Differentiate between internal documentation and external documentation.

Internal documentation is the one in which various information regarding the program is enlisted in the program itself i.e. in the form of comments. On the contrary, external documentation is the one that is prepared separately to inform the users about the system.

26. What is Program testing?

- Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected.
- If the program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction

27. Explain the aim of testing.

The aim of the testing process is to identify all defects existing in a software product.

28. Differentiate between verification and validation.

Verification	**Validation**
1. Verification is a static practice of verifying documents, design, code and program.	1. Validation is a dynamic mechanism of validating and testing the actual product.
2. It does not involve executing the code.	2. It always involves executing the code.
3. It is human based checking of documents and files.	3. It is computer based execution of program.

4. Verification uses methods like inspections, reviews, walkthroughs, and Desk-checking etc.	4. Validation uses methods like black box (functional) testing, gray box testing, and white box (structural) testing etc.
5. **Verification** is to check whether the software conforms to specifications.	5. **Validation** is to check whether software meets the customer expectations and requirements.
6. It can catch errors that validation cannot catch. It is low level exercise.	6. It can catch errors that verification cannot catch. It is High Level Exercise.
7. Target is requirements specification, application and software architecture, high level, complete design, and database design etc.	7. Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.
8. Verification is done by QA team to ensure that the software is as per the specifications in the SRS document.	8. Validation is carried out with the involvement of testing team.
9. It generally comes first-done before validation.	9. It generally follows after **verification**.

29. Define failure.

A manifestation of an error (or defect or bug). But, the mere presence of an error may not necessarily lead to a failure.

30. What is Test case?

This is the triplet [I,S,O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system

31. Define Test suite?

The set of all test cases with which a given software product is to be tested.

32. Differentiate between functional testing (black-box testing) and structural testing(white-box testing).

- ✓ Test cases are designed using only the functional specification of the software that is without any knowledge of the internal structure of the software called black-box testing (functional testing).
- ✓ Designing test cases requires thorough knowledge about the internal structure of software called white-box testing (structural testing).

33. Differentiate between testing in the large and testing in the small.

- Software products are normally tested first at the individual component (or unit) level. This is referred to as testing in the small.

- After testing all the components individually, the components are slowly integrated and tested at each level of integration (integration testing).
- Finally, the fully integrated system is tested (called system testing). Integration and system testing are known as testing in the large.

34. What is unit testing?

Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation

35. What is black box testing?

In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required.

36. Explain mutation testing.

In mutation testing, the software is first tested by using an initial test suite built up from the different white box testing strategies. After the initial testing is complete, mutation testing is taken up. The idea behind mutation testing is to make few arbitrary changes to a program at a time.

37. What is Debugging?

Debugging is the process of locating and fixing or bypassing bugs (errors) in computer program code or the engineering of a hardware device. To debug a program or hardware device is to start with a problem, isolate the source of the problem, and then fix it.

38. What are the three approaches of debugging?

- ✓ Brute Force Method
- ✓ Backtracking
- ✓ Cause Elimination Method

39. What is meant by a program analysis tool?

A program analysis tool means an automated tool that takes the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, etc.

40. What are the categories of program analysis tool?

The program analysis tool is categorized into

- ✓ Static Analysis tools
- ✓ Dynamic Analysis tools

41. What is Static analysis tools (Kiviat Chart)?

Tools, which are used to summarize the results of analysis of every function in a polar chart. It shows the analyzed values for cyclomatic complexity, number of source lines, percentage of comment lines, etc..

42. What is Dynamic Analysis Tools?

Tools, which are used to evaluate several program characteristics based on analysis of the run-time behavior of a program. It record and analyze the actual behavior of a program while it is being executed.

43. What is Integration testing?

In integration testing, individual software modules are combined and tested as a group. It is to test the module interfaces, i.e. there are no errors in the parameter passing, when one module invokes another module. Different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realize the full system.

44. What are the approaches of Integration testing?

The Integration test approaches are

- Big bang approach
- Top-down approach
- Bottom-up approach
- Mixed-approach

45. What is Big-bang integration testing?

A integration testing, in which all the modules makeup a system that are integrated in a single step.

46. What is Bottom-up integration testing?

Large software products are often made up of several subsystem. A Subsystems consist of many modules which communicate among each other through well-defined interface.

47. What is Top-down integration testing?

Top-down integration testing starts with the root module and one or two subordinate modules in the system.

48. What is mixed integration testing?

The mixed(sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches.

49. What is the difference between Phased and Incremental Integration Testing?

- ✓ In incremental integration testing, only one new module is added to the partially integrated system each time.
- ✓ In phased integration, a group of related modules are added to the partial system each time.

50. What is Grey-Box Testing of Object-oriented programs?

For object-oriented programs, several types of test cases can be designed based on the design models of object-oriented programs. These are called the grey-box test cases

51. List out types of grey-box testing based on UML models?

a. State-model-based testing

b. Use case-based testing

c. Class diagram-based testing

d. Sequence diagram-based testing

52. Give Integration Testing of Object-oriented Programs approaches.

Two main approaches

i. Thread-based

ii. Use based

53. What is system testing?

System tests are designed to validate a fully developed system to assure that it meets its requirements. The test cases are therefore designed solely based on the SRS document.

54. What are the types of system testing?

The three main kinds of system testing are

• **Alpha Testing.** Alpha testing refers to the system testing carried out by the test team within the developing organization.

• **Beta testing.** Beta testing is the system testing performed by a select group of friendly customers.

• **Acceptance Testing.** Acceptance testing is the system testing per formed by the customer to determine whether he should accept the delivery of the system

55. Define Performance testing

- Performance testing is carried out to check whether the system needs the non-functional requirements identified in the SRS document.
- There are several types of performance testing.
    - Stress testing
    - Volume testing
    - Configuration testing
    - Compatibility testing
    - Regression testing
    - Recovery testing
    - Maintenance testing
    - Documentation testing

    - Usability testing

56. Describe Error seeding

- ✓ Sometimes the customer might specify the maximum number of allowable errors that may be present in the delivered system.
- ✓ These are often expressed in terms of maximum number of allowable errors per line of source code.
- ✓ Error seed can be used to estimate the number of residual errors in a system

57. What are General Issues associated with Testing?

- ✓ How to document the results of testing and how to perform regression testing
- ✓ Test summary Report:
- A piece of documentation produced towards the end of testing.
- It specifies what is the total number of tests that were applied to a subsystem, Out of the total number of tests how many tests were successful, and How many were unsuccessful, and the degree to which they were unsuccessful.
- ✓ Regressing testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix.