SIDDHARTH INSTITUTE OF ENGINEERING & TECHNOLOGY :: PUTTUR
(AUTONOMOUS)
Siddharth Nagar, Narayanavanam Road – 517583

QUESTION BANK (DESCRIPTIVE)

Subject with Code: Software Engineering (20CS0518)	**Course & Branch:** B.Tech – CSE, CSM, CIC
Year & Sem : III B.Tech & I-Sem	**Regulation :** R20

UNIT –I

INTRODUCTION AND INTRODUCTION TO AGILITY

1	**a**	Define Software and describe the characteristics of software.	**[L2][CO1]**	**[6M]**

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

  IEEE defines software engineering as:
  (1) The application of a systematic, disciplined, quantifiable approach to the development of software.
  (2) The study of approaches as in the above statement.

Characteristics of software

1) Software is developed or engineered; it is not manufactured in the Classical Sense.
2) Software doesn't "Wear Out"
3) Although the industry is moving toward component-based construction, most software continues to be custom built.

	b	Write in detail about the nature of software.	**[L1][CO1]**	**[6M]**

Software takes Dual role of Software. It is a **Product** and at the same time a **Vehicle for delivering a product**.
Software delivers the most important product of our time is called **information**

Defining Software

Software is defined as

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

2	**a**	What is Software Process? Distinguish any two Process Models.	**[L4][CO2]**	**[6M]**

A ***process*** is a collection of **activities, actions, and tasks** that are performed when some work product is to be created.

An ***activity*** strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied.

An ***action*** encompasses a set of tasks that produce a major work product (e.g., an architectural design model).

A ***task*** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

A ***process framework*** establishes the foundation for a complete software engineering process by identifying a small number of ***framework activities*** that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of ***umbrella activities*** that are applicable across the entire software process.

A generic process framework for software engineering encompasses **five** activities:

- **Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer. The intent is to understand stakeholders objectives for the project and to gather requirements that help define software features and functions.
- **Planning.** Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a *software project plan*—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.
- **Modeling.** Creation of models to help developers and customers understand the requires and software design
- **Construction.** This activity combines code generation and the testing that is required to uncover errors in the code.
- **Deployment.** The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These **five** generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems.

	b	How Process framework is created for software? Explain.	**[L2][CO1]**	**[6M]**

A ***process framework*** establishes the foundation for a complete software engineering process by identifying a small number of ***framework activities*** that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of ***umbrella activities*** that are applicable across the entire software process.

A generic process framework for software engineering encompasses **five** activities:

- Communication.
- Planning
- Modeling.
- Construction.
- Deployment.

These **five** generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems.

Framework activities helps to solve a problem using umbrella Activities:

Software project tracking and control—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

Risk management—assesses risks that may affect the outcome of the project or the quality of the product.

Software quality assurance—defines and conducts the activities required to ensure software quality.

Technical reviews—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

Measurement—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders needs; can be used in conjunction with all other framework and umbrella activities.

Software configuration management—manages the effects of change throughout the software process.

3	Discuss briefly about different types of Software Myths.	**[L2][CO1]**	**[12M]**

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

Management Myths:

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

Reality:

1) The book of standards may very well exist, but is it used?
2) Are software practitioners aware of its existence?
3) Does it reflect modern software engineering practice?
4) Is it complete?
5) Is it adaptable?

Myth: *If we get behind schedule, we can add more programmers and catch up*
Reality: Software development is not a mechanistic process like manufacturing. "Adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort.
Myth: *If we decide to outsource the software project to a third party, I can just relax and let that firm build it.*
Reality: If an organization does not understand how to manage and control software project internally, it will invariably struggle when it out sources software project.

Customer Myths

Myth : *A general statement of objectives is sufficient to begin writing programs - we can fill in details later.*
Reality : Although a comprehensive and stable statement of requirements is not always possible, an ambiguous statement of objectives is a recipe for disaster. Unambiguous requirements are developed only through effective and continuous communication between customer and developer.
Myth : *Project requirements continually change, but change can be easily accommodated because software is flexible.*
Reality : It's true that software requirement change, but the impact of change varies with the time at which it is introduced. When requirement changes are requested early, cost impact is relatively small. However, as time passes, cost impact grows rapidly – resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Practitioner's myths.

Myths that are still believed by software practitioners have been fostered by 50 years of programming culture. During the early days of software, programming was viewed as an art form. Old ways and attitudes die hard.
Myth: *Once we write the program and get it to work, our job is done.*
Reality: Someone once said that "the sooner you begin 'writing code', the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended

after it is delivered to the customer for the first time.

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

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

Myth: *The only deliverable work product for a successful project is the working program.* ***Reality:*** A working program is only one part of a *software configuration* that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.

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

Reality: Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times. Many software professionals recognize the fallacy of the myths just described.

4	**a**	How umbrella activities help in solving a software problem? Explain.	**[L2][CO1]**	**[6M]**

U**mbrella Activities**. In general, **umbrella activities** are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

Framework activities helps to solve a problem using umbrella Activities:

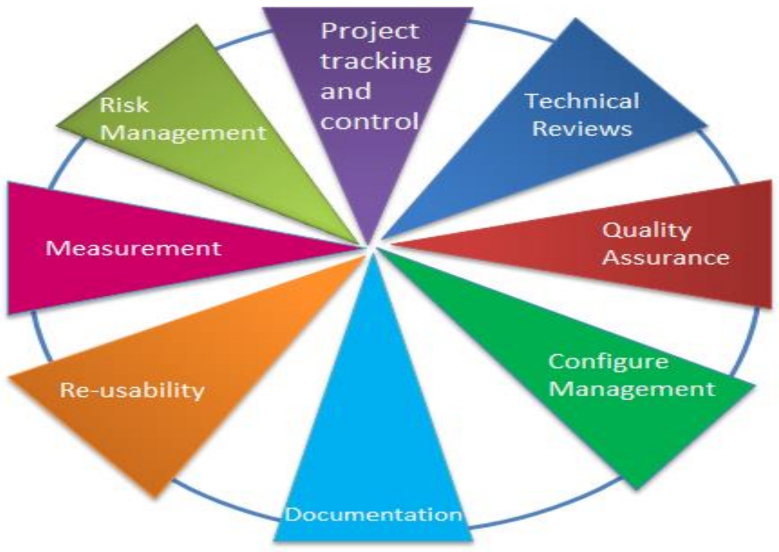


Fig: Umbrella Activities in Software Engineering

- **Software project tracking and control**—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.
- **Risk management**—assesses risks that may affect the outcome of the project or the quality of the product.
- **Software quality assurance**—defines and conducts the activities required to ensure software quality.
- **Technical reviews**—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.
- **Measurement**—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders needs; can be used in conjunction with all other framework and umbrella activities.
- **Software configuration management**—manages the effects of change throughout the software process.

	b	Write about process frame work activities.	**[L1][CO1]**	**[6M]**

A ***process framework*** establishes the foundation for a complete software engineering process by identifying a small number of ***framework activities*** that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of ***umbrella activities*** that are applicable across the entire software process.
A generic process framework for software engineering encompasses **five** activities:

- **Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer. The intent is to understand stakeholders objectives for the project and to gather requirements that help define software features and functions.
- **Planning.** Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a *software project plan*—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.
- **Modeling.** Creation of models to help developers and customers understand the requires and software design
- **Construction.** This activity combines code generation and the testing that is required to uncover errors in the code.
- **Deployment.** The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

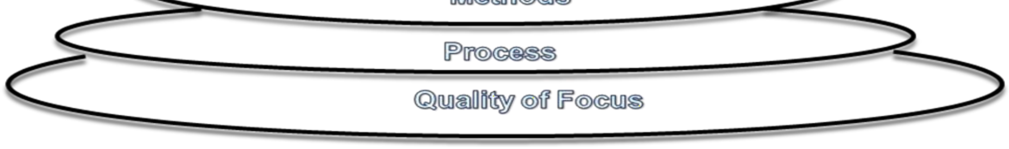
5	**a**	Define the term Software Engineering – A Layered Technology	**[L1][CO1]**	**[6M]**

In order to build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities

- **Problem should be understood before software solution is developed**
- **Design is a pivotal Software Engineering activity**
- **Software should exhibit high quality**
- **Software should be maintainable**

These simple realities lead to one conclusion. Software in all of its forms and across all of its application domains should be **engineered**.

Software Engineering is a **layered technology**. Software Engineering encompasses a **Process, Methods** for managing and engineering software and **tools**. The following Figure represents **Software engineering Layer**



Software engineering is a layered technology. Referring to above Figure, any engineering approach must rest on an organizational commitment to **quality**.
The bedrock that supports software engineering is a **quality focus**.

The foundation for software engineering is the ***process*** **layer**. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. **Process** defines a **framework** that must be established for effective delivery of software engineering technology.

Software engineering ***methods*** provide the technical **how-to's** for building software. **Methods** encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support.

Software engineering ***tools*** provide **automated or semi-automated** support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering,* is established.

	b	How principles of Software Engineering help in building a software	**[L2][CO1]**	**[6M]**

- Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.
- Software engineering is an engineering branch associated with development of software product using well-defined **scientific principles**, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

- **Fritz Bauer**, a German computer scientist, defines software engineering as:
- Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and work efficiently on real machine

The principles are:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self–organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

6	For what kind of software Spiral model is used? Examine in detail about it.	**[L3][CO1]**	**[12M]**

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

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

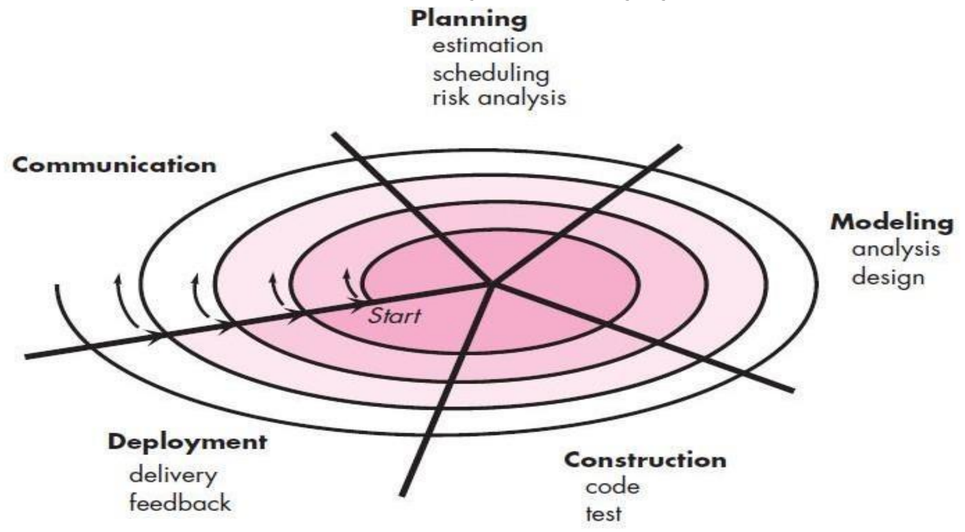


Fig : The Spiral Model

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

The first circuit around the spiral might result in the development of a **product** specification; subsequent passes around the spiral might be used to develop a **prototype** and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan.

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

7	**a**	What is an Iterative Model? How Iterative is best than classical life cycle model. Explain	**[L2][CO1]**	**[6M]**

An iterative process flow repeats one or more of the activities before proceeding to the next. An evolutionary process flow executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software.

Evolutionary models are **iterative**. They are characterized in a manner that enables you to develop increasingly more complete versions of the software with each iteration. There are **two** common evolutionary process models.

Prototyping Model : Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a ***prototyping paradigm*** may offer the best approach. Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models. The prototyping paradigm begins with **communication**. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned **quickly**, and **modeling** (in the form of a “quick design”) occurs. A **quick design** focuses on a representation of those aspects of the software that will be visible to end users. The quick design leads to the **construction of a prototype**. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements.

Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

The prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools that enable working programs to be generated quickly. The prototype can serve as **"the first system**."

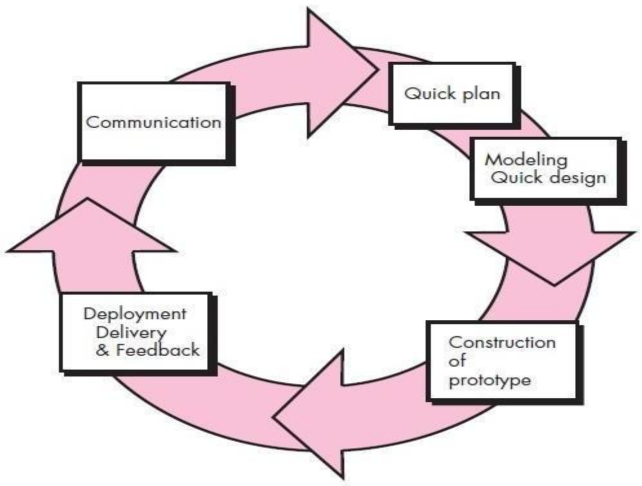


Fig : prototyping paradigm

Prototyping can be **problematic** for the following reasons:

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability.
2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability.

   Although problems can occur, prototyping can be an **effective paradigm** for software engineering.

	b	What is SDLC? How it is used in Software Development Process?	**[L2][CO2]**	**[6M]**

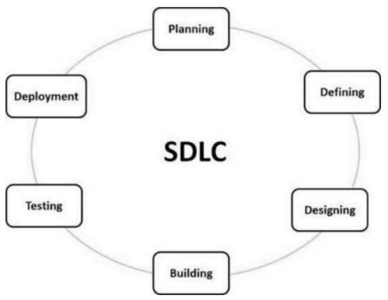
The Software Development Life Cycle (SDLC) is **a structured process that enables the production of high-quality, low-cost software, in the shortest possible production time**. The goal of the SDLC is to produce superior software that meets and exceeds all customer expectations and demands.

SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.

The following figure is a graphical representation of the various stages of a typical SDLC.

A typical Software Development Life Cycle consists of the following stages –

Stage 1: Planning and Requirement Analysis
Stage 2: Defining Requirements
Stage 3: Designing the Product Architecture
Stage 4: Building or Developing the Product
Stage 5: Testing the Product
Stage 6: Deployment in the Market and Maintenance



8	**a**	Dissect in brief about Unified Process Model with neat diagram.	**[L4][CO2]**	**[6M]**

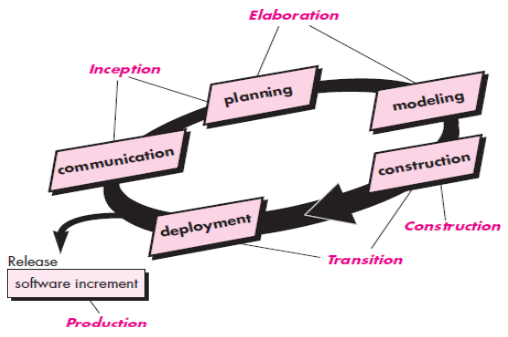
Unified process (UP) is an architecture-centric, use-case driven, iterative and incremental development process. UP is also referred to as the **unified software development process**.

The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models, but characterize them in a way that implements many of the best principles of **agile software development.** The Unified Process recognizes the importance of **customer communication** and streamlined methods for describing the customer's view of a system. It emphasizes the important role of software architecture and "helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse" . It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

Phases of the Unified Process

This process divides the development process into **five** phases:

- Inception
- Elaboration
- Conception
- Transition
- Production



The ***inception phase*** of the UP encompasses both customer communication and planning activities. By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is proposed; and a plan for the iterative, incremental nature of the ensuing project is developed.

The ***elaboration phase*** encompasses the communication and modeling activities of the generic process model. Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include **five different views** of the software—***the use case model, the requirements model, the design model, the implementation model, and the deployment model***. Elaboration creates an "**executable architectural baseline**" that represents a "**first cut**" executable system.

The ***construction phase*** of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users. To accomplish this, requirements and design models that were started during the elaboration phase are completed to reflect the final version of the software increment. All necessary and required features and functions for the software increment (i.e., the release) are then implemented in **source code**.

The ***transition phase*** of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity. Software is given to end users for **beta testing and user feedback** reports both defects and necessary changes. At the conclusion of the transition phase, the software increment becomes a usable software release.

The ***production phase*** of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

	b	Write about RAD Model.	**[L1][CO2]**	**[6M]**

RAD is a linear sequential software development process model that emphasizes a concise development cycle using an element based construction approach. If the requirements are well understood and described, and the project scope is a constraint, the RAD process enables a development team to create a fully functional system within a concise time period.

RAD (Rapid Application Development) is a concept that products can be developed faster and of higher quality through:

- Gathering requirements using workshops or focus groups
- Prototyping and early, reiterative user testing of designs
- The re-use of software components
- A rigidly paced schedule that refers design improvements to the next product version
- Less formality in reviews and other team communication

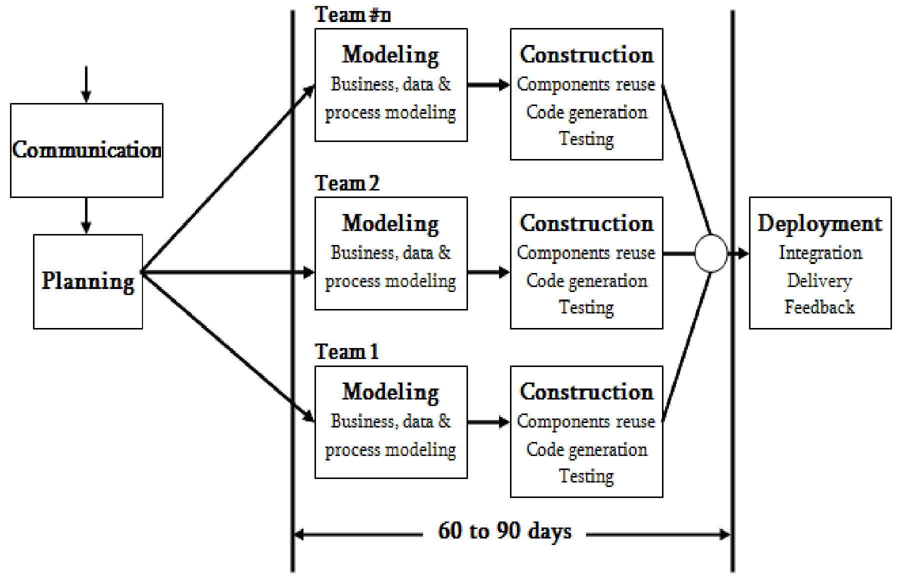


Figure : Flowchart of RAD model

RAD is a linear sequential software development process model that emphasizes a concise development cycle using an element based construction approach. If the requirements are well understood and described, and the project scope is a constraint, the RAD process enables a development team to create a fully functional system within a concise time period.

RAD (Rapid Application Development) is a concept that products can be developed faster and of higher quality through:

- Gathering requirements using workshops or focus groups
- Prototyping and early, reiterative user testing of designs
- The re-use of software components
- A rigidly paced schedule that refers design improvements to the next product version
- Less formality in reviews and other team communication

9	What is Agile Process? How Extreme Programming (XP) is an effective agile Model? Explain with neat sketch.	**[L2][CO2]**	**[12M]**

Any agile software process is characterized in a manner that addresses a number of key assumptions about the majority of software projects:

1. It is difficult to predict in advance which software requirements will persist and which will change.
2. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable as we might like.

Given these three assumptions, an important question arises: How do we create a process that can manage unpredictability?

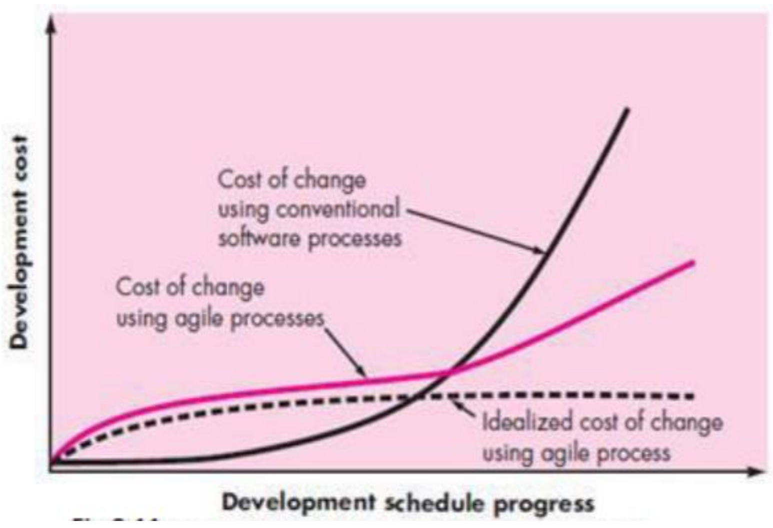
The answer, as I have already noted, lies in process **adaptability**. An agile process, therefore, must be adaptable.

AGILITY AND THE COST OF CHANGE

- The costs of doing this work are minimal, and the time required will not adversely affect the outcome of the project.
- The change requires a modification to the architectural design of the software, the design and

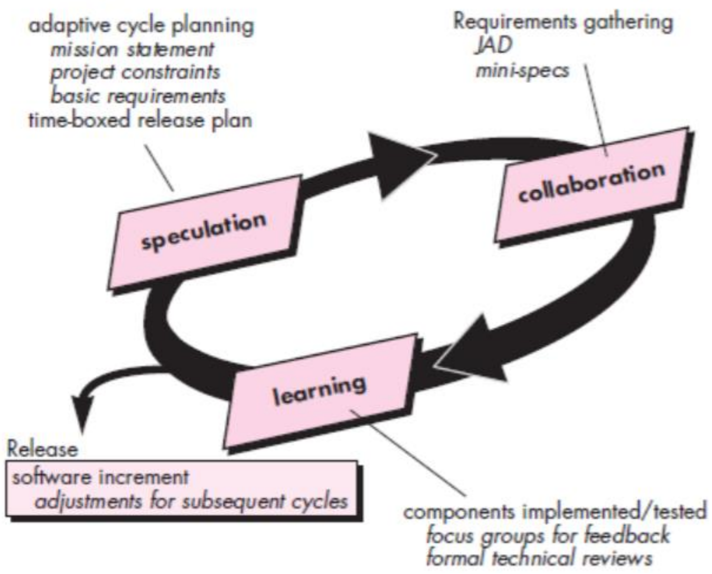
construction of three new components, modifications to another five components, the design of new tests, and so on.

- The time and cost required to ensure that the change is made without unintended side effects is nontrivial. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence to suggest that a significant reduction in the cost of change can be achieved.



The XP Process:

Extreme Programming uses an object-oriented approach and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure illustrates the XP process.



10	**a**	What is Agility? Illustrate any four Agile Process Models.	**[L3][CO2]**	**[6M]**

AGILITY

➔ An agile team is a nimble team able to appropriately respond to changes.
➔ Change is what software development is very much about.
➔ Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product.
➔ It emphasizes rapid delivery of software and de-emphasizes the importance of intermediate work products.
➔ Agility can be applied to any software process.
➔ However, to accomplish this, it is essential that the process be designed in a way that allows the project team to adapt tasks.

Extreme Programming (XP)

Extreme Programming (XP), the most widely used approach to agile software development.

XP Values:

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

Adaptive Software Development (ASD)

- Adaptive Software Development (ASD) has been proposed by Jim Highsmith as a technique for building complex software and systems. ASD focus on human collaboration and team self-organization. ASD can be defined as "life cycle" that incorporates three phases, speculation, collaboration, and learning.

Scrum

- Scrum is an agile software development method that was proposed by Jeff Sutherland in the early 1990s. Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks occur within a process pattern called a ***sprint***.

Dynamic Systems Development Method (DSDM)

The Dynamic Systems Development Method (DSDM) is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment". **principle**—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete (100 percent) application.

	b	Write a note on Agile Unified Process.	**[L1][CO2]**	**[6M]**

The Agile Unified Process (AUP) adopts a "serial in the large" and "iterative in the small" philosophy for building computer-based systems. By adopting the classic UP phased activities—inception, elaboration, construction, and transition—AUP provides a serial overlay that enables a team to visualize the overall process flow for a software project. Each AUP iteration addresses the following activities.

• ***Modeling.*** UML representations of the business and problem domains are created. However, to stay agile, these models should be "just barely good enough" to allow the
team to proceed.

• ***Implementation.*** Models are translated into source code.

• ***Testing.*** Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.

• ***Deployment.*** Like the generic process activity. Deployment in this context focuses on the delivery of a software increment and the acquisition of feedback from end users.

• ***Configuration and project management.*** In the context of AUP, configuration management addresses change management, risk management, and the control of any persistent work products that are produced by the team.

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

UNIT-II

1	Discuss about Requirement Engineering steps.	**[L4][CO1]**	**[12M]**

REQUIREMENTS ENGINEERING

Requirements analysis, also called requirements engineering, It is the process of understanding of requirements. Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system. It encompasses seven distinct tasks: inception, elicitation, elaboration, negotiation, specification, validation, and management.
Inception : It establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.
Elicitation: In this, ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day- to-day basis.
Problems of scope. The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
Problems of understanding. The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or un testable.
Problems of volatility. The requirements change over time.
Elaboration: The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information. Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system.

Negotiation: To negotiate the requirements of a system to be developed, it is necessary to identify conflicts and to resolve those conflicts. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

Specification: The term specification means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

Validation: Requirement's validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected.
The primary requirements validation mechanism is the technical review. The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies, conflicting requirements, or unrealistic requirements.
Requirements management: Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any

time as the project proceeds. Many of these activities are identical to the software configuration management (SCM) techniques.

2	**a**	Who is a stakeholder? In what way he/she is being used in Software Development Process.	**[L1][CO2]**	**[6M]**

Identifying Stakeholders

A *stakeholder* is anyone who has a direct interest in or benefits from the system that is to be developed. At inception, you should create a list of people who will contribute input as requirements are elicited.

Recognizing Multiple Viewpoints

Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. The information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another.

Working toward Collaboration

The job of a requirements engineer is to identify areas of commonality and areas of conflict or inconsistency. It is, of course, the latter category that presents a challenge. Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion"(e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

Asking the First Questions

Questions asked at the inception of the project should be "context free" . The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

The next set of questions enables you to gain a better understanding of the problem andallows the customer to voice his or her perceptions about a solution:

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication activity itself. Gause and Weinberg call these "meta-questions" and propose the following list:

- Are you the right person to answer these questions? Are your answers
- "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?

- Can anyone else provide additional information?
- Should I be asking you anything else?

These questions will help to "break the ice" and initiate the communication that is essential to successful elicitation.

	b	How to establish the groundwork for understanding of software requirements. Explain the steps in it.	**[L2][CO2]**	**[6M]**

ESTABLISHING THE GROUNDWORK

Identifying Stakeholders

A *stakeholder* is anyone who has a direct interest in or benefits from the system that is to be developed. At inception, you should create a list of people who will contribute input as requirements are elicited

Recognizing Multiple Viewpoints

Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. The information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another.

Working toward Collaboration

The job of a requirements engineer is to identify areas of commonality and areas of conflict or inconsistency. It is, of course, the latter category that presents a challenge. Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion"(e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

Asking the First Questions

Questions asked at the inception of the project should be "context free" . The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

The next set of questions enables you to gain a better understanding of the problem andallows the customer to voice his or her perceptions about a solution:

- How would you characterize "good" output that would be generated by a successful solution?

- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication activity itself. Gause and Weinberg call these “meta-questions” and propose the following list:

- Are you the right person to answer these questions? Are your answers
- “official”?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

These questions will help to “break the ice” and initiate the communication that is essential to successful elicitation.

3	**a**	Illustrate Eliciting Requirements and narrate the steps in it in detail.	**[L3][CO1]**	**[6M]**

ELICITING REQUIREMENTS

Requirements elicitation (also called *requirements gathering*) combines elements of problem solving, elaboration, negotiation, and specification

<u>Collaborative Requirements Gathering</u>

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

- Meetings are conducted and attended by both software engineers and other stakeholders.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A “facilitator” (can be a customer, a developer, or an outsider) controls the meeting.
- A “definition mechanism” (can be work sheets, flip charts, or wall stickers or
- an electronic bulletin board, chat room, or virtual forum) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

During inception basic questions and answers establish the scope of the problem and the overall perception of a solution. Out of these initial meetings, the developer and customers writea one- or two-page “product request.”

A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate. The product request is distributed to all attendees before the meeting date.

While reviewing the product request in the days before the meeting, each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of services that manipulate or interact with the objects. Finally, lists of constraints (e.g., cost, size, business rules) and performance criteria (e.g., speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system.

The lists of objects can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive-backed sheets, or written on a wall board. After individual lists are presented in one topic area, the group creates a combined list by eliminating redundant entries, adding any new ideas that come up during the discussion, but not deleting anything.

Quality Function Deployment

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD "concentrates on maximizing customer satisfaction from the software engineering process". To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.

QFD identifies three types of requirements :

- Normal requirements. The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.
- Expected requirements. These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction.
- Exciting requirements. These features go beyond the customer's expectations and proveto be very satisfying when present.

Usage Scenarios

As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a threadof usage for the system to be constructed. The scenarios, often called *use cases*, provide a description of how the system will be used.

Elicitation Work Products

The work products produced as a consequence of requirements elicitation will vary depending onthe size of the system or product to be built. For most systems, the work products include

- A statement of need and feasibility.

- A bounded statement of scope for the system or product.
- A list of customers, users, and other stakeholders who participated in requirements elicitation.
- A description of the system's technical environment.
- A list of requirements and the domain constraints that apply to each.
- A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
- Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in requirementselicitation.

	b	What is Functional and Non-Functional Requirements? How is collected and differentiated. Explain	**[L2][CO1]**	**[6M]**

Requirements analysis is very critical process that enables the success of a system or software project to be assessed. Requirements are generally split into two types: *Functional* and *Non-functional requirements*.

Functional Requirements: These are the requirements that the end user specifically demands as basic facilities that the system should offer. All these functionalities need to be necessarily incorporated into the system as a part of the contract. These are represented or stated in the form of input to be given to the system, the operation performed and the output expected. They are basically the requirements stated by the user which one can see directly in the final product, unlike the non-functional requirements.

Non-functional requirements: These are basically the quality constraints that the system must satisfy according to the project contract. The priority or extent to which these factors are implemented varies from one project to other. They are also called non-behavioral requirements.
They basically deal with issues like:

- Portability
- Security
- Maintainability
- Reliability
- Scalability
- Performance
- Reusability
- Flexibility

Following are the differences between Functional and Non Functional Requirements

Functional Requirements	Non Functional Requirements
A functional requirement defines a system or its component.	A non-functional requirement defines the quality attribute of a software system.
It specifies "What should the software system do?"	It places constraints on "How should the software system fulfill the functional requirements?"
Functional requirement is specified by User.	Non-functional requirement is specified by technical peoples e.g. Architect, Technical leaders and software developers.
It is mandatory.	It is not mandatory.

It is captured in use case.	It is captured as a quality attribute.
Defined at a component level.	Applied to a system as a whole.
Helps you verify the functionality of the software.	Helps you to verify the performance of the software.
Functional Testing like System, Integration, End to End, API testing, etc are done.	Non-Functional Testing like Performance, Stress, Usability, Security testing, etc are done.
Usually easy to define.	Usually more difficult to define.
Example **1)** Authentication of user whenever he/she logs into the system. **2)** System shutdown in case of a cyber attack. **3)** A Verification email is sent to user whenever he/she registers for the first time on some software system.	**Example** **1)** Emails should be sent with a latency of no greater than 12 hours from such an activity. **2)** The processing of each request should be done within 10 seconds **3)** The site should load in 3 seconds when the number of simultaneous users are > 10000

4	**a**	How Use-Case are developed from collected requirements. Devise with an example of use-case diagram.	**[L4][CO3]**	**[6M]**

DEVELOPING USE CASES

Use cases are defined from an actor's point of view. An actor is a role that people (users) or devices play as they interact with the software.

The first step in writing a use case is to define the set of "actors" that will be involved in the story. *Actors* are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described.

Actors represent the roles that people (or devices) play as the system operates. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case. Different people may play the role of each actor.

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors during the first iteration and secondary actors as more is learned about the system.

Primary actors interact to achieve required system function and derive the intended benefit from the system. *Secondary actors* support the system so that primary actors can do their work. Once actors have been identified, use cases can be developed.

Jacobson suggests a number of questions that should be answered by a use case:

- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?

- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

The basic use case presents a high-level story that describes the interaction between the actor andthe system.

Ex:

For SafeHome project requirements, we define four actors: homeowner (a user), setup manager (likely the same person as homeowner, but playing a different role), sensors (devices attached to the system), and the monitoring and response subsystem (the central station that monitors the *SafeHome* home security function). For the purposes of this example, we consider only the homeowner actor. The homeowner actor interacts with the home security function in a number of different ways using either the alarm control panel or a PC:

- Enters a password to allow all other interactions.
- Inquires about the status of a security zone.
- Inquires about the status of a sensor.
- Presses the panic button in an emergency.
- Activates/deactivates the security system.

Use case: *InitiateMonitoring*
Primary actor: Home owner.
Goal in context: To set the system to monitor sensors when the homeowner leaves the house or remains inside.
Preconditions: System has been programmed for a password and to recognize various sensors.
Trigger: The homeowner decides to "set" the system, i.e., to turn on the
alarm functions.
Scenario:
1. Homeowner: observes control panel
2. Homeowner: enters password
3. Homeowner: selects "stay" or "away"
4. Homeowner: observes read alarm light to indicate that *SafeHome* has been armed
Exceptions:
1. Control panel is *not ready:* homeowner checks all sensors to determine which are open; closes them.
2. Password is incorrect (control panel beeps once): homeowner re enters correct password.
3. Password not recognized: monitoring and response subsystem must be contacted to reprogram password.
4. *Stay* is selected: control panel beeps twice and a *stay* light is lit; perimeter sensors are activated.
5. *Away* is selected: control panel beeps three times and an *away* light is lit; all sensors are activated.
Priority: Essential, must be implemented

When available: First increment

Frequency of use: Many times per day

Channel to actor: Via control panel interface

Secondary actors: Support technician, sensors

Channels to secondary actors:

Support technician: phone line

Sensors: hardwired and radio frequency interfaces

Open issues:

1. Should there be a way to activate the system without the use of a password or with an abbreviated password?
2. Should the control panel display additional text messages?
3. How much time does the homeowner have to enter the password from the time the first key is pressed?
4. Is there a way to deactivate the system before it actually activates?

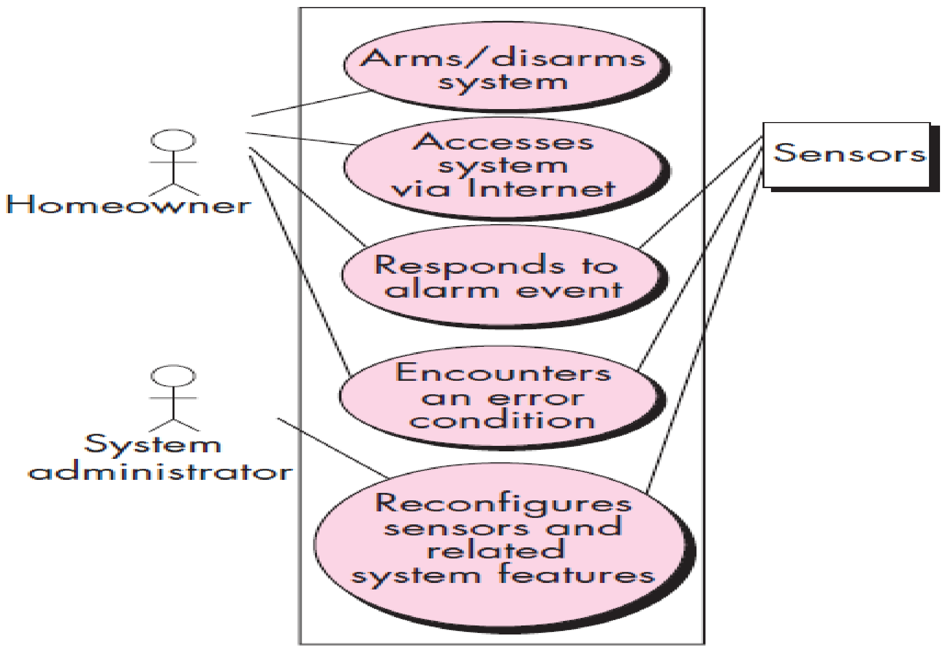

Arms/disarms system
Accesses system via Internet
Homeowner
Responds to alarm event
Sensors
Encounters an error condition
System administrator
Reconfigures sensors and related system features

Fig: UML use case diagram for *SafeHome* home security function

	b	What are the elements in Requirement Model. How it helps in Analyzing the Requirements?	**[L2][CO1]**	**[6M]**

BUILDING THE REQUIREMENTS MODEL

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as you learn more about the system to be built, and other stakeholders understand more about what they really require..

5.1 Elements of the Requirements Model

The specific elements of the requirements model are dictated by the analysis modeling methodthat is to be used. However, a set of generic elements is common to most requirements models.

- Scenario-based elements. The system is described from the user's point of view using a scenario-based approach.
- Class-based elements. Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors.
- Behavioral elements. The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior.
- Flow-oriented elements. Information is transformed as it flows through a computer- based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms.

5.2 Analysis Patterns

Analysis patterns suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

Two benefits that can be associated with the use ofanalysis patterns:

First, analysis patterns speed up the development of abstract analysis models that capture the main requirements of the concrete problem by providing reusable analysis models with examplesas well as a description of advantages and limitations.

Second, analysis patterns facilitate the transformation of the analysis model into a design modelby suggesting design patterns and reliable solutions for common problems.

Analysis patterns are integrated into the analysis model by reference to the pattern name

5	**a**	Why Requirement Negotiation is important? Discuss in detail	**[L4][CO1]**	**[6M]**

The intent of negotiation is to develop a project plan that meets stakeholder needs whileat the same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team. The best negotiations strive for a "win-win" result. That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you win by working to realistic and achievable budgets and deadlines.

Boehm defines a set of negotiation activities at the beginning of each software process iteration. Rather than a single customer communication activity, the following activities are defined:

1. Identification of the system or subsystem's key stakeholders.
2. Determination of the stakeholders' "win conditions."
3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned.

Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to subsequent software engineering activities.

	b	What kind of questions were addressed by Requirement team while validating the requirements?	**[L1][CO1]**	**[6M]**

VALIDATING REQUIREMENTS

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. The requirements represented by the model are prioritized by the stakeholders and grouped within requirements packages that will be implemented as software increments.

A review of the requirements model addresses the following questions:

- Is each requirement consistent with the overall objectives for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?
- Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
- Have requirements patterns been used to simplify the requirements model?
- Have all patterns been properly validated? Are all patterns consistent with customer requirements?

These and other questions should be asked and answered to ensure that the requirements model is an accurate reflection of stakeholder needs and that it provides a solid foundation for design.

6	**a**	What is the need of Requirements Analysis and how it is done? Explain the steps in it.	**[L2][CO1]**	**[6M]**

REQUIREMENTS ANALYSIS

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering.

The requirements modeling action results in one or more of the following types of models:

- *Scenario-based models* of requirements from the point of view of various system "actors"
- *Data models* that depict the information domain for the problem
- *Class-oriented models* that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements
- *Flow-oriented models* that represent the functional elements of the system and how they transform data as it moves through the system
- *Behavioral models* that depict how the software behaves as a consequence of external "events"

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model provides the developer and the customer with the means to assess quality once software is built.

Throughout requirements modeling, primary focus is on *what*, not *how*. What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined, and what constraints apply?

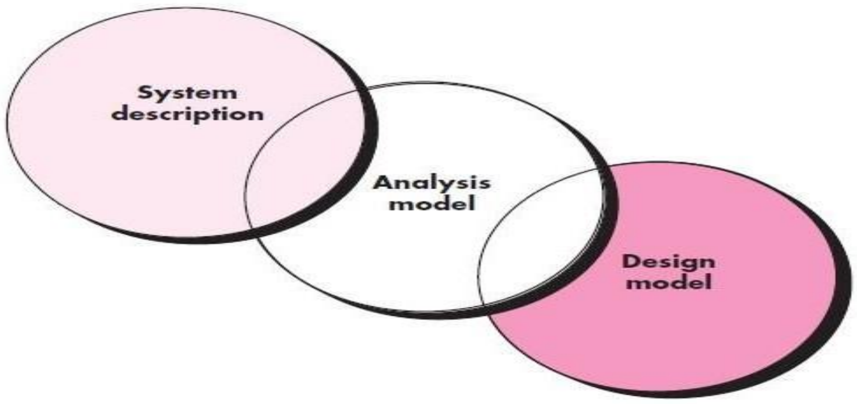


Fig : The requirements model as a bridge between the system description and the design model

	b	Discuss Domain analysis in detail with a neat sketch.	[L3][CO3]	[6M]

Domain Analysis

Domain analysis doesn't look at a specific application, but rather at the domain in whichthe application resides.

The "specific application domain" can range from avionics to banking, from multimediavideo games to software embedded within medical devices. The goal of domain analysis is straightforward: to identify common problem solving elements that are applicable to all applications within the domain, to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.

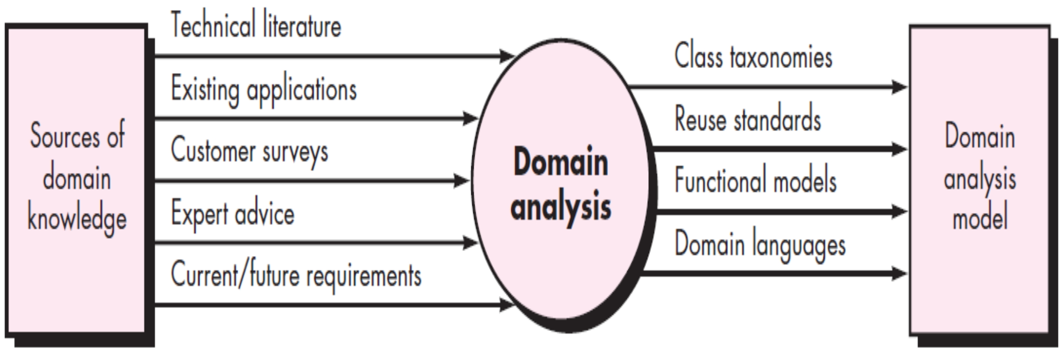


Fig: Input and output for domain analysis

7	**a**	Justify the approaches in Requirements Modeling with diagram	**[L5][CO1]**	**[6M]**

Requirements Modeling Approaches

One view of requirements modeling, called *structured analysis,* considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their *attributes and relationships*.

A second approach to analysis modeling, called *object-oriented analysis,* focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process are predominantly object oriented.

Each element of the requirements model is represented in following figure presents the problem from a different point of view.

Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships between the objects, and the collaborations that occur between the classes that are defined.

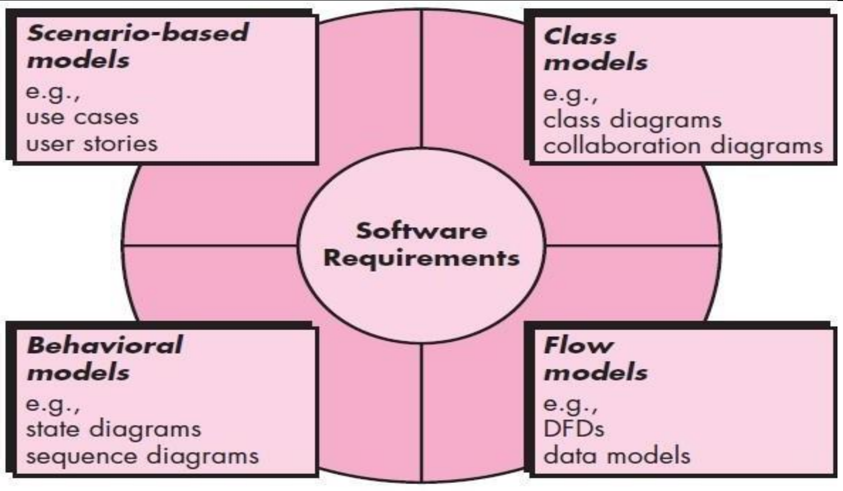


Fig : Elements of the analysis model

Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally,

Flow-oriented elements represent the system as an information transform, depicting howdata objects_are transformed as they flow through various system functions.

	b	Differentiate Behavioral Model Vs Structural Model	**[L4][CO3]**	**[6M]**

The difference between structural models and behavioral models are:-
Structural Models

Structural models of software display the organization of a system in terms of the components and their relationships.
Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executed.
Structural models are created when discussing and designing the system architecture.

Behavioral Models

Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
You can think of these stimuli as being of two types:
•A. Data Some data arrives that has to be processed by the system.
•B. Events Some event happens that triggers system processing. Events may have associated data, although this is not always the case.
A sequence diagram and State Transition Diagram may be used for the behavioral model.

8	**a**	What is Scenario-Based Modeling? Devise with an example	**[L4][CO3]**	**[6M]**

SCENARIO-BASED MODELING

Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

Creating a Preliminary Use Case

A use case describes a specific usage scenario in straightforward language from the point of view of a defined actor. These are the questions that must be answered if use cases are to providevalue as a requirements modeling tool.

(1) what to write about,

(2) how much to write about it,

(3) how detailed to make your description, and

(4) how to organize the description?

To begin developing a set of use cases, list the functions or activities performed by a specific actor.

Ex: Consider Safehome project

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the “surveillance” from the major function buttons.
6. The homeowner selects “pick a camera.”
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the “view” button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Refining a Preliminary Use Case

Each step in the primary scenario is evaluated by asking the following questions:

- *Can the actor take some other action at this point?*
- *Is it possible that the actor will encounter some error condition at this point?* If so, what might it be?
- *Is it possible that the actor will encounter some other behavior at this point (e.g.,behavior that is invoked by some event outside the actor’s control)?* If so, what might it be?

In addition to the three generic questions suggested earlier in this section, the following issues should also be explored:

- *Are there cases in which some “validation function” occurs during this use case?* This implies that validation function is invoked and a potential error condition might occur.
- *Are there cases in which a supporting function (or actor) will fail to respond appropriately?* For example, a user action awaits a response but the function that is to

respond times out.

- *Can poor system performance result in unexpected or improper user actions?* For example, a Web-based interface responds too slowly, resulting in a user making multiple selects on a processing button. These selects queue inappropriately and ultimately generate an error condition.

Writing a Formal Use Case

The typical outline for formal use cases can be in following manner

- The *goal in context* identifies the overall scope of the use case.
- The *precondition* describes what is known to be true before the use case is initiated.
- The *trigger* identifies the event or condition that "gets the use case started"
- The *scenario* lists the specific actions that are required by the actor and the appropriate system responses.
- *Exceptions* identify the situations uncovered as the preliminary use case is refined

Ex:

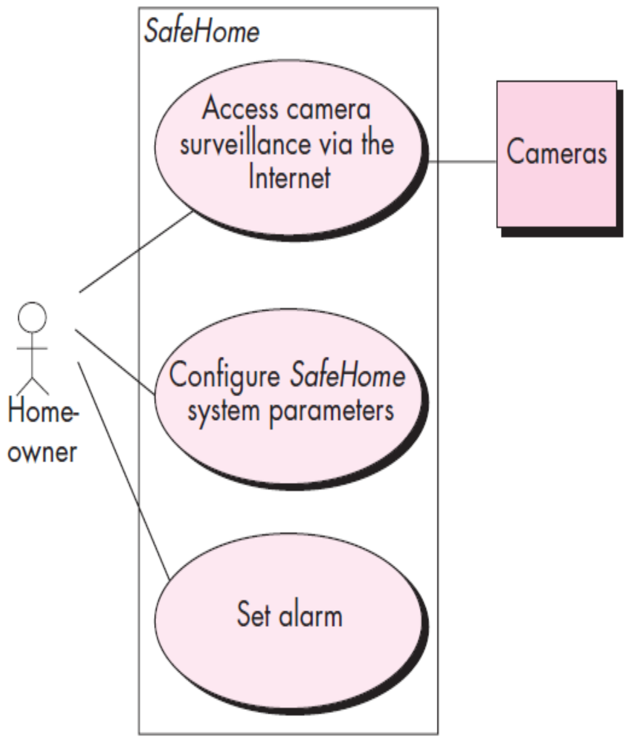

SafeHome
Access camera surveillance via the Internet
Cameras
Configure SafeHome system parameters
Home-owner
Set alarm

Fig: Preliminary use-case diagram for the *SafeHome* system

	b	Discuss about "Establishing the groundwork" steps involved in inception phase of requirement engineering.	**[L2][CO3]**	**[6M]**

ESTABLISHING THE GROUNDWORK

Identifying Stakeholders

A *stakeholder* is anyone who has a direct interest in or benefits from the system that is to be developed. At inception, you should create a list of people who will contribute input as requirements are elicited.

Recognizing Multiple Viewpoints

Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. The information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another.

Working toward Collaboration

The job of a requirements engineer is to identify areas of commonality and areas of conflict or inconsistency. It is, of course, the latter category that presents a challenge. Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion"(e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

Asking the First Questions

Questions asked at the inception of the project should be "context free" . The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

The next set of questions enables you to gain a better understanding of the problem andallows the customer to voice his or her perceptions about a solution:

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication activity itself. Gause and Weinberg call these "meta-questions" and propose the following list:

- Are you the right person to answer these questions? Are your answers
- "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?

- Should I be asking you anything else?

These questions will help to “break the ice” and initiate the communication that is essential to successful elicitation.

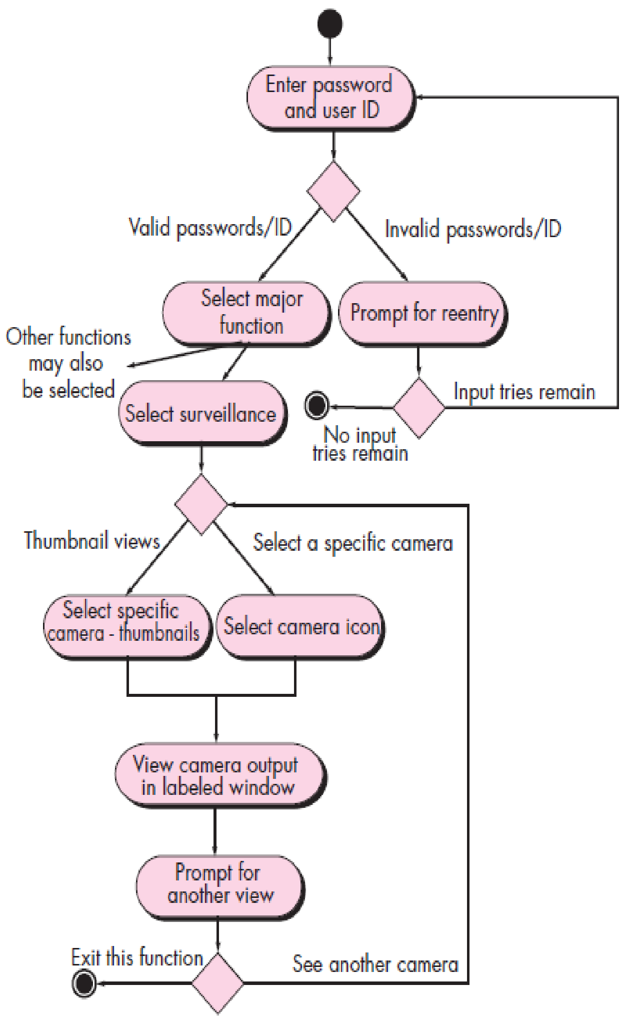
9	**a**	What are all the UML Models that supplement the Use-case diagram? Explain.	**[L2][CO3]**	**[6M]**

UML MODELS THAT SUPPLEMENT THE USE CASE

Developing an Activity Diagram

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart, an activity diagram uses rounded rectangles to imply a specific system function, arrows to represent flow through thesystem, decision diamonds to depict a branching decision (each arrow emanating from the diamond is labeled), and solid horizontal lines to indicate that parallel activities are occurring. i.eA UML activity diagram represents the actions and decisions that occur as some function is performed.

Ex:



10.2 Swimlane Diagrams

The UML *swimlane diagram* is a useful variation of the activity diagram and allows you to represent the flow of activities described by the use case and at the same time indicate which actor or analysis class has responsibility for the action described by an activity rectangle.Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

The following figure represents *swimlane diagram for SafeHome System*

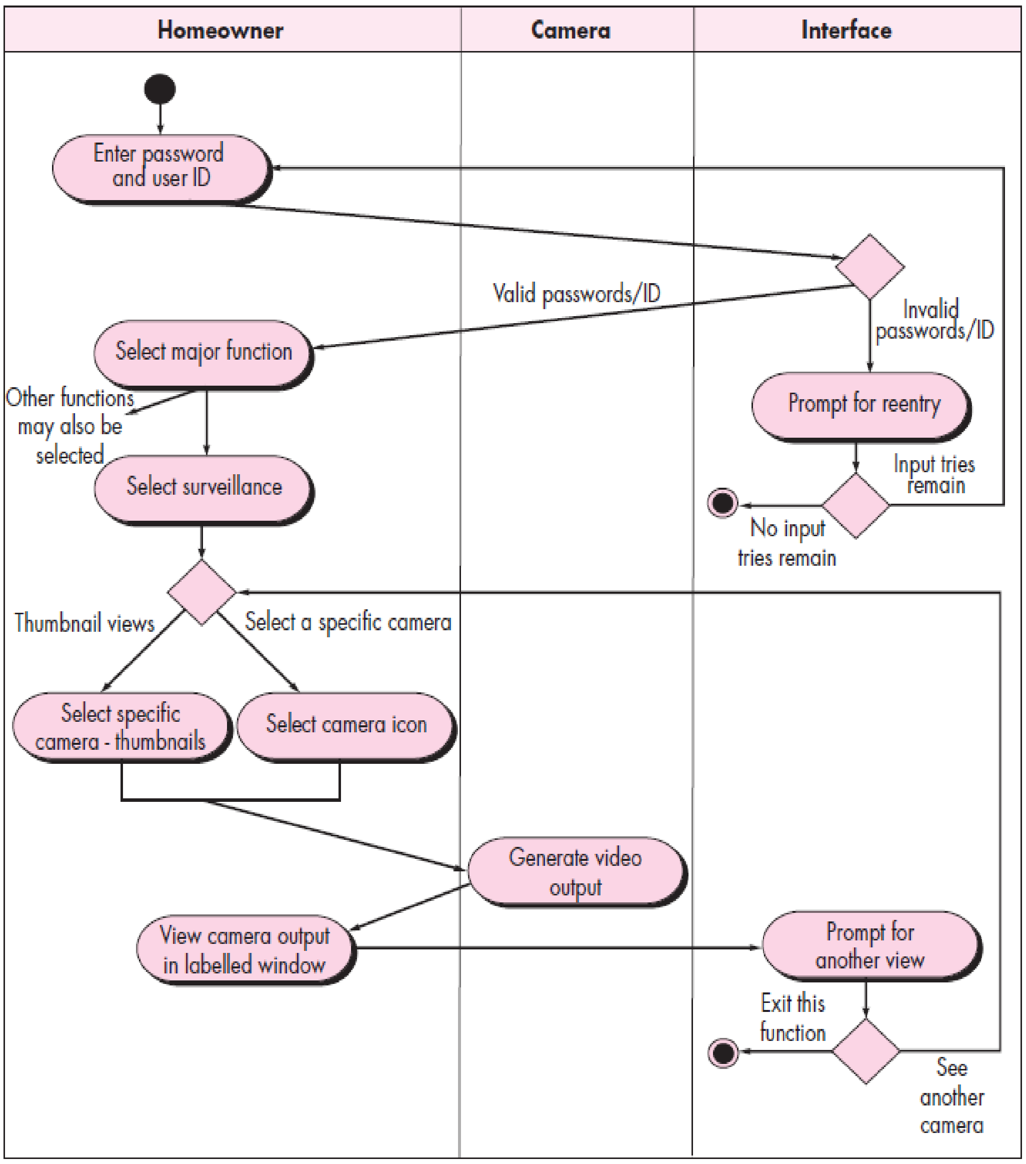


Fig : Swimlane diagram for SafeHome System

	b	Explain in detail about Data Modeling Concepts.	[L2][CO3]	[6M]

DATA MODELING CONCEPTS

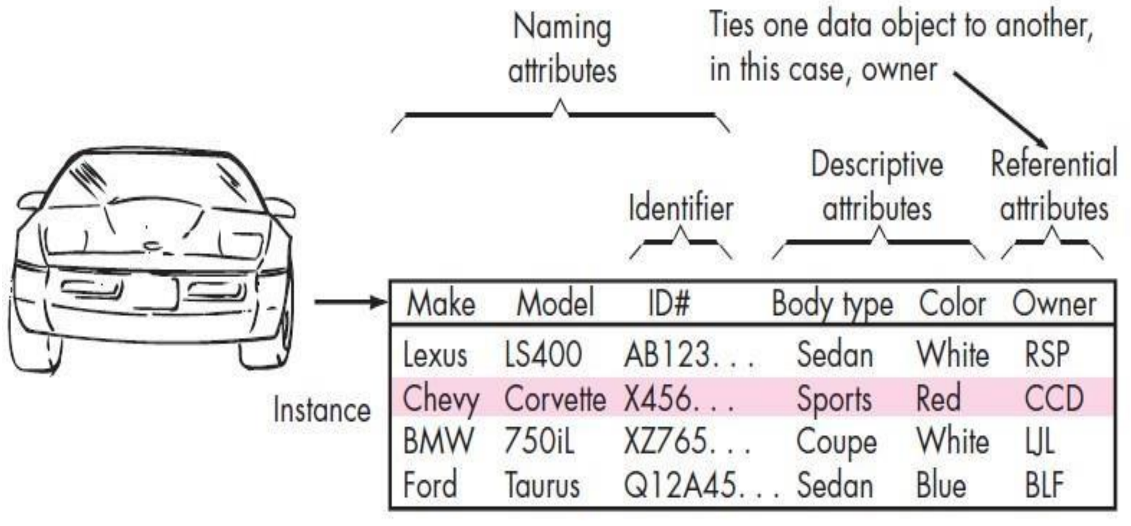
Data modeling is the process of documenting a complex software system design as an easily understood diagram, using text and symbols to represent the way data needs to flow. The diagram can be used as a blueprint for the construction of new software or for re-engineering a legacy application. The most widely used data Model by the Software engineers is E*ntity- Relationship Diagram* (ERD), it addresses the issues and represents all data objects that are entered, stored, transformed, and produced within an application.

Data Objects

A *data object* is a representation of composite information that must be understood by software. A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file).

For example, a person or a car can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data. Therefore, the data object can be represented as a table as shown in following table. The headings in the table reflect attributes of the object.



Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

Fig : Tabular representation of data objects

Data Attributes

Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table.

Relationships

Data objects are connected to one another in different ways. Consider the two data objects, person and car. These objects can be represented using the following simple notation and relationships are 1) A person *owns* a car, 2) A person *is insured to drive* a car

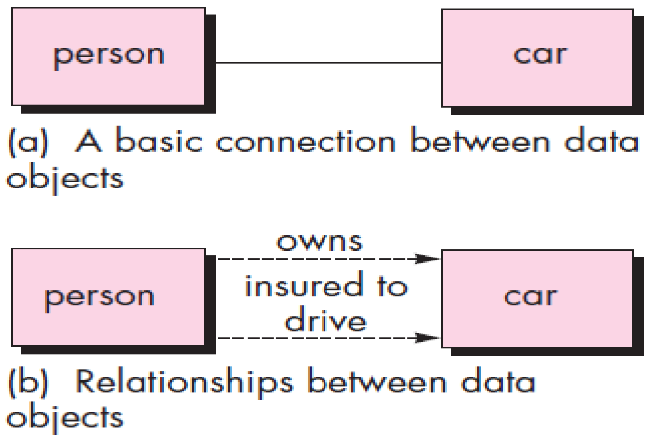

person
car
(a) A basic connection between data objects
owns
person
insured to drive
car
(b) Relationships between data objects

Fig : Relationships between data objects

10	**a**	Construct Class-Based Modeling briefly.	**[L3][CO3]**	**[6M]**

Class-Based Modeling

Class-based modeling is a stage of requirements modeling. In the context of software engineering, requirements modeling examines the requirements a proposed software application or system must meet in order to be successful. Typically, requirements modeling begins with scenario-based modeling, which develops a use case that will help with the next stages, like data and class-based modeling. Class-based modeling takes the use case and extracts from it the classes, attributes, and operations the application will use. Like all modeling stages, the end result of class-based modeling is most often a diagram or series of diagrams, most frequently created using UML, or rather, Unified Modeling Language.

Example Use Case

To better discuss what class-based modeling entails, let's assume scenario-based modeling has already been completed and resulted in the following use case: Sunny Beach Hotel purchases an advertisement in the "Best Hotels" travel book published by Acme Publishing. The client for the software development project is Acme Publishing, which has commissioned an application to manage the whole publication process, from selling an advertisement to a client to creating the book's content to finally printing, shipping, and selling the book.

Class-Based Model Method

Class-based modeling begins by identifying the classes in the use case.

One way to identify potential analysis classes is by performing what is called a grammatical parse of the problem statement or use case. This is done by underlining each noun or noun clause in the problem statement or use case and ignoring verbs.

A grammatical parse of the example use case looks like this (please note that these nouns, normally underlined, are marked in bold):

With this done, class-based modeling then uses several general classifications to help identify elements that can be considered potential classes:

1. External entities that produce or consume information to be used by the application (in the example use case, Sunny Beach Hotel)
2. Things that are part of the information domain of the problem (the advertisement that Sunny Beach Hotel purchases)
3. Occurrences or events that occur within the context of system operation (the transaction of Sunny Beach Hotel purchasing the advertisement from Acme Publishing)
4. Roles played by people who interact with the system (the marketing representative at Sunny Beach Hotel that places the order for the advertisement with Acme Publishing and the salesperson at Acme

Publishing that receives the order)

5. Organizational units that are relevant to an application (divisions, groups or teams, like the sales and editorial departments at Acme Publishing)
6. Places that establish the context of the problem and the overall function of the system (Sunny Beach Hotel's location; Acme Publishing's office location)
7. Structures that define a class of objects or related classes of objects (The "Best Hotels" travel book is one of several travel books that Acme Publishing publishes, so within this context "travel book" is a class of objects.)

Once the potential classes have been identified, six selection characteristics are used to decide which classes to include in the model:

1. **Retained information**, where information must be remembered about the system over time. (the fact that the advertisement has been sold to the Sunny Beach Hotel)
2. **Needed services**, where a set of operations that can change the attributes of a class (The number of advertisements sold will change the number of pages of the "Best Hotels" travel book and also the amount of income it has generated for Acme Publishing.)
3. **Multiple attributes**, where a class must have multiple attributes. If something identified as a potential class has only a single attribute, it's best included as a variable. (The "Best Hotels" travel book has multiple attributes, like number of pages, page size, and retail price.)
4. **Common attributes**, where all instances of a class share the same attributes (All travel books published by Acme Publishing will have the same attributes: number of pages, page size, and retail price. The values of these attributes may differ from one travel book to another. For example, one travel book may have 170 pages, while another will have 200 pages, but they both still have a number of pages, which is one of their common attributes.)
5. **Common operations**, where a set of operations apply to all instances of a class (All travel books published by Acme Publishing have pages that need to be filled with advertisements or other content, they all need to be printed and shipped to point of sale.)
6. **Essential requirements**, where entities produce or consume information (The "Best Hotels" travel book consumes information by storing the advertisement's content in its pages and produces information with the amount of income earned for the sale of that advertisement.)

	b	Explain how to create a Behavioral Model with a use case diagram.	**[L2][CO3]**	**[6M]**

BEHAVIORAL MODEL

The *behavioral model* indicates how software will respond to external events or stimuli.

To create the model, you should perform the following steps:

1. Evaluate all use cases to fully understand the sequence of interaction within the system.

2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.

3. Create a sequence for each use case.

4. Build a state diagram for the system.

5. Review the behavioral model to verify accuracy and consistency.

Identifying Events with the Use Case

The use case represents a sequence of activities that involves actors and the system. To illustrate, we reconsider the use case for a portion of the *SafeHome* security function. The homeowner uses the keypad to key in a four-digitpassword. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

An actor should be identified for each event; the information that is exchanged should be noted, and any conditions or constraints should be listed. Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events .

State Representations

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its Function. The first indicates how an individual class changes state based on external events and the second shows the behavior ofthe software as a function of time.

State diagrams for analysis classes: One component of a behavioral model is a UML state diagram that represents active states for each class and the events (triggers) that cause changes between these active states. The following figure illustrates a state diagram for the ControlPanelobject in the *SafeHome* security function. Each arrow shown in figure represents a transition

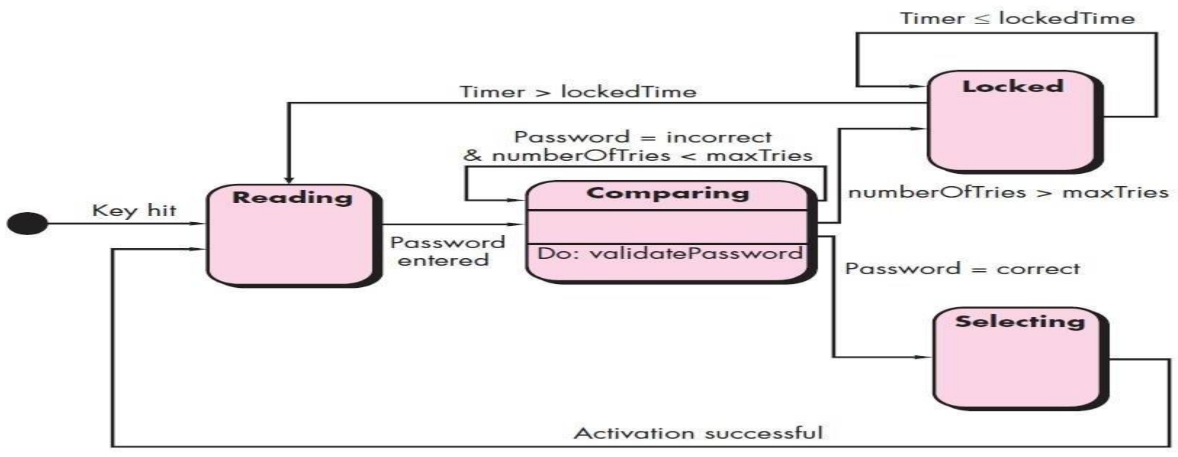
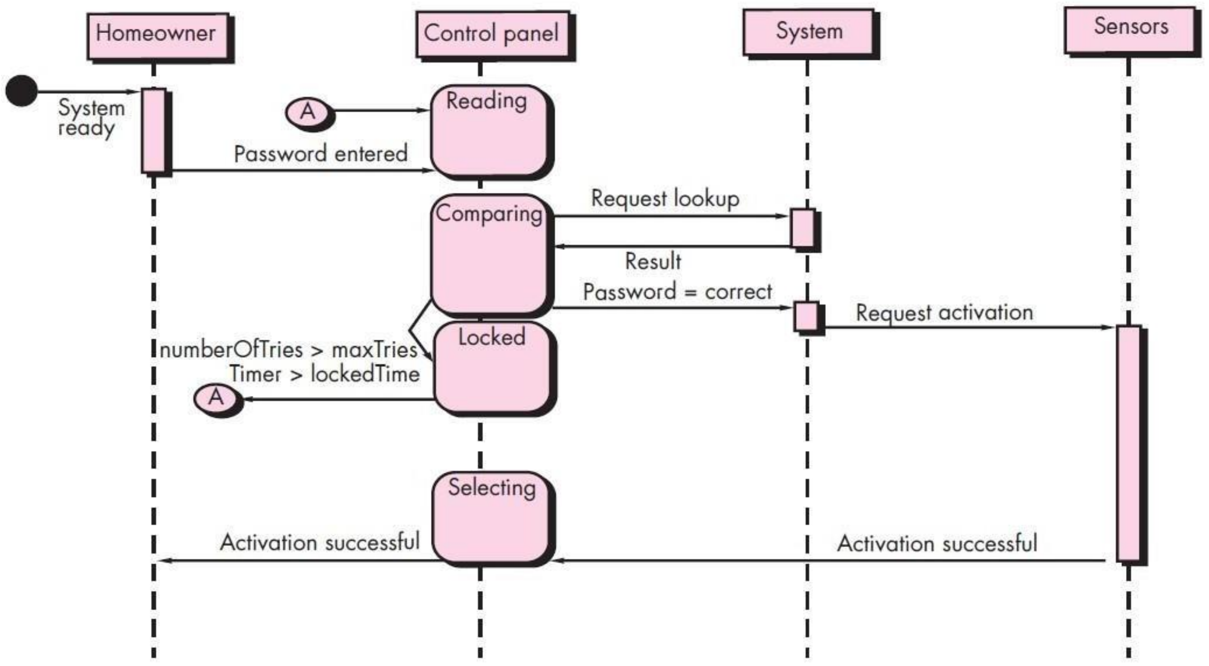


Fig : State diagram for the Control Panel class

from one active state of an object to another. The labels shown for each arrow represent the eventthat triggers the transition

Sequence diagrams: The second type of behavioral representation, called a *sequence diagram* inUML, indicates how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time. In essence, the sequence diagram is a shorthand version of the use case. It represents key classes and the events that causebehavior to flow from class to class.

		 Homeowner Control panel System Sensors System ready A Reading Password entered Comparing Request lookup Result Password = correct Request activation Locked numberOfTries > maxTries Timer > lockedTime A Selecting Activation successful Activation successful Fig : Sequence diagram (partial) for the *SafeHome* security function

UNIT –III

DESIGN CONCEPTS AND ARCHITECTURAL DESIGN

1	a	What is the Design process? Discuss software quality guidelines.	[L2][CO3]	[6M]

THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a“blueprint” for constructing the software. That is, the design is represented at a high level ofabstraction—a level that can be directly traced to the specific system objective and moredetailed data, functional, and behavioral requirements.

Software Quality Guidelines and Attributes:

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews. The three characteristics that serve as a guide for the evaluation of a good design:

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

Quality Guidelines.

In order to evaluate the quality of a design representation, the software team must establish technical criteria for good design. Guide lines are as follows

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
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 software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

	b	Explain common characteristics in the evolution of software design.	[L2][CO1]	[6M]

The Evolution of Software Design:

Software Evolution is a term which refers to the process of developing software initially, then timely updating it for various reasons, i.e., to add new features or to remove obsolete functionalities etc. The evolution process includes fundamental activities of change analysis, release planning, system implementation and releasing a system to customers.

The cost and impact of these changes are accessed to see how much system is affected by the change and how much it might cost to implement the change. If the proposed changes are accepted, a new release of the software system is planned. During release planning, all the proposed changes (fault repair, adaptation, and new functionality) are considered.

A design is then made on which changes to implement in the next version of the system. The process of change implementation is an iteration of the development process where the revisions to the system are designed, implemented and tested.

The necessity of Software evolution: Software evaluation is necessary just because of the following reasons:
a) Change in requirement with time: With the passes of time, the organization's needs and modus Operandi of working could substantially be changed so in this frequently changing time the tools(software) that they are using need to change for maximizing the performance.
b) Environment change: As the working environment changes the things(tools) that enable us to work in that environment also changes proportionally same happens in the software world as the working environment changes then, the organizations need reintroduction of old software with updated features and functionality to adapt the new environment.
c) Errors and bugs: As the age of the deployed software within an organization increases their preciseness or impeccability decrease and the efficiency to bear the increasing complexity workload also continually degrades. So, in that case, it becomes necessary to avoid use of obsolete and aged software. All such obsolete Softwares need to undergo the evolution process in order to become robust as per the workload complexity of the current environment.

d) Security risks: Using outdated software within an organization may lead you to at the verge of various software-based cyberattacks and could expose your confidential data illegally associated with the software that is in use. So, it becomes necessary to avoid such security breaches through regular assessment of the security patches/modules are used within the software. If the software isn't robust enough to bear the current occurring Cyber attacks so it must be changed (updated).
e) For having new functionality and features: In order to increase the performance and fast data processing and other functionalities, an organization need to continuously evolute the software throughout its life cycle so that stakeholders & clients of the product could work efficiently.

2	Determine software design concepts in detail.	[L3][CO3]	[12M]

DESIGN CONCEPTS

Design concepts has evolved over the history of software engineering. Each concept provides the software designer with a foundation from which more sophisticated design methods can be applied.

2.1 Abstraction: When you consider a modular solution to any problem, many levels of abstraction can be posed. At the *highest level of abstraction*, a solution is stated in *broad terms* using the language of the problem environment. At *lower levels* of *abstraction*, a *more detailed description* of the solution is provided. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

A procedural abstraction refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. *A data abstraction* is a named collection of data that describes a data object.

2.2 Architecture: Software architecture alludes to "*the overall structure of the software andthe ways in which that structure provides conceptual integrity for a system*".
In its simplest form, *architecture* is the structure or organization of program

components (modules), the manner in which these components interact, and the structure of data that are used by the components.
One goal of software design is to derive an architectural rendering of a system. A set of architectural patterns enables a software engineer to solve common design problems.Shaw and Garlan describe a set of properties as part of an architectural design:

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters).
Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems.

2.3 Patterns: A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns.
The intent of each design pattern is to provide a description that enables a designer to determine
(1) whether the pattern is applicable to the current work
(2) whether the pattern can be reused (hence, saving design time)
(3) whether the pattern can serve as a guide for developing a similar, but functionally orstructurally different pattern.

2.4 Separation of Concerns: Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is *subdivided into pieces* that can each be solved and/or optimized independently.

For two problems, p1 and p2, if the perceived complexity of p1 is greater than the perceived complexity of p2, it follows that the effort required to solve p1 is greater than the effort requiredto solve p2. As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem. It also follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. This leads to *a divide-and-conquer strategy*—it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to software modularity.

2.5 Modularity: Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements. It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Referring to Figure , the effort (cost) to develop an individual software module does decrease as the total number of modules increases.

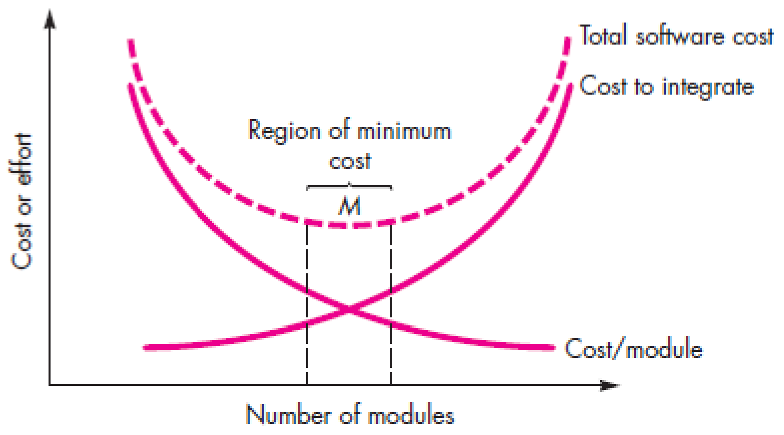


Fig: Modularity and Software cost

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.The curves shown in Figure do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of M. *Undermodularity* or *overmodularity* should be avoided.

2.6 Information Hiding: The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, 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. Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.

2.7 Functional Independence: The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with "singleminded" function and an "aversion" to excessive interaction with other modules. Independence is assessed using two qualitative criteria: cohesion and coupling.
Cohesion is an indication of the relative functional strength of a module. Coupling is an indicationof the relative interdependence among modules.

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 software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

2.8 Refinement: Stepwise refinement is a *top-down design strategy.* A program is developed by successively refining levels of procedural detail. Refinement is actually a process of elaboration begins with a statement of function (or description of information) that is defined at a high level of abstraction. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs. Refinement helps you to reveal low-level details as design progresses.

2.9 Aspects: As requirements analysis occurs, a set of "concerns" is uncovered. Theseconcerns "include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts". Ideally, arequirements model can be organized in a way that allows you to isolate each concern(requirement) so that it can be considered independently.

2.10 Refactoring: An important design activity for many agile methods is refactoring a reorganization technique that simplifies the design (or code) of a component without changing itsfunction or behavior. "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior 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. The result will be software that is easier to integrate, easier to test, and easier to maintain.

2.11 Object-Oriented Design Concepts: The object-oriented (OO) paradigm is widely usedin modern software engineering. OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others.

2.12 Design Classes: As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.
Five different types of design classes, each representing a different layer of the design architecture, can be developed.

- *User interface classes* define all abstractions that are necessary for human computer interaction (HCI).
- *Business domain classes* are often refinements of the analysis classes. The classes identifythe attributes and

services (methods) that are required to implement some element of the business domain.

• *Process classes* implement lower-level business abstractions required to fully manage the business domain classes.

• *Persistent classes* represent data stores (e.g., a database) that will persist beyond the execution of the software.

• *System classes* implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

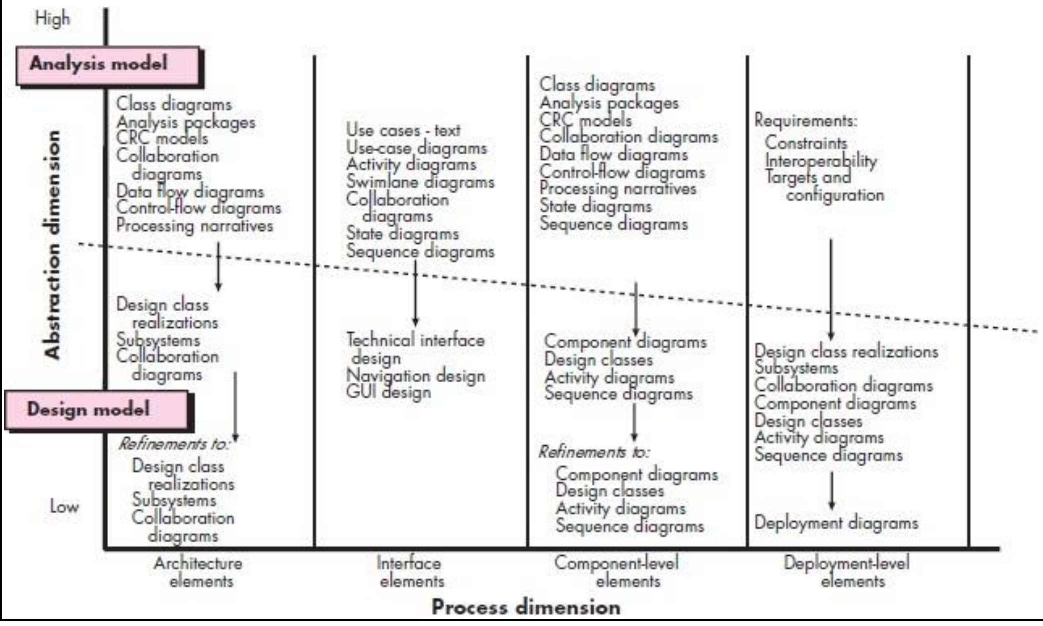
They define four characteristics of a well-formed design class:
Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class.
Primitiveness. 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. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled, the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the *Law of Demeter*, suggests that a method should only send messages to methods in neighboring classes._

3	**a**	Describe a Design model with various kinds of elements	**[L2][CO3]**	**[6M]**

There are two dimensions in which a design model can be illustrated: process and abstraction dimension. Process dimension points the changes in the design model as the design tasks are implemented as part of software process. Abstraction dimension represents the level of detail as each element of analysis model is transformed into design equivalent and refined iteratively.

Design model elements are not developed in a sequential fashion. The design model has four major elements:
- Data
- Architecture
- Components
- Interface

Data design focuses on files or databases at an architectural level and on a component level, data design considers the data structures that are required to implement local data objects.



The **architectural model** is derived from the application domain, the analysis model and available styles and patterns.

The **interface design elements** for software tells us how the information flows in and out of the system and how it is communicated among the components. There are three parts to interface design elements: the user interface, interfaces to systems external to application and interfaces to components within the application.
User interface elements include aesthetic elements, ergonomic elements and technical elements. External interface design requires information about entity to which information is sent or received. Internal interface design is closely related to component level design.

The **component level design elements** describes the internal detail of software component. The component level design defines data structures for all local data objects and algorithmic detail processing that occurs within a component and an interface that allows access to all behaviors.

	b	Prioritize the Quality Attributes in Software Design	**[L4][CO3]**	**[6M]**

Quality Attributes. Hewlett-Packard developed a set of software quality attributes that has beengiven the acronym FURPS—functionality, usability, reliability, performance, and supportability. The FURPS quality attributes represent a target for all software design:

- *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- *Usability* is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and thepredictability of the program.
- *Performance* is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- *Supportability* combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, maintainability—and in addition, testability, compatibility, configurability, the ease with which a system can be installed, and the ease with which problems can be localized.

4	**a**	How Software Quality Guidelines is framed? List out the guidelines	**[L2][CO3]**	**[6M]**

Quality Guidelines. In order to evaluate the quality of a design representation, the software team must establish technical criteria for good design. Guide lines are as follows

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
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 implementedand 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 software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

	b	Write about cohesion and coupling.	[L2][CO3]	[6M]

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with "singleminded" function and an "aversion" to excessive interaction with other modules. Independence is assessed using two qualitative criteria: **cohesion** and **coupling**.

Cohesion is an indication of the relative functional strength of a module. Coupling is an indication of the relative interdependence among modules.

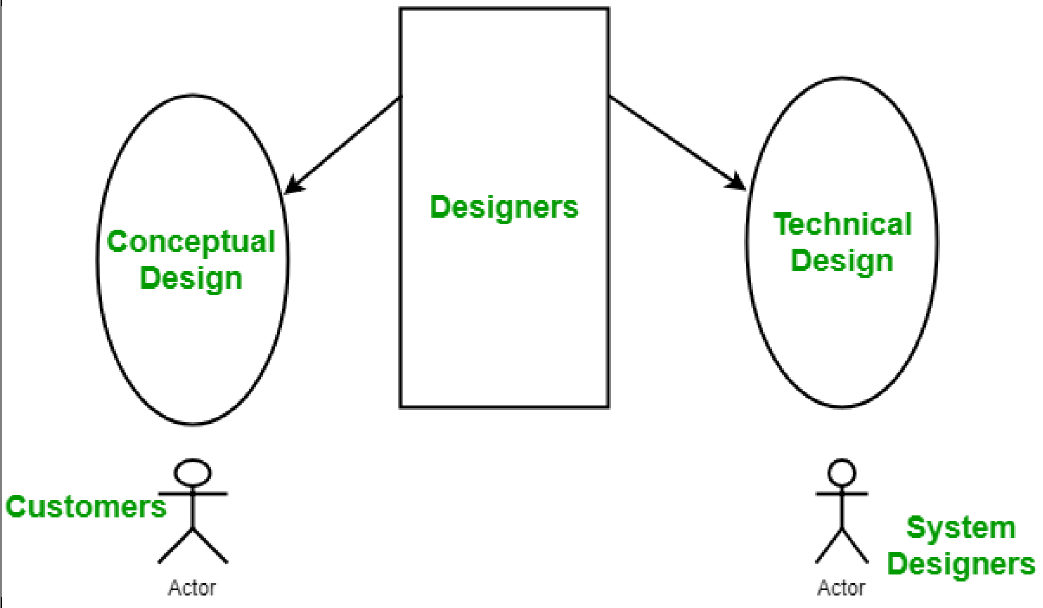
Coupling and Cohesion are two key concepts in software engineering that are used to measure the quality of a software system's design.

Coupling refers to the degree of interdependence between software modules. High coupling means that modules are closely connected and changes in one module may affect other modules. Low coupling means that modules are independent and changes in one module have little impact on other modules.

Cohesion refers to the degree to which elements within a module work together to fullfill a single, well-defined purpose. High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes.

Both coupling and cohesion are important factors in determining the maintainability, scalability, and reliability of a software system. High coupling and low cohesion can make a system difficult to change and test, while low coupling and high cohesion make a system easier to maintain and improve.

Basically, design is a two-part iterative process. First part is Conceptual Design that tells the customer what the system will do. Second is Technical Design that allows the system builders to understand the actual hardware and software needed to solve customer's problem.



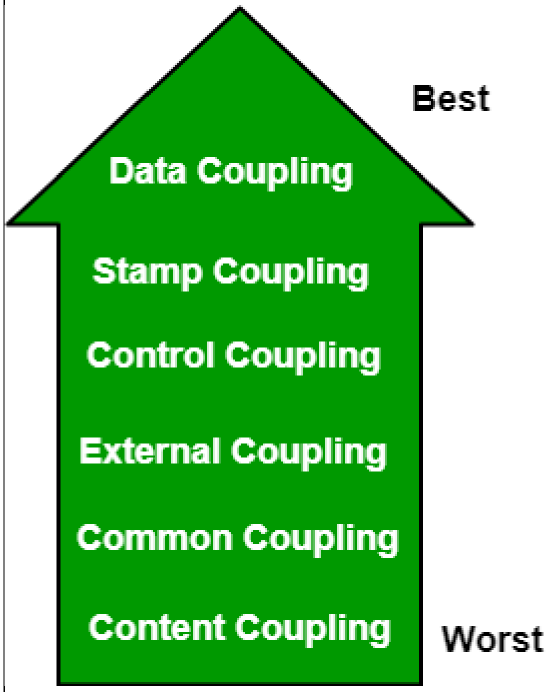
5	a	Devise to assess alternate Architectural design.	[L4][CO3]	[6M]

Assessing Alternative Architectural Design
Design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved.
Two different approaches for the assessment of alternative architectural designs.
(1) The first method uses an iterative method to assess design trade-offs.
(2)The second approach applies a pseudo-quantitative technique for assessing design quality.

An Architecture Trade-Off Analysis Method
The Software Engineering Institute (SEI) has developed an architecture trade-off analysis method (ATAM) that establishes an iterative evaluation process for software architectures.
The design analysis activities that follow are performed iteratively.
1. Collect scenarios : A set of use cases is developed to represent the system from the user's point of view.
2. Elicit (Bring out) requirements, constraints, and environment description. This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.
3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements. The architectural style(s) should be described using one of the following architectural views…
Module view for analysis of work assignments with components and the degree to which information hiding has been achieved.
Process view for analysis of system performance.
Data flow view for analysis of the degree to which the architecture meets functional requirements.
4. Evaluate quality attributes : Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.
5.Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style. This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variationin the architecture are termed sensitivity points..
6. Critique (Assess) candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.
The Software Engineering Institute (SEI) describes this approach in the following manner
Once the architectural sensitivity points have been determined, finding trade-off points is simply the identification of architectural elements to which multiple attributes are sensitive. For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). . . . The number of servers, then, is a trade-off point with respect to this architecture.

	b	Why Cohesion and Coupling is used in Software Design Process. What are the types in it. Differentiate all the types	**[L4][CO3]**	**[6M]**

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



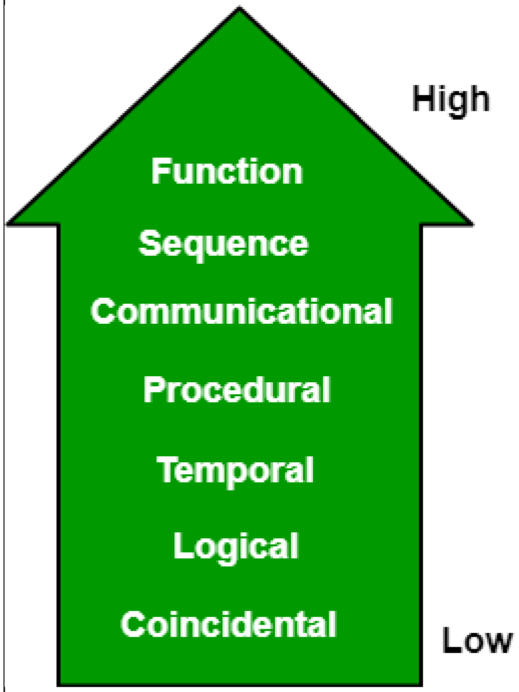
Types of Coupling:

- **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent of each other and communicate through data. Module communications don't contain tramp data. Example-customer billing system.
- **Stamp Coupling** In stamp coupling, the complete data structure is passed from one module to

another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice was made by the insightful designer, not a lazy programmer.

- **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.
- **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.
- **Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses, and reduced maintainability.
- **Content Coupling:** In a content coupling, one module can modify the data of another module, or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

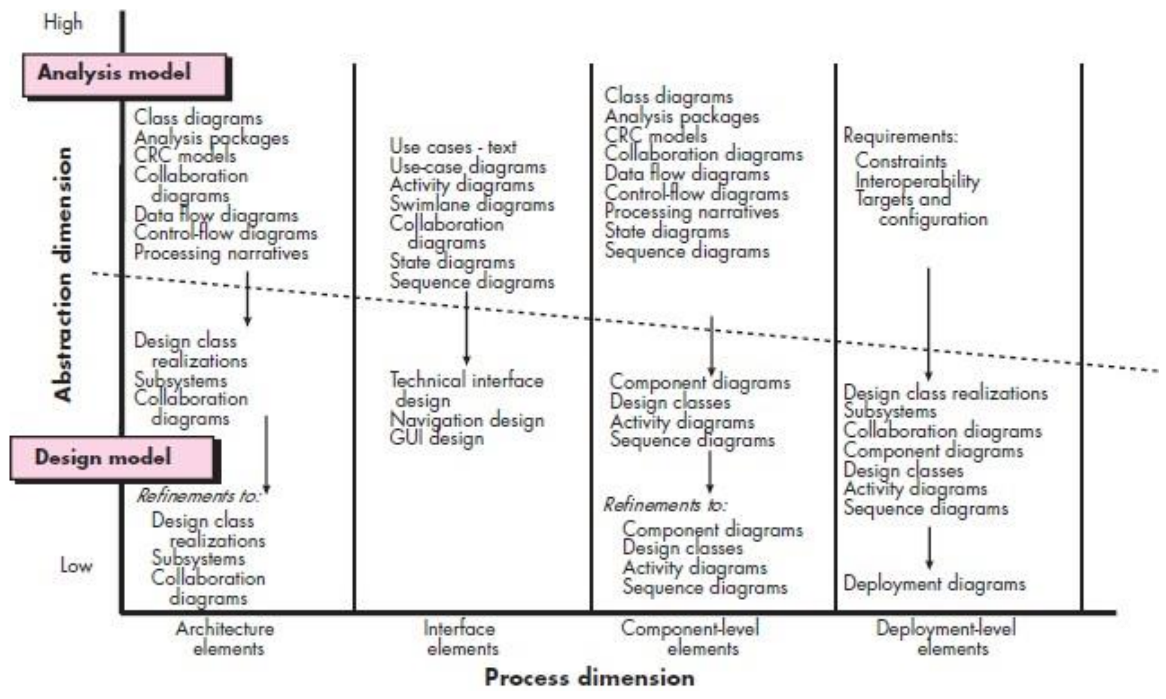
Cohesion: Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good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



Types of Cohesion:

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

6	**a**	Distinguish between Analysis Model and Design Model	**[L5][CO4]**	**[6M]**



	b	What is UI Design? Why it is so important?	**[L4][CO4]**	**[6M]**

UI design is the design field that focuses on the look and feel of an app or website that we interact with. Unlike **UX design**, which focuses on the user experience, UI design focuses on aesthetics. The look and feel of an app can be the colors users see, the animations they interact with, the buttons they press, or the text they read. These fall under the job description of a UI designer.

The UI designer works on bringing **wireframes** to life and **creating prototypes** that incorporate both the input from the UX designers as well as accommodating all of the brand needs. UI designers tend to every detail — from typography to logo use — and make sure that every project they're working on matches other touch-points. For example, you want your app, website and landing pages to all look cohesive.

At the same time, UI designers can bring innovation to the design. Since they get to focus more on the creative side of the project, they can come up with more innovative ways to display information. Of course, it's always good practice to check in with the UX designers again to make sure the interface design isn't affecting the user journey.

UI design important

Since users can only interact with your app or website through the UI, its design will determine their main impressions. If you've tried any low-effort app or website you'll know exactly how frustrating it can be to deal with a badly designed UI. Apps are now competing for users' time, so you don't have long to impress a user. In fact, according to **Localytics**, 71% of all app users churn within 90 days.

User interface designers bring a creative edge to the project, making your app or website stand out amongst the millions of other options out there. Even if you're following a broader template of what a ride-hailing app or booking website should look like, creative UI design is how you can stand out.

On top of that, user interface design can make a difference in a product's **emotional appeal**. Done right, UI design can pique your user's interest and get them excited about exploring the product more. But otherwise, it could send them in the opposite direction.

Brand awareness also plays a huge role in UI design. By having all your designs follow the same guidelines, you can make sure that your product or service can become more recognizable. UI design doesn't just stop with the design of a specific asset. By having it as a strong focus for your project, designers are able to design for the bigger picture of how it will all look and work together in the finished result.

7	Describe architectural genres for software-based systems.	**[L2][CO4]**	**[12M]**

ARCHITECTURAL GENRES

Although the underlying principles of architectural design apply to all types of architecture, *the architectural genre will often dictate the specific architectural approach to the structure that must be built.*

***Grady Booch* suggests the following architectural genres for software-based systems:**

- **Artificial intelligence**—Systems that simulate or augment human cognition, locomotion, orother organic processes.
- **Commercial and nonprofit**—Systems that are fundamental to the operation of a businessenterprise.
- **Communications**—Systems that provide the infrastructure for transferring and managingdata, for connecting users of that data, or for presenting data at the edge of an infrastructure.
- **Content authoring**—Systems that are used to create or manipulate textual or multimediaartifacts.
- **Devices**—Systems that interact with the physical world to provide some point service for anindividual.
- **Entertainment and sports**—Systems that manage public events or that provide a large groupentertainment experience.
- **Financial**—Systems that provide the infrastructure for transferring and managing money andother securities.
- **Games**—Systems that provide an entertainment experience for individuals or groups.
- **Government**—Systems that support the conduct and operations of a local, state, federal,global, or other political entity.
- **Industrial**—Systems that simulate or control physical processes.
- **Legal**—Systems that support the legal industry.
- **Medical**—Systems that diagnose or heal or that contribute to medical research.
- **Military**—Systems for consultation, communications, command, control, and intelligence (C4I)as well as offensive and defensive weapons.
- **Operating systems**—Systems that sit just above hardware to provide basic software services.
- **Platforms**—Systems that sit just above operating systems to provide advanced services.
- **Scientific**—Systems that are used for scientific research and applications.
- **Tools**—Systems that are used to develop other systems.
- **Transportation**—Systems that control water, ground, air, or space vehicles.
- **Utilities**—Systems that interact with other software to provide some point service.

8	Express the various types of Architectural styles briefly.	**[L6][CO4]**	**[12M]**

Architectural style describes a system category that encompasses

(1) a set of components (e.g., a database, computational modules) that perform a functionrequired by a system;

(2) a set of connectors that enable "communication, coordination and cooperation" among **components;**

(3) constraints that define how components can be integrated to form the system; and

(4) semantic models that enable a designer to understand the overall properties of a system byanalyzing the known properties of its constituent parts.

An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture. However, a pattern differs from a style in a number of fundamental ways:

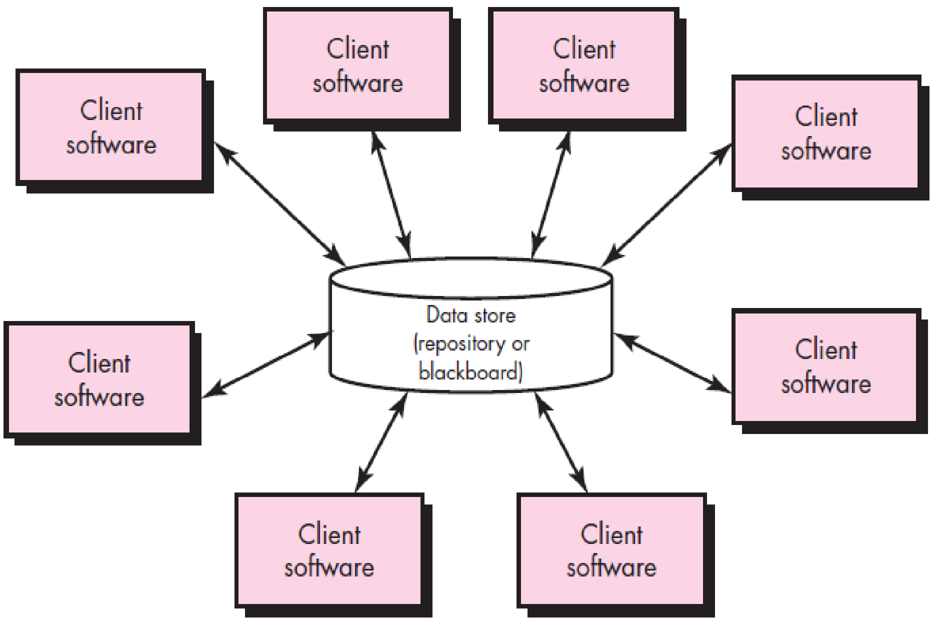
(1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather thanthe architecture in its entirety;

(2) a pattern imposes a rule on the architecture, describing how the software will handle someaspect of its functionality at the infrastructure level (e.g., concurrency)

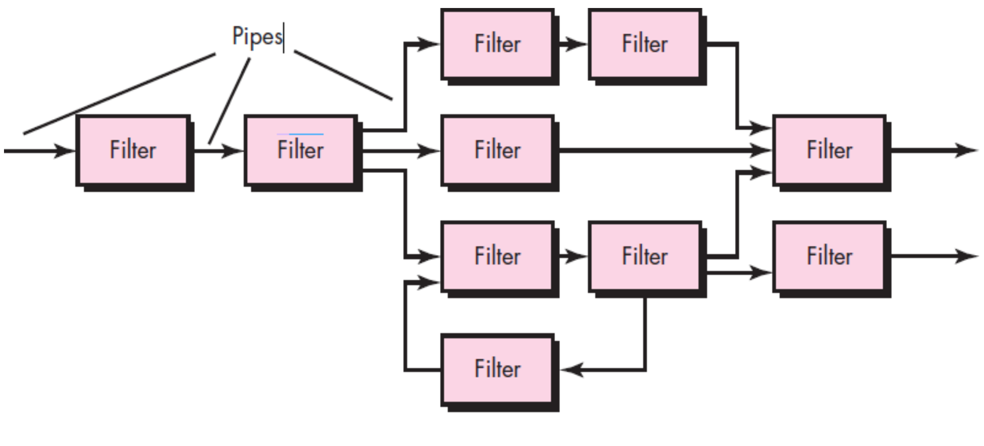
(3) architectural patterns tend to address specific behavioral issues within the context of thearchitecture (e.g., how real-time applications handle synchronization or interrupts).

6.1 A Brief Taxonomy of Architectural Styles: Although millions of computer-based systemshave been created over the past 60 years, the vast majority can be categorized into one of a relatively small number of architectural styles:

Data-centered architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 9.1 illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. Data-centered architectures promote integrability. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.



Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern (Figure) has a set of components, called filters, connected by pipes that transmit datafrom one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.

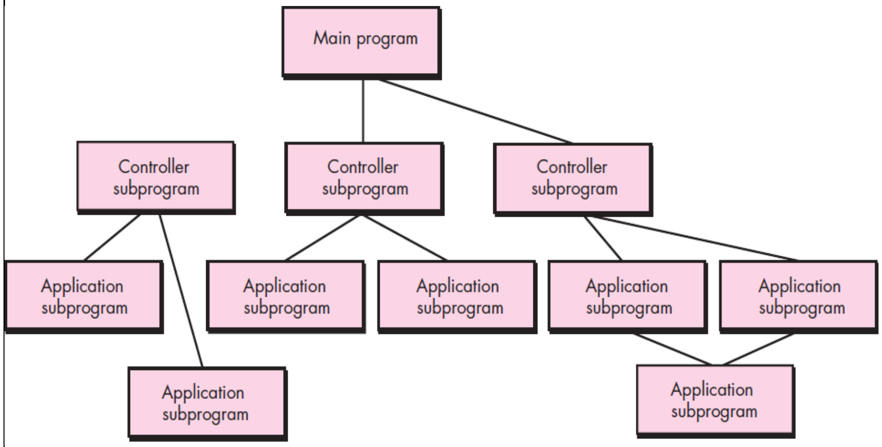


Pipes and filters

Call and return architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles exist within this category:

• *Main program/subprogram architectures*. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other components. Figure illustrates an architecture of this type.

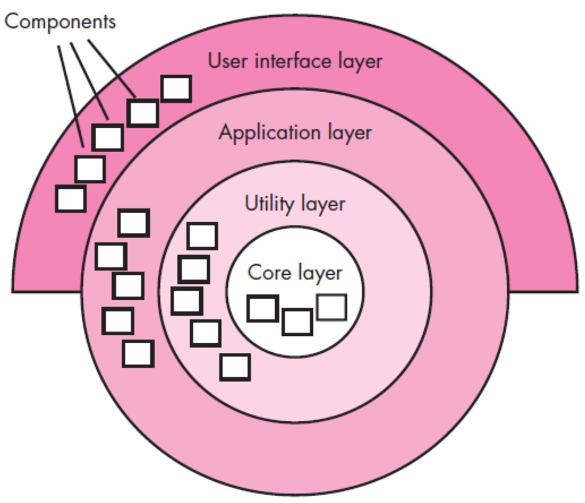
• *Remote procedure call architectures*. The components of a main program/subprogram architecture are distributed across multiple computers on a network.



Object-oriented architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

Layered architectures. The basic structure of a layered architecture is illustrated in Figure. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions. These architectural styles are only a small subset of those available. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. For example, a layered style (appropriate for most systems) can be combined with a data- centered architecture in many database applications.

Fig: Layered architecture



Architectural Patterns: Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

For example, the overall architectural style for an application might be call-and-return or object- oriented. But within that style, you will encounter a set of common problems that might best be addressed with specific architectural patterns.

Organization and Refinement: Because the design process often leaves you with anumber of architectural alternatives, it is important to establish a set of design criteria that can beused to assess an architectural design that is derived. The following questions provide insight into an architectural style:

Control. How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system?
Data. How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer? Do data components (e.g., a blackboard or repository) exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system?
These questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

9	Discuss briefly about Architectural design and their tasks.	**[L2][CO4]**	**[12M]**

ARCHITECTURAL DESIGN

The design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes. *An archetype is an abstraction (similar to a class) that represents one element of system behavior*. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail.

Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure

has been derived.

7.1 Representing the System in Context: *Architectural context represents how the software interacts with entities external to its boundaries*. At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural contextdiagram is illustrated in Figure.

• *Superordinate systems*—those systems that use the target system as part of some higher-level processing scheme.

• *Subordinate systems*—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

• *Peer-level systems*—those systems that interact on a peer-to-peer basis i.e., information is either produced or consumed by the peers and the target system.

• *Actors—entities* (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing. Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

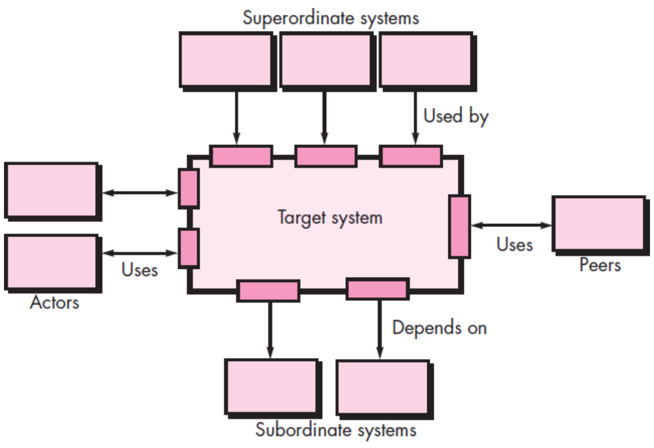


Fig: Architectural context Diagram

7.2 Defining Archetypes: *Archetypes are the abstract building blocks of an architectural design.* An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

The following are the archetypes for safeHome: Node, Detector, Indicator, Controller.

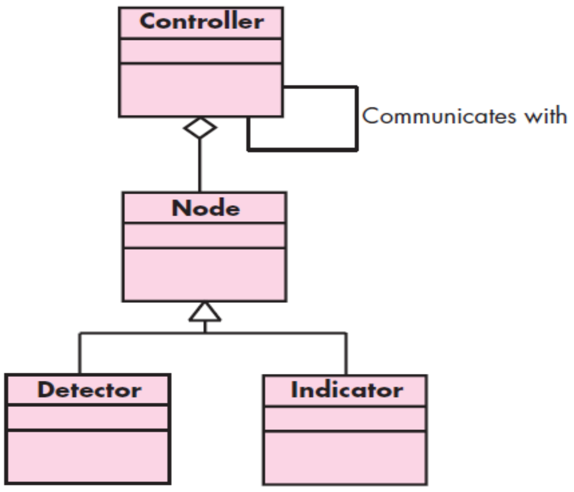


Fig: UML relationships for archetype

7.3 Refining the Architecture into Components: As the software architecture is refined into components, the structure of the system begins to emerge. The architecture must accommodate many infrastructure components that enable application components but have no businessconnection to the application domain. As an example for SafeHome home security, the set of top-level components that address the following functionality:

• *External communication management*—coordinates communication of the security functionwith external entities such as other Internet-based systems and external alarm notification.

- *Control panel processing*—manages all control panel functionality.
- *Detector management*—coordinates access to all detectors attached to the system.
- *Alarm processing*—verifies and acts on all alarm conditions.
-

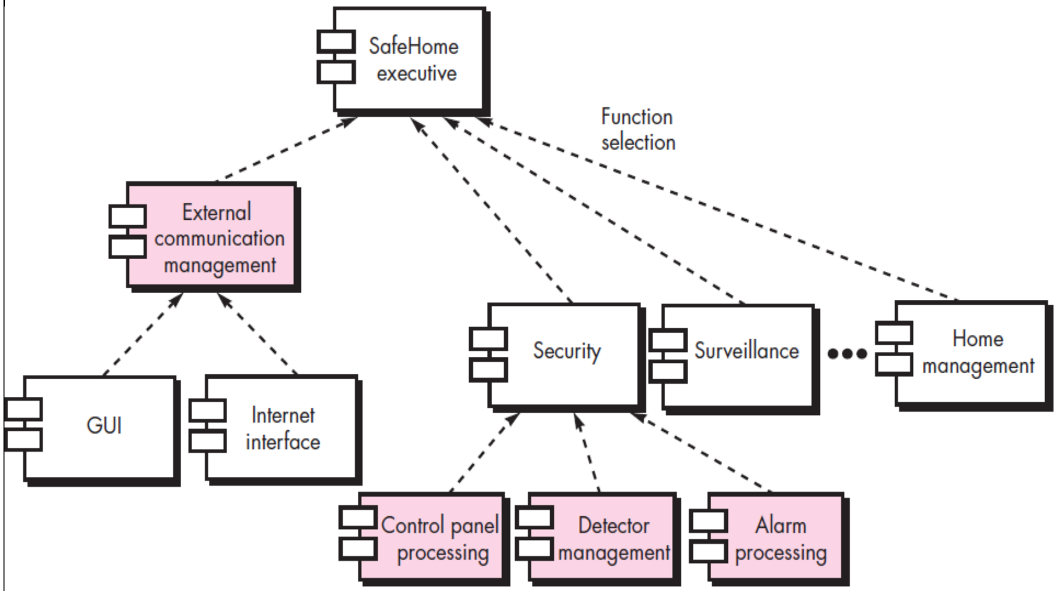


Fig: Overall architectural structure for safeHome with top-level components

7.4 Describing Instantiations of the System: The architectural design that has been modeled to this point is still relatively *high level*. The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain havebeen defined, the overall structure of the system is apparent, and the major software components have been identified. However, *further refinement* is still necessary.

	a	Justify the Assessing of Alternative Architectural Designs for Software	**[L5][CO4]**	**[6M]**

10

ASSESSING ALTERNATIVE ARCHITECTURAL DESIGNS

Design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved. Two different approaches for the assessment of alternative architectural designs. The first method uses an iterative method to assess design trade-offs. The second approach applies a pseudo-quantitative technique for assessing design quality.

8.1 An Architecture Trade-Off Analysis Method: The Software Engineering Institute (SEI) has developed an architecture trade-off analysis method (ATAM) that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

1. *Collect scenarios*. A set of use cases is developed to represent the system from the user's **point of view.**
2. *Elicit requirements*, constraints, and environment description. This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.
3. *Describe the architectural styles/patterns* that have been chosen to address the scenarios andrequirements. The architectural style(s) should be described using one of the following architectural views:
    - Module view for analysis of work assignments with components and the degree to which

information hiding has been achieved.

- Process view for analysis of system performance.
- Data flow view for analysis of the degree to which the architecture meets functional requirements.

4. *Evaluate quality attributes* by considering each attribute in isolation. The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability,portability, reusability, and interoperability.
5. *Identify the sensitivity of quality attributes* to various architectural attributes for a specific architectural style. This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed sensitivity points.
6. *Critique candidate architectures* using the sensitivity analysis conducted in step 5.The SEI describes this approach in the following manner.

Once the architectural sensitivity points have been determined, finding trade-off points is simplythe identification of architectural elements to which multiple attributes are sensitive.

8.2 Architectural Complexity: A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the system.

The three types of dependencies:

***Sharing dependencies* represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers. For example, for twocomponents u and v, if u and v refer to the same global data, then there exists a shared dependence relationship between u and v.**

Flow dependencies represent dependence relationships between producers and consumers **of resources. For example, for two components u and v, if u must complete before control flows into v (prerequisite), or if u communicates with v by parameters, then there exists a flow dependence relationship between u and v.**

Constrained dependencies **represent constraints on the relative flow of control among a set of activities. For example, for two components u and v, u and v cannot execute at the same time (mutual exclusion), then there exists a constrained dependence relationship between u and v.**

8.3 Architectural Description Languages: Architectural description language (ADL) provides a semantics and syntax for describing a software architecture. Hofmann and his colleagues suggest that an ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components. Once descriptive, language based techniques for architectural design have been established, it is more likely that effective assessment methods for architectures will be established as the design evolves.

	b	Write down the steps in refining the Architecture into Components	**[L2][CO4]**	**[6M]**

An Architecture Trade-Off Analysis Method: The Software Engineering Institute (SEI) has developed an architecture trade-off analysis method (ATAM) that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

Collect scenarios. A set of use cases is developed to represent the system from the user's **point of view.**

Elicit requirements, constraints, and environment description. This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.

Describing the architectural styles/patterns that have been chosen to address the scenarios andrequirements.

The architectural style(s) should be described using one of the following architectural views:

- Module view for analysis of work assignments with components and the degree to which information hiding has been achieved.
- Process view for analysis of system performance.
- Data flow view for analysis of the degree to which the architecture meets functional requirements.

Evaluating quality attributes by considering each attribute in isolation. The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability,portability, reusability, and interoperability.

Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style. This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed sensitivity points.

Critique candidate architectures using the sensitivity analysis conducted in step 5.The SEI describes this approach in the following manner.

Once the architectural sensitivity points have been determined, finding trade-off points is simplythe identification of architectural elements to which multiple attributes are sensitive.

Architectural Complexity: A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the system.

The three types of dependencies:

***Sharing dependencies* represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers. For example, for twocomponents u and v, if u and v refer to the same global data, then there exists a shared dependence relationship between u and v.**

Flow dependencies represent dependence relationships between producers and consumers **of resources. For example, for two components u and v, if u must complete before control flows into v (prerequisite), or if u communicates with v by parameters, then there exists a flow dependence relationship between u and v.**

***Constrained dependencies* represent constraints on the relative flow of control among a set of activities. For example, for two components u and v, u and v cannot execute at the same time (mutual exclusion), then there exists a constrained dependence relationship between u and v.**

Architectural Description Languages: Architectural description language (ADL) provides a semantics and syntax for describing a software architecture. Hofmann and his colleagues suggest that an ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components. Once descriptive, language based techniques for architectural design have been established, it is more likely that effective assessment methods for architectures will be established as the design evolves.

UNIT –IV

USER INTERFACE DESIGN AND WEB APP DESIGN

1	Briefly explain about golden rules in the user interface design.	**[L2][CO4]**	**[12M]**

THE GOLDEN RULES

The three golden rules on interface design are

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guidethis important aspect of software design.

1.1 Place the User in Control:

During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface. Mandel defines a number of design principles that allow the user to maintain control:

- **Define interaction modes in a way that does not force a user into unnecessary or undesired actions.**

An interaction mode is the current state of the interface. For example, if spell check is selected in a word-processor menu, the software moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

- **Provide for flexible interaction.**

Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi touch screen, or voice recognition commands.

- **Allow user interaction to be interruptible and undoable.**

Even when involved in a sequenceof actions, the user should be able to interrupt the sequence to do something else (without losingthe work that had been done). The user should also be able to "undo" any action.

- **Streamline interaction as skill levels advance and allow the interaction to be customized.** Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.
- **Hide technical internals from the casual user.**

The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology.

- **Design for direct interaction with objects that appear on the screen**.

The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to "stretch" an object is an implementation of direct manipulation.

Reduce the User's Memory Load:

The more a user has to remember, the more error- prone the interaction with the system will be.

- **Reduce demand on short-term memory.**
  When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results.
- **Establish meaningful defaults.**
  The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences.
- **Define shortcuts that are intuitive.**
  When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember.
- **The visual layout of the interface should be based on a real-world metaphor.**
  For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process.
- **Disclose information in a progressive fashion**.
  The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

1.3 Make the Interface Consistent:

The interface should present and acquire information in aconsistent fashion. This implies that
(1) all visual information is organized according to design rules that are maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined and implemented.

- **Allow the user to put the current task into a meaningful context**.
  Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, 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 (or products)should 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 isa compelling reason to do so.

2	**a**	Devise the golden rules to form the basis for a set of user interface design principles.	**[L4][CO4]**	**[6M]**

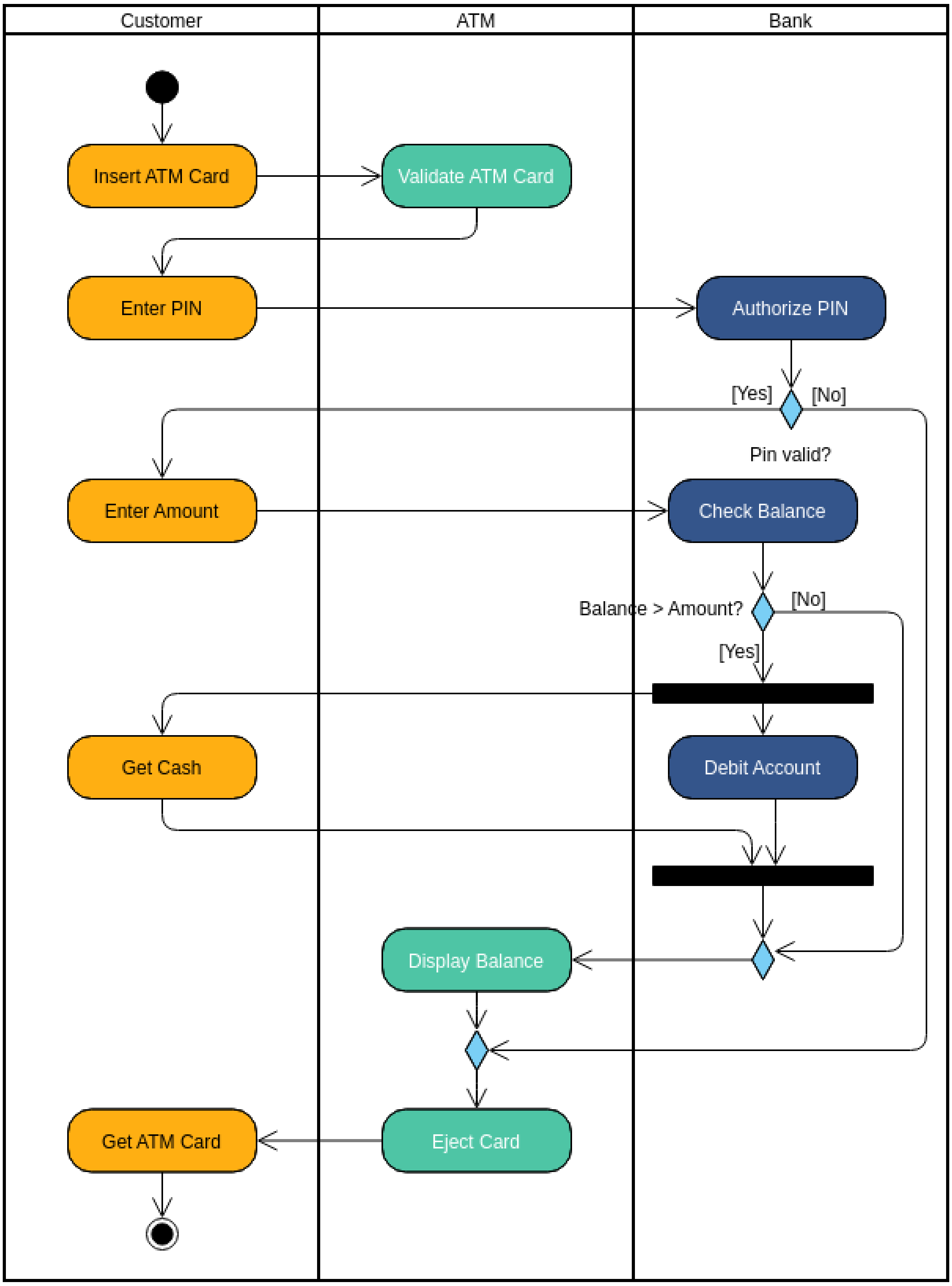
The overall process for analyzing and designing a user interface begins with the creation of different models of system function. Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.

2.1 Interface Analysis and Design Models:

Four different models come into play when a user interface is to be analyzed and designed. To build an effective user interface, “all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality. In addition, users can be categorized as:

- ***Novices***. No syntactic knowledge of the system and little semantic knowledge of the application or computer usage in general.
- ***Knowledgeable, intermittent users***. Reasonable semantic knowledge of the application but

		relatively low recall of syntactic information necessary to use the interface. • ***Knowledgeable***, ***frequent users***. Good semantic and syntactic knowledge that often leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction. The user's ***mental model*** (system perception) is the image of the system that end users carry in their heads. For example, if the user of a particular word processor were asked to describe its operation, the system perception would guide the response. ***The implementation model*** combines the outward manifestation of the computer based system(look and feel interface), coupled with all supporting information (books, manuals, videotapes, help) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively. In essence, these models enable the interface designer to satisfy a key element of the most important principle of user interface design: "*Know the user, know the tasks*."		
	b	Design the swim-lane diagram for ATM system	**[L6][CO3]**	**[6M]**


Customer
ATM
Bank
Insert ATM Card
Validate ATM Card
Enter PIN
Authorize PIN
[Yes]
[No]
Pin valid?
Enter Amount
Check Balance
Balance > Amount?
[No]
[Yes]
Get Cash
Debit Account
Display Balance
Get ATM Card
Eject Card

3	**a**	Briefly explain about User Interface Design Process	**[L2][CO4]**	**[6M]**

The analysis and design process for user interfaces is iterative and can berepresented using a spiral model. Referring to Figure, the four distinct framework activities:
(1) interface analysis and modeling, (2) interface design, (3) interface construction, and (4) interface validation. The spiral shown in Figure implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration ofrequirements and the resultant design.

Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, you work to understand the system perception for each class of users.

Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated.

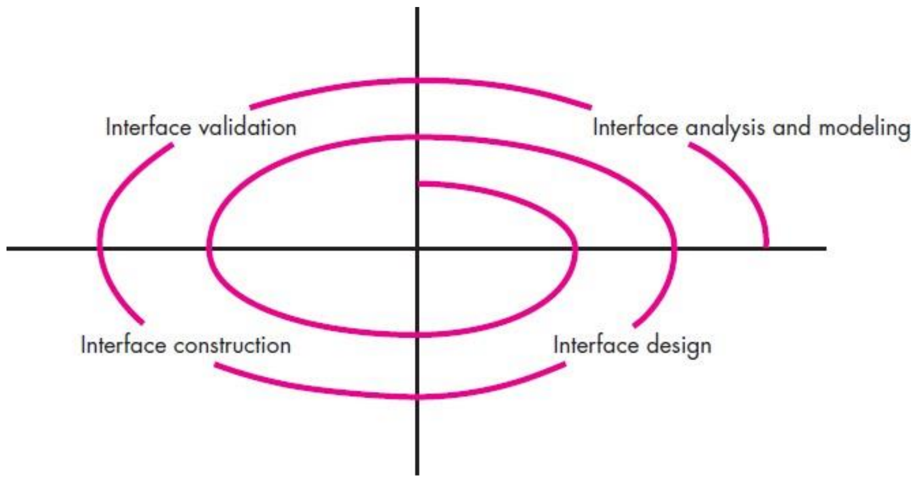


Fig 11.1: The user interface design process

Finally, analysis of the user environment focuses on the physical work environment. Among the questions to be asked are
• Where will the interface be located physically?
• Will the user be sitting, standing, or performing other tasks unrelated to the interface?
• Does the interface hardware accommodate space, light, or noise constraints?
• Are there special human factors considerations driven by environmental factors?

The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.
Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.
Interface validation focuses on (1) the ability of the interface to implement every user task

correctly, to accommodate all task variations, and to achieve all general user requirements; (2) the degree to which the interface is easy to use and easy to learn, and (3) the users' acceptance of the interface as a useful tool in their work.
Subsequent passes through the process elaborate task detail, design information, and theoperational features of the interface.

	b	Express the rules of User Interface Design.	**[L2][CO4]**	**[6M]**

The overall process for analyzing and designing a user interface begins with the creation of different models of system function. Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.

nterface Analysis and Design Models:
Four different models come into play when a user interface is to be analyzed and designed. To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality. In addition, users can be categorized as:

- ***Novices***. No syntactic knowledge of the system and little semantic knowledge of the application or computer usage in general.
- ***Knowledgeable, intermittent users***. Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.
- ***Knowledgeable***, ***frequent users***. Good semantic and syntactic knowledge that often leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction.

The user's ***mental model*** (system perception) is the image of the system that end users carry in their heads. For example, if the user of a particular word processor were asked to describe its operation, the system perception would guide the response.
The implementation model combines the outward manifestation of the computer based system(look and feel interface), coupled with all supporting information (books, manuals, videotapes, help) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively. In essence, these models enable the interface designer to satisfy a key element of the most important principle of user interface design: "*Know the user, know the tasks*."

4	**a**	Explain in detail about Task Analysis and Modeling	**[L2][CO2]**	**[6M]**

3.2 Task Analysis and Modeling:
The goal of task analysis is to answer the following questions:
• What work will the user perform in specific circumstances?
• What tasks and subtasks will be performed as the user does the work?
• What specific problem domain objects will the user manipulate as work is performed?
• What is the sequence of work tasks—the workflow?
• What is the hierarchy of tasks?
To answer these questions, you must use techniques that are applied to the user interface.
Use cases.
When used as part of task analysis, the use case is developed to show how an enduser performs some specific work-related task. The use case provides a basic description of oneimportant work

task for the computer-aided design system. From it, you can extract tasks,objects, and the overall flow of the interaction.

Task elaboration.

The stepwise elaboration is (also called functional decomposition or stepwise refinement) a mechanism for refining the processing tasks that are required for software to accomplish some desired function.

Task analysis can be applied in two ways. An interactive, computer-based system is often used to replace a manual or semi manual activity. To understand the tasks that must be performed to accomplish the goal of the activity, you must understand the tasks that people currently perform (when using a manual approach) and then map these into a similar (but not necessarily identical) set of tasks that are implemented in the context of the user interface.

Alternatively, you can study an existing specification for a computer-based solution and derive a set of user tasks that will accommodate the user model, the design model, and the system perception. Regardless of the overall approach to task analysis, you must first define andclassify tasks. Each of the major tasks can be elaborated into subtasks.

Object elaboration.

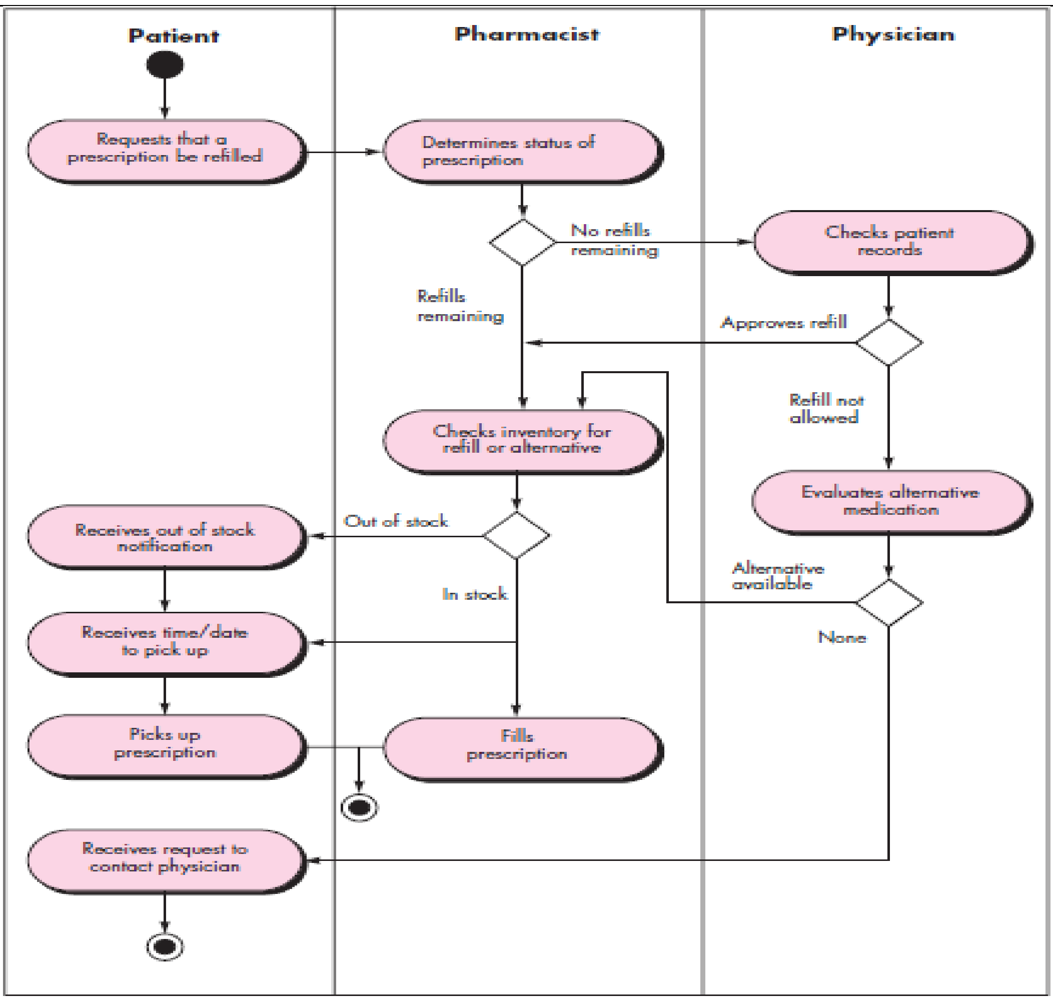
Rather than focusing on the tasks that a user must perform, you can examine the use case and other information obtained from the user and extract the physical objects that are used by the interior designer. These objects can be categorized into classes. Attributes of each class are defined, and an evaluation of the actions applied to each object provide a list of operations. The user interface analysis model would not provide a literal implementation for each of these operations. However, as the design is elaborated, the details ofeach operation are defined.

Workflow analysis.

When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply workflow analysis. This technique allows you to understand how a work process is completed when several people (and roles) are involved. Consider a company that intends to fully automate the process of prescribing and delivering prescription drugs. The entire process will revolve around a Web-based application that is accessible by physicians (or their assistants), pharmacists, and patients. Workflow can be represented effectively with a UML swimlane diagram (a variation on the activity diagram). See Figure

Regardless, the flow of events (shown in the figure) enables you to recognize a number of key interface characteristics:

1. Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different than the one defined for pharmacists or physicians.
2. The interface design for pharmacists and physicians must accommodate access to and display of information from secondary information sources (e.g., access to inventory for the pharmacist and access to information about alternative medications for the physician).
3. Many of the activities noted in the swimlane diagram can be further elaborated using task analysis and/or object elaboration (e.g., Fills prescription could imply a mail-order delivery, a visit to a pharmacy, or a visit to a special drug distribution center).


Patient
Pharmacist
Physician
Requests that a prescription be refilled
Determines status of prescription
No refills remaining
Checks patient records
Refills remaining
Approves refill
Checks inventory for refill or alternative
Refill not allowed
Evaluates alternative medication
Receives out of stock notification
Out of stock
Alternative available
In stock
None
Receives time/date to pick up
Picks up prescription
Fills prescription
Receives request to contact physician

Hierarchical representation. A process of elaboration occurs as you begin to analyze the interface. Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of each task identified for the user. For example, consider the following user task and subtask hierarchy.

User task: Requests that a prescription be refilled
• Provide identifying information.
• Specify name.
• Specify userid.
• Specify PIN and password.
• Specify prescription number.
• Specify date refill is required.
To complete the task, three subtasks are defined. One of these subtasks, provide identifying information, is further elaborated in three additional sub-subtasks.

	b	Write a short note on (i) Analysis of the Work Environment (ii) Analysis of Display Content	**[L2][CO3]**	**[6M]**

Analysis of the Work Environment: Hackos and Redish stated the importance of work environment analysis as:

People do not perform their work in isolation. They are influenced by the activity around them, the physical characteristics of the workplace, the type of equipment they are using, and the work relationships they have with other people. If the products you design do not fit into the environment, they may be difficult or frustrating to use.

Analysis of Display Content:

The user tasks lead to the presentation of a variety of different types of content.

These data objects may be

(1) generated by components (unrelated to the interface) in other parts of an application, (2) acquired from data stored in a database that is accessible from the application, or (3) transmitted from systems external to the application in question.

During this interface analysis step, the format and aesthetics of the content (as it is displayed by the interface) are considered. Among the questions that are asked and answered are:

- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right-hand corner)?
  • Can the user customize the screen location for content?
  • Is proper on-screen identification assigned to all content?
  • If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collectionsof data?
  • Will graphical output be scaled to fit within the bounds of the display device that is used?

The answers to these (and other) questions will help you to establish requirements for content presentation.

4	Dissect in brief about the various steps of Interface Design.	**[L4][CO5]**	**[12M]**

Once interface analysis has been completed, all tasks (or objects and actions) required by the end user have been identified in detail and the interface design activity commences. Interface design is an iterative process. Although many different user interface design models (e.g., have been proposed, all suggest some combination of the following steps:

1. Using information developed during interface analysis define interface objects and actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change. Modelthis 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 theinterface.

4.1 Applying Interface Design Steps:

The definition of interface objects and the actions that are applied to them is an important step in interface design. To accomplish this, user scenarios are parsed. That is, a use case is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

Once the objects and actions have been defined and elaborated iteratively, they are categorized by type.

Target, source, and application objects are identified. A source object (e.g., a report icon) is dragged and dropped onto a target object (e.g., a printer icon).

When you are satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed. Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items are conducted.

4.2 User Interface Design Patterns:

Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. As an example of a commonly encountered interface design problem, consider a situation in which a user must enter one or more calendar dates, sometimes months in advance. A vast array of interface design patterns has been proposed over the past decade.

4.3 Design Issues: As the design of a user interface evolves, four common design issues almost always surface: <u>*system response time*, *user help facilities*, *error information handling*, *and command labeling*</u>

- **Response time.** In general, system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.

<u>*System response time has two important characteristics*</u>: length and variability. If system response is too long, user frustration and stress are inevitable. Variability refers to the deviation from average response time.

- **Help facilities.** computer-based system requires help now and then. In most cases, however, modern software provides online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface. A number of design issues must be addressed when a help facility is considered:

- Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for all functions.
- How will the user request help? Options include a help menu, a special function key, or a HELP command.
- How will help be represented? Options include a separate window, a reference to a printed document (less than ideal), or a one- or two-line suggestion produced in a fixed screen location.
- How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or control sequence.

- **Error handling.** Error messages and warnings are "bad news" delivered to users of interactive systems when something has gone wrong. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration.

In general, every error message or warning produced by an interactive system should have the following characteristics:

- The message should describe the problem that the user can understand.
- The message should provide constructive advice for recovering from the error.
- The message should indicate any negative consequences of the error.
- The message should be accompanied by an audible or visual cue.
- The message should be "nonjudgmental." That is, the wording should never place blame on the user.

- **Menu and command labeling.** The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. A number of

design issues arise when typed commands or menu labels are provided as a mode of interaction:
• Will every menu option have a corresponding command?

- What form will commands take? Options include a control sequence (e.g., alt-P), function keys,or a typed word.
- How difficult will it be to learn and remember the commands? What can be done if a commandis forgotten?

• Can commands be customized or abbreviated by the user?
• Are menu labels self-explanatory within the context of the interface?
• Are submenus consistent with the function implied by a master menu item?

- **Application accessibility**. Accessibility for users (and software engineers) who may be *physically challenged* is an imperative for ethical, legal, and business reasons. A variety of accessibility guidelines - many designed for Web applications but often applicable to all types of software—provide detailed suggestions for designing interfaces that achieve varying levels of accessibility. Others provide specific guidelines for "assistive technology" that addresses the needs of thosewith visual, hearing, mobility, speech, and learning impairments.

- **Internationalization**. Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages. The challenge for interface designers is to create "globalized" software.

A variety of internationalization guidelines are available to software engineers. These guidelines address broad design issues (e.g., screen layouts may differ in various markets) and discrete implementation issues. The *Unicode standard* has been developed to address the daunting challenge of managing dozens of natural languages with hundreds of characters and symbols.

5	Examine the elements of interface analysis with examples.	**[L3][CO5]**	**[12M]**

INTERFACE ANALYSIS

A key tenet of all software engineering process models is: understand the problem before you attempt to design a solution. In the case of user interface design, understanding the problem means understanding (1) the people (end users) who will interact with the system through the interface, (2) the tasks that end users must perform to do their work, (3) the content that is presented as part of the interface, and (4) the environment in which these tasks will be conducted. We examine these elements of interface analysis with the intent of establishing a solidfoundation for the design tasks that follow.

3.1 User Analysis:
The only way that you can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use the system. Information from a broad array of sources can be used to accomplish this:
User Interviews. The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups.

Sales input. Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.

Marketing input. Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.

Support input. Support staff talks with users on a daily basis. They are the most likely source of information on what works and what doesn't, what users like and what they dislike, what features generate questions and what features are easy to use.

The following set of questions will help you to better understand the users of a system:

- Are users trained professionals, technicians, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?
- Do users work normal office hours or do they work until the job is done?
- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology that sits behind the interface?

Once these questions are answered, you'll know who the end users are, what is likely to motivate and please them, how they can be grouped into different user classes or profiles, what their mental models of the system are, and how the user interface must be characterized to meettheir needs.

3.3 Task Analysis and Modeling:

The goal of task analysis is to answer the following questions:

- What work will the user perform in specific circumstances?
- What tasks and subtasks will be performed as the user does the work?
- What specific problem domain objects will the user manipulate as work is performed?
- What is the sequence of work tasks—the workflow?
- What is the hierarchy of tasks?

To answer these questions, you must use techniques that are applied to the user interface.

Use cases.

When used as part of task analysis, the use case is developed to show how an enduser performs some specific work-related task. The use case provides a basic description of oneimportant work task for the computer-aided design system. From it, you can extract tasks,objects, and the overall flow of the interaction.

Task elaboration.

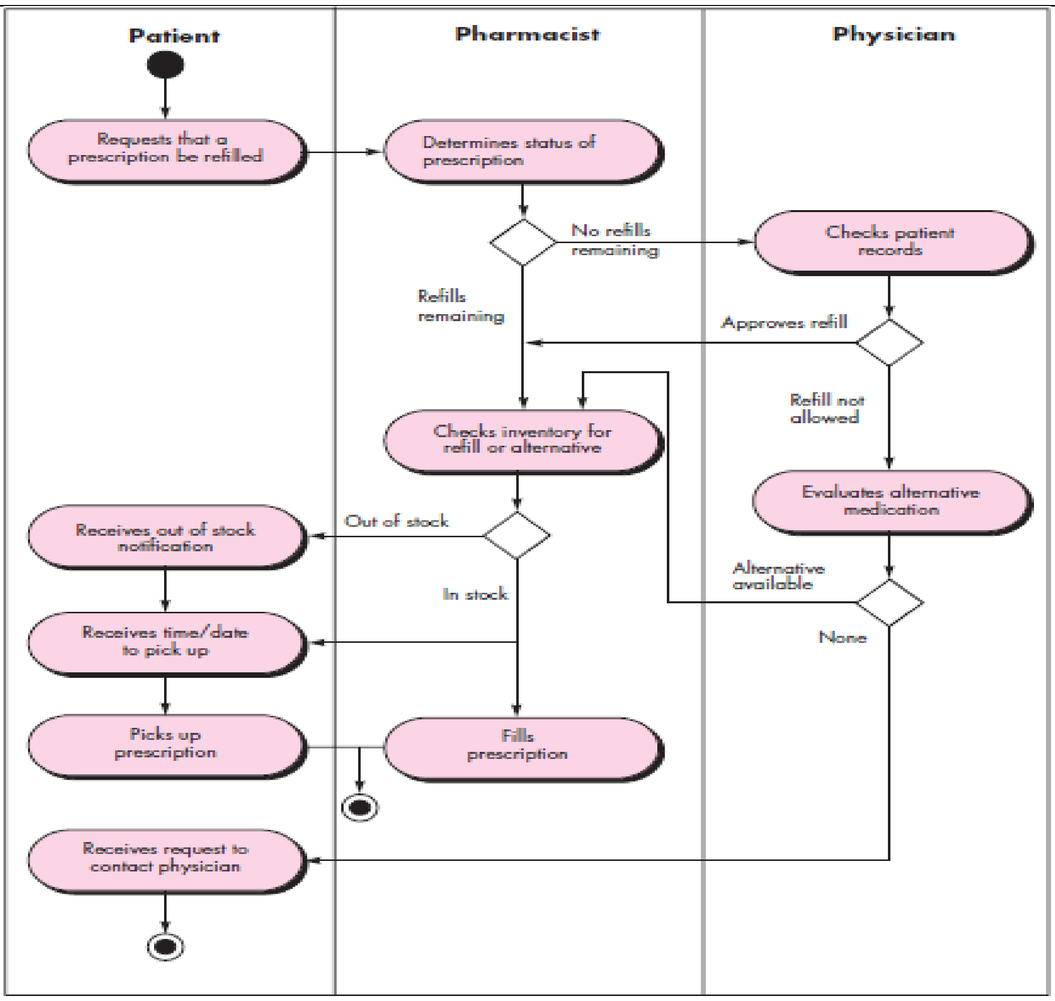
The stepwise elaboration is (also called functional decomposition or stepwise refinement) a mechanism for refining the processing tasks that are required for software to accomplish some desired function.

Task analysis can be applied in two ways. An interactive, computer-based system is often used to replace a manual or semi manual activity. To understand the tasks that must be performed to accomplish the goal of the activity, you must understand the tasks that people currently perform (when using a manual approach) and then map these into a similar (but not necessarily identical) set of tasks that are implemented in the context of the user interface.

Alternatively, you can study an existing specification for a computer-based solution and derive a set of user tasks that will accommodate the user model, the design model, and the system perception. Regardless of the overall approach to task analysis, you must first define andclassify tasks. Each of the major tasks can be elaborated into subtasks.

Object elaboration.
Rather than focusing on the tasks that a user must perform, you can examine the use case and other information obtained from the user and extract the physical objects that are used by the interior designer. These objects can be categorized into classes. Attributes of each class are defined, and an evaluation of the actions applied to each object provide a list of operations. The user interface analysis model would not provide a literal implementation for each of these operations. However, as the design is elaborated, the details ofeach operation are defined.
Workflow analysis.
When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply workflow analysis. This technique allows you to understand how a work process is completed when several people (and roles) are involved. Consider a company that intends to fully automate the process of prescribing and delivering prescription drugs. The entire process will revolve around a Web-based application that is accessible by physicians (or their assistants), pharmacists, and patients. Workflow can be represented effectively with a UML swimlane diagram (a variation on the activity diagram). See Figure
Regardless, the flow of events (shown in the figure) enables you to recognize a number of key interface characteristics:

4. Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different than the one defined for pharmacists or physicians.
5. The interface design for pharmacists and physicians must accommodate access to and display of information from secondary information sources (e.g., access to inventory for the pharmacist and access to information about alternative medications for the physician).
6. Many of the activities noted in the swimlane diagram can be further elaborated using task analysis and/or object elaboration (e.g., Fills prescription could imply a mail-order delivery, a visitto a pharmacy, or a visit to a special drug distribution center).


Patient
Pharmacist
Physician
Requests that a prescription be refilled
Determines status of prescription
No refills remaining
Checks patient records
Refills remaining
Approves refill
Refill not allowed
Checks inventory for refill or alternative
Evaluates alternative medication
Receives out of stock notification
Out of stock
Alternative available
In stock
None
Receives time/date to pick up
Picks up prescription
Fills prescription
Receives request to contact physician

Hierarchical representation. A process of elaboration occurs as you begin to analyze the interface. Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of each task identified for the user. For example, consider the following user task and subtask hierarchy.

User task: Requests that a prescription be refilled
• Provide identifying information.
• Specify name.
• Specify userid.
• Specify PIN and password.
• Specify prescription number.
• Specify date refill is required.
To complete the task, three subtasks are defined. One of these subtasks, provide identifyinginformation, is further elaborated in three additional sub-subtasks.
Analysis of Display Content:
The user tasks lead to the presentation of a variety of different types of content.
These data objects may be
(1) generated by components (unrelated to the interface) in other parts of an application, (2) acquired from data stored in a database that is accessible from the application, or (3) transmitted from systems external to the application in question.
During this interface analysis step, the format and aesthetics of the content (as it is displayed by the interface) are considered. Among the questions that are asked and answered are:

• Are different types of data assigned to consistent geographic locations on the screen (e.g.,photos

always appear in the upper right-hand corner)?
• Can the user customize the screen location for content?
• Is proper on-screen identification assigned to all content?
• If a large report is to be presented, how should it be partitioned for ease of understanding?

- Will mechanisms be available for moving directly to summary information for large collectionsof data?

• Will graphical output be scaled to fit within the bounds of the display device that is used?
The answers to these (and other) questions will help you to establish requirements for content presentation.

3.4 Analysis of the Work Environment: Hackos and Redish stated the importance of work environment analysis as:
People do not perform their work in isolation. They are influenced by the activity around them, the physical characteristics of the workplace, the type of equipment they are using, and the work relationships they have with other people. If the products you design do not fit into the environment, they may be difficult or frustrating to use.

6	a	Explain Interface Design workflow for WebApps.	[L2][CO5]	[6M]

Interface Design Workflow for WebApps:
Information contained within the requirements model forms the basis for the creation of a screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. The following tasks represent a rudimentary workflow for WebApp interface design:

1. Review information contained in the requirements model and refine as required.

2. Develop a rough sketch of the WebApp interface layout. An interface prototype (including the layout) may have been developed as part of the requirements modeling activity.

3. Map user objectives into specific interface actions. For the vast majority of WebApps, the user will have a relatively small set of primary objectives. These should be mapped into specific interface actions.

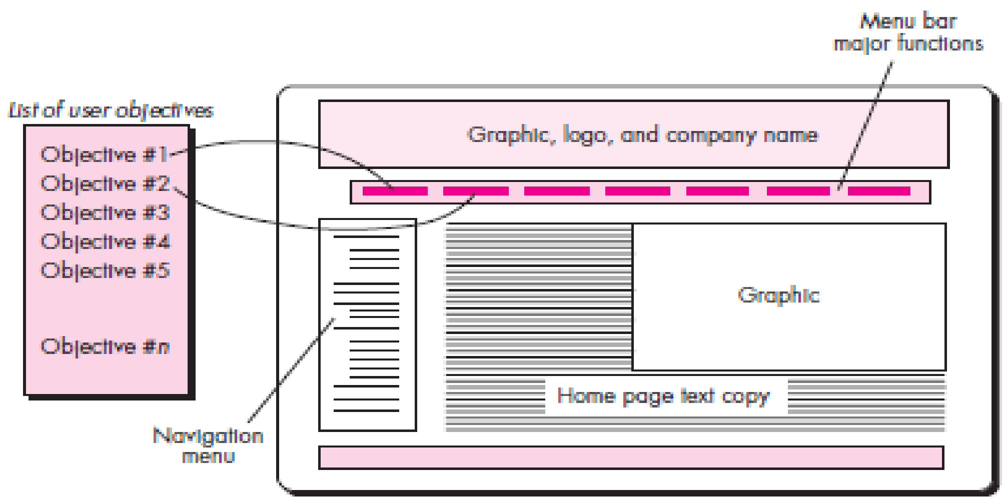


Fig: Mapping User Objectives into interface actions

4. Define a set of user tasks that are associated with each action. Each interface action (e.g., “buy a product”) is associated with a set of user tasks. These tasks have been identified during requirements modeling.

5. Storyboard screen images for each interface action. As each action is considered, a sequence of storyboard images (screen images) should be created to depict how the interface responds to user interaction.

6. Refine interface layout and storyboards using input from aesthetic design. In most cases, you'll be responsible for rough layout and storyboarding. Aesthetic design is integrated with the work performed by the interface designer.

7. Identify user interface objects that are required to implement the interface. This task may require a search through an existing object library to find those reusable objects (classes) that are appropriate for the WebApp interface.

8. Develop a procedural representation of the user's interaction with the interface. This optional task uses UML sequence diagrams and/or activity diagrams to depict the flow of activities (and decisions) that occur as the user interacts with the WebApp.

9. Develop a behavioral representation of the interface. This optional task makes use of UML state diagrams to represent state transitions and the events that cause them. Control mechanisms (i.e., the objects and actions available to the user to alter a WebApp state) are defined.

10. Describe the interface layout for each state. Using design information developed in Tasks 2 and 5, associate a specific layout or screen image with each WebApp state described in Task

11. Refine and review the interface design model. Review of the interface should focus on usability. It is important to note that the final task set you choose should be adapted to the special requirements of the application that is to be built.

	b	Organize the steps involved in WebApp Interface Design.	**[L4][CO5]**	**[6M]**

The user interface of a WebApp is its "first impression."

Effective interfaces do not concern the user with the inner workings of the system. Effective applications and services perform a maximum of work, while requiring a minimum of information from users.

In order to design WebApp interfaces that exhibit these characteristics:

- **Communication**. The interface should communicate the status of any activity initiated by the user. Communication can be obvious (e.g., a text message) or subtle (e.g., an image of a sheet of paper moving through a printer to indicate that printing is under way).
- **Consistency**. The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout) should be consistent throughout the WebApp
- **Controlled autonomy.** The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- **Efficiency**. The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the developer who designs and builds it or the client server environment that executes it.
- **Flexibility**. The interface should be flexible enough to enable some users to accomplish tasks directly and others to explore the WebApp in a somewhat random fashion.
- **Focus**. The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.
- **Fitt's law.** "The time to acquire a target is a function of the distance to and size of the target".
- **Human interface objects**. A vast library of reusable human interface objects has been developed for WebApps. Use them.
- **Latency reduction.** Rather than making the user wait for some internal operation to complete, the WebApp should use multitasking in a way that lets the user proceed with work as if the operation has been completed.
- **Learnability**. A WebApp interface should be designed to minimize learning time, and once

learned, to minimize relearning required when the WebApp is revisited.

- **Metaphors**. An interface that uses an interaction metaphor is easier to learn and easier to use, as long as the metaphor is appropriate for the application and the user.
- **Maintain work product integrity.** A work product must be automatically saved so that it will not be lost if an error occurs.
- **Readability**. All information presented through the interface should be readable by young and old.
- **Track state**. When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return.
- **Visible navigation.** A well-designed WebApp interface provides "the illusion that users are in the same place, with the work brought to them".

7	**a**	Define five quality attributes of WebApp Design.	**[L1][CO5]**	**[6M]**

Usability :
Global site understandability
Online feedback and help feature
Interface and aesthetic features
Special features
Functionality :-
Navigation and browsing features
Searching and retrieving capability
Application domain-related features
Reliability :-
Correct link processing
Error Recovery
User input validation and recovery
Efficiency :-
Response time performance
Page generation speed
Graphics generation speed
Maintainability :-
Ease of correction
Adaptability
Extendibility

	b	Explain in detail about Aesthetic design.	**[L2][CO5]**	**[6M]**

Aesthetic design, also called *graphic design*, is an artistic endeavor that complements the technical aspects of WebApp design.

Layout Issues:
Like all aesthetic issues, there are no absolute rules when screen layout is designed. However, a number of general layout guidelines are worth considering:

- **Don't be afraid of white space**. It is inadvisable to pack every square inch of a Web page with information.
- **Emphasize content.** Nielsen suggests that the typical Web page should be 80 percent content with the remaining real estate dedicated to navigation and other features.
- **Organize layout elements from top-left to bottom-right**. The vast majority of users will scan a Web page in much the same way as they scan the page of a book—top-left to bottom-right. If layout elements have specific priorities, high-priority elements should be placed in the upper-left portion of the page real estate.

- **Group navigation, content, and function geographically within the page.** Humans look for patterns in virtually all things.
- **Don't extend your real estate with the scrolling bar.** Although scrolling is often necessary, most studies indicate that users would prefer not to scroll.
- **Consider resolution and browser window size when designing layout.** Rather than defining fixed sizes within a layout, the design should specify all layout items as a percentage of availablespace.

Graphic Design Issues:

Graphic design considers every aspect of the look and feel of a WebApp. The graphic design process begins with layout and proceeds into a consideration of global color schemes; text types, sizes, and styles; the use of supplementary media (e.g., audio, video, animation); and all other aesthetic elements of an application.

8	Write a short note on Content Design.	**[L2][CO3]**	**[12M]**

CONTENT DESIGN

Content design focuses on two different design tasks. First, a design representation for *content objects* and the mechanisms required to establish their relationship to one another is developed. Second, the *information* within a specific content object is created.

8.1 Content Objects:.

A content object has attributes that include content-specific information (normally defined during WebApp requirements modeling) and implementation-specific attributes that are specified as part of design.

8.2 Content Design Issues:

Once all content objects are modeled, the information that each object is to deliver must be authored and then formatted to best meet the customer's needs. Content authoring is the job of specialists in the relevant area who design the content object by providing an outline of information to be delivered and an indication of the types of generic content objects (e.g., descriptive text, graphic images, photographs) that will be used to deliver the information. Aesthetic design may also be applied to represent the proper look and feel for the content.

1 Content Architecture:

The design of content architecture focuses on the definition of the overall hypermedia structure of the WebApp. Although custom architectures are sometimes created, you always have the option of choosing from four different content structures

Linear structures **are encountered when a predictable sequence of interactions (with some variation or diversion) is common.**

Grid structures **are an architectural option that you can apply when WebApp content can be organized categorically in two (or more) dimensions.**

Hierarchical structures are undoubtedly the most common WebApp architecture.

A networked or "pure web" **structure is similar in many ways to the architecture that evolves for object-oriented systems.**

The architectural structures can be combined to form *composite structures*.

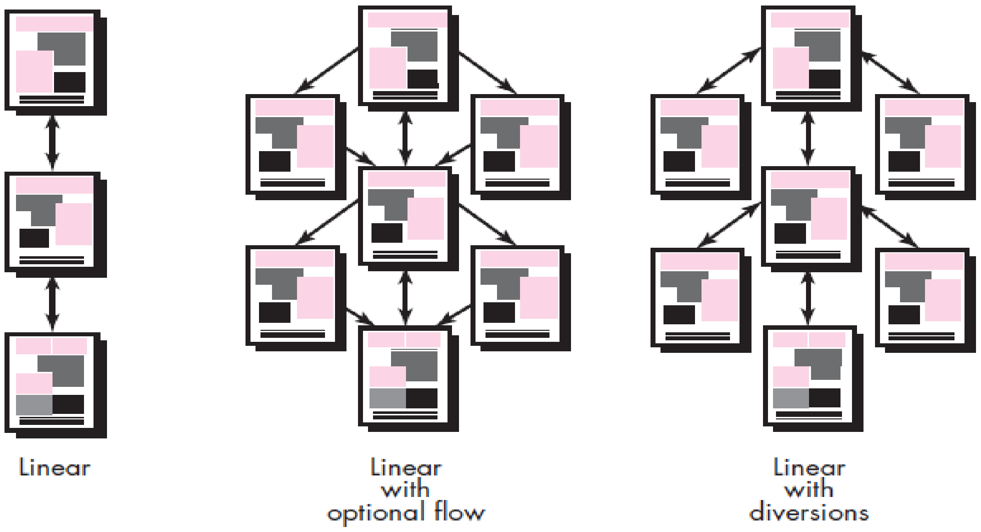


Fig 13.4: Linear Structures

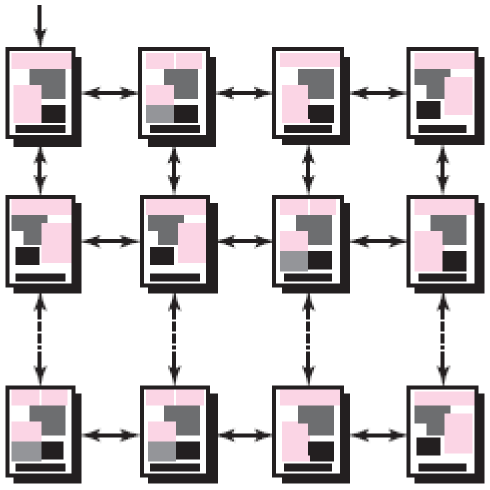


Fig 13.5: Grid Structure

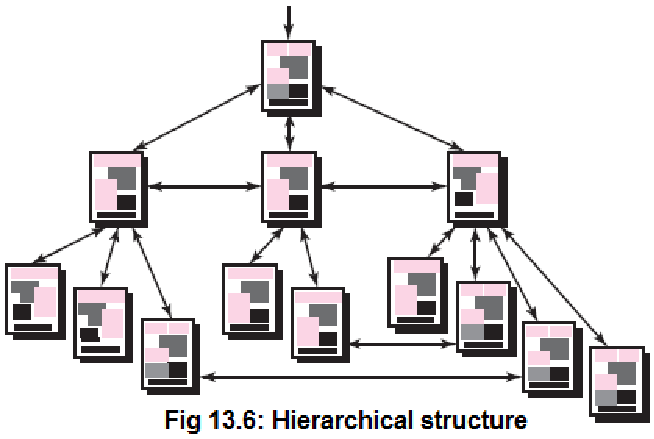


Fig 13.6: Hierarchical structure

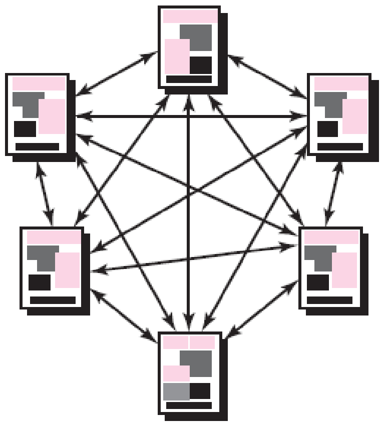


Fig 13.7: Network structure

9	Give detailed notes on architecture design.	[L2][CO5]	[12M]

ARCHITECTURE DESIGN

Content architecture focuses on the manner in which content objects (or composite objects such as Web pages) are structured for presentation and navigation.

In most cases, architecture design is conducted in parallel with interface design, aesthetic design, and content design.

Content Architecture:

The design of content architecture focuses on the definition of the overall hypermedia structure of the WebApp. Although custom architectures are sometimes created, you always have the option of choosing from four different content structures

Linear structures are encountered when a predictable sequence of interactions(with some variation or diversion) is common.
Grid structures are an architectural option that you can apply when WebAppcontent can be organized categorically in two (or more) dimensions.
Hierarchical structures are undoubtedly the most common WebApp architecture.
A networked or "pure web" structure is similar in many ways to the architecture thatevolves for object-oriented systems.
The architectural structures can be combined to form *composite structures*.

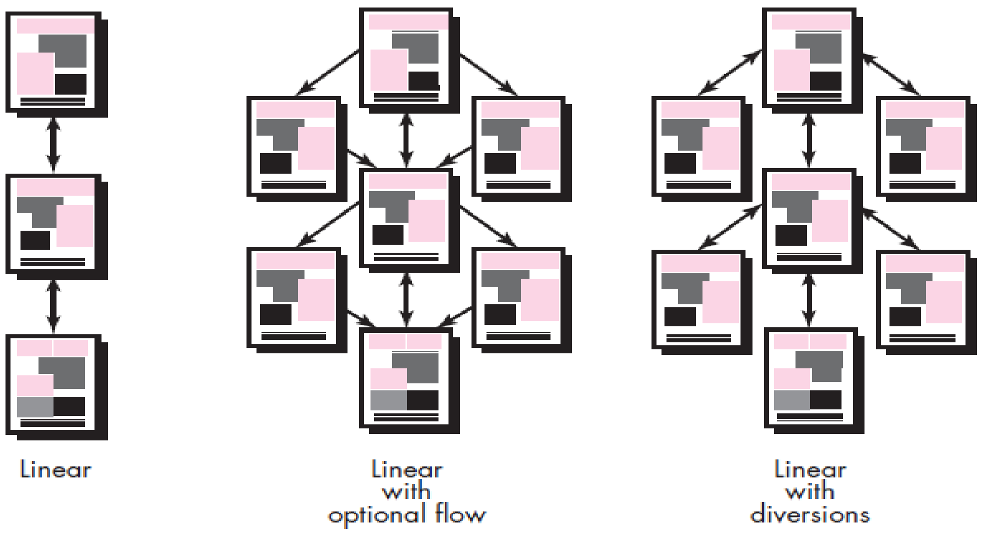


Fig 13.4: Linear Structures

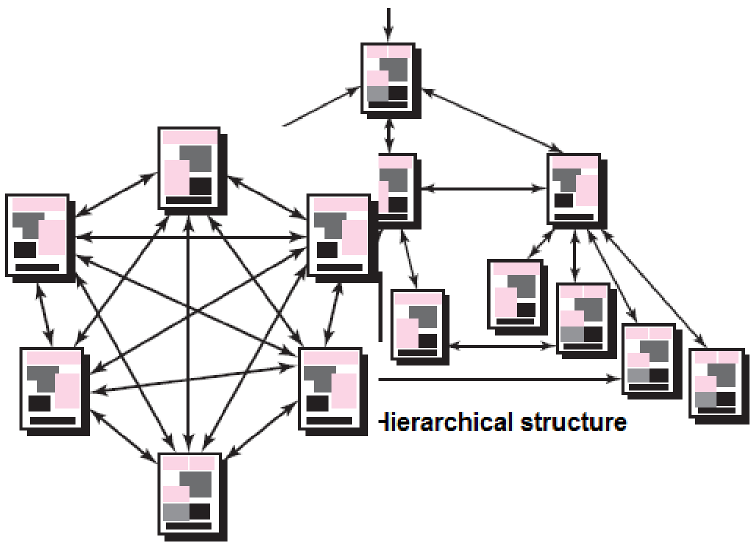


Fig 13.7: Network structure

10	**a**	Dissect in brief about the various steps of Navigation Design.	**[L4][CO5]**	**[6M]**

10 NAVIGATION DESIGN

Once the WebApp architecture has been established and the components (pages, scripts, applets, and other processing functions) of the architecture have been identified, you must define navigation pathways that enable users to access WebApp content and functions. To accomplish this, you should (1) identify the semantics of navigation for different users of the site, and (2) define the mechanics (syntax) of achieving the navigation.

10.1 Navigation Semantics:

Like many WebApp design actions, navigation design begins with a consideration of the user hierarchy and related use cases developed for each category
of user (actor). Each actor may use the WebApp somewhat differently and therefore have different navigation requirements. A series of navigation semantic units (NSUs)—“a set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements. The overall navigation structure for a WebApp may be organized as a hierarchy of NSUs.

10.2 Navigation Syntax: As design proceeds, your next task is to define the mechanics of navigation. A number of options are available as you develop an approach for implementing each NSU:

- *Individual navigation link*—includes text-based links, icons, buttons and switches, and graphical metaphors. You must choose navigation links that are appropriate for the content and consistent with the heuristics that lead to high-quality interface design.
- *Horizontal navigation bar*—lists major content or functional categories in a bar containing appropriate links. In general, between four and seven categories are listed.
- *Vertical navigation column*—(1) lists major content or functional categories, or (2) lists virtually all major content objects within the WebApp. If you choose the second option, such navigation columns can “expand” to present content objects as part of a hierarchy (i.e., selecting an entry in the original column causes an expansion that lists a second layer of related content objects).
- *Tabs*—a metaphor that is nothing more than a variation of the navigation bar or column, representing content or functional categories as tab sheets that are selected when a link is required.
- *Site maps*—provide an all-inclusive table of contents for navigation to all content objects and functionality contained within the WebApp. In addition to choosing the mechanics of navigation, you should also establish appropriate navigation conventions and aids.

	b	Examine the elements of component level design.	**[L3][CO5]**	**[6M]**

Modern WebApps deliver increasingly sophisticated processing functions that (1) perform localized processing to generate content and navigation capability in a dynamic fashion, (2) provide computation or data processing capability that are appropriate for the WebApp’s business domain, (3) provide sophisticated database query and access, and (4) establish data interfaces with external corporate systems. To achieve these (and many other) capabilities, you must design and construct program components that are identical in form to software components for traditional software.

UNIT –V

TESTING AND TESTING CONVENTIONAL APPLICATIONS

1	**a**	Distinguish between Verification and Validation with example	**[L5][CO4]**	**[6M]**

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

Verification: "Are we building the product right?"

Validation: "Are we building the right product?"

s.no	Verification	Validation
1	Verification addresses the concern: "Are you building it right?"	Validation addresses the concern: "Are you building th right thing?"
2	Ensures that the software system meets all the functionality.	Ensures that the functionalities meet the intended behavior.
3	Verification takes place first and includes the checking for documentation, code, etc.	Validation occurs after verification and mainly involves the checking of the overall product.
4	Done by developers.	Done by testers.
5	It has static activities, as it includes collecting reviews, walkthroughs, and inspections to verify a software.	It has dynamic activities, as it includes executing the software against the requirements.
6	It is an objective process and no subjective decision should be needed to verify a software.	It is a subjective process and involves subjective decisions on how well a software works.

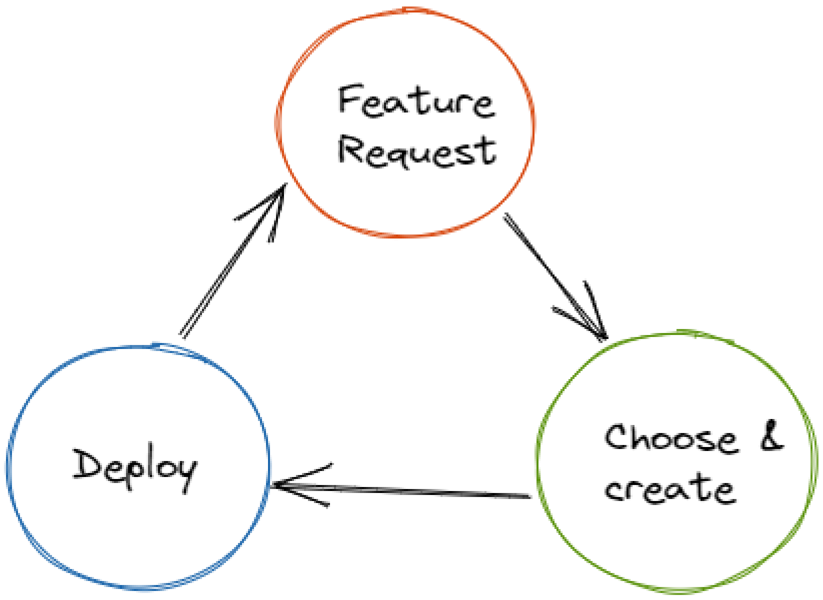
	b	What is Software Testing? Why it is important before deploying the software.	**[L4][CO5]**	**[6M]**

Software Testing is a set of activities that can be planned in advance and conducted systematically.

Software deployment" is different from "software release," which refers to an application's iterative development process. More functionality, bug-fixing optimization, and other features could be included in a new software release.

The generalized deployment process can be broken down into three steps:

1. Examine a request for a new feature.
2. Choose, create, and release
3. Install the application.



The release phase of the deployment process entails moving all of the activities required for packaging the application to the platform on which it will execute.

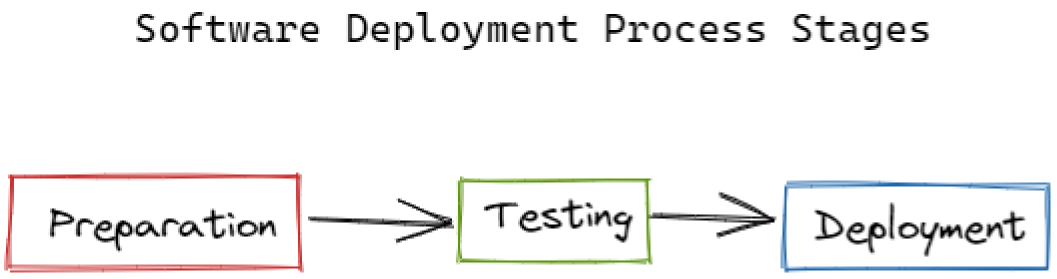
The deployment process can be carried out manually or automatically. Automated deployment has supplanted human approaches in recent years because of its efficiency.

There is currently a wide range of software deployment platforms available. Two of the most popular are Amazon Web Services (AWS) and Google Cloud Platform (GCP).

Software Deployment Process Stages

For everything to go smoothly, you'll need a well-thought-out software deployment process. Organizations should create their processes, which might be based on their specific business needs or simply on industry best practices, depending on the situation.

The deployment process is divided into three stages.



1. **Preparation**
   Begin by gathering all of the code that has to be deployed. This is a crucial step in the process since it ensures that only the correct code is deployed. The code that has to be deployed, as well as any configuration files, libraries, or resources, can be packaged as a single resource at this point. To do so, look through the project management software for completed user stories or simply the original message that prompted the deployment to begin.

2. **Testing**
   It is not recommended to deploy new software until it has been thoroughly tested. This is done to ensure that there are no flaws or bugs later on. The software must first be deployed to a staging environment, where all required tests can be run through a set of pre-configured tests. The results

must then be reviewed by developers, who must subsequently remedy any bugs discovered.

3. **Deployment**
   The software can only go live when these critical stages have been completed. The new code is merged into the production environment at this point. This process is followed exactly as it is in the test environment, thus there should be little to no likelihood of issues at this point. Checking for issues on the live server is a good idea just to be safe.

2	**a**	Explain in brief about the levels/steps in Software Testing	**[L2][CO4]**	**[6M]**

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. The steps are shown in Figure. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name *unit testing.*

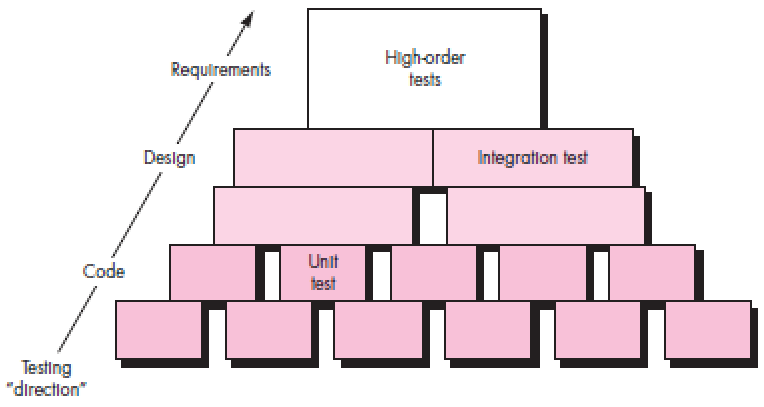


Fig: Software Testing Steps

Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.
Integration testing addresses the issues associated with the dual problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration
Validation testing provides final assurance that software meets all informational, functional, behavioral, and performance requirements.
System testing: The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). *System testing* verifies that all elements mesh properly and that overall system function/performance is achieved.

	b	Discriminate the strategic approach to software testing.	**[L5][CO5]**	**[6M]**

The software process may be viewed as the spiral illustrated in Figure. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established.

A strategy for software testing may also be viewed in the context of the spiral. ***Unit testing*** begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code. ***integration testing***, where the focus is on design and the construction of the software architecture.

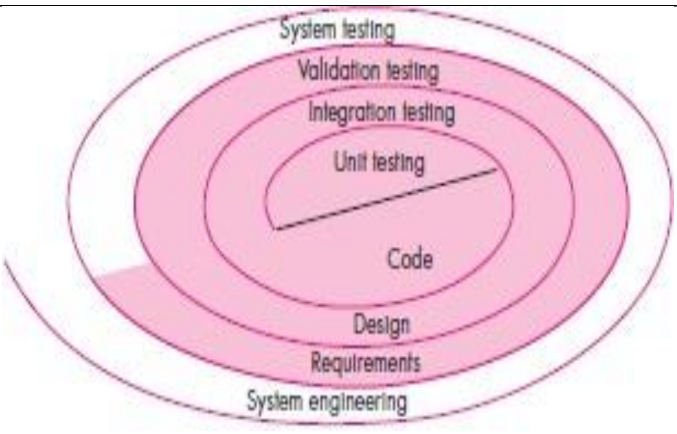


Fig: Testing Strategy

validation testing, where requirements established as part of requirements modeling are validated against the software that has been constructed. Finally, ***system testing***, where the software and other system elements are tested as a whole.

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. The steps are shown in Figure. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name *unit testing.*

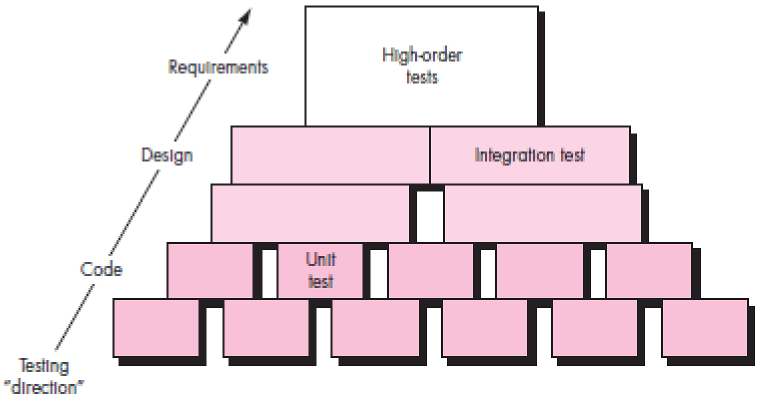


Fig: Software Testing Steps

Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.
Integration testing addresses the issues associated with the dual problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration
Validation testing provides final assurance that software meets all informational, functional, behavioral, and performance requirements.
System testing: The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). *System testing* verifies that all elements mesh properly and that overall system function/performance is achieved.

3	Explain about the importance of test strategies in conventional software.	**[L2][CO5]**	**[12M]**

TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

There are many strategies that can be used to test software.

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.

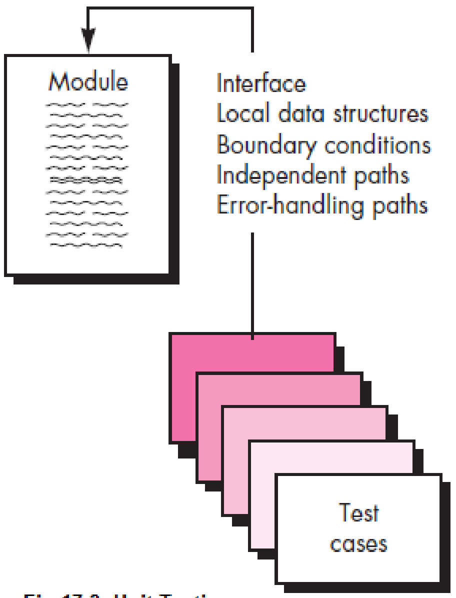


Fig 17.3: Unit Testing

Unit-test considerations. Unit tests are illustrated schematically in Figure. 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.

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.

Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the n*th* 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. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories. Each test case should be coupled with a set of expected results.

Because a component is not a stand-alone program, *driver* and/or *stub* software must often be developed for each unit test. The unit test environment is illustrated in Figure. 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.

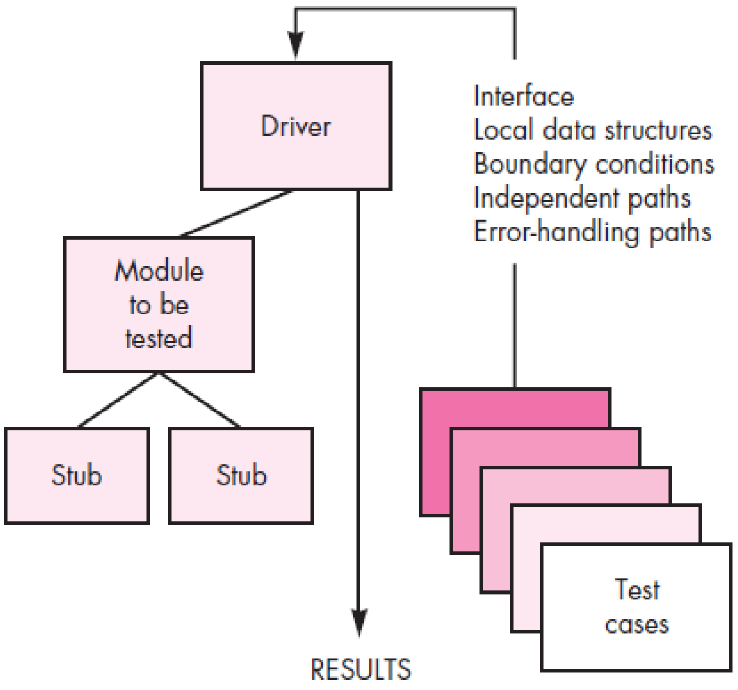


Fig 17.4: Unit-test environment

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.

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

There is often a tendency to attempt non incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. *The entire program is tested as a whole*.

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.

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

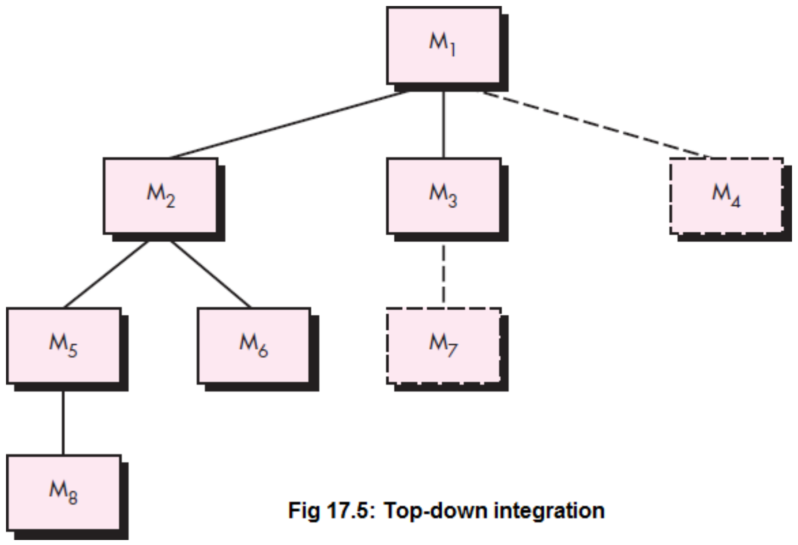


Fig 17.5: Top-down integration

Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner. Referring to Figure, depth- first integration integrates all components on a major control path of the program structure. Breadth-first integration incorporates allcomponents directly subordinate at each level, moving across the structure horizontally.

From the figure, components M2, M3, and M4 would be integrated first. The next control level,M5, M6, and so on, follows. The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all componentsdirectly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinatestubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced.The process continues from step 2 until the entire program structure is built.

The top-down integration strategy verifies major control or decision points early in the test process.
As a tester, you are left with three choices:
(1) delay many tests until stubs are replaced with actual modules,
(2) develop stubs that perform limited functions that simulate the actual module, or
(3) integrate the software from the bottom of the hierarchy upward.

Bottom-up integration. Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

Integration follows the pattern illustrated in Figure

.

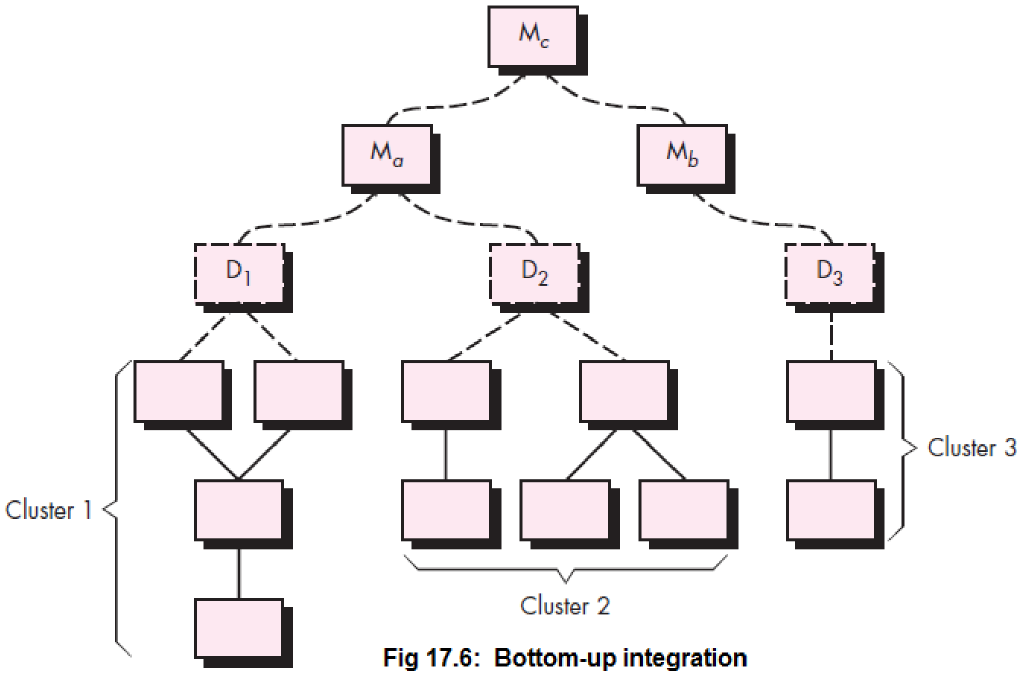


Fig 17.6: Bottom-up integration

Regression testing. Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, *regression testing is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.*
Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
Regression testing may be conducted manually, by re executing a subset of all test cases or using automated capture/playback tools. The regression test suite (the subset of tests to be executed) 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 software components that have been changed.
As integration testing proceeds, the number of regression tests can grow quite large.

Smoke testing. Smoke testing is an integration testing approach that is commonly used when product software is developed. It is designed as a pacing mechanism for *time-critical projects*, allowing the software team to assess the project on a frequent basis. In essence, the smoke- testing approach encompasses the following activities:
1. Software components that have been translated into code are integrated into a build. A build includes all data files, libraries, reusable modules, and engineered components that are requiredto implement one or more product functions.
2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "showstopper" errors that have the highest likelihood of throwing the software project behind schedule.
3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

Strategic options. There has been much discussion about the relative advantages and disadvantages of top-down versus bottom-up integration testing. In general, *the advantages of one strategy tend to result in disadvantages for the other strategy*. Selection of an integration strategy depends upon software characteristics and, sometimes, project schedule. In general, a combined approach (sometimes called sandwich testing) that uses top-down tests for upper levels of the program structure, coupled with bottom-up tests for subordinate levels may be the best compromise.

As integration testing is conducted, the tester should identify critical modules. A critical module has one or more of the following characteristics: (1) addresses several software requirements,
(2) has a high level of control (resides relatively high in the program structure), (3) is complex or error prone, or (4) has definite performance requirements. Critical modules should be tested as early as is possible. In addition, regression tests should focus on critical module function.

Integration test work products. An overall plan for integration of the software and a description of specific tests is documented in a Test Specification. Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software. The following criteria and corresponding tests are applied for all test phases:
Interface integrity. Internal and external interfaces are tested as each module (or cluster) is incorporated into the structure.
Functional validity. Tests designed to uncover functional errors are conducted.
Information content. Tests designed to uncover errors associated with local or global data structures are conducted.
Performance. Tests designed to verify performance bounds established during software design are conducted.

4	Explain in brief about System Testing. How it differs from Validation Testing.	**[L2][CO6]**	**[12M]**

SYSTEM TESTING

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system.

Recovery Testing: *Many computer-based systems must recover from faults and resume processing with little or no downtime*.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the *mean-time-to-repair* (MTTR) is evaluated to determine whether it is within acceptable limits.

Security Testing: *Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.*

During security testing, the tester plays the role(s) of the individual who desires to penetrate the system.

Stress Testing: Stress testing executes a system in a manner that demands resources inabnormal quantity, frequency, or volume.

Performance Testing: Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. That is, it is often necessary to measure *resource utilization* (e.g., processor cycles) in an exacting fashion.

Deployment Testing: In many cases, software must execute on a variety of platforms and under more than one operating system environment. *Deployment testing, sometimes called configuration testing, exercises the software in each environment in which it is to operate*. In addition, deployment testing examines all installation procedures and specialized installation software (e.g., “installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.

Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. At the validation or system level, the distinction between conventional software, object-oriented software, and WebApps disappears. *Testing focuses on user-visible actions and user-recognizable output from the system*.

Validation-Test Criteria: Software validation is achieved through a series of tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied.

After each validation test case has been conducted, one of two possible conditions exists: (1) The function or performance characteristic conforms to specification and is accepted or (2) a deviation from specification is uncovered and a deficiency list is created.

Configuration Review: An important element of the validation process is a configuration rw. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities. The configuration review, sometimes called an **audit**.

Alpha and Beta Testing: *Conducted by the end user rather than software engineers, an **acceptance test** can range from an informal “test drive” to a planned and systematically executed series of tests*. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

The ***alpha test*** is conducted at the developer’s site by a representative group of end users. The software is used in a natural setting with the developer “looking over the shoulder” of the users and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The ***beta test*** is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a “live” application of the software in an environment that cannot be controlled by the developer.

A variation on beta testing, called ***customer acceptance testing***, is sometimes performed when custom software is delivered to a customer under contrac

5	**a**	Discuss the process of Art of Debugging.	**[L2][CO5]**	**[6M]**

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art.

The Debugging Process: Debugging is not testing but often occurs as a consequence of testing. Referring to Figure, the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the non corresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will usually have one of two outcomes:
(1) the cause will be found and corrected or
(2) the cause will not be found.

Why is debugging so difficult?:
1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by non errors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.

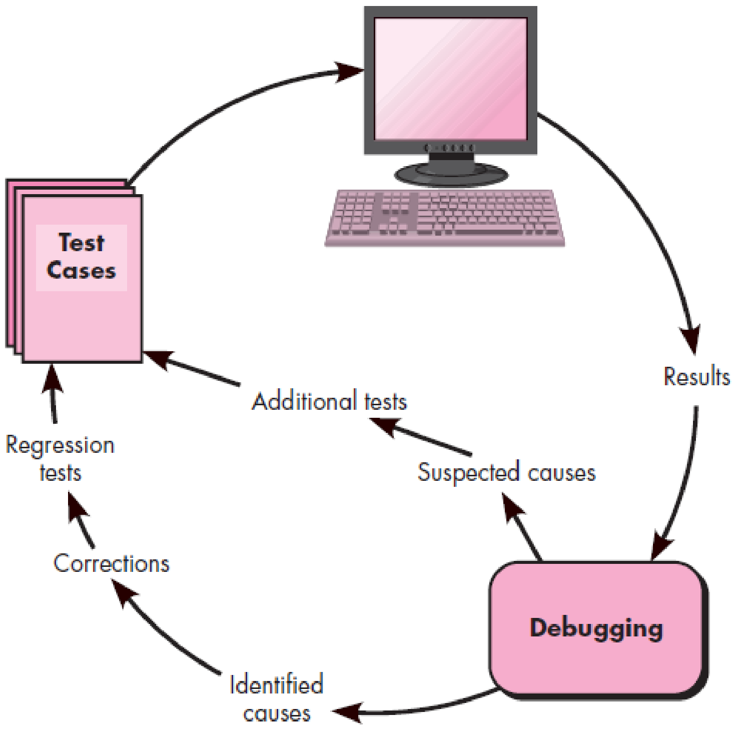


Fig 17.7: The debugging process

5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions.
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

	b	Difference between Alpha and Beta testing?	**[L4][CO6]**	**[6M]**

Alpha and Beta Testing: *Conducted by the end user rather than software engineers, an* ***acceptance test*** *can range from an informal "test drive" to a planned and systematically executed series of tests*. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

The ***alpha test*** is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The ***beta test*** is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer

6	**a**	Write a short note on fundamentals of software testing.	**[L2][CO4]**	**[6M]**

SOFTWARE TESTING FUNDAMENTALS

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. The tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

Testability. "Software testability is simply how easily [a computer program] can be tested." The following characteristics lead to testable software.

Operability. "The better it works, the more efficiently it can be tested." If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests.

Observability. "What you see is what you test." Inputs provided as part of testing produce distinct outputs.

Controllability. "The better we can control the software, the more the testing can be automated and optimized." All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured.

Decomposability. "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."

Simplicity. "The less there is to test, the more quickly we can test it." The program should exhibit functional simplicity.

Stability. "The fewer the changes, the fewer the disruptions to testing." Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests.

Understandability. "The more information we have, the smarter we will test." The architectural design and the dependencies between internal, external, and shared components are well understood.

Test Characteristics : *A good test has a high probability of finding an error.* To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.

A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test.

A good test should be "best of breed". In a group of tests that have a similar intent, time and

resource limitations may mitigate toward the execution of only a subset of these tests.
A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors.

	b	Identify the Object-Oriented Testing Methods and Explain	**[L3][CO6]**	**[6M]**

4.1 Unit Testing in the OO Context: 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 packages attributes (data) and the operations that manipulate these data. An encapsulated class is usually the focus of unit testing. However, operations (methods) within the class are the smallest testable units. 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 tactics applied to unit testing must change.

4.2 Integration Testing in the OO Context: Because object-oriented software does not have an obvious hierarchical control structure, traditional top-down and bottom-up integration strategies have little meaning.
There are two different strategies for integration testing of OO systems. The first, *thread-based testing*, integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. *Regression testing* is applied to ensure that no side effects occur. The second integration approach, *use-based testing*, begins the construction of the system by testing those classes (called independent classes) that use very few (if any) server classes. After the independent classes are tested, the next layer of classes, calleddependent classes, that use the independent classes are tested. This sequence of testing layers of *dependent classes* continues until the entire system is constructed.

7	Explain in detail about Black box testing with its types	**[L2][CO6]**	**[12M]**

Black-box testing, also called *behavioral testing*, focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than whitebox methods.
Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external database access, (4) behavior or performance errors, and (5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, blackbox testing tends to be applied during later stages of testing. Tests are designed to answer the following questions:
• How is functional validity tested?
• How are system behavior and performance tested?
• What classes of input will make good test cases?
• Is the system particularly sensitive to certain input values?
• How are the boundaries of a data class isolated?
• What data rates and data volume can the system tolerate?
• What effect will specific combinations of data have on system operation?

By applying black-box techniques, you derive a set of test cases that satisfy the following criteria

(1) Test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonabletesting, and
(2) Test cases that tell you something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

10.1 Graph-Based Testing Methods: The first step in black-box testing is to understand the objects that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another". Stated in another way, software testing begins by creating a graph of important objects and their relationships and

then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

To accomplish these steps, you begin by creating a graph—a collection of nodes that represent *objects*, *links* that represent the relationships between objects, node weights that describe the properties of a node (e.g., a specific data value or state behavior), and *link weights* that describe some characteristic of a link.

The symbolic representation of a graph is shown in Figure 18.8a. Nodes are represented as circles connected by links that take a number of different forms.

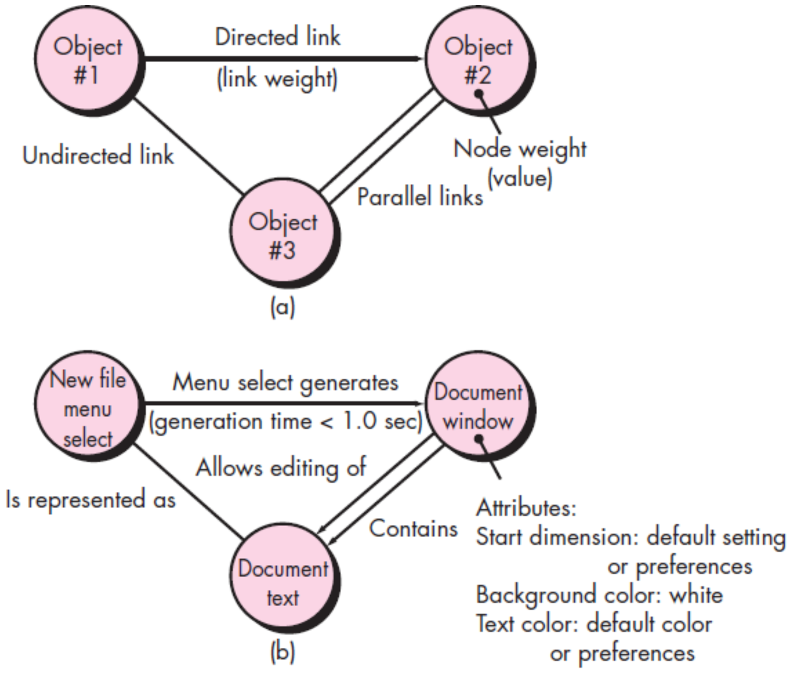


Fig 18.8: a)Graph notation b) Simple Example

A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction. A *bidirectional link*, also called a *symmetric link*, implies that the relationship applies in both directions. Parallel links are used when a number of different relationships are established between graph nodes.

As a simple example, consider a portion of a graph for a word-processing application (Figure 18.8b) where

Object #1 _ **newFile** (menu selection)
Object #2 _ **documentWindow**
Object #3 _ **documentText**

Referring to the figure, a menu select on **newFile** generates a document window. The node weight of

documentWindow provides a list of the window attributes that are to be expected when the window is generated. The *link weight* indicates that the window must be generated in less than 1.0 second. An *undirected link* establishes a symmetric relationship between the **newFile** menu selection and **documentText**, and parallel links indicate relationships between **documentWindow** and **documentText**.

Transaction flow modeling. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an online service), and the links represent the logical connection between steps (e.g., **flightInformationInput** is followed by **validationAvailabilityProcessing**). The data flow diagram can be used to assist in creating graphs of this type.

Finite state modeling. The nodes represent different user-observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., orderInformation is verified during **inventoryAvailabilityLook-up** and is followed by **customerBillingInformation input**). The state diagram can be used to assist in creating graphs of this type.
Data flow modeling. The nodes are data objects, and the links are the transformations that occur to translate one data object into another. For example, the node FICA tax withheld (FTW)is computed from gross wages (GW) using the relationship.
Timing modeling. The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

10.2 Equivalence Partitioning: *Equivalence partitioning* is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.
Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. If a set of objects can be linked by relationships that are *symmetric, transitive, and reflexive*, an equivalence class is present. An equivalence class represents a set of *valid* or *invalid states* for input conditions.
Equivalence classes may be defined according to the following guidelines:
1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classesare defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence classare defined.
4. If an input condition is Boolean, one valid and one invalid class are defined.

10.3 Boundary Value Analysis: A greater number of errors occurs at the boundaries of the input domain rather than in the "center." It is for this reason that boundary value analysis (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class.

Guidelines for BVA are similar in many respects to those provided for *equivalence partitioning*:
1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.

2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.
4. If internal program data structures have prescribed boundaries, be certain to design a test case to exercise the data structure at its boundary.

10.4 Orthogonal Array Testing: *Orthogonal array testing* can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding *region faults*—an error category associated with faulty logic within a software component.

To illustrate the difference between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has three input items, X, Y, and Z. Each of these input items has three discrete values associated with it. There are $3^3 = 27$ possible test cases.

Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure).

When orthogonal array testing occurs, an L9 *orthogonal array* of test cases is created. The L9 orthogonal array has a "*balancing property*". That is, test cases (represented by dark dots in the figure) are "dispersed uniformly throughout the test domain," as illustrated in the right-hand cube in Figure 18.9. Test coverage across the input domain is more complete.

To illustrate the use of the L9 orthogonal array, consider the send function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the send function.
If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1,P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3). Phadke assesses these test cases by stating:

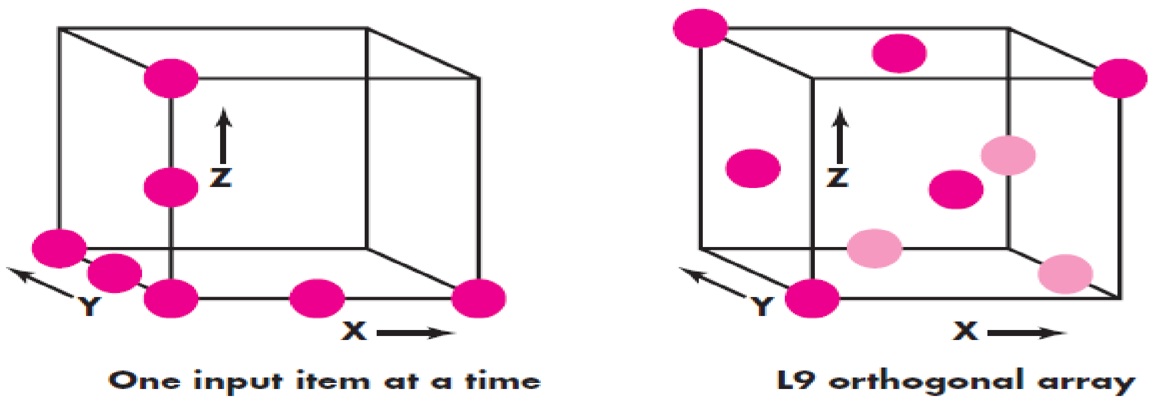
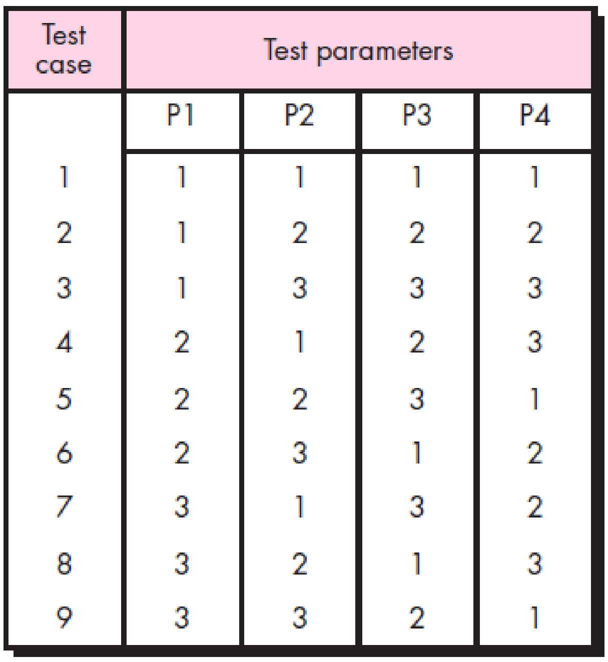


Fig 18.9: A geometric view of test cases

The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax send function is illustrated in Figure 18.10.



Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Fig 18.10: An L9 Orthogonal array

Phadke assesses the result of tests using the L9 orthogonal array in the following manner:

Detect and isolate all single mode faults. A single mode fault is a consistent problem with any level of any single parameter. For example, if all test cases of factor P1 =1 cause an error condition, it is a single mode failure.

Detect all double mode faults. If there exists a consistent problem when specific levels of two parameters occur together, it is called a double mode fault. Indeed, a double mode fault is an indication of pairwise incompatibility or harmful interactions between two test parameters.

Multimode faults. Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multimode faults are also detected by these tests.

8	Describe briefly about White box testing with its types	**[L2][CO6]**	**[12M]**

White-box testing, sometimes called *glass-box testing*, is a test- case design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, you can derive test cases that

(1) guarantee that all independent paths within a module have been exercised at least once,

(2) exercise all logical decisions on their true and false sides,

(3) execute all loops at their boundaries and within their operational bounds, and

(4) exercise internal data structures to ensure their validity.

A major White box testing technique is Code Coverage analysis. Code Coverage analysis eliminates gaps in a Test Case suite. It identifies areas of a program that are not exercised by a set of test cases. Once gaps are identified, you create test cases to verify untested parts of the code, thereby increasing the quality of the software product

There are automated tools available to perform Code coverage analysis. Below are a few coverage analysis techniques a box tester can use:

Statement Coverage:- This technique requires every possible statement in the code to be tested at least once during the testing process of software engineering.

Branch Coverage – This technique checks every possible path (if-else and other conditional loops) of a software application.

Apart from above, there are numerous coverage types such as Condition Coverage, Multiple Condition Coverage, Path Coverage, Function Coverage etc. Each technique has its own merits and attempts to test (cover) all parts of software code. **Using Statement and Branch coverage you generally attain 80-90% code coverage which is sufficient.**

Following are important WhiteBox Testing Techniques:

- Statement Coverage
- Decision Coverage
- Branch Coverage
- Condition Coverage
- Multiple Condition Coverage
- Finite State Machine Coverage
- Path Coverage
- Control flow testing
- Data flow testing

Types of White Box Testing

White box testing encompasses several testing types used to evaluate the usability of an application, block of code or specific software package. There are listed below —

- **Unit Testing:** It is often the first type of testing done on an application. Unit Testing is performed on each unit or block of code as it is developed. Unit Testing is essentially done by the programmer. As a software developer, you develop a few lines of code, a single function or an object and test it to make sure it works before continuing Unit Testing helps identify a majority of bugs, early in the software development lifecycle. Bugs identified in this stage are cheaper and easy to fix.
- **Testing for Memory Leaks**: Memory leaks are leading causes of slower running applications. A QA specialist who is experienced at detecting memory leaks is essential in cases where you have a slow running software application.

Apart from the above, a few testing types are part of both black box and white box testing. They are listed below

- **White Box Penetration Testing:** In this testing, the tester/developer has full information of the application's source code, detailed network information, IP addresses involved and all server information the application runs on. The aim is to attack the code from several angles to expose security threats.
- **White Box Mutation Testing**: Mutation testing is often used to discover the best coding techniques to use for expanding a software solution.

9	**a**	What are the Testing Methods applicable at the Class Level? Explain.	**[L1][CO6]**	**[6M]**
		Testing "in the small" *focuses on a single class* and the *methods* that are encapsulated by the class. *Random testing* and *partitioning* are methods that can be used to exercise a class during OO testing.		

Random Testing for OO Classes: To provide brief illustrations of these methods, consider a banking application in which an Account class has the following operations: open(), setup(), deposit(), withdraw(), balance(), summarize(), creditLimit(), and close(). Each of these operations may be applied for Account, but certain constraints are implied by the nature of the problem. Even with these constraints, there are many permutations of the operations.

Partition Testing at the Class Level: Partition testing reduces the number of test cases required to exercise the class in much the same manner as equivalence partitioning for traditional software. Input and output are categorized and test cases are designed to exercise each category. But how are the partitioning categories derived?

State-based partitioning categorizes class operations based on their ability to change the stateof the class.

Attribute-based partitioning categorizes class operations based on the attributes that they use. For the Account class, the attributes balance and creditLimit can be used to define partitions. Operations are divided into three partitions: (1) operations that use creditLimit, (2) operations that modify creditLimit, and (3) operations that do not use or modify creditLimit. Test sequences are then designed for each partition.

Category-based partitioning categorizes class operations based on the *generic function* that each performs. For example, operations in the Account class can be categorized in initialization operations (open, setup), computational operations (deposit, withdraw), queries (balance, summarize, creditLimit), and termination operations (close).

	b	Illustrate Testing Strategies for Object Oriented software	**[L3][CO6]**	**[6M]**

Test-case design methods for object-oriented software continue to evolve. However, an overall approach to OO test-case design has been suggested by Berard

1. Each test case should be uniquely identified and explicitly associated with the class to be tested.
2. The purpose of the test should be stated.
3. A list of testing steps should be developed for each test and should contain:
    a. A list of specified states for the class that is to be tested
    b. A list of messages and operations that will be exercised as a consequence of the test
    c. A list of exceptions that may occur as the class is tested
    d. A list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)
    e. Supplementary information that will aid in understanding or implementing the test.

The Test-Case Design Implications of OO Concepts: "Testing requires reporting on the concrete and abstract state of an object." Yet, encapsulation can make this information somewhat difficult to obtain. Unless built-in operations are provided to report the values for class attributes, a snapshot of the state of an object may be difficult to acquire.

Inheritance may also present you with additional challenges during test-case design. It is already noted that each new usage context requires retesting, even though reuse has beenachieved. In addition, multiple inheritance complicates testing further by increasing the numberof contexts for which testing is required.

Applicability of Conventional Test-Case Design Methods: The white-box testing methods can be applied to the operations defined for a class. Basis path, loop testing, or data flow techniques can help to ensure that every statement in an operation has been tested.

Fault-Based Testing: The tester looks for plausible faults. To determine whether these faults exist, test cases are designed to exercise the design or code. If real faults in an OO system are perceived to be implausible, then this approach is really no better than any random testing technique.
Integration testing looks for plausible faults in operation calls or message connections. Three types of faults are encountered in this context: unexpected result, wrong operation/message used, and incorrect invocation. To determine plausible faults as functions (operations) are invoked, the behavior of the operation must be examined. Integration testing applies to attributes as well as to operations. The "behaviors" of an object are defined by the values that its attributes are assigned.

13.4 Test Cases and the Class Hierarchy: *Inheritance* complicate the testing process. Consider the following situation. A **class Base** contains operations **inherited()** and **redefined().** A **class Derived** redefines redefined() to serve in a local context.
It is important to note, however, that only a subset of all tests for Derived::inherited() may have to be conducted. If part of the design and code for inherited() does not depend on redefined(), that code need not be retested in the derived class. Base::redefined() and Derived::redefined() are two different operations with different specifications and implementations. Those test requirements probe for plausible faults: *integration faults, condition faults, boundary faults, andso forth*. But the operations are likely to be similar. Their sets of test requirements will overlap.

Scenario-Based Test Design: Fault-based testing misses two main types of errors:
(1) incorrect specifications and
(2) interactions among subsystems.
When errors associated with an incorrect specification occur, the product doesn't do what the customer wants. It might do the wrong thing or omit important functionality.

Scenario-based testing concentrates on what the user does, not what the product does. Scenarios uncover interaction errors. But to accomplish this, test cases must be more complex and more realistic than fault-based tests.

Testing Surface Structure and Deep Structure: *Surface structure* refers to the *externally observable structure of an OO program*. That is, the structure that is immediately obvious to an end user. Rather than performing functions, the users of many OO systems may be given objects to manipulate in some way.

Deep structure refers to the internal technical details of an OO program, that is, the structure thatis understood by examining the design and/or code. Deep structure testing is designed to exercise dependencies, behaviors, and communication mechanisms that have been established as part of the design model for OO software.

10	**a**	How to test Specialized Environments, Architectures and Applications.	**[L2][CO6]**	**[6M]**

Unique guidelines and approaches to testing are sometimes warranted when specialized environments, architectures, and applications are considered.

11.1 Testing GUIs: Graphical user interfaces (GUIs) will present you with interesting testing challenges. Finite-state modeling graphs may be used to derive a series of tests that address specific data and program objects that are relevant to the GUI. Because of the large number of permutations associated with GUI operations, GUI testing should be approached using automated tools. A wide array of GUI testing tools has appeared on the market over the past few years.

11.2 Testing of Client-Server Architectures: In fact, recent industry studies indicate a significant increase in testing time and cost when client-server environments are developed.

In general, the testing of client-server software occurs at three different levels:

(1) Individual client applications are tested in a "disconnected" mode; the operation of the server and the underlying network are not considered. (2) The client software and associated server applications are tested in concert, but network operations are not explicitly exercised. (3) The complete client-server architecture, including network operation and performance, is tested. Although many different types of tests are conducted at each of these levels of detail, the following testing approaches are commonly encountered for client-server applications:

• **Application function tests**. The functionality of client applications is tested. In essence, the application is tested in stand-alone fashion in an attempt to uncover errors in its operation.

• **Server tests.** The coordination and data management functions of the server are tested. Server performance (overall response time and data throughput) is also considered.

• **Database tests.** The accuracy and integrity of data stored by the server is tested. Transactions posted by client applications are examined to ensure that data are properly stored, updated, and retrieved. Archiving is also tested.

• **Transaction tests.** A series of tests are created to ensure that each class of transactions is processed according to requirements. Tests focus on the correctness of processing and also on performance issues.

• **Network communication tests.** These tests verify that communication among the nodes of the network occurs correctly and that message passing, transactions, and related network traffic occur without error. Network security tests may also be conducted as part of these tests.

	b	Describe interclass test case design.	**[L2][CO6]**	**[6M]**

Test-case design becomes more complicated as integration of the object-oriented system begins. *It is at this stage that testing of collaborations between classes must begin*.

Like the testing of individual classes, class collaboration *testing can be accomplished by applying random and partitioning methods, as well as scenario-based testing and behavioral testing.*

Multiple Class Testing:

The following sequence of steps togenerate multiple class random test cases:

1. For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes.

2. For each message that is generated, determine the collaborator class and the corresponding operation in the server object.

3. For each operation in the server object, determine the messages that it transmits.

4. For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.

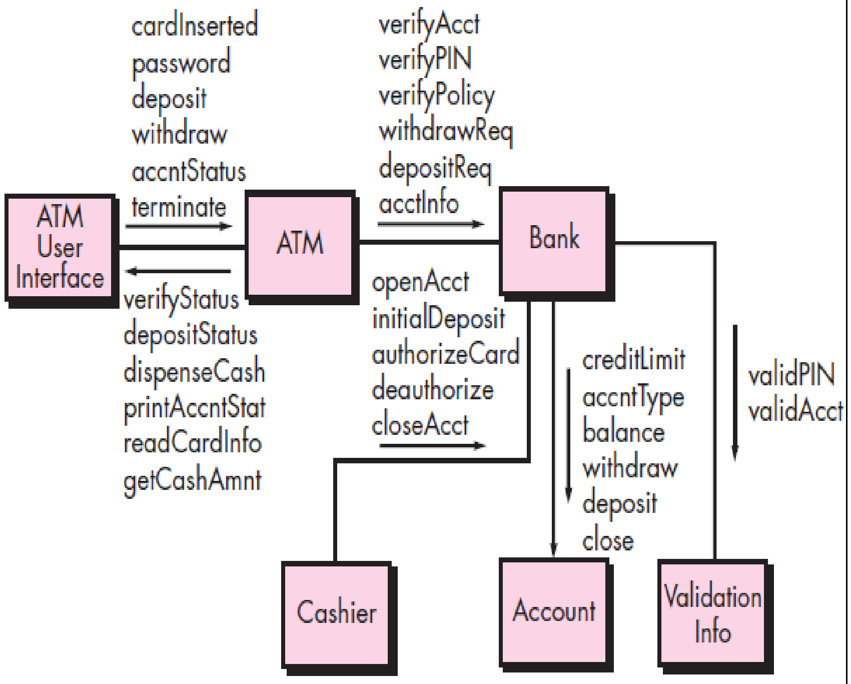


Fig 19.2: Class Collaboration diagram for banking application

15.2 Tests Derived from Behavior Models: The use of the *state diagram* as a model that represents the dynamic behavior of a class. Figure 19.3 illustrates a state diagram for the *Account class*.

The tests to be designed should achieve coverage of every state. That is, the operation sequences should cause the Account class to make transition through all allowable states:

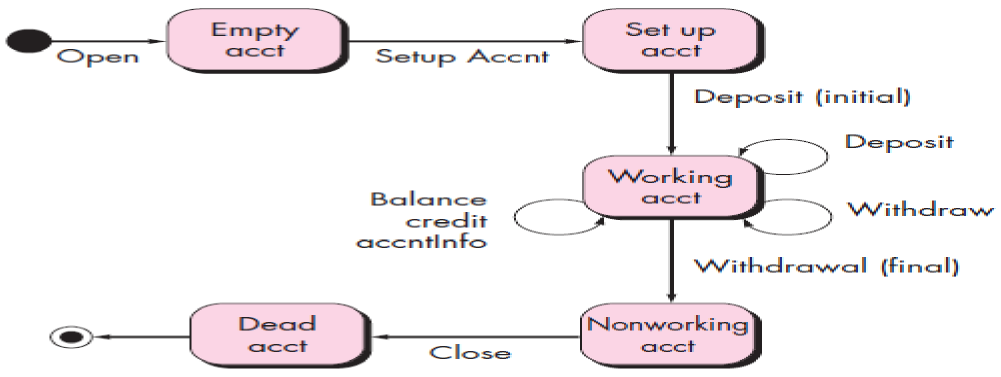


Fig 19.3: State diagram for the Account class

Prepared by:
Dr. B.Geethavani, Prof., CSE
Dr.R.G.Kumar, Assoc. Prof., CSE
Mr.B.Sarvesan, Assoc. Prof., CSE