UNIT-3

1Q)Analysis Modelling Approaches:

- **Requirements Analysis:**
  It results in specification of software's operational characteristics, indicates software's interface with other system elements and establishes constraints that s/w must meet.
- All elements of analysis model are directly traceable to parts of the design model. A clear division of design and analysis tasks between these two important modeling activities is not always possible.
- The analysis model and requirements specification provide a means for assessing quality once the software is built.

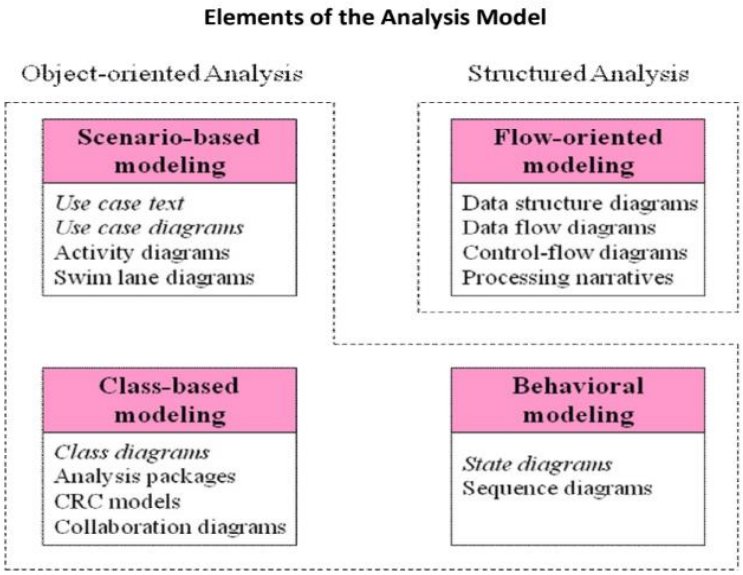
Analysis Modeling Approaches:

1.Structured analysis:

- Considers data and the processes that transform the data as separate entities
- Data is modeled in terms of only attributes and relationships (but no operations)
- Processes are modeled to show the
  1) input data,
  2) the transformation that occurs on that data, and
  3) the resulting output data

2)Object-oriented analysis: Focuses on the definition of classes and the manner in which they collaborate with one another to fulfill customer requirements

- UML and unified process are predominantly object oriented.
- Analysis modeling leads to derivation of each of the modeling elements as shown below:



The specific content of each element (diagrams/models used to construct element)differs from project to project.

1.Structured Analysis:

Flow -Oriented Modeling:

- **Data Modeling:**

Analysis modeling often begins with data modeling. Its concepts are:

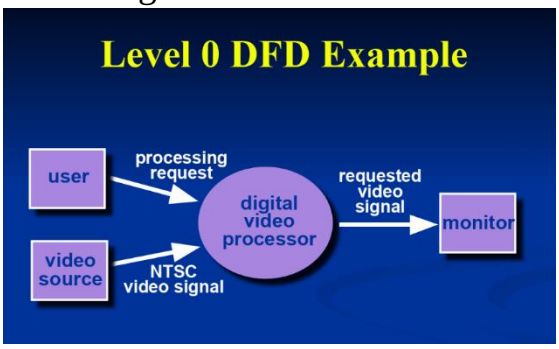
a. Data objects (Entities)
b. Data attributes
c. Relationship
d. Cardinality (number of occurrences)

- **Data Flow and Control Flow**

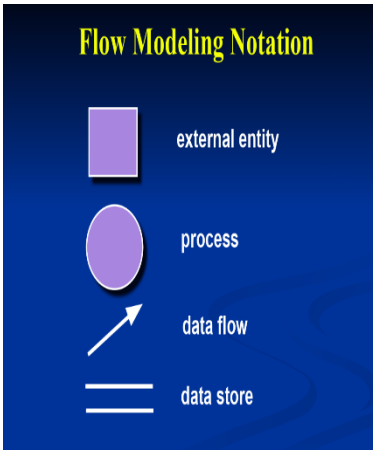
1) **Data Flow Diagram**:

- Data Flow Diagrams(DFD) can be used to complement UML diagrams and provide additional insight into system requirements and data flow.
- DFD takes an input-process-output view of a system ,i.e, data objects flow into the s/w ,transformed by processing elements and resultant data objects flow out of s/w.

- DFD is represented in a hierarchical fashion i.e, the first data flow model (sometimes called a level 0 DFD or context diagram) represents system as a whole. Subsequent DFD's refines context diagram, providing increasing detail with each level.

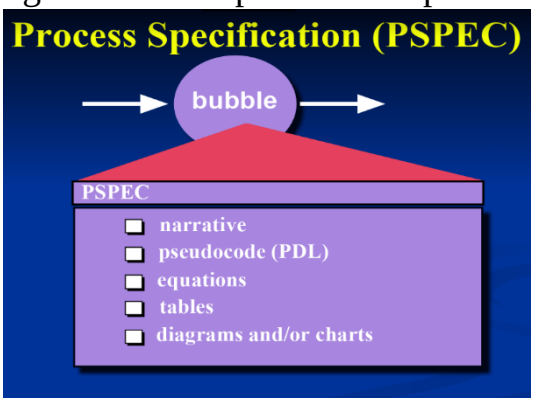


- Creating a Data Flow Model:
  Data Flow Diagram enables the s/w engineer to develop models of information and functional domains at the same time.

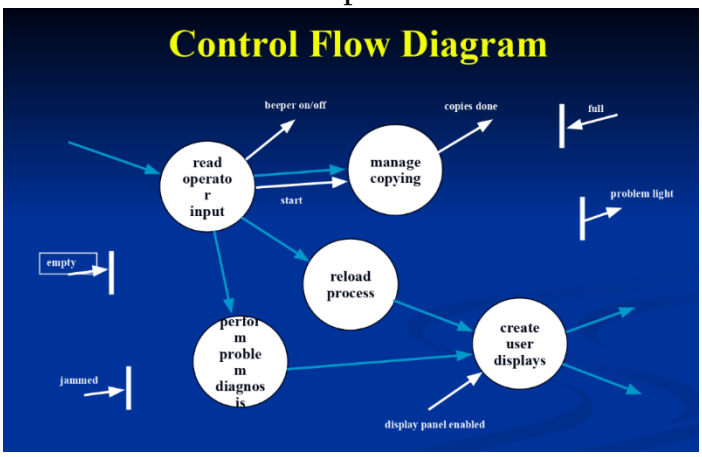


2) **Process Specification**:
   - Describes data flow processing at the lowest level of refinement in the data flow diagrams
   - The process specification is used to describe all flow model processes that appear at final level of refinement.
   - The content of PSPEC include narrative text, a Program Design language(PDL)description of process algorithm, mathematical equations, tables, diagrams or charts.

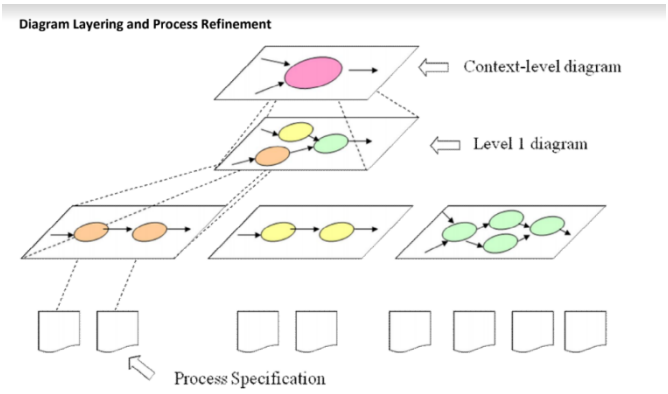
- By providing PSPEC to accompany each bubble in flow model, s/w engineer creates a "mini-spec" that can serve as
  guide for design of s/w component to implement process.



3) **<u>Control Flow Diagram</u>**:
   - Illustrates how events affect the behavior of a system through the use of state diagrams
   - The control flow diagram is "superimposed" on the DFD and shows events that control the processes noted in the DFD



   - Control flows do not physically activate/deactivate the processes. This is done via the CSPEC.
   - <u>Control specification</u>:
     The control specification(CSPEC)represents behavior of the system in two different ways:
     i. It contains a state diagram that is sequential specification of behavior.
     ii. It can also contain a program activation table- a combinational specification of behavior.



2.Object-Oriented Analysis:

- The intent of OOA is to define all classes(and relationships and behavior associated with them)that are relevant to the problem to be solved.
- To accomplish this, a no. of tasks must occur:

1. Basic user requirements must be communicated b/w user &S.E.
2. Classes must be identified(i.e, attributes and methods are defined).
3. A class hierarchy is defined.
4. Object-to-object relation should be represented.
5. Object behavior must be modeled.
6. Repeat tasks 1 to 5 until model is complete.

- OOA builds a class-oriented model that relies on understanding of OO concepts.

i. Scenario-Based Modeling:

- Analysis modeling with UML begins with creation of scenarios in the form of use-cases, activity diagrams and swimlane diagrams.

Writing use cases:

- "A use-case is defined as a set of sequence of actions performed by an actor to obtain a specific output."
- "An actor may be a person that use a system or product, or a system itself, anything that performs an action in system".
- A use-case captures the interactions that occur between producers and consumers of information and system itself
- The concept of a use-case is relatively easy to understand-
  describe a specific usage scenario in straight forward language from point of view of defined actor.

What to write about?

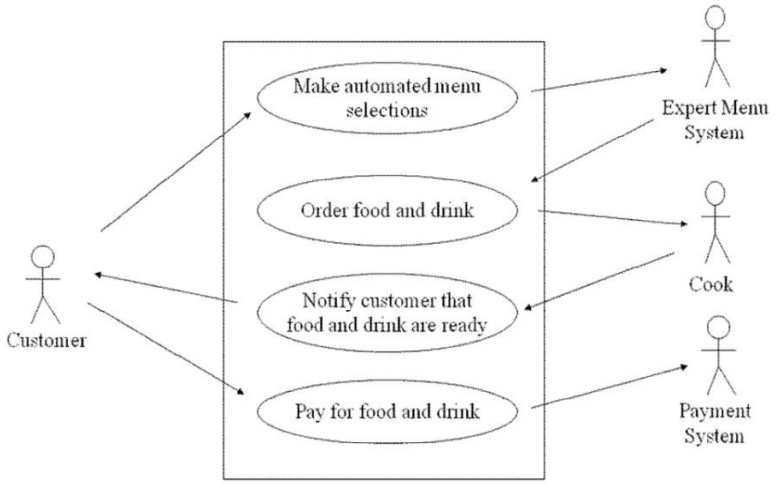
- The first 2 RE tasks-inception and elicitation-provide us the information we need to begin writing use cases.
- To begin developing a set of use-cases, the functions or activities performed by a specific actor are listed.
- As conversations with stakeholder progress, the RE team develops use-cases for each of activities noted.
- A variation of a narrative(formal) use-case presents the interaction as an ordered sequence of user actions. Each action is represented as a declarative sentence
- It is important to note that sequential presentation does not consider any alternative interactions. Such use-cases are referred to as "primary scenarios".
- A description of alternative interactions is essential for complete understanding of the function ,by asking the following questions:[Answers result secondary use-cases].
  1. Can the actor take some other action at that point?
  2. Is it possible that actor will encounter some error at this point? If so, what might it be?
  3. Is it possible that actor will encounter some other behavior? If so, what might it be?

• It is effective to use the first person “I” to describe how the actor interacts with the software

• Format of the text part of a use case:

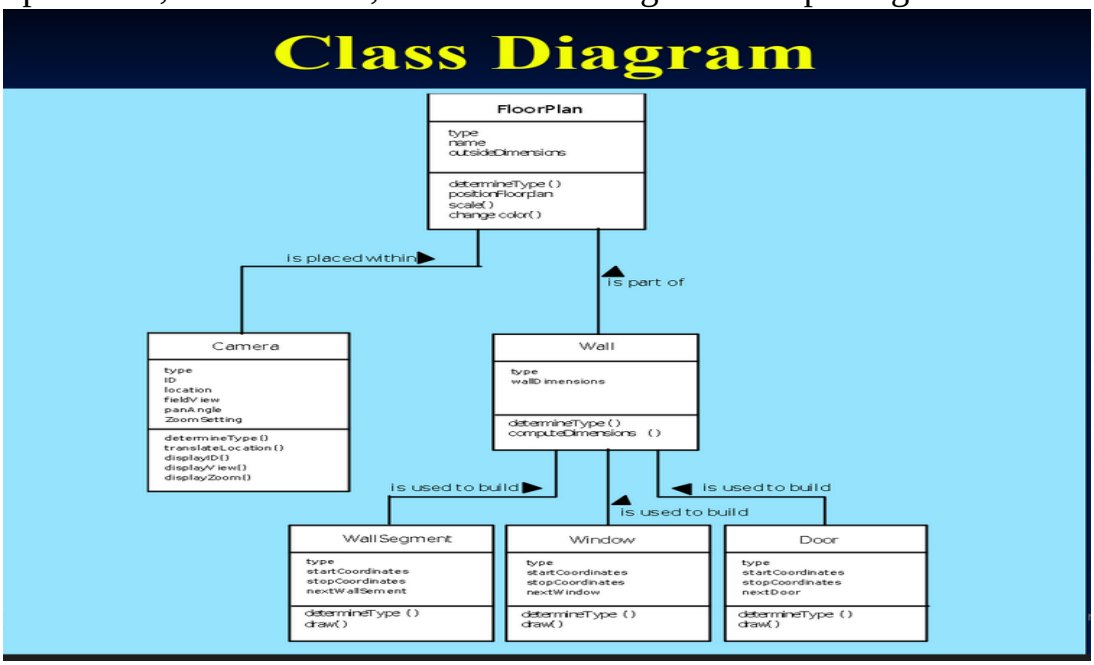
- The figure below is an example of use-case diagram for “Restaurant Management System”:

Example Use Case Diagram



ii. Class-based Modelling:

- Class-based modeling represents:
  - objects that the system will manipulate
  - operations (also called methods or services) that will be applied to the objects to effect the manipulation
  - relationships (some hierarchical) between the objects
  - collaborations that occur between the classes that are defined.
- The elements of a class-based model include classes and objects, attributes, operations, CRC models, collaboration diagrams and packages.



- **Class-Responsibility-Collaborator(CRC)Modeling:**
  - "CRC Modleing provides a simple means for identifying and organizing the classes that are relevant to system or product requirements."
  - The intent of CRC model is to develop an organized representation of classes , using actual or virtual index cards.
  - Responsibilities are the attributes and operations that relevant for the class.
  - Collaborators are those classes that are requiered to provide a class with information needed to complete a responsibility.
  - Types of classes in CRC:

1.Entity class: Contains information important to users. It is also called model or business classes, are executed directly from statement of the problem. These classes represent things that are to be stored in a DB and persist throughout duration of the application.

2.Boundary Classes: Used to create interface that the user sees and interacts with as the s/w is used. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.

3.Controller Classes: Manage a "unit of work" from start to finish. These can be designed to manage,

i. Creation or update of entity objects;

ii. Complex communication b/w sets of objects;

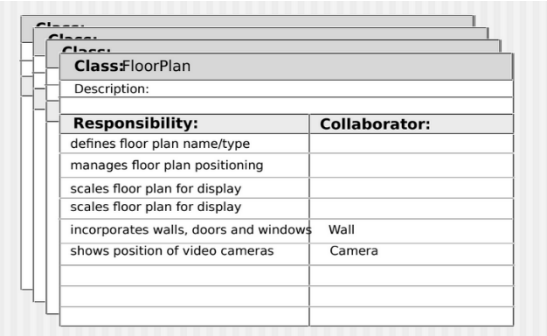
iii.Instantiation of boundary objects ,as they obtain information from entity objects.

iv.Validation of data communicated b/w objects or b/w user &application.

- Controller classes are not considered until design has begun

- **Collaborations**
  - Classes fulfill their responsibilities in one of two ways:
    A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or a class can collaborate with other classes.
  - Collaborations identify relationships between classes

- Collaborations are identified by determining whether a class can fulfill each responsibility itself three different generic relationships between classes :
  (a) the is-part-of relationship
  (b) the has-knowledge-of relationship
  (c) the depends-upon relationship
- The figure below is an example for CRC model.



iii. Behavioral Modeling

The behavioral model indicates how software will respond to external events or stimuli. To create the model, the analyst must perform the following steps:

i. Evaluate all use-cases to fully understand the sequence of interaction within the system.
ii. Identify events that drive the interaction sequence and understand how these events relate to specific objects.
iii. Create a sequence for each use-case.
iv. Build a state diagram for the system.
v. Review the behavioral model to verify accuracy and consistency.

- **Creating a Behavioral Model**

1) Identify events found within the use cases and implied by the attributes in the class diagrams

2) Build a state diagram for each class, and if useful, for the whole software system

- **Identifying Events in Use Cases**

• An event occurs whenever an actor and the system exchange information

• An event is NOT the information that is exchanged, but rather the fact that information has been exchanged

• Some events have an explicit impact on the flow of control, while others do not

– An example is the reading of a data item from the user versus comparing the data item to some possible value

- **Building a State Diagram**

• A state is represented by a rounded rectangle

• A transition (i.e., event) is represented by a labeled arrow leading from one state to another

– Syntax: trigger-signature [guard]/activity

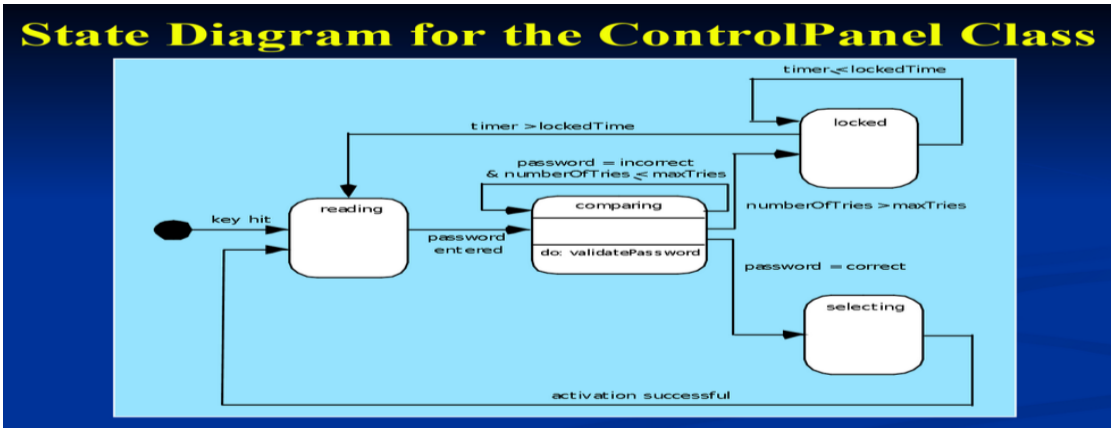
• The active state of an object indicates the current overall status of the object as is goes through transformation or processing

– A state name represents one of the possible active states of an object

• The passive state of an object is the current value of all of an object's attributes

– A guard in a transition may contain the checking of the passive state of an object

- Example State Diagram



------x----x------

2Q) UML Diagrams

UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.

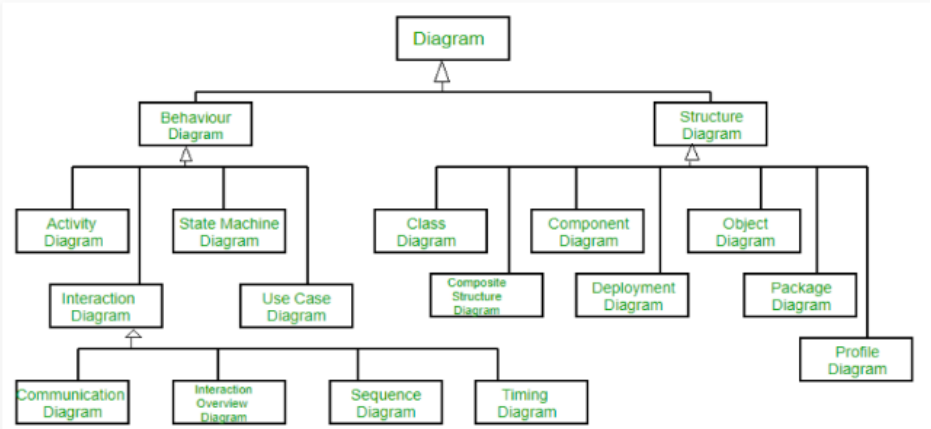
UML includes the following nine diagrams,

- Class diagram
- Object diagram
- Use case diagram
- Sequence diagram
- Collaboration diagram
- Activity diagram
- Statechart diagram
- Deployment diagram
- Component diagram

There are two broad categories of diagrams and they are again divided into subcategories −

- Structural Diagrams
- Behavioral Diagrams

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



i. Structural Diagrams

- The structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram, which forms the main structure and are therefore stable.
- These static parts are represented by classes, interfaces, objects, components, and nodes. The four structural diagrams are −

1) Class diagram
2) Object diagram
3) Component diagram
4) Deployment diagram

ii. Behavioural Diagrams

- Any system can have two aspects, static and dynamic. So, a model is considered as complete when both the aspects are fully covered.
- Behavioral diagrams basically capture the dynamic aspect of a system. Dynamic aspect can be further described as the changing/moving parts of a system.
- UML has the following five types of behavioral diagrams −

1) Use case diagram
2) Sequence diagram
3) Collaboration diagram
4) Statechart diagram

5) Activity diagram

i. Use Case Diagram

- Use case diagrams are a set of use cases, actors, and their relationships. They represent the use case view of a system.
- A use case represents a particular functionality of a system. Hence, use case diagram is used to describe the relationships among the functionalities and their internal/external controllers. These controllers are known as **actors.**

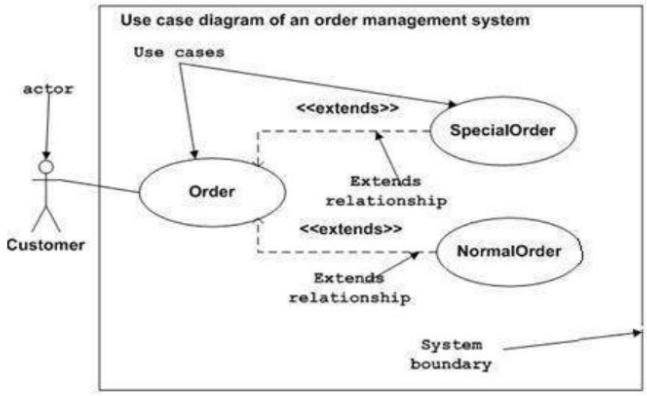
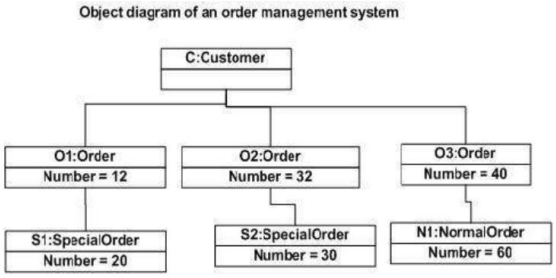


Figure: Sample Use Case diagram

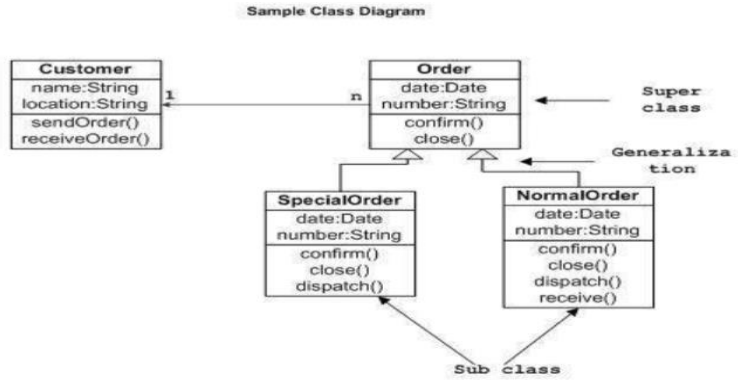
ii. Object Diagram

- Object diagrams can be described as an instance of class diagram. Thus, these diagrams are more close to real-life scenarios where we implement a system.
- Object diagrams are a set of objects and their relationship is just like class diagrams. They also represent the static view of the system.
- The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system from a practical perspective.



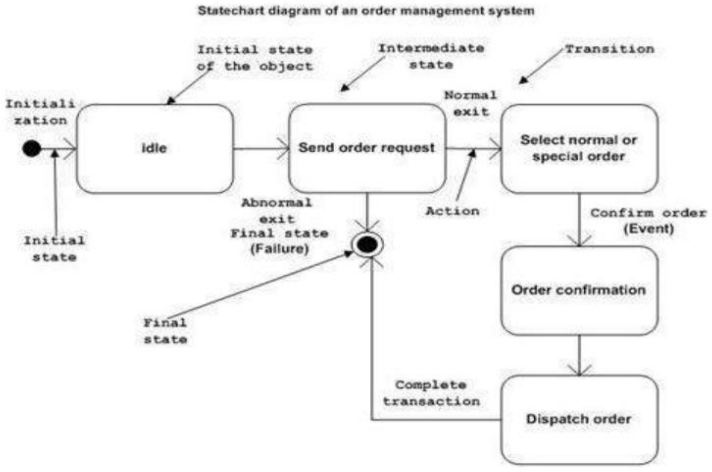
iii. Class Diagram

- Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, associations, and collaboration. Class diagrams basically represent the object-oriented view of a system, which is static in nature.
- Active class is used in a class diagram to represent the concurrency of the system.
- Class diagram represents the object orientation of a system. Hence, it is generally used for development purpose. This is the most widely used diagram at the time of system construction.



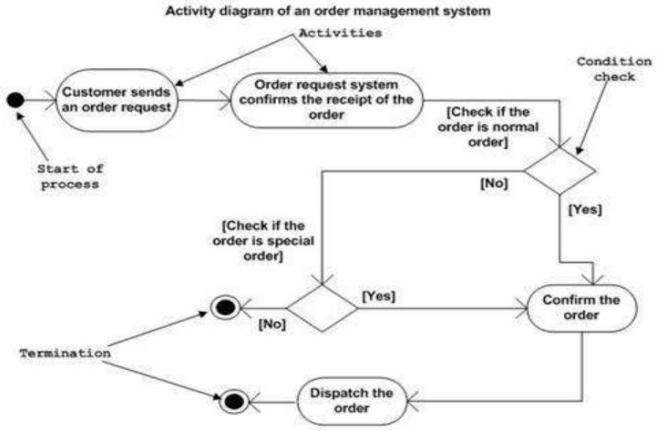
iv. Statechart Diagram

- Any real-time system is expected to be reacted by some kind of internal/external events. These events are responsible for state change of the system.
- Statechart diagram is used to represent the event driven state change of a system. It basically describes the state change of a class, interface, etc.
- State chart diagram is used to visualize the reaction of a system by internal/external factors.



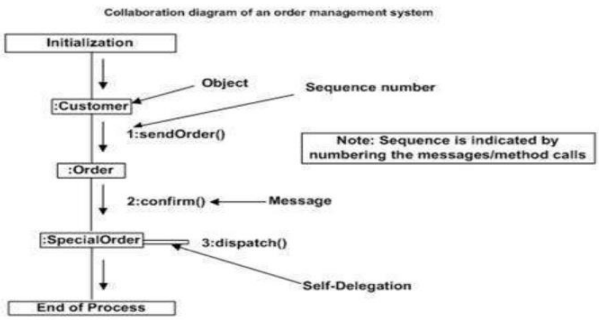
v. Activity Diagram

- Activity diagram describes the flow of control in a system. It consists of activities and links. The flow can be sequential, concurrent, or branched.
- Activities are nothing but the functions of a system. Numbers of activity diagrams are prepared to capture the entire flow in a system.
- Activity diagrams are used to visualize the flow of controls in a system. This is prepared to have an idea of how the system will work when executed.



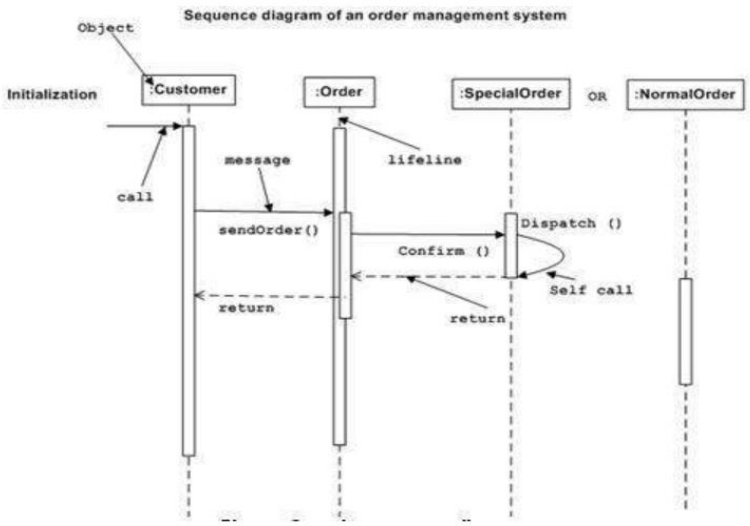
vi. Collaboration Diagram

- Collaboration diagram is another form of interaction diagram. It represents the structural organization of a system and the messages sent/received. Structural organization consists of objects and links.
- The purpose of collaboration diagram is similar to sequence diagram. However, the specific purpose of collaboration diagram is to visualize the organization of objects and their interaction.



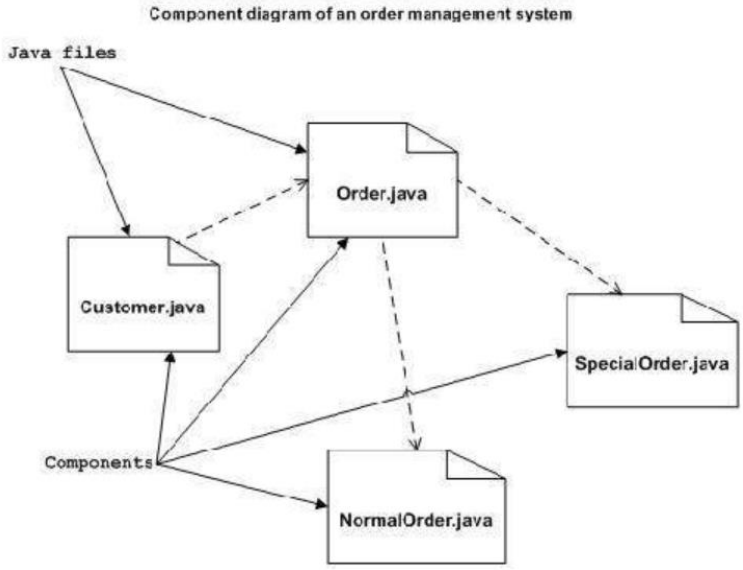
vii. Sequence Diagram

- A sequence diagram is an interaction diagram. From the name, it is clear that the diagram deals with some sequences, which are the sequence of messages flowing from one object to another.
- Interaction among the components of a system is very important from implementation and execution perspective. Sequence diagram is used to visualize the sequence of calls in a system to perform a specific functionality.



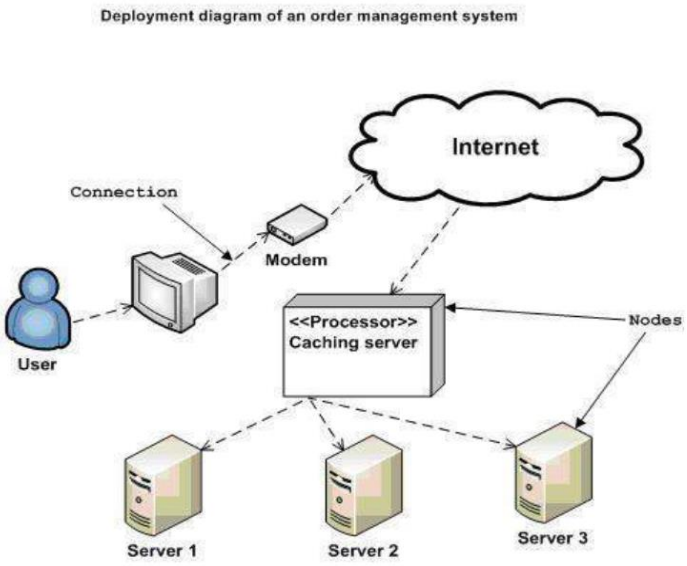
viii. Component Diagram

- Component diagrams represent a set of components and their relationships. These components consist of classes, interfaces, or collaborations. Component diagrams represent the implementation view of a system.
- During the design phase, software artifacts (classes, interfaces, etc.) of a system are arranged in different groups depending upon their relationship. Now, these groups are known as components.
- Finally, it can be said component diagrams are used to visualize the implementation.



ix. Deployment Diagram

- Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed.
- Deployment diagrams are used for visualizing the deployment view of a system. This is generally used by the deployment team.



----------X----------X----------

3Q) Definition, Goals & Purpose of Design. Concepts (Abstraction, Information hiding, Cohesion, Coupling, Refactoring etc...) Characteristics of good design

- **Design Engineering:**

  i. Design engineering encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product.
  ii. Design creates a representation or model of the s/w, but unlike the analysis model (that focuses on describing required data, function and behaviour), the design model provides detail about s/w data structures, architecture, interfaces, and components that are necessary to implement the system.
  iii. Design is the place where s/w quality is established.
  iv. Work product: A design model that encompasses architectural, interface, component-level, and deployment representations is the primary work product that is produced during s/w design.

- **Goals of a Good Design:**

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

- **Purpose of Design:**

• Design is where customer requirements, business needs, and technical considerations all come together in the formulation of a product or system

• The design model provides detail about the software data structures, architecture, interfaces, and components

• The design model can be assessed for quality and be improved before code is generated and tests are conducted

– Does the design contain errors, inconsistencies, or omissions?

– Are there better design alternatives?

– Can the design be implemented within the constraints, schedule, and cost that have been

established?

• A designer must practice diversification and convergence

– The designer selects from design components, component solutions, and knowledge available through catalogs, textbooks, and experience

– The designer then chooses the elements from this collection that meet the requirements defined by requirements engineering and analysis modeling

– Convergence occurs as alternatives are considered and rejected until one particular configuration of components is chosen

• Software design is an iterative process through which requirements are translated into a blueprint for constructing the software

– Design begins at a high level of abstraction that can be directly traced back to the data, functional, and behavioral requirements

– As design iteration occurs, subsequent refinement leads to design representations at much lower levels of abstraction

- **Characteristics of a good design:**

1) A design should exhibit an architecture that

(a) has been created using recognizable architectural styles or patterns,

(b) is composed of components that exhibit good design characteristics and

(c) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.

2) A design should be modular.
3) A design should contain distinct representations of data, architecture, interfaces and components.

4) A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5) A design should lead to components that exhibit independent functional characteristics.
6) A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7) A design should be derived using a repeatable method that is driven by information obtained during s/w requirements analysis.
8) A design should be represented using a notation that effectively communicates its meaning.

- **Design Concepts:**
  Fundamental s/w design concepts provide the necessary framework for "getting it right".

i. Abstraction
ii. Architecture
iii. Patterns
iv. Modularity
v. Information Hiding
vi. Functional Independence
vii. Refinement
viii. Refactoring
ix. Design Classes

1. **Abstraction :**

- When we consider a modular solution to any problem, many levels of abstraction can be posed. As we move through different levels of abstraction, we work to create procedural and data abstractions.
- A procedural abstraction refers to a sequence of instructions that have a specific and limited function.
- A data abstraction is a named collection of data that describes a data object.
- The procedural abstraction would make use of information contained in the attributes of the data abstraction.

2. **Architecture:**

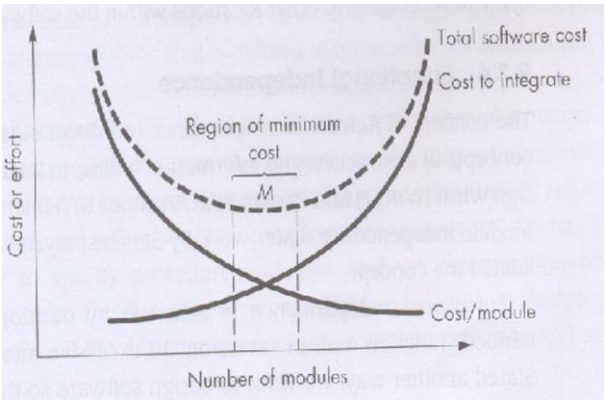
- Architecture is the structure or organization of program components, the manner in which these components interact, and the structure of data that are used by the components.

3. <u>Patterns:</u>

- A design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.
- The intent of each design pattern is to provide a description that enables a designer to determine
- Whether the pattern is applicable to the current work,
- Whether the pattern can be reused, and
- Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

4. <u>Modularity:</u>

- S/w architecture and design patterns embody modularity ie., s/w is divided into separately named and addressable components, called modules, that are integrated to satisfy problem requirements.
- The effort (cost) to develop an individual s/w module does decrease as the total number of module increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows.
- These characteristics lead to a total cost or effort curve shown in the figure.
- There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.



5. **Information Hiding:**

- Modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information
- Abstraction helps to define the procedural (or informational) entities that make up the s/w.
- Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

6. **Functional Independence:**

- The concept of functional independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding.
- We want to design s/w so that each module addresses a specific sub-function of requirements and has simple interface when viewed from other parts of the program structure.
- Functional independence is a key to good design, and design is the key to software quality.
- Independence is assessed using two qualitative criteria : **cohesion and coupling.**
- **Cohesion** is an indication of relative functional strength of a module.
- A cohesive module performs a single task, requiring little interaction with other components in other parts of a program
- **Coupling** is an indication of interconnection among modules in a s/w structure.
- In s/w design, we strive for lowest possible coupling.

7. **Refinement:**

- Refinement is actually a process of elaboration.
- It is a top-down design strategy.
- We begin with a statement of function that is defined at a high level of abstraction. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.
- Abstraction and refinement are complementary concepts. Both concepts aid the designer in creating a complete design model as the design evolves.

8. **Refactoring:**

- Re-factoring is a reorganization technique that simplifies the design (or code) of a component without changing its functions or behavior.
- When s/w is re-factored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

9. **Design classes:**

- As the design model evolves, the s/w team must define a set of design classes.

------X-----X-----

4Q) Design Process, Quality, Evaluation, Patterns - Design Engineering Analysis to Design (Characteristics of a well formed design class)

- **Design Process:**

1) Examine the information domain model and design appropriate data structures for data objects and their attributes

2) Using the analysis model, select an architectural style (and design patterns) that are appropriate for the software

3) Partition the analysis model into design subsystems and allocate these subsystems within the architecture

a) Design the subsystem interfaces

b) Allocate analysis classes or functions to each subsystem

4) Create a set of design classes or components

a) Translate each analysis class description into a design class

b) Check each design class against design criteria; consider inheritance issues

c) Define methods associated with each design class

d) Evaluate and select design patterns for a design class or subsystem

5) Design any interface required with external systems or devices

6) Design the user interface

7) Conduct component-level design

a) Specify all algorithms at a relatively low level of abstraction

b) Refine the interface of each component

c) Define component-level data structures

d) Review each component and correct all errors uncovered

8) Develop a deployment model

a) Show a physical layout of the system, revealing which components will be located where in the physical computing environment

- **Design Quality:**

• The importance of design is quality

• Design is the place where quality is fostered

– Provides representations of software that can be assessed for quality

– Accurately translates a customer's requirements into a finished software product or system

– Serves as the foundation for all software engineering activities that follow

• Without design, we risk building an unstable system that

– Will fail when small changes are made

– May be difficult to test

– Cannot be assessed for quality later in the software process when time is short and most of the budget has been spent

• The quality of the design is assessed through a series of formal technical reviews or design

- **Design patterns**

- Mature engineering disciplines make use of thousands of design patterns for such things as buildings, highways, electrical circuits, factories, weapon systems, vehicles, and computers
- Design patterns also serve a purpose in software engineering
    - Address a specific element of the design such as an aggregation of components or solve some design problem, relationships among components, or the mechanisms for effecting inter-component communication
    - Consist of creational, structural, and behavioural patterns

- **Characteristics of Well-formed design:**

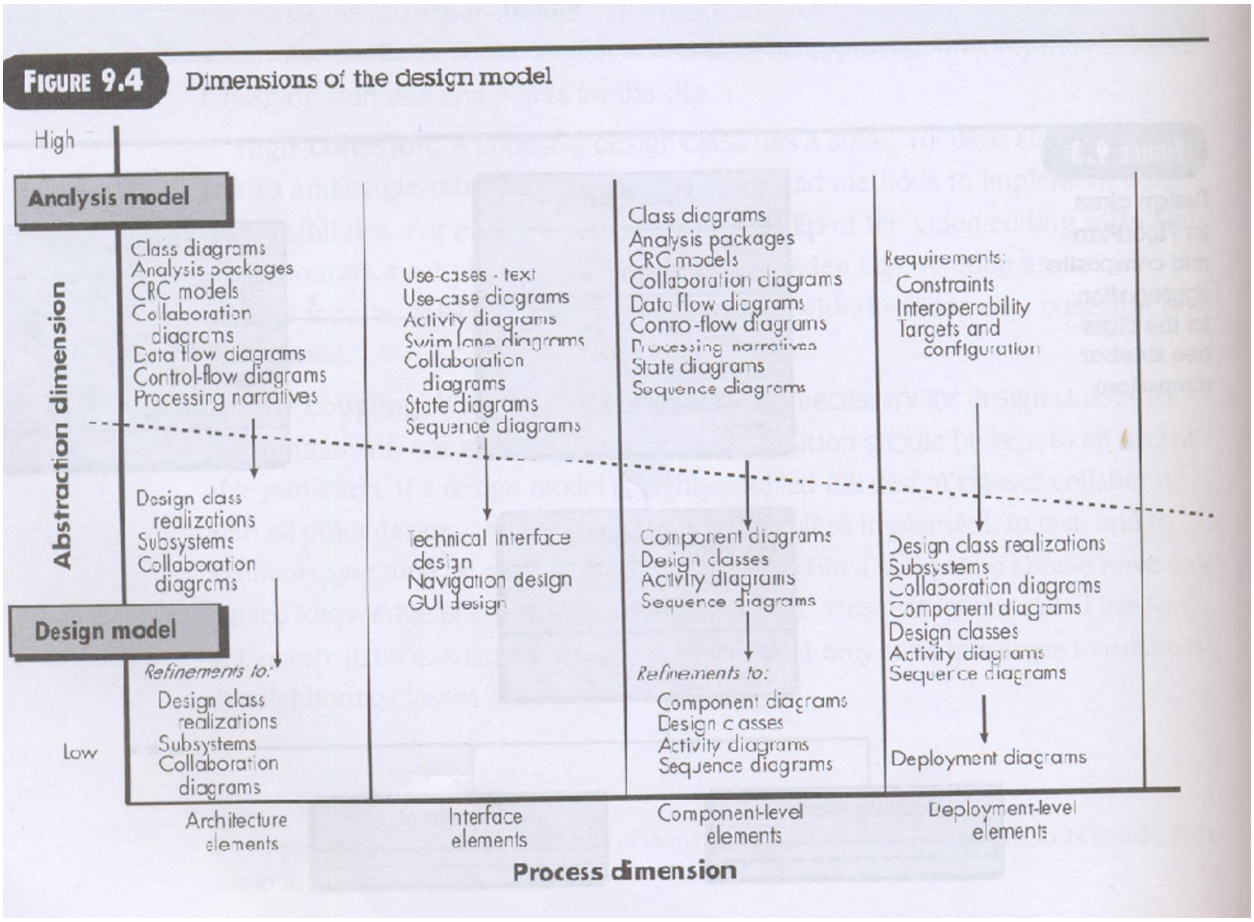
- The four characteristics of a well-formed design class are defined as:
- Complete and sufficient : A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class.
- Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.
- **Primitiveness** : Methods associated with a design class should be focused on accomplishing one service for the class.
- **High Cohesion** : A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.
- **Low Coupling** : Design classes within a subsystem should have only limited knowledge of classes in other subsystems.

-----x-----x-----

5Q)Dimensions of design model.

The design model can be viewed in two different dimensions.

- The **process dimension** indicates the evolution of the design model as design tasks are executed as part of the s/w process.
- The **abstraction dimension** represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.



UNIT-4

1. Software Architecture

Software architecture is the defining and structuring of a solution that meets technical and operational requirements. Software architecture optimizes attributes involving a series of decisions, such as security, performance and manageability. These decisions ultimately impact application quality, maintenance, performance and overall success. Software architecture is a structured framework used to conceptualize software elements, relationships and properties. The software architecture process works through the abstraction and separation of these concerns to reduce complexity.

2. Architectural Styles & Patterns

In software engineering, an *Architectural Pattern* is a general and reusable solution to an occurring problem in a particular context. It is a recurring solution to a recurring problem. The purpose of *Architectural Patterns* is to understand how the major parts of the system fit together and how messages and data flow through the system. We can have multiple patterns in a single system to optimize each section of our code. *Architectural Patterns* are high-level strategies that concern large-scale components, the global properties and mechanisms of a system.

Architectural Style is a name given to a recurrent Architectural Design. It doesn't exist to solve a problem. A single architecture can contain several Architectural Styles, and each Architectural Style can make use of several Architectural Patterns. An Architecture Patterns can be a subset of an Architectural Styles targeting a specific scope. We can use the same words used by the Building Architecture domain, where an Architectural Style is characterized by the features that make a building notable and historically identifiable. A style may include such elements as form, a method of construction or building materials.

Some major Architectural Patterns and Architectural Patterns Styles:

Knowing what we know, let's now have a brief overview of some major *Architectural Patterns* and *Architectural Styles*.

Layered

This pattern is used to structure programs that can be decomposed into groups of subtasks. It partitions the concerns of the application into layers.

Most of the time, we have four layers:

- Presentation layer or UI layer
- Application layer or Service layer
- Business logic layer or Domain layer
- Data access layer or Persistence layer

Event-Driven

Also called EDA, this pattern organizes a system around the production, detection and consumption of events. Such a system consists of event *Emitters* and event *Consumers*. *Emitters* are decoupled from *Consumers*, which are also decoupled from each other. This kind of architecture is often used for asynchronous systems or user interfaces.

Domain Driven Design

This *Architectural Style*, also known has *DDD*, is an object-oriented approach. Here, the idea is to design software based on the *Business Domain*, its elements and behaviors, and the relationships between them.

Pipes and Filters

This *Architectural Style* decomposes a task that performs complex processing into a series of separate elements that can be reused. In other words, it consists of any number of components, called *Filters*, that transform or filter data, before passing it to other components through connectors called *Pipes*.

Microservices

The goal of a *Microservices* architecture is, instead of building one single big monolithic application, to create several tiny programs.

3. Software Architectural Reuse:

4. Software Component:

A software component is a modular building block for computer software . It is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. A component communicates and collaborates with Other components and Entities outside the boundaries of the system Three different views of a component :-

– An object-oriented view

– A conventional view

– A process-related view

5. Component Design:

Component-level design occurs after the first iteration of the architectural design It strives to create a design model from the analysis and architectural models. The translation can open the door to subtle errors that are difficult to find and correct later. A component-level design can be represented using some intermediate representation (e.g. graphical, tabular, or text-based) that can be translated into source code The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors

The Software Component

• A software component is a modular building block for computer software

– It is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces

• A component communicates and collaborates with

– Other components

– Entities outside the boundaries of the system

• Three different views of a component

– An object-oriented view

– A conventional view

– A process-related view

Object-oriented View

• A component is viewed as a set of one or more collaborating classes .Each problem domain (i.e., analysis) class and infrastructure (i.e., design) class is elaborated to identify all attributes and operations that apply to its implementation .This also involves defining the interfaces that enable classes to communicate and collaborate .This elaboration activity is applied to every component defined as part of the architectural design

• Once this is completed, the following steps are performed

– Provide further elaboration of each attribute, operation, and interface

– Specify the data structure appropriate for each attribute

Design the algorithmic detail required to implement the processing logic associated with each operation .

– Design the mechanisms required to implement the interface to include the messaging that occurs between objects

Conventional View

A component is viewed as a functional element (i.e., a module) of a program that incorporates

– The processing logic

– The internal data structures that are required to implement the processing logic

– An interface that enables the component to be invoked and data to be passed to it

• A component serves one of the following roles

– A control component that coordinates the invocation of all other problem domain components

– A problem domain component that implements a complete or partial function that is required by the customer

– An infrastructure component that is responsible for functions that support the processing required in the problem domain

• Once this is completed, the following steps are performed for each transform

– Define the interface for the transform (the order, number and types of the parameters)

– Define the data structures used internally by the transform

– Design the algorithm used by the transform (using a stepwise refinement approach)

Process-related View

• Emphasis is placed on building systems from existing components maintained in a library rather than creating each component from scratch

• As the software architecture is formulated, components are selected from the library and used to populate the architecture

• Because the components in the library have been created with reuse in mind, each contains the following:

– A complete description of their interface

– The functions they perform

6. Component development – Components -Off -The -Shelf (COTS)

Component-based development (CBD) is a procedure that accentuates the design and development of computer-based systems with the help of reusable software components. With CBD, the focus shifts from software programming to software system composing.

Component-based development techniques involve procedures for developing software systems by choosing ideal off-the-shelf components and then assembling them using a well-defined software architecture. With the systematic reuse of coarse-grained components, CBD intends to deliver better quality and output. The idea behind CBD is to integrate the related parts and reuse them collectively. These integrated parts are known as components.

Component-based development techniques consist of non-conventional development routines, including component evaluation, component retrieval, etc. It is important that the CBD is carried out within a middleware infrastructure that supports the process, for example, Enterprise Java Beans.

The key goals of CBD are as follows:

- Save time and money when building large and complex systems: Developing complex software systems with the help of off-the-shelf components helps reduce software development time substantially. Function points or similar techniques can be used to verify the affordability of the existing method.
- Enhance the software quality: The component quality is the key factor behind the enhancement of software quality.
- Detect defects within the systems: The CBD strategy supports fault detection by testing the components; however, finding the source of defects is challenging in CBD.

Some advantages of CBD include:

- Minimized delivery:
    - Search in component catalogs
    - Recycling of pre-fabricated components
- Improved efficiency:
    - Developers concentrate on application development
- Improved quality:
    - Component developers can permit additional time to ensure quality
- Minimized expenditures

The specific routines of CBD are:

- Component development
- Component publishing

- Component lookup as well as retrieval
- Component analysis
- Component assembly

7. Object Constraint Language(OCL)

The Object Constraint Language (OCL) started as a complement of the UML notation with the goal to overcome the limitations of UML in terms of precisely specifying detailed aspects of a system design. Since then, OCL has become a key component of any model-driven engineering (MDE) technique as the default language for expressing all kinds of (meta)model query, manipulation and specification requirements. Among many other applications, OCL is frequently used to express model transformations well-formedness rules , or code-generation templates .

8. Data and Database Engineer/Design

Database design, as the name suggests, is the process of creating a detailed data model of the database. It contains all the logical design and physical design parameters and attributes which will be used to create a database. The data model should address all the attributes in great detail for each entity.

Database development, as is suggestive of its name, is the process of realizing the database design and constructing the database according to the specification of the schema. This also involves choosing the right database for the requirements of the business overcoming the software and hardware limitations.

Thus Database Design and Development is the entire process of putting business model requirements into a well planned database.

The database design must conform to design standards. If the design standards are not followed, there would surely be destructive data anomalies. Without a design standards, it is nearly impossible to construct a proper design, evaluate this design and trace the problems in this design. The processing speed or the access time of data through a database is a big consideration for database design and development .Even if the database design is created following the standard guidelines but cannot serve the end user with speed, such a perfect design is of no use to the business.

9. The golden rules for user interface design

Place the User in Control

• Define interaction modes in a way that does not force a user into unnecessary or undesired actions .The user shall be able to enter and exit a mode with little or no effort (e.g., spell check □ edit text □ spell check)

• Provide for flexible interaction .The user shall be able to perform the same action via keyboard commands, mouse movement, or voice recognition

• Allow user interaction to be interruptible and "undo"able .The user shall be able to easily interrupt a sequence of actions to do something else (without losing the work that has been done so far)

• Design for direct interaction with objects that appear on the screen . The user shall be able to manipulate objects on the screen in a manner similar to what would occur if the object were a physical thing (e.g., stretch a rectangle, press a button, move a slider)

Reduce the User's Memory Load

• Reduce demand on short-term memory . The interface shall reduce the user's requirement to remember past actions and results by providing visual cues of such actions

• Establish meaningful defaults . The system shall provide the user with default values that make sense to the average user but allow the user to change these defaults

• Define shortcuts that are intuitive .The user shall be provided mnemonics (i.e., control or alt combinations) that tie easily to the action in a way that is easy to remember such as the first letter.

• Disclose information in a progressive fashion . When interacting with a task, an object or some behavior, the interface shall be organized hierarchically by moving the user progressively in a step-wise fashion from an abstract concept to a concrete action (e.g., text format options □ format dialog box) .

Make the Interface Consistent

• The interface should present and acquire information in a consistent fashion All visual information shall be organized according to a design standard that is maintained throughout all screen displays .

• Allow the user to put the current task into a meaningful context . The interface shall provide indicators (e.g., window titles, consistent color coding) that enable the user to know the context of the work at hand .

• Maintain consistency across a family of applications . A set of applications performing complimentary functionality shall all implement the same design rules so that consistency is maintained for all interaction .

• If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so .

10. Steps in user interface analysis & design

To perform user interface analysis, the practitioner needs to study and understand four elements

– The users who will interact with the system through the interface

– The tasks that end users must perform to do their work

– The content that is presented as part of the interface

– The work environment in which these tasks will be conducted

User Analysis

• The analyst strives to get the end user's mental model and the design model to converge by understanding “The users themselves” and “ How these people use the system “.

• Information can be obtained from

– User interviews with the end users

– Sales input from the sales people who interact with customers and users on a regular basis

– Marketing input based on a market analysis to understand how different population segments might use the software

– Support input from the support staff who are aware of what works and what doesn't, what users like and dislike, what features generate questions, and what features are easy to use.

Task Analysis and Modeling

• Task analysis strives to know and understand

– The work the user performs in specific circumstances

– The tasks and subtasks that will be performed as the user does the work

– The specific problem domain objects that the user manipulates as work is performed

– The sequence of work tasks (i.e., the workflow)

– The hierarchy of tasks

• Use cases Show how an end user performs some specific work-related task . Enable the software engineer to extract tasks, objects, and overall workflow of the interaction . Helps the software engineer to identify additional helpful features

Content Analysis

• The display content may range from character-based reports, to graphical displays, to multimedia information

• Display content may be

– Generated by components in other parts of the application

– Acquired from data stored in a database that is accessible from the application

– Transmitted from systems external to the application in question

• The format and aesthetics of the content (as it is displayed by the interface) needs to be considered

Work Environment Analysis

• Software products need to be designed to fit into the work environment, otherwise they may be difficult or frustrating to use .Factors to consider include

– Type of lighting

– Display size and height

– Keyboard size, height and ease of use

– Mouse type and ease of use

– Surrounding noise

– Space limitations for computer and/or user.

User interface design is an iterative process, where each iteration elaborate and refines the information developed in the preceding step

• General steps for user interface design

1) Using information developed during user interface analysis, define user interface objects and actions (operations).

2) Define events (user actions) that will cause the state of the user interface to change; model this behavior.

3) Depict each interface state as it will actually look to the end user.

4) Indicate how the user interprets the state of the system from information provided through the interface .

Unit-5

2Q)Verification and Validation

• Software testing is part of a broader group of activities called verification and validation that are involved in software quality assurance

– The set of activities that ensure that software correctly implements a specific function or algorithm

• Validation (Does it meet user requirements?)

– The set of activities that ensure that the software that has been built is traceable to customer requirements

Validation Testing

• Validation testing follows integration testing

• The distinction between conventional and object-oriented software disappears

• Focuses on user-visible actions and user-recognizable output from the system

• Demonstrates conformity with requirements

• Designed to ensure that

– All functional requirements are satisfied

– All behavioral characteristics are achieved

– All performance requirements are attained

– Documentation is correct

– Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)

• After each validation test

– The function or performance characteristic conforms to specification and is accepted

– A deviation from specification is uncovered and a deficiency list is created

• A configuration review or audit ensures that all elements of the software configuration have been properly developed, cataloged, and have the necessary detail for entering the support phase of the software life cycle.

FORMAL TECHNICAL REVIEW

A formal technical review (FTR) is a software quality control activity performed by software engineers (and others). The objectives of an FTR are:

(1) to uncover errors in function, logic, or implementation for any representation of the software;

(2) to verify that the software under review meets its requirements;

(3) to ensure that the software has been represented according to predefined standards;

(4) to achieve software that is developed in a uniform manner; and

(5) to make projects more manageable. In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation.

- The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen.
- The FTR is actually a class of reviews that includes walkthroughs and inspections.
- Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended. In the sections that follow, guidelines similar to those for a walkthrough are presented as a representative formal technical review.

-----x----x-----

3Q)TESTING:

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

Characteristics of good testing:

• A good test has a high probability of finding an error

– The tester must understand the software and how it might fail

• A good test is not redundant

– Testing time is limited; one test should not serve the same purpose as another test

• A good test should be "best of breed"

– Tests that have the highest likelihood of uncovering a whole class of errors should be used

• A good test should be neither too simple nor too complex

– Each test should be executed separately; combining a series of tests could cause side effects and mask certain errors.

Goals of testing:

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

Test cases:

1) Using the design or code as a foundation, draw a corresponding flow graph

2) Determine the cyclomatic complexity of the resultant flow graph

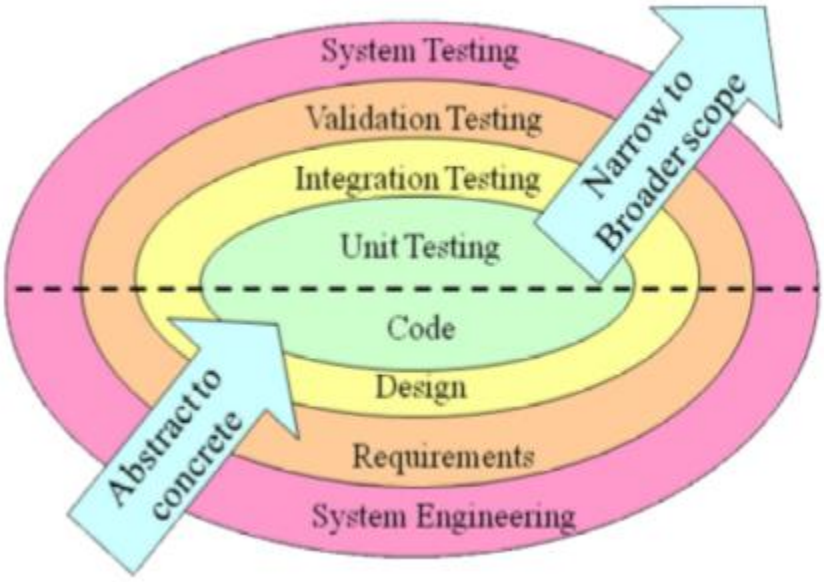
3) Determine a basis set of linearly independent paths

4) Prepare test cases that will force execution of each path in the basis set

Test oracle:

"A fundamental assumption of this testing is that there is some mechanism, a test oracle, that will determine whether or not the results of a test execution are correct".In essence, a test oracle establishes the basis for any determination of the correctness of the output. In most cases, the oracle is the requirements model, but it could also be another document or application, data recorded elsewhere, or even a human expert.

Conventional Test Strategies:



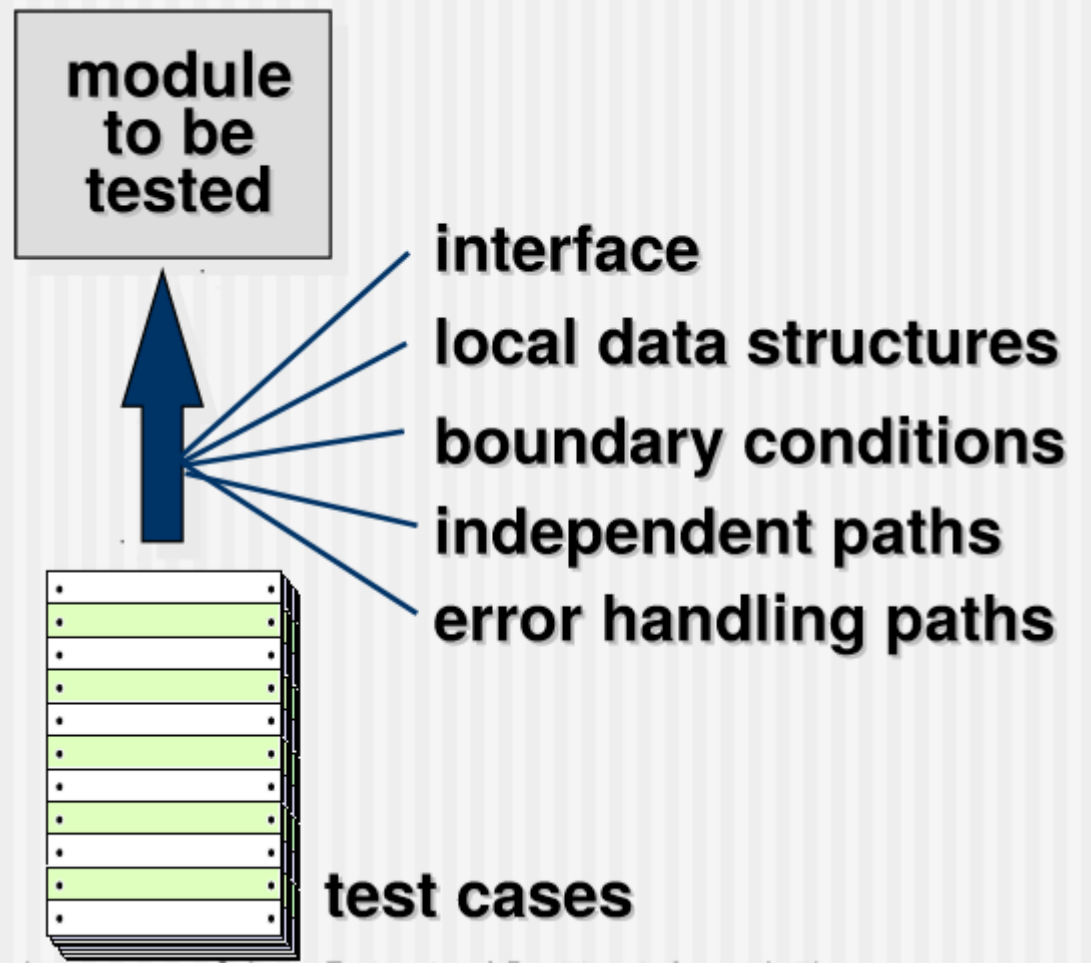
Levels of Testing for Conventional Software

• Unit testing

– Concentrates on each component/function of the software as implemented in the source code .

– Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection.

– Components are then assembled and integrated

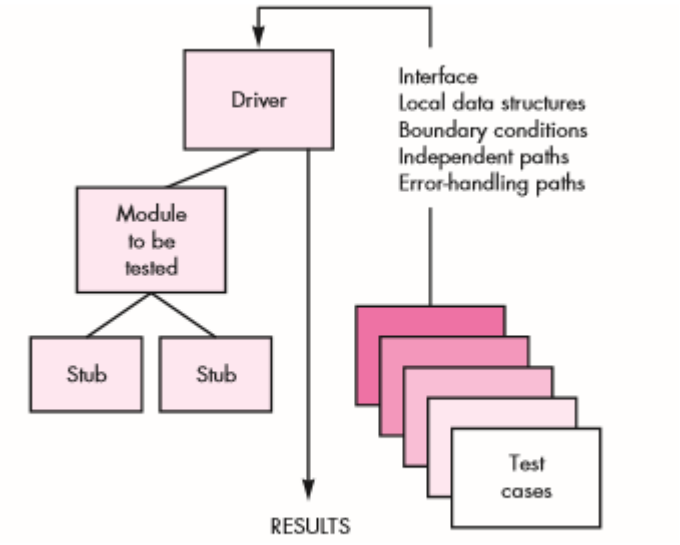


.

Unit-test considerations: The module interface is tested to ensure that information properly flows into and out of the program unit under test.

- Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. And finally, all error-handling paths are tested.
- Data flow across a component interface is tested before any other testing is initiated.
- Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the nth element of an n-dimensional array is processed, when the ith repetition of a loop with i passes is invoked, when the maximum or minimum allowable value is encountered.
- Unit-test procedures: Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated. Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test. In most applications a driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints
  relevant results. Stubs serve to replace modules that are subordinate (invoked by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data

manipulation, prints verification of entry, and returns control to the module



undergoing testing.

- Drivers and stubs represent testing "overhead." That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step
- Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

• Integration testing

– Focuses on the design and construction of the software architecture

- Focuses on inputs and outputs, and how well the components fit together and work together

-The objective is to take unit-tested components and build a program structure that has been dictated by design.

Integration Testing Strategies

Options:

- the “big bang” approach
- an incremental construction strategy

  Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.

• Validation testing

– Requirements are validated against the constructed software

– Provides final assurance that the software meets all functional, behavioral, and performance requirements

• System testing

– The software and other system elements are tested as a whole

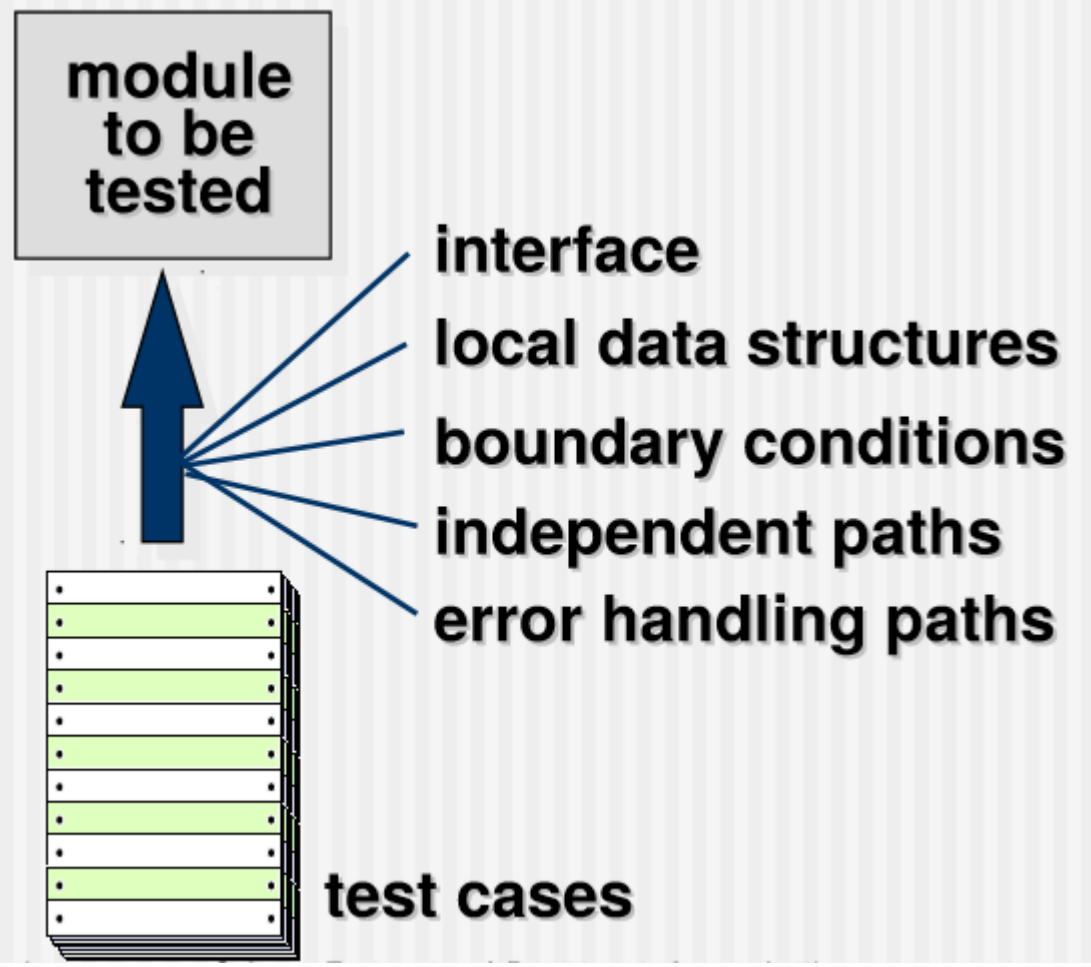
– Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved.

Unit testing

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module.

The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing.

The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.



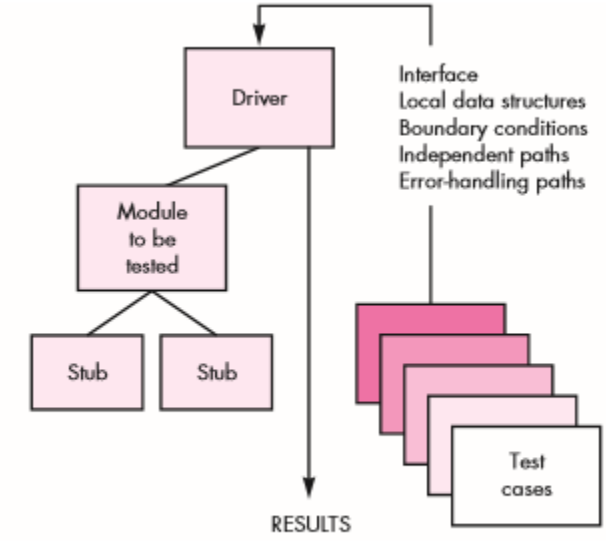
Unit-test considerations: The module interface is tested to ensure that information properly flows into and out of the program unit under test.

- Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once
- . Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. And finally, all error-handling paths are tested.
- Data flow across a component interface is tested before any other testing is initiated.

- Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the nth element of an n-dimensional array is processed, when the ith repetition of a loop with i passes is invoked, when the maximum or minimum allowable value is encountered.

- Unit-test procedures: Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated. Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test. In most applications a driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results.

- Stubs serve to replace modules that are subordinate (invoked by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.



- Drivers and stubs represent testing "overhead." That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept

simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with “simple” overhead software. In such cases, complete testing can be postponed until the integration test step

- Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

White Box Testing

• Uses the control structure part of component-level design to derive the test cases

• These test cases

– Guarantee that all independent paths within a module have been exercised at least once

– Exercise all logical decisions on their true and false sides

– Execute all loops at their boundaries and within their operational bounds

– Exercise internal data structures to ensure their validity

Basis Path Testing

• White-box testing technique proposed by Tom McCabe

• Enables the test case designer to derive a logical complexity measure of a procedural design

• Uses this measure as a guide for defining a basis set of execution paths

• Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing

Flow Graph Notation

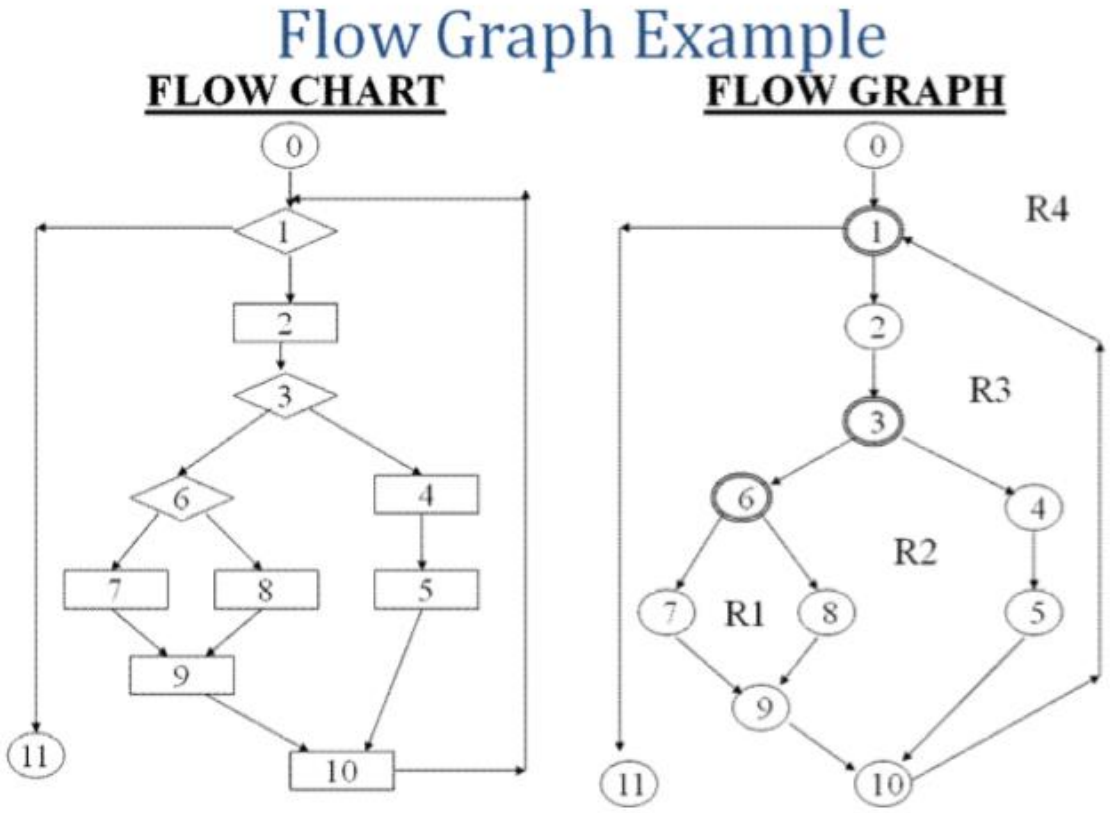
• A circle in a graph represents a node, which stands for a sequence of one or more procedural statements

• A node containing a simple conditional expression is referred to as a predicate node

– Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node

– A predicate node has two edges leading out from it (True and False)

• An edge, or a link, is a an arrow representing flow of control in a specific direction
– An edge must start and terminate at a node
– An edge does not intersect or cross over another edge
• Areas bounded by a set of edges and nodes are called regions
• When counting regions, include the area outside the graph as a region, too



Independent Program Paths
• Defined as a path through the program from the start node until the end node that introduces at least one new set of processing statements or a new condition (i.e., new nodes)
• Must move along at least one edge that has not been traversed before by a previous path
• Basis set for flow graph on previous slide
– Path 1: 0-1-11
– Path 2: 0-1-2-3-4-5-10-1-11
– Path 3: 0-1-2-3-6-8-9-10-1-11
– Path 4: 0-1-2-3-6-7-9-10-1-11

• The number of paths in the basis set is determined by the cyclomatic complexity.

Cyclomatic Complexity

• Provides a quantitative measure of the logical complexity of a program
• Defines the number of independent paths in the basis set
• Provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once
• Can be computed three ways
– The number of regions
– V(G) = E – N + 2, where E is the number of edges and N is the number of nodes in graph G
• Results in the following equations for the example flow graph
– Number of regions = 4
– V(G) = 14 edges – 12 nodes + 2 = 4
– V(G) = 3 predicate nodes + 1 =4

Black Box testing :

• Complements white-box testing by uncovering different classes of errors

• Focuses on the functional requirements and the information domain of the software

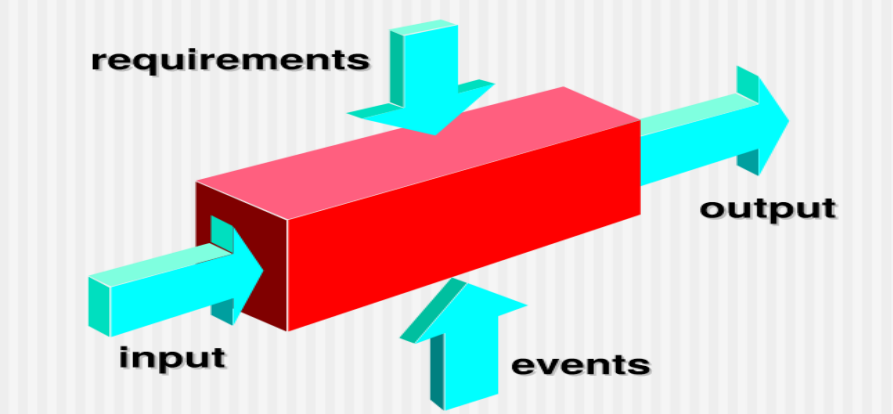
• Used during the later stages of testing after white box testing has been performed

• The tester identifies a set of input conditions that will fully exercise all functional requirements for a program

• The test cases satisfy the following:

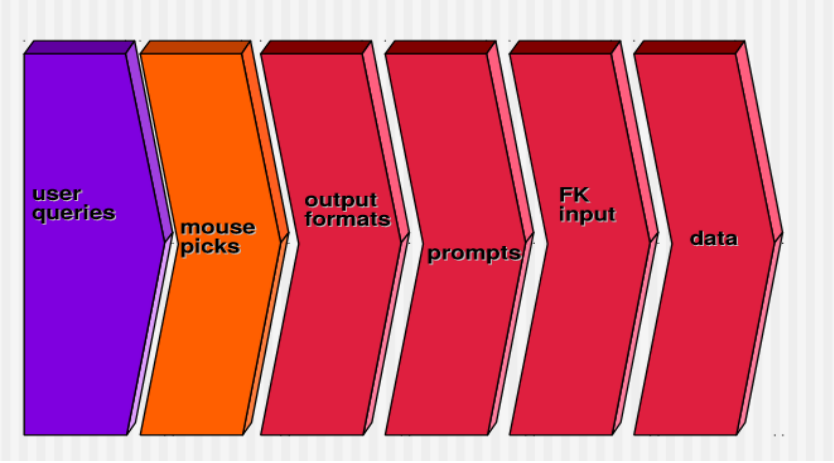
– Reduce, by a count greater than one, the number of additional test cases that must be designed to achieve reasonable testing

– Tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific task at hand



Equivalence Partitioning

- A black-box testing method that divides the input domain of a program into classes of data from which test cases are derived
- An ideal test case single-handedly uncovers a complete class of errors, thereby reducing the total number of test cases that must be developed
- Test case design is based on an evaluation of equivalence classes for an input condition
- An equivalence class represents a set of valid or invalid states for input conditions
- From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercise at once.



Guidelines for Defining Equivalence Classes

• If an input condition specifies a range, one valid and two invalid equivalence classes are defined

– Input range: 1 – 10 Eq classes: {1..10}, {x < 1}, {x > 10}

• If an input condition requires a specific value, one valid and two invalid equivalence classes are defined

– Input value: 250 Eq classes: {250}, {x < 250}, {x > 250}

• If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined

– Input set: {-2.5, 7.3, 8.4} Eq classes: {-2.5, 7.3, 8.4}, {any other x}

• If an input condition is a Boolean value, one valid and one invalid class are define

– Input: {true condition} Eq classes: {true condition}, {false condition}

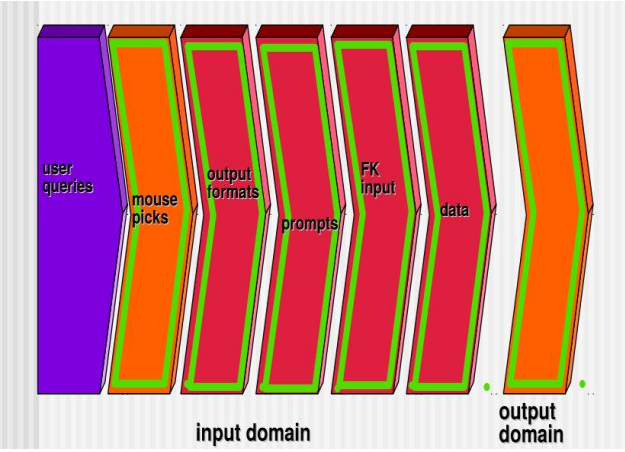
Boundary Value Analysis

• A greater number of errors occur at the boundaries of the input domain rather than in the "center"

• Boundary value analysis is a test case design method that complements equivalence partitioning

– It selects test cases at the edges of a class

– It derives test cases from both the input domain and output domain



Guidelines for Boundary Value Analysis

- 1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b as well as values just above and just below a and b
- 2. If an input condition specifies a number of values, test case should be developed that exercise the minimum and maximum numbers. Values just above and just below the minimum and maximum are also tested
- Apply guidelines 1 and 2 to output conditions; produce output that reflects the minimum and the maximum values expected; also test the values just below and just above
- If internal program data structures have prescribed boundaries (e.g., an array), design a test case to exercise the data structure at its minimum and maximum boundaries

Integration testing

- Defined as a systematic technique for constructing the software architecture

– At the same time integration is occurring, conduct tests to uncover errors associated with interfaces

- Objective is to take unit tested modules and build a program structure based on the prescribed design
- Two Approaches

– Non-incremental Integration Testing
– Incremental Integration Testing

Incremental Integration Testing

- Three kinds

– Top-down integration
– Bottom-up integration
– Sandwich integration

- The program is constructed and tested in small increments
- Errors are easier to isolate and correct
- Interfaces are more likely to be tested completely
- A systematic test approach is applied

Top-down Integration

• Modules are integrated by moving downward through the control hierarchy, beginning with the main module

• Subordinate modules are incorporated in either a depth-first or breadth-first fashion

– DF: All modules on a major control path are integrated

– BF: All modules directly subordinate at each level are integrated

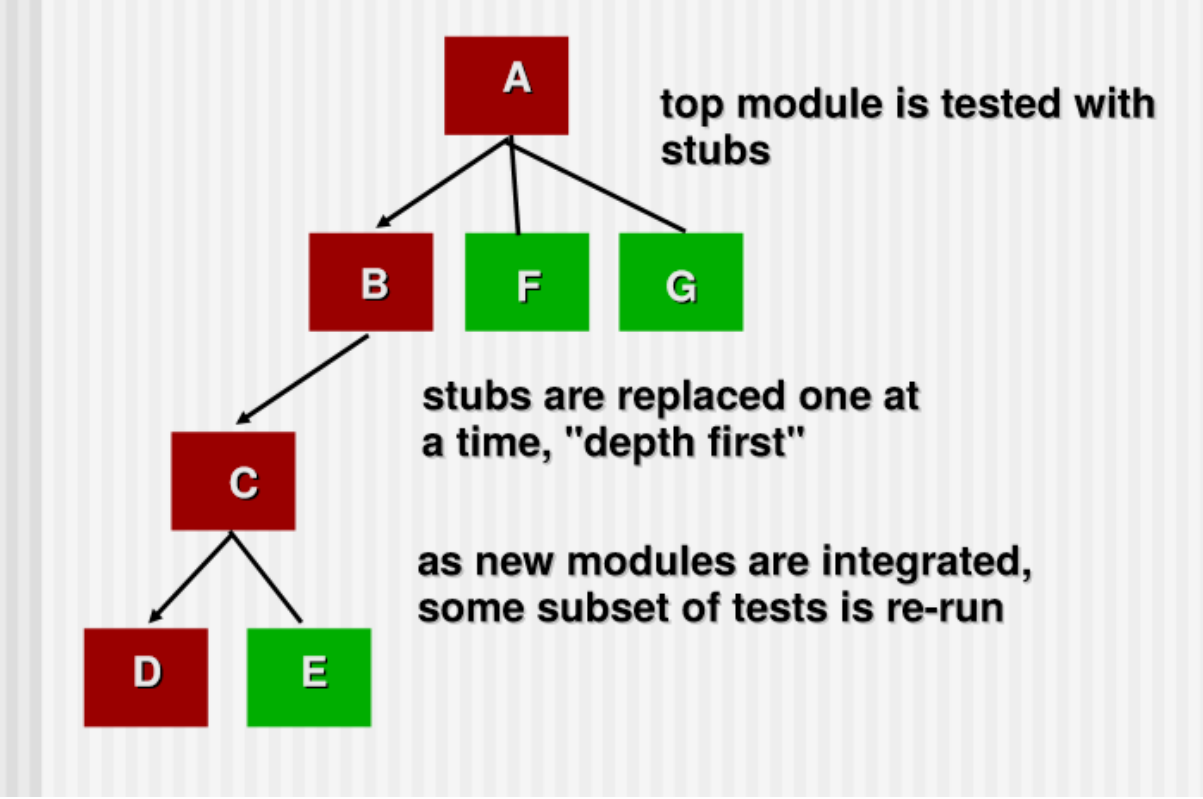
• Advantages

– This approach verifies major control or decision points early in the test process

• Disadvantages

– Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded

– Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process



Bottom-up Integration

• Integration and testing starts with the most atomic modules in the control hierarchy

• Advantages

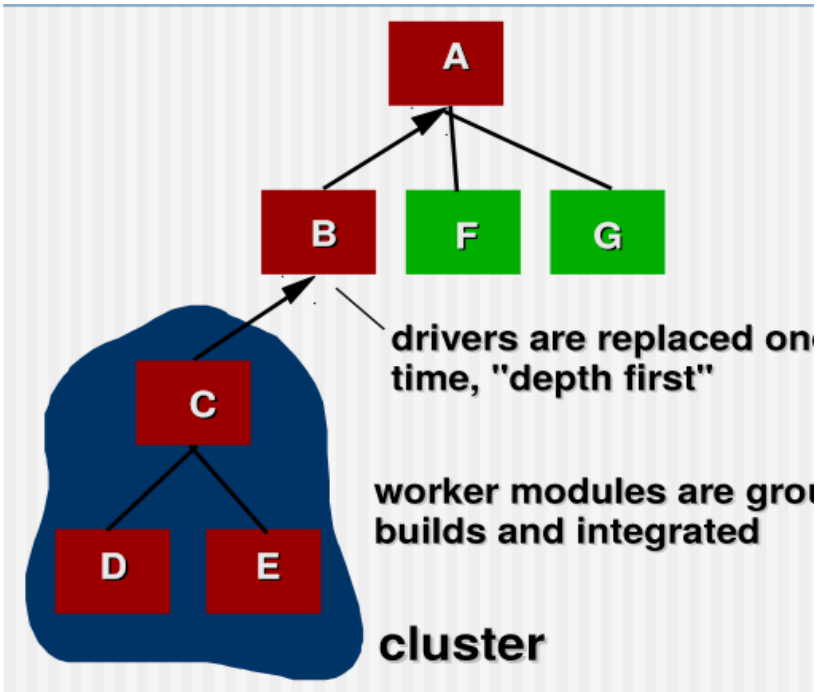
– This approach verifies low-level data processing early in the testing process

– Need for stubs is eliminated

• Disadvantages

– Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version

– Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available .



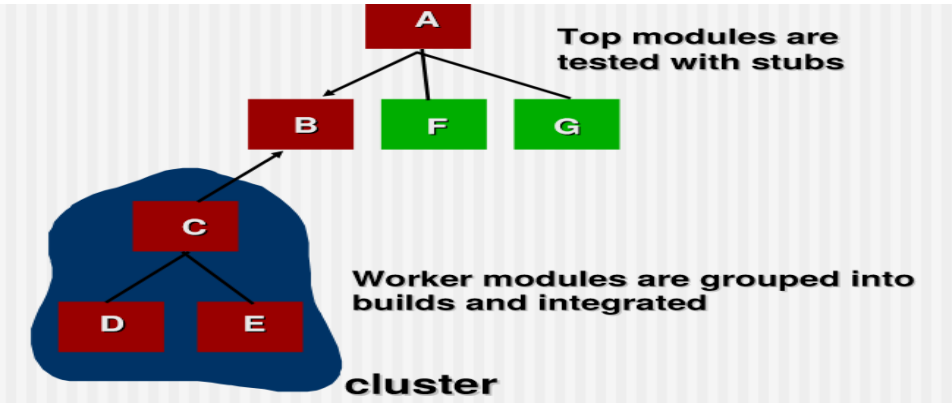
Sandwich Integration

• Consists of a combination of both top-down and bottom-up integration

• Occurs both at the highest level modules and also at the lowest level modules

• Proceeds using functional groups of modules, with each group completed before the next

– High and low-level modules are grouped based on the control and data processing they provide for a specific program feature
– Integration within the group progresses in alternating steps between the high and low level modules of the group
– When integration for a certain functional group is complete, integration and testing moves onto the next group
• Reaps the advantages of both types of integration while minimizing the need for drivers and stubs
• Requires a disciplined approach so that integration doesn't tend towards the "big bang" scenario.



System Testing

Different Types :

Regression Testing
• Each new addition or change to base lined software may cause problems with functions that previously worked flawlessly
• Regression testing re-executes a small subset of tests that have already been conducted
– Ensures that changes have not propagated unintended side effects
– Helps to ensure that changes do not introduce unintended behaviour or additional errors
– May be done manually or through the use of automated capture/playback tools

• Regression test suite contains three different classes of test cases
– A representative sample of tests that will exercise all software functions
– Additional tests that focus on software functions that are likely to be affected by the change
– Tests that focus on the actual software components that have been changed.

Big-Bang

Non-incremental Integration Testing :
• Commonly called the "Big Bang" approach
• All components are combined in advance
• The entire program is tested as a whole
• Chaos results
• Many seemingly-unrelated errors are encountered
• Correction is difficult because isolation of causes is complicated
• Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop.

Stress Testing:
Stress testing is a continuation of load testing, but in this instance the variables, N, T, and Dare forced to meet and then exceed operational limits. The intent of these tests is to answer each of the following questions:

- Does the system degrade "gently," or does the server shut down as capacity is exceeded?
- Does server software generate "server not available" messages? More generally, are users aware that they cannot reach the server?
- Does the server queue resource requests and empty the queue once capacity demands diminish?
- Are transactions lost as capacity is exceeded?

Is data integrity affected as capacity is exceeded?

- What values of N, T, and D force the server environment to fail? How does failure manifest itself? Are automated notifications sent to technical support staff at the server site?

- If the system does fail, how long will it take to come back online?
- Are certain WebApp functions (e.g., compute intensive functionality, data streaming capabilities) discontinued as capacity reaches the 80 or 90 percent level?

A variation of stress testing is sometimes referred to as spike/bounce testing . In this testing regime, load is spiked to capacity, then lowered quickly to normal operating conditions, and then spiked again. By bouncing system loading, you can determine how well the server can marshal resources to meet very high demand and then release them when normal conditions reappear (so that they are ready for the next spike).

Load Testing

The intent of load testing is to determine how the WebApp and its server-side environment will respond to various loading conditions. As testing proceeds, permutations to the following variables define a set of test conditions:

N, number of concurrent users

T, number of online transactions per unit of time

D, data load processed by the server per transaction

In every case, these variables are defined within normal operating bounds of the system. As each test condition is run, one or more of the following measures are collected: average user response, average time to download a standardized unit of data, or average time to process a transaction. You should examine these measures to determine whether a precipitous decrease in performance can be traced to a specific combination of N, T, and D.

Load testing can also be used to assess recommended connection speeds for users of the WebApp. Overall throughput, P, is computed in the following manner:

P =N*T*D

As an example, consider a popular sports news site. At a given moment, 20,000 concurrent users submit a request (a transaction, T) once every 2 minutes on average. Each transaction requires the WebApp to download a new article that averages 3K bytes in length. Therefore, throughput can be calculated as:

P=[20,000 * 0.5* 3Kb]/60 =500 Kbytes/sec
= 4 megabits per second

The network connection for the server would therefore have to support this data rate and should be tested to ensure that it does.

Validation Testing
Alpha and Beta Testing
• Alpha testing
– Conducted at the developer's site by end users
– Software is used in a natural setting with developers watching intently
– Testing is conducted in a controlled environment
• Beta testing
– Conducted at end-user sites
– Developer is generally not present
– It serves as a live application of the software in an environment that cannot be controlled by the developer
– The end-user records all problems that are encountered and reports these to the developers at regular intervals
• After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base .

Object Oriented Test Strategies
Class Based:

- When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class (object) packages attributes (data) and the operations (also known as methods or services) that manipulate these data. Rather than testing an individual module, the smallest testable unit is the encapsulated class. Because a class can contain a number of different operations and a particular operation may exist as part of a number of different classes, the meaning of unit testing changes dramatically.

- We can no longer test a single operation in isolation (the conventional view of unit testing) but rather, as part of a class. To illustrate, consider a class hierarchy in which an operation X() is defined for the superclass and is inherited by a number of subclasses. Each subclass uses operation X(), but it is applied within the context of the private attributes and operations that have been defined for each subclass. Because the context in which operation X()is used varies in subtle ways, it is necessary to test operation X() in the context of each of the subclasses. This means that testing operation X() in a vacuum (the traditional unit-testing approach) is ineffective in the object-oriented context.
- Class testing is the equivalent of unit testing

- operations within the class are tested.
- the state behaviour of the class is examined.

Model-Based Testing:

 Analyse an existing behavioural model for the software or create one.

- Recall that a behavioural model indicates how software will respond to external events or stimuli.

 Traverse the behavioural model and specify the inputs that will force the software to make the transition from state to state.

- The inputs will trigger events that will cause the transition to occur.

 Review the behavioural model and note the expected outputs as the software makes the transition from state to state.

 Execute the test cases.

 Compare actual and expected results and take corrective action as required.

------x----x-------

4Q)DEBUGGING:

Debugging occurs as a consequence of successful testing. The debugging process begins with the execution of a test case. Results are assessed and the difference between expected and actual performance is encountered. This difference is a symptom of an underlying cause that lies hidden.The debugging process attempts to match symptom with cause, thereby leading to error correction.

DEBUGGING STRATEGIES:

Objective of debugging is to find and correct the cause of a software error

• Bugs are found by a combination of systematic evaluation, intuition, and luck.

• Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code.

• There are three main debugging strategies
– **Brute force**
– **Backtracking**
– **Cause elimination**

Brute Force:

• Most commonly used and least efficient method
• Used when all else fails
• Involves the use of memory dumps, run-time traces, and output statements
• Leads many times to wasted effort and time

Backtracking:

• Can be used successfully in small programs
• The method starts at the location where a symptom has been uncovered
• The source code is then traced backward (manually) until the location of the cause is found
• In large programs, the number of potential backward paths may become unmanageably large

Cause Elimination:

• Involves the use of induction or deduction and introduces the concept of binary partitioning.

– Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true.
– Deduction (general to specific): Show that a specific conclusion follows from a set of general
Premises.
• Data related to the error occurrence are organized to isolate potential causes
• A cause hypothesis is devised, and the aforementioned data are used to prove or disprove the hypothesis
• Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each cause
• If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

----------x----------x----------

5Q)REFACTORING OF SOFTWARE:

Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code [design] yet improves its internal structure. When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

SOFTWARE CONFIGURATION MANAGEMENT:

Software configuration management (SCM) is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM activities are developed to identify change, control change, ensure that change is being properly implemented, and report changes to others who may have an interest.
The output of the software process is information that may be divided into three broad categories:
(1) computer programs (both source level and executable forms),
(2) work products that describe the computer programs (targeted at various stakeholders),
and

(3) data or content (contained within the program or external to it). The items that comprise all information produced as part of the software process are collectively called a *software configuration*.

Software configuration management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process. In the sections that follow, I describe major SCM tasks and important concepts that can help you to manage change.

----------x----------x----------

6Q)METRICS FOR DIFFERENT PHASES OF SDLC:

Phase 1:Requirements phase
Function-based metrics: use the function point as a normalizing factor or as a measure of the
"size" of the specification.
Specifcation metrics: used as an indication of quality by measuring number of requirements
by type.

Phase 2:Design phase
Architectural design metrics
Structural complexity = g(fan-out)
Data complexity = f(input & output variables, fan-out)
System complexity = h(structural & data complexity)
HK metric: architectural complexity as a function of fan-in and fan-out
Morphology metrics: a function of the number of modules and the number of interfaces
between modules.

Phase 3:Coding phase

Halstead's Software Science: a comprehensive collection of metrics all predicated on the number (count and occurrence) of operators and operands within a component or program.
It should be noted that Halstead's "laws" have generated substantial controversy, and many believe that the underlying theory has faws. However,experimental verifcation for selected programming languages has been performed.

Phase 4:Testing phase
Testing efort can also be estimated using metrics derived from Halstead measures. Binder [Bin94] suggests a broad array of design metrics that have a direct influence on
the "testability" of an OO system.
Back of cohesion in methods (BCOM).
 Percent public and protected (PAP).
 Public access to data members (PAD).
 Number of root classes (NOR).
 Fan-in (FIN).
 Number of children (NOC) and depth of the inheritance tree (DIT).

Phase 5:Maintanence phase
IEEE Std. 982.1-1988 [IEE94] suggests a *software maturity index* (SMI) that provides an indication of the stability of a software product (based on changes that occur for each
release of the product). The following information is determined:
• *MT* = the number of modules in the current release
• *Fc* = the number of modules in the current release that have been changed
• *Fa* = the number of modules in the current release that have been added
• *Fd* = the number of modules from the preceding release that were deleted in the current
release
The software maturity index is computed in the following manner:
• SMI = [*MT* - (*Fa* + *Fc* + *Fd*)]/*MT*
As SMI approaches 1.0, the product begins to stabilize.

- ELEMENTS OF PRODUCT METRICS:

A *measure* provides a quantitative indication of the extent, amount, dimension, capacity, or
size of some attribute of a product or process
The IEEE glossary defnes a *metric* as "a quantitative measure of the degree to which a
system, component, or process possesses a given attribute."
An *indicator* is a metric or combination of metrics that provide insight into the software
process, a software project, or the product itself.

Measurement Principles

The objectives of measurement should be established before data collection begins;
Each technical metric should be defned in an unambiguous manner;
Metrics should be derived based on a theory that is valid for the domain of application (e.g., metrics for design should draw upon basic design concepts and principles and attempt to provide an indication of the presence of an attribute that is deemed desirable);
Metrics should be tailored to best accommodate specifc products and processes.

Measurement Process

Formulation. The derivation of software measures and metrics appropriate for the representation of the software that is being considered.
Collection. The mechanism used to accumulate data required to derive the formulated metrics.
Analysis. The computation of metrics and the application of mathematical tools.
Interpretation. The evaluation of metrics results in an efort to gain insight into the quality of the representation.
Feedback. Recommendations derived from the interpretation of product metrics transmitted to the software team.

- FUNCTION POINT:

The *function point* (FP) *metric* can be used effectively as a means for measuring the
functionality delivered by a system.4 Using historical data, the FP metric can then be

used to (1) estimate the cost or effort required to design, code, and test the software;
(2) predict the number of errors that will be encountered during testing; and (3) forecast
the number of components and/or the number of projected source lines in the implemented system.

Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and qualitative assessments of software complexity. Information domain values are defined in the following manner:

Number of external inputs (EIs). Each *external input* originates from a user or is transmitted from another application and provides distinct application-oriented data or control information. Inputs are often used to update *internal logical files* (ILFs). Inputs should be distinguished from inquiries, which are counted separately.

Number of external outputs (EOs). Each *external output* is derived data within the application that provides information to the user. In this context external output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

Number of external inquiries (EQs). An *external inquiry* is defined as an online input that results in the generation of some immediate software response in the form of an online output (often retrieved from an ILF).

Number of internal logical files (ILFs). Each *internal logical file* is a logical grouping of data that resides within the application's boundary and is maintained via external inputs.

Number of external interface files (EIFs). Each *external interface file* is a logical grouping of data that resides external to the application but provides information that may be of use to the application.

Once these data have been collected, the table in Figure 23.1 is completed and a complexity
value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average,
or complex. Nonetheless, the determination of complexity is somewhat subjective.

To compute function points (FP), the following relationship is used:

FP = count total *[0.65 + 0.01 * sigma(*Fi*)] (23.1)
where count total is the sum of all FP entries obtained from Figure23.1

FIGURE 23.1 Computing function points

Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)		×	3	4	6	=	
External Outputs (EOs)		×	4	5	7	=	
External Inquiries (EQs)		×	3	4	6	=	
Internal Logical Files (ILFs)		×	7	10	15	=	
External Interface Files (EIFs)		×	5	7	10	=	
Count total	→						

The *Fi* (*i* _ 1 to 14) are *value adjustment factors* (VAF) based on responses to the following questions [Lon02]:

1. Does the system require reliable backup and recovery?
2. Are specialized data communications required to transfer information to or from the application?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require online data entry?
7. Does the online data entry require the input transaction to be built over multiple screens or operations?
8. Are the ILFs updated online?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential). The constant values in Equation (23.1) and the weighting factors that are applied to information domain counts are determined empirically.

----------x---------x----------

7Q)QUALITY:

Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

RELIABILITY:

Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time".

SQA PLAN:

The *SQA Plan* provides a road map for instituting software quality assurance. Developed by the SQA group (or by the software team if an SQA group does not exist), the plan serves as a template for SQA activities that are instituted for each software project. A standard for SQA plans has been published by the IEEE [IEE93]. The standard recommends a structure that identifies: (1) the purpose and scope of the plan, (2) a description of all software engineering work products (e.g., models, documents, source code) that fall within the purview of SQA, (3) all applicable standards and practices that are applied during the software process, (4) SQA actions and tasks (including reviews and audits) and their placement throughout the software
process, (5) the tools and methods that support SQA actions and tasks, (6) software configuration management (Chapter 22) procedures, (7) methods for assembling, safeguarding, and maintaining all SQA-related records, and (8) organizational roles and responsibilities relative to product quality.

ISO-9000:

A *quality assurance system* may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management [ANS87]. Quality assurance systems are created to help organizations ensure their products and services satisfy customer expectations by meeting their specifications.These systems cover a wide variety of activities encompassing a product's entire life cycle including planning, controlling measuring, testing and reporting, and improving quality levels throughout the development and manufacturing process. ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services

offered.To become registered to one of the quality assurance system models contained in ISO 9000, a company's quality system and operations are scrutinized by third-party auditors for compliance to the standard and for effective operation. Upon successful registration,a company is issued a certificate from a registration body represented by the auditors.Semiannual surveillance audits ensure continued compliance to the standard.

SIX SIGMA:

Six Sigma is the most widely used strategy for statistical quality assurance in industry
today. Originally popularized by Motorola in the 1980s, the Six Sigma strategy "is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company's operational performance by identifying and eliminating defects' in manufacturing and service-related processes" [ISI08]. The term Six Sigma is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard. The Six Sigma methodology defines three core steps:

• *Define* customer requirements and deliverables and project goals via well defined methods of customer communication.
• *Measure* the existing process and its output to determine current quality performance (collect defect metrics).
• *Analyze* defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps:

• *Improve* the process by eliminating the root causes of defects.
• *Control* the process to ensure that future work does not reintroduce the causes of defects.

These core and additional steps are sometimes referred to as the DMAIC (define, measure, analyze, improve, and control) method.

If an organization is developing a software process (rather than improving an existing process), the core steps are augmented as follows:

• *Design* the process to (1) avoid the root causes of defects and (2) to meet customer requirements.
• *Verify* that the process model will, in fact, avoid defects and meet customer requirements.

This variation is sometimes called the DMADV (define, measure, analyze, design, and verify) method.
A comprehensive discussion of Six Sigma is best left to resources dedicated to the subject.

CAPABILITY MATURITY MODEL:

A *maturity model* is applied within the context of an SPI framework. The intent of the maturity model is to provide an overall indication of the "process maturity" exhibited by a software organization.

There are five levels of maturity:

Level 5, Optimized—The organization has quantitative feedback systems in place to identify process weaknesses and strengthen them pro-actively. Project teams analyze defects to determine their causes; software processes are evaluated and updated to prevent known types of defects from recurring.

Level 4, Managed—Detailed software process and product quality metrics establish the quantitative evaluation foundation. Meaningful variations in process performance can be distinguished from random noise, and trends in process and product qualities can be predicted.

Level 3, Defined—Processes for management and engineering are documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing software.

Level 2, Repeatable—Basic project management processes are established to track cost, schedule, and functionality. Planning and managing new products is based on experience with similar projects.

Level 1, Initial—Few processes are defined, and success depends more on individual
heroic efforts than on following a process and using a synergistic team effort.

CAPABILITY MATURITY MODEL INTEGRATION:
the *Capability Maturity Model Integration* (CMMI) [CMM07], a comprehensive process
meta-model that is predicated on a set of system and software engineering capabilities
that should be present as organizations reach different levels of process capability and maturity.
The CMMI represents a process meta-model in two different ways: (1) as a "continuous" model and (2) as a "staged" model. The continuous CMMI metamodel
describes a process in two dimensions as illustrated in Figure 30.2. Each
process area (e.g., project planning or requirements management) is formally assessed against specific goals and practices and is rated according to the following capability levels:
Level 0: *Incomplete*—the process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability for the process area.
Level 1: *Performed*—all of the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.
Level 2: *Managed*—all capability level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are "monitored, controlled, and reviewed; and are evaluated for adherence to the process description" [CMM07].
Level 3: *Defined*—all capability level 2 criteria have been achieved. In addition, the process is "tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets" [CMM07].

Level 4: *Quantitatively managed*—all capability level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as criteria in managing the process" [CMM07].
Level 5: *Optimized*—all capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.
The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals. *Specific goals* establish the characteristics that must exist if the activities implied by a process area are to be effective. *Specific practices* refine a goal into a set of process-related activities.

PEOPLE CAPABILITY MATURITY MODEL:
The *People Capability Maturity Model* "is a roadmap for implementing workforce practices that continuously improve the capability of an organization's workforce". Like the CMM, CMMI, and related SPI frameworks, the People CMM defines a set of five organizational maturity levels that provide an indication of the relative sophistication of workforce practices and processes. These maturity levels [CMM08] are tied to the existence (within an organization) of a set of key process areas (KPAs).

Level	Focus	Process Areas
Optimized	*Continuous improvement*	Continuous workforce innovation Organizational performance alignment Continuous capability improvement
Predictable	*Quantifies and manages knowledge, skills, and abilities*	Mentoring Organizational capability management Quantitative performance management Competency-based assets Empowered workgroups Competency integration
Defined	*Identifies and develops knowledge, skills, and abilities*	Participatory culture Workgroup development Competency-based practices Career development Competency development Workforce planning Competency analysis
Managed	*Repeatable, basic people management practices*	Compensation Training and development Performance management Work environment Communication and co-ordination Staffing
Initial	*Inconsistent practices*	

The People CMM complements any SPI framework by encouraging an organization
to nurture and improve its most important asset—its people. As important, it establishes a workforce atmosphere that enables a software organization to "attract, develop, and retain outstanding talent".

SOFTWARE MATURITY INDEX(SMI):

A *software maturity index* (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:

MT =number of modules in the current release

Fc =number of modules in the current release that have been changed

Fa = number of modules in the current release that have been added

Fd =number of modules from the preceding release that were deleted in the Current release The software maturity index is computed in the following manner:

SMI =MT - (Fa+Fc+Fd)/MT

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as a metric for planning software maintenance activities. The mean time to produce a release of a software product can be correlated with SMI, and empirical models for maintenance effort can be developed.

------X----------X--------