

1	<p>a) Describe the essence of software engineering practice.</p> <p>1. Understand the problem (communication and analysis): Who has a stake in the solution to the problem? That is, who are the stakeholders? What are the unknowns? What data, functions, and features are required to properly solve the problem? Can the problem be compartmentalized? Is it possible to represent smaller problems that may be easier to understand? Can the problem be represented graphically? Can an analysis model be created?</p> <p>2. Plan a solution (modelling a software design): Have you seen similar problems before? Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required? Has a similar problem been solved? If so, are elements of the solution reusable? Can sub problems be defined? If so, are solutions readily apparent for the sub problems? Can you represent a solution in a manner that leads to effective implementation? Can a design model be created?</p> <p>3. Carry out a plan (code generation): The design we create serves as a road map the system we want to create. Does the solution conform the plan? Is the source code traceable to the design model? Is each part of the component part of the solution provably correct? Has the design and code had correctness proofs been applied to the algorithm?</p> <p>4. Examine the result for accuracy (testing and quality assurance): You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible. Is it possible to test each component part of the solution? Has a reasonable testing strategy been implemented? Does the solution produce results that conform to the data, functions, and features that are required? Has the software been validated against all stakeholder requirements?</p>
	<p>b) Briefly explain various specialized process models.</p> <p>Component-Based Development: Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software components. Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):</p> <ol style="list-style-type: none"> 1. Available component-based products are researched and evaluated for the application domain in question. 2. Component integration issues are considered. 3. A software architecture is designed to accommodate the components. 4. Components are integrated into the architecture. 5. Comprehensive testing is conducted to ensure proper functionality. The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits.

The Formal Methods Model:

The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called cleanroom software engineering, is currently applied by some software development organizations.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through ad hoc review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable you to discover and correct errors that might otherwise go undetected. Although not a mainstream approach, the formal methods model offers the promise of defect-free software. Yet, concern about its applicability in a business environment has been voiced:

- The development of formal models is currently quite time consuming and expensive.
- Because few software developers have the necessary background to apply formal methods, extensive training is required.
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

Aspect-Oriented Software Development:

Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modeled as components (e.g., object oriented classes) and then constructed within the context of a system architecture. As modern computer-based systems become more sophisticated (and complex), certain concerns—customer required properties or areas of technical interest—span the entire architecture. Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).

When concerns cut across multiple system functions, features, and information, they are often referred to as crosscutting concerns. Aspectual requirements define those crosscutting concerns that have an impact across the software architecture. Aspect-oriented software development (AOSD), often referred to as aspect-oriented programming (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing aspects.

c) Write 12 agility principles for those who want to achieve agility in their software development process.

The Agile Alliance defines 12 agility principles for those who want to achieve agility:

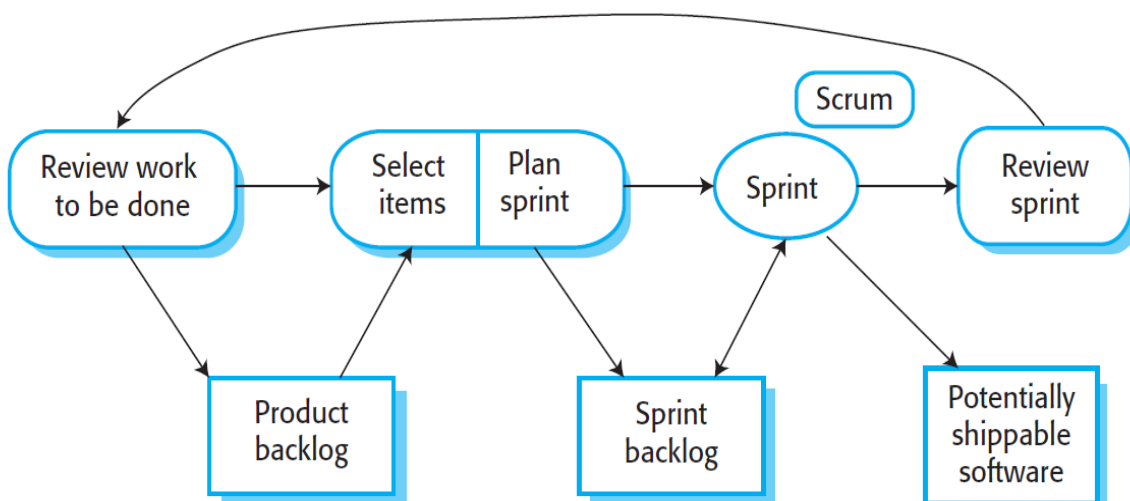
1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of 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. Give them the environment and support they need, and trust them to get the job done.
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. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
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.

2 a) **What is meant by Industrial XP? Write an XP user story that describes the “Favorites” feature available on most of the web browsers.**

- Internet bookmarks are stored webpage locations that can be retrieved. The main purpose is to easily catalog and access web pages that a user has visited and choose to save.
- Saved links are called “favorites”, and by virtue of the browser’s large market share, the term favorite has been synonymous with bookmark since the early days of widely – distributed browsers.
- Bookmark are normally visible in a browser menu and stored on the user’s computer and commonly a metaphor is be used for organization.

b) **With a neat diagram, describe the overall flow of the Scrum process.**



Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development activities:

- Backlog —a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time. The product manager assesses the backlog and updates priorities as required.
- Sprints —consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box 10 (typically 30 days). Changes are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.

		<ul style="list-style-type: none"> ➤ Scrum meetings —are short (typically 15-minute) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members: <ul style="list-style-type: none"> • What did you do since the last team meeting? • What obstacles are you encountering? • What do you plan to accomplish by the next team meeting? ➤ A team leader, called a Scrum master, leads the meeting and assesses the responses from each person. ➤ The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to “knowledge socialization” and thereby promote a self-organizing team structure. ➤ Demos —deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. ➤ It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.
	c)	<p>Provide three examples of software projects that would be amenable to the component-based model. Explain your answer with justification.</p> <p>A holiday booking management system which includes hotel, air tickets booking, cab/taxi booking in a particular destination.</p> <p>Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software components. Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):</p> <ol style="list-style-type: none"> 1. Available component-based products are researched and evaluated for the application domain in question. 2. Component integration issues are considered. 3. A software architecture is designed to accommodate the components. 4. Components are integrated into the architecture. 5. Comprehensive testing is conducted to ensure proper functionality. The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits.

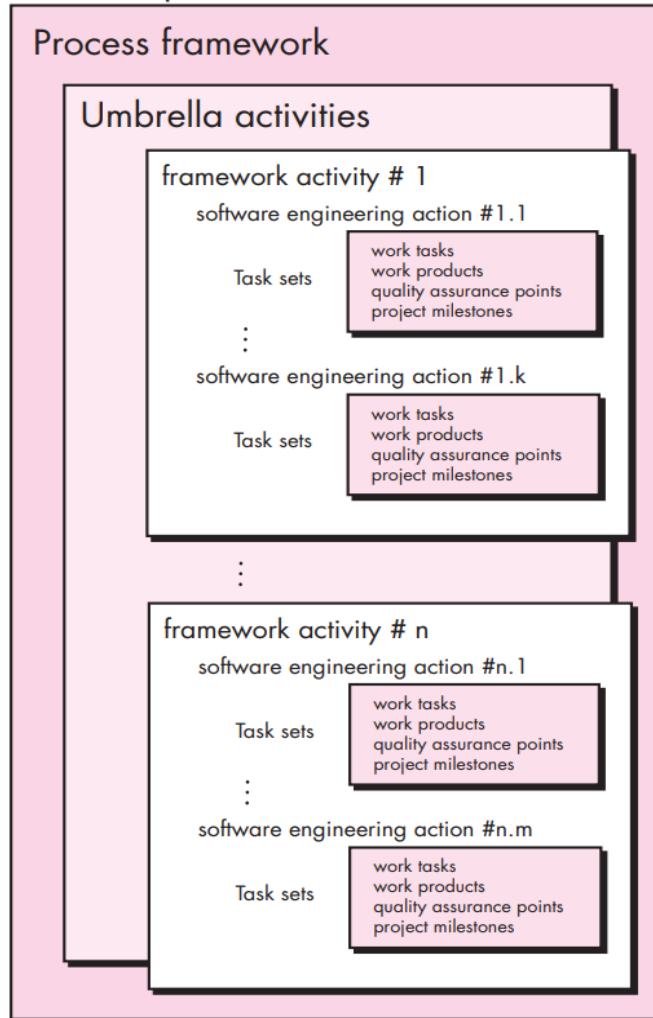
SE MODEL PAPER ANSWERS

1	A) *	Explain how both waterfall model and the prototyping model can be accommodated in the incremental delivery model of the software process.
	B)	Analyze the Generic Process model for the Software Development process with a neat representation.

FIGURE 2.1

A software
process
framework

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 and other stakeholders. 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.

Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction.

This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

Deployment.

The software (as a complete entity or as a partially completed increment) 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. The details of the software process will be quite different in each case, but the framework activities remain the same.

2 A) Explain the best practices of extreme programming method.

Planning.

The planning activity (also called the planning game) begins with listening—a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality. Listening leads to the creation of a set of “stories” (also called user stories) that describe required output, features, and functionality for software to be built. Each story is written by the customer and is placed on an index card. The customer assigns a value (i.e., a priority) to the story based on the overall business value of the feature or function. Members of the XP team then assess each story and assign a cost—measured in development weeks—to it. If the story is estimated to require more than three development weeks, the customer is asked to split the story into smaller stories and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time. Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team. As development work proceeds, the customer can add stories, change the value of an existing story, split stories, or eliminate them. The XP team then reconsiders all remaining releases and modifies its plans accordingly.

Design.

XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality (because the developer assumes it will be required later) is discouraged. XP encourages the use of CRC cards as an effective mechanism for thinking about the software in an object-oriented context. CRC (class-responsibility collaborator) cards identify and organize the object-oriented classes that are relevant to the current software increment. A central notion in XP is that design occurs both before and after coding commences. Refactoring means that design occurs continuously as the system is constructed.

Coding

After stories are developed and preliminary design work is done, the team does not move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment). Once the unit test has been created, the developer is better able to focus on what must be implemented to pass the test. Once the

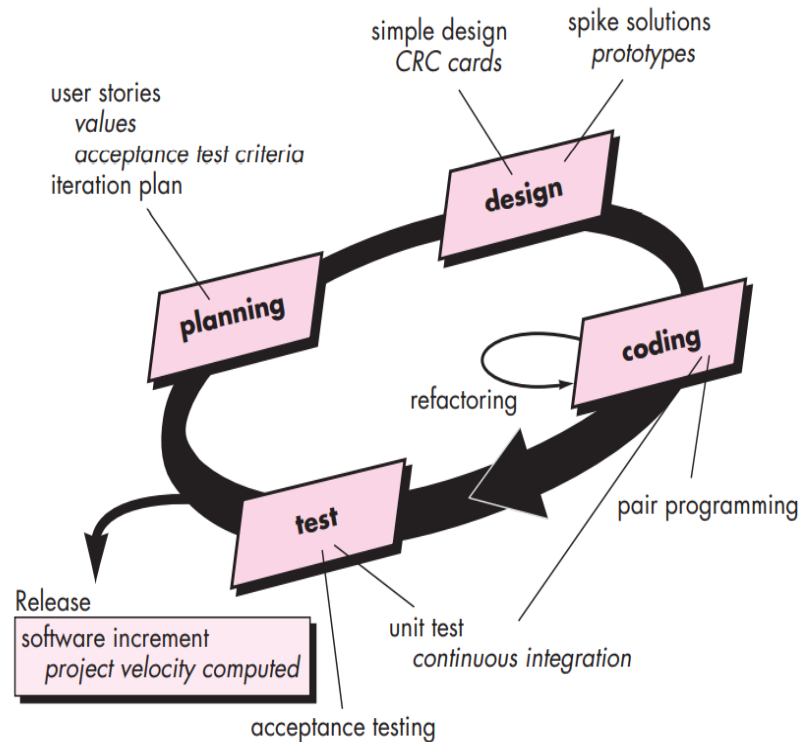
code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

Testing.

I have already noted that the creation of unit tests before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and repeatedly). As the individual unit tests are organized into a “universal testing suite”, integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry.

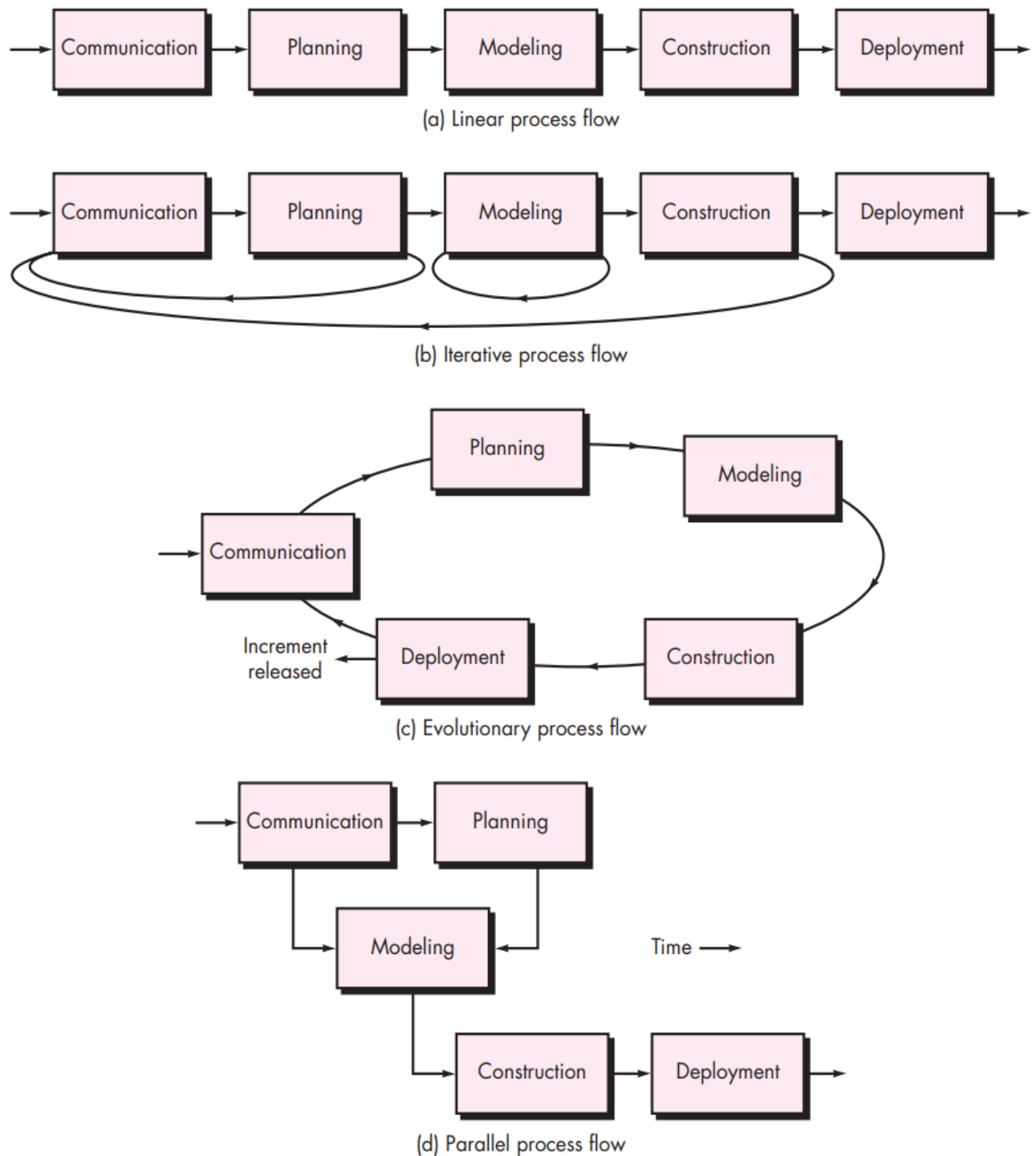
FIGURE 3.2

The Extreme Programming process



B) With neat diagrams, explain process flow of the software process.

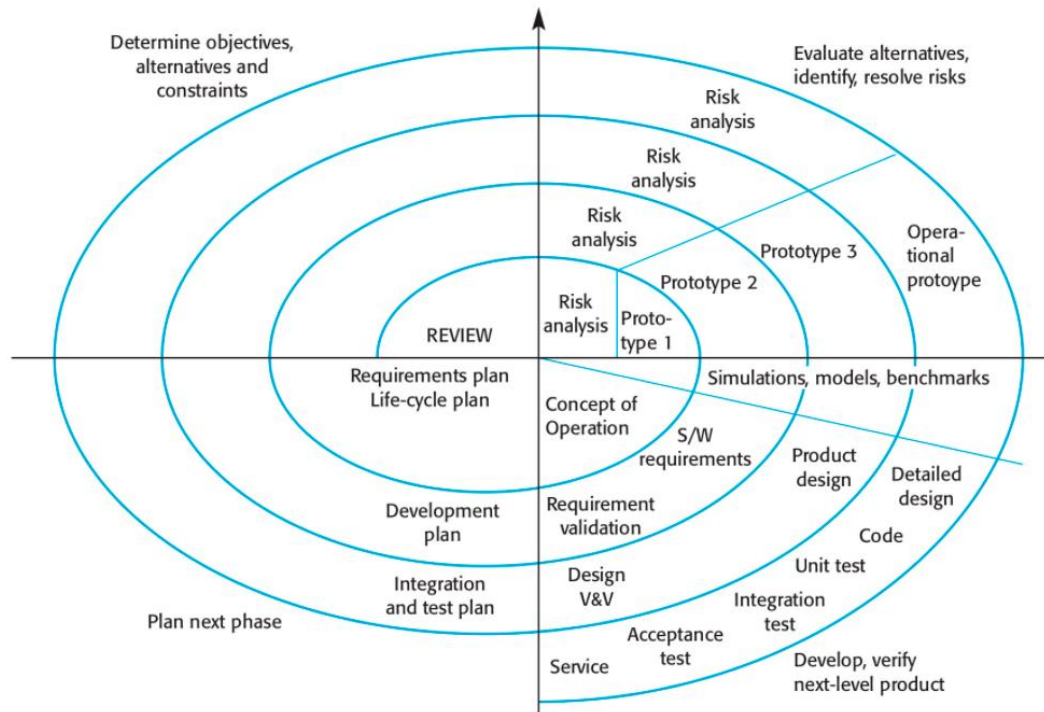
FIGURE 2.2 Process flow



- A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.
- 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.
- A parallel process flow executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).

C) With a neat diagram, explain Boehm’s spiral model of the software process.

Barry Boehm (Boehm, 1988) proposed a risk-driven software process framework (the spiral model) that integrates risk management and incremental development. The software process is represented as a spiral rather than a sequence of activities with some backtracking from one activity to another. Each loop in the spiral represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design and so on. The spiral model combines change avoidance with change tolerance. It assumes that changes are a result of project risks and includes explicit risk management activities to reduce these risks.



Each loop in the spiral is split into four sectors:

1. *Objective setting*: Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified. Alternative strategies, depending on these risks, may be planned.
2. *Risk assessment and reduction*: For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.
3. *Development and validation*: After risk evaluation, a development model for the system is chosen. For example, throw-away prototyping may be the best development approach if user interface risks are dominant. If safety risks are the main consideration, development based on formal transformations may be the most appropriate process, and so on. If the main identified risk is sub-system integration, the waterfall model may be the best development model to use.
4. *Planning*: The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

The main difference between the spiral model and other software process models is its explicit recognition of risk. A cycle of the spiral begins by elaborating objectives such as performance and functionality. Alternative ways of achieving these objectives, and dealing with the constraints on each of them are then enumerated.

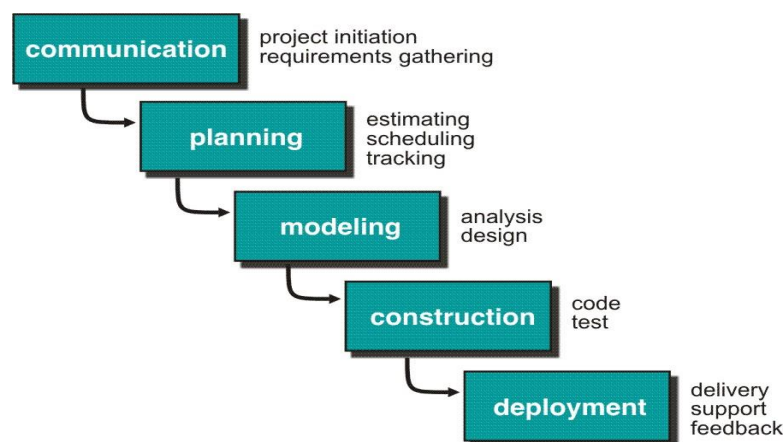
Each alternative is assessed against each objective and sources of project risk are identified. The next step is to resolve these risks by information-gathering activities such as more detailed analysis, prototyping and simulation.

Once risks have been assessed, some development is carried out, followed by a planning activity for the next phase of the process. Informally, risk simply means something that can go wrong. For example, if the intention is to use a new programming language, a risk is that the available compilers are unreliable or do not produce sufficiently efficient object code. Risks lead to proposed software changes and project problems such as schedule and cost overrun, so risk minimization is a very important project management activity.

1. a) Identify the important SDLC models. Explain any ONE in detail.

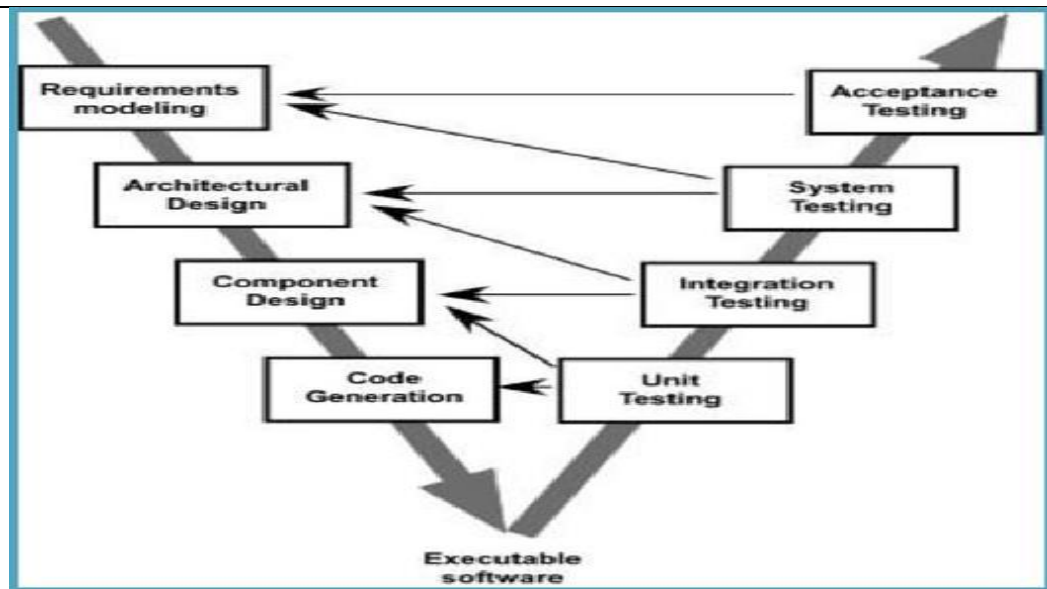
1. The Waterfall Model:

- a. The waterfall is a universally accepted SDLC model. In this method, the whole process of software development is divided into various phases.
- b. The waterfall model is a continuous software development model in which development is seen as flowing steadily downwards (like a waterfall) through the steps of requirements analysis, design, implementation, testing (validation), integration, and maintenance.
- c. Linear ordering of activities has some significant consequences. First, to identify the end of a phase and the beginning of the next, some certification techniques have to be employed at the end of each step. Some verification and validation usually do this mean that will ensure that the output of the stage is consistent with its input (which is the output of the previous step), and that the output of the stage is consistent with the overall requirements of the system.



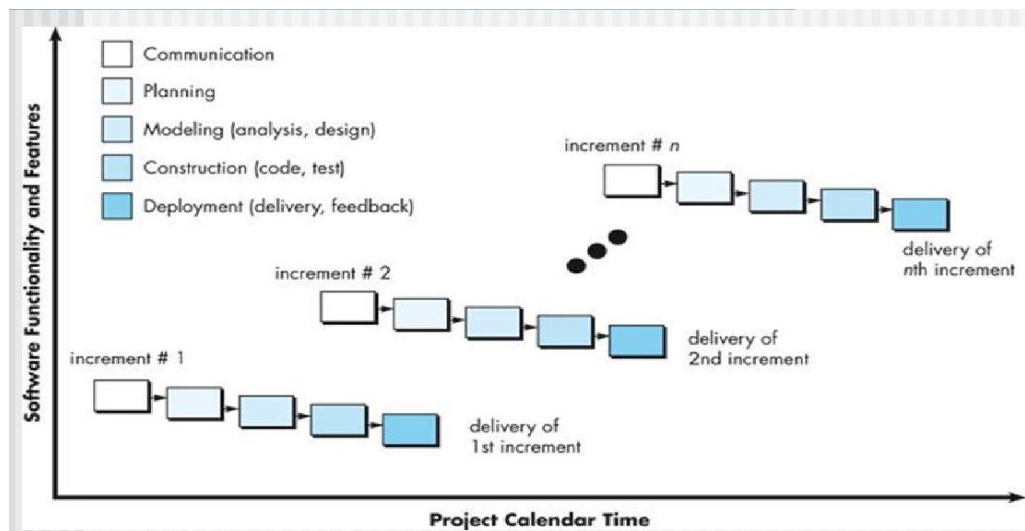
2. The V model:

- a. A variation in the representation of the waterfall model is called the V-model.
- b. As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution.
- c. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moves down the left side.
- d. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.



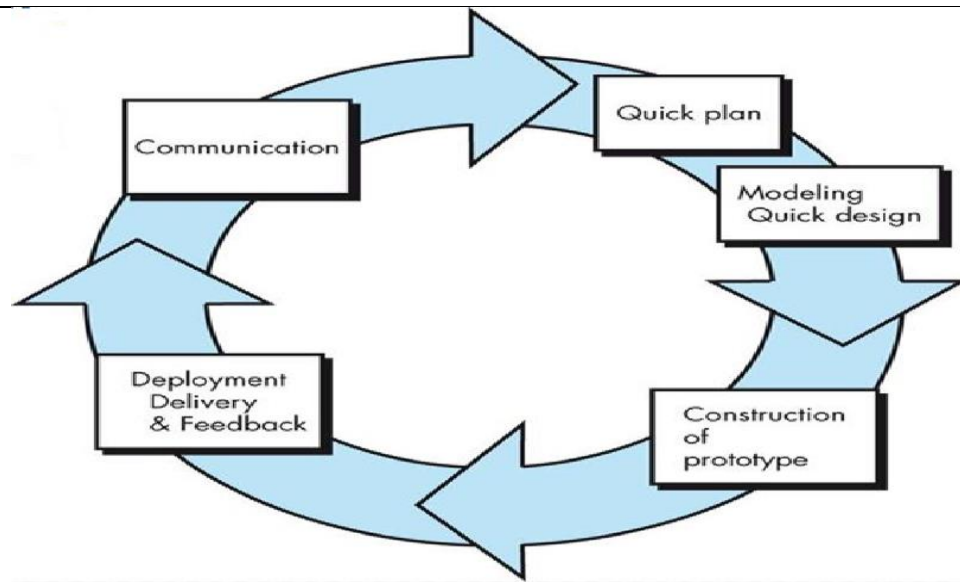
3. Incremental Model:

- a. The incremental model is not a separate model. It is necessarily a series of waterfall cycles.
- b. The requirements are divided into groups at the start of the project. For each group, the SDLC model is followed to develop software.
- c. The SDLC process is repeated, with each release adding more functionality until all requirements are met.
- d. In this method, each cycle act as the maintenance phase for the previous software release. Modification to the incremental model allows development cycles to overlap. After that subsequent cycle may begin before the previous cycle is complete.



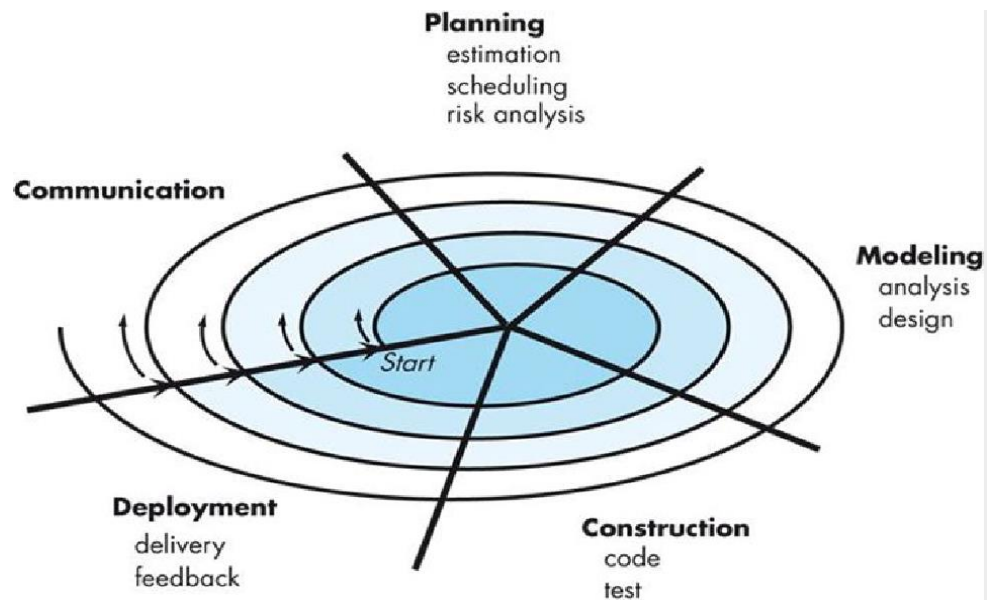
4. Prototyping Model:

- a. Prototyping is used when customers requirements are fuzzy.
- b. OR the developer may not be sure of the efficiency of algorithm, the adaptability of an Operating System or the form that Human Computer interaction should take
- c. But we have to throw away the prototype once the customer requirements are clear & met for better quality. The product must be rebuilt using software engineering practices for long term quality.

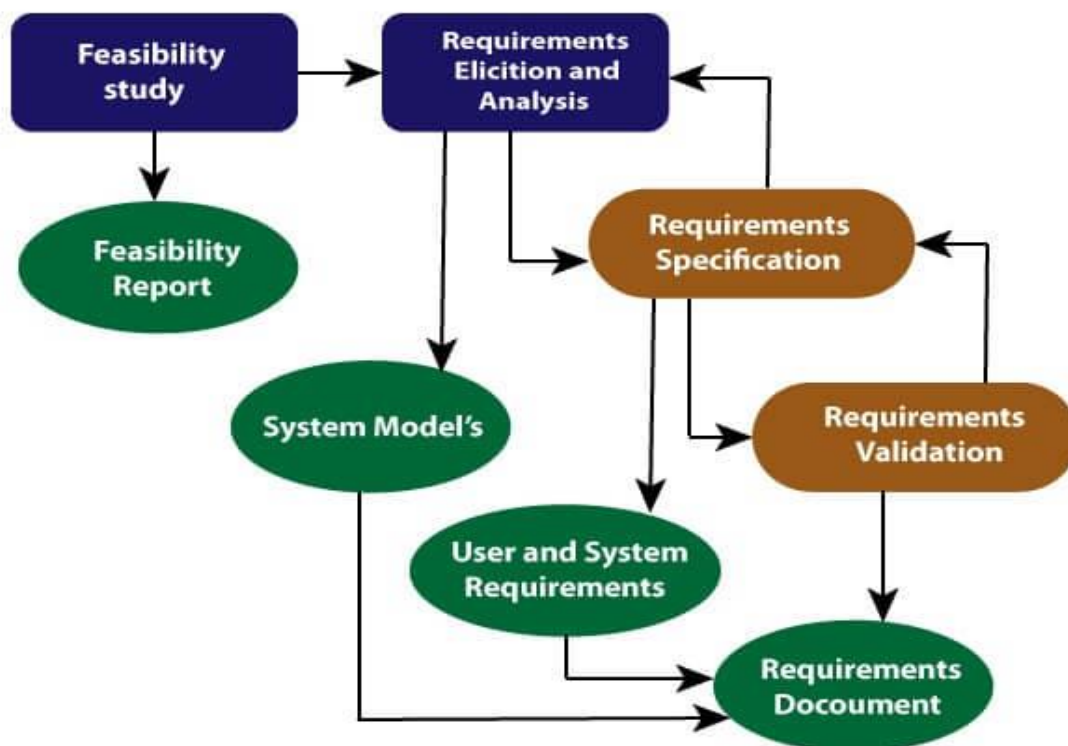


5. The Spiral Model:

- a. The spiral model is a **risk-driven process model**. This SDLC model helps the group to adopt elements of one or more process models like a waterfall, incremental, waterfall, etc. The spiral technique is a combination of rapid prototyping and concurrency in design and development activities.
- b. Each cycle in the spiral begins with the identification of objectives for that cycle, the different alternatives that are possible for achieving the goals, and the constraints that exist. This is the first quadrant of the cycle (upper-left quadrant).
- c. The next step in the cycle is to evaluate these different alternatives based on the objectives and constraints. The focus of evaluation in this step is based on the risk perception for the project.
- d. The next step is to develop strategies that solve uncertainties and risks. This step may involve activities such as benchmarking, simulation, and prototyping.



- b) Illustrate the importance of Requirements Engineering process with a neat block diagram.



Requirement Engineering Process

Requirement Engineering is the process of defining, documenting and maintaining the requirements. It is a process of gathering and defining service provided by the system. Requirements Engineering Process consists of the following main activities:

- Requirements elicitation
- Requirements specification
- Requirements verification and validation
- Requirements management

Requirements Elicitation:

It is related to the various ways used to gain knowledge about the project domain and requirements. The various sources of domain knowledge include customers, business manuals, the existing software of same type, standards and other stakeholders of the project.

The techniques used for requirements elicitation include interviews, brainstorming, task analysis, Delphi technique, prototyping, etc. Some of these are discussed [here](#). Elicitation does not produce formal models of the requirements understood. Instead, it widens the domain knowledge of the analyst and thus helps in providing input to the next stage.

Requirements specification:

This activity is used to produce formal software requirement models. All the requirements including the functional as well as the non-functional requirements and the constraints are specified by these models in totality. During specification, more knowledge about the problem may be required which can again trigger the elicitation process.

The models used at this stage include ER diagrams, data flow diagrams(DFDs), function decomposition diagrams(FDDs), data dictionaries, etc.

Requirements verification and validation:

Verification: It refers to the set of tasks that ensures that the software correctly implements a specific function.

Validation: It refers to a different set of tasks that ensures that the software that has been

built is traceable to customer requirements.

If requirements are not validated, errors in the requirement definitions would propagate to the successive stages resulting in a lot of modification and rework.

The main steps for this process include:

- The requirements should be consistent with all the other requirements i.e no two requirements should conflict with each other.
- The requirements should be complete in every sense.
- The requirements should be practically achievable.

Reviews, buddy checks, making test cases, etc. are some of the methods used for this.

Requirements management:

Requirement management is the process of analyzing, documenting, tracking, prioritizing and agreeing on the requirement and controlling the communication to relevant stakeholders. This stage takes care of the changing nature of requirements. It should be ensured that the SRS is as modifiable as possible so as to incorporate changes in requirements specified by the end users at later stages too. Being able to modify the software as per requirements in a systematic and controlled manner is an extremely important part of the requirements engineering process.

Attention reader! Don't stop learning now. Get hold of all the important CS Theory concepts for SDE interviews with the **CS Theory Course** at a student-friendly price and become industry ready.

The Requirement Engineering (RE) is the most important phase of the Software Development Life Cycle (SDLC). This phase is used to translate the imprecise, incomplete needs and wishes of the potential users of software into complete, precise and formal specifications. The specifications act as the contract between the software users and the developers. Therefore the importance of Requirement Engineering is enormous to develop effective software and in reducing software errors at the early stage of the development of software. Since Requirement Engineering (RE) has great role in different stages of the SDLC, its consideration in software development is crucial.

Develop the SRS for Online Fashion Store using SRS IEEE template.

https://www.academia.edu/29499527/SOFTWARE_REQUIREMENT_SPECIFICATION_FOR_ONLINE_FASHION_STORE

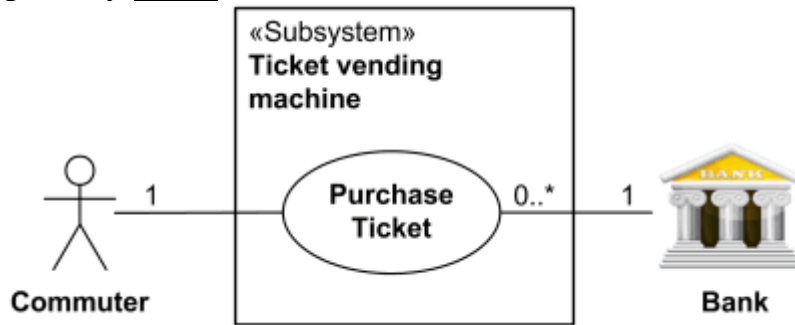
c) Analyze the use of UML diagrams in building the Analysis Model with a real time example.

A UML diagram is a diagram based on the UML (Unified Modeling Language) with the purpose of **visually representing a system** along with its main actors, roles, actions, artifacts or classes, in order to better understand, alter, maintain, or document information about the system.

Mainly, UML has been used as a general-purpose modeling language in the field of software engineering. However, it has now found its way into the documentation of several business processes or workflows. For example, activity diagrams, a type of UML diagram, can be used as a replacement for flowcharts. They provide both a more standardized way of modeling workflows as well as a wider range of features to improve readability and efficacy.

Ticket vending machine, i.e. vending machine that sells and produces tickets to commuters, is a **subject** of the example use case diagram. This kind of a machine is a combination of both hardware and software, and it is only a part of the whole system selling tickets to the customers. So we will use «**Subsystem**» stereotype.

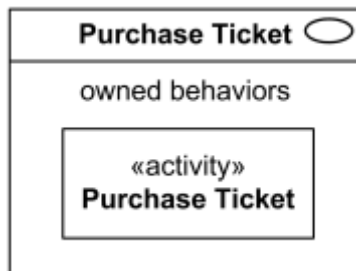
Ticket vending machine allows commuters to buy tickets. So **Commuter** is our primary **actor**.



Ticket vending machine provides Purchase Ticket use case for the Commuter and Bank actors.

The ultimate goal of the Commuter in relation to our ticket vending machine is to buy a ticket. So we have **Purchase Ticket use case**. Purchasing ticket might involve a bank, if payment is to be made using a debit or credit card. So we are also adding another actor - **Bank**. Both actors participating in the use case are connected to the use case by association.

Use case **behaviors** may be described in a natural language text (opaque behavior), which is current common practice, or by using UML **behavior diagrams**. UML tools should allow binding behaviors to the described use cases. Example of such binding of the Purchase Ticket use case to the behavior represented by activity is shown below using UML 2.5 notation.



- 1 a) Propose several software quality guidelines and attributes for a good design.
- b) Briefly explain the taxonomy of architectural styles.
- 2 a) Develop a component level design for Hospital Management System.

- 3 a) Discuss some of the problems that occur when requirements must be elicited from different customers.

The following are the problems that occur when requirements must be elicited from three or four different customers.
 - The requirements of the customer cannot be understood easily. The customer's requirements will change over

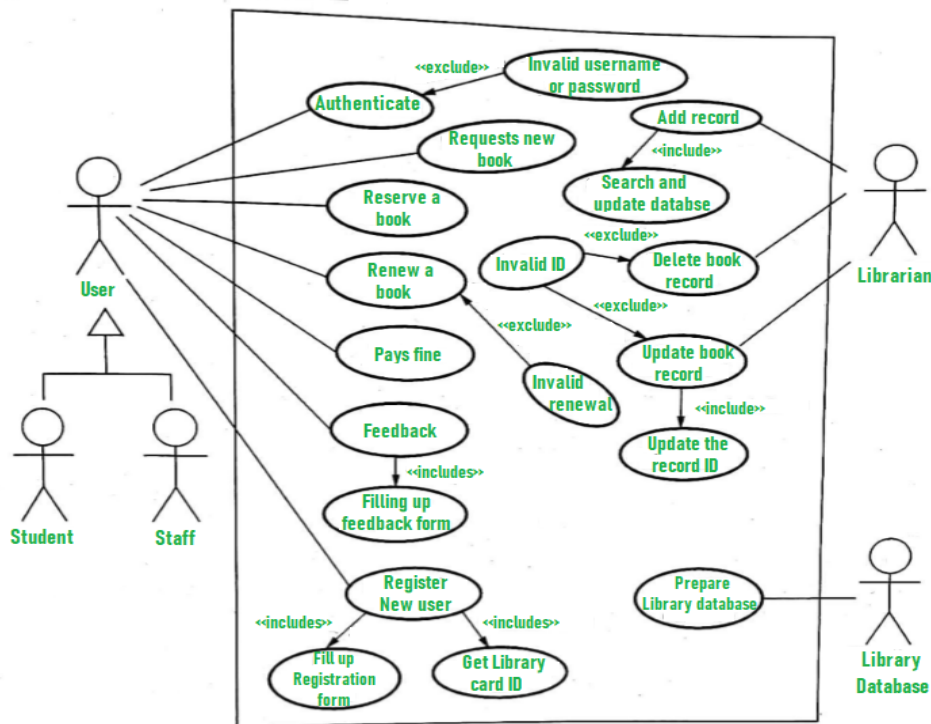
time such that a customer with a set of requirements at one time can include another set of requirements afterward.

- It is very difficult to understand the requirements of the customers.
- The customers will have a wide range of expectations such that it may lead to disappointments at most of the time.
- The customers will change their requirements rapidly.

b) Develop a complete use cases for the following activities:

*

- University Library System
- Buying a Stock using an online brokerage account
- Using credit card at a Restaurant



case diagram for the library management system. Some scenarios of the system are as follows :

- User who registers himself as a new user initially is regarded as staff or student for the library system.
 - For the user to get registered as a new user, registration forms are available that is needed to be fulfilled by the user.
 - After registration, a library card is issued to the user by the librarian. On the library card, an ID is assigned to cardholder or user.
- After getting the library card, a new book is requested by the user as per there requirement.
- After, requesting, the desired book or the requested book is reserved by the user that means no other user can request for that book.
- Now, the user can renew a book that means the user can get a new due date for the desired book if the user has renewed them.
- If the user somehow forgets to return the book before the due date, then the user pays fine. Or if the user forgets to renew the book till the due date, then the book will be overdue and the user pays fine.
- User can fill the feedback form available if they want to.

		<p>(vii) Librarian has a key role in this system. Librarian adds the records in the library database about each student or user every time issuing the book or returning the book, or paying fine.</p> <p>(viii) Librarian also deletes the record of a particular student if the student leaves the college or passed out from the college. If the book no longer exists in the library, then the record of the particular book is also deleted.</p> <p>(ix) Updating database is the important role of Librarian.</p>
	c) *	<p>What is the purpose of domain analysis? How is it related to the concept of requirements patterns? Illustrate with suitable example.</p> <p>Domain Analysis is the process that identifies the relevant objects of an application domain. The goal of Domain Analysis is Software Reuse. The higher is the level of the life-cycle object to reuse, the larger are the benefits coming from its reuse, the harder is the definition of a workable process.</p> <p>Step 1 of 3 592-8-3P SA CODE: 4478 SR CODE: 4475</p> <p>Domain analysis is an on-going software engineering activity that is not connected to any one software project</p> <p>Purpose of domain analysis: The key to reusable software is captured in domain analysis in that it stresses the reusability of analysis and design.</p>
4	a)	<p>Define requirements engineering. List and explain seven distinct tasks of requirements engineering.</p> <p>The broad spectrum of tasks and techniques that lead to an understanding of requirements is called requirements engineering. From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. It must be adapted to the needs of the process, the project, the product, and the people doing the work. It encompasses seven distinct tasks: inception, elicitation, elaboration, negotiation, specification, validation, and management.</p> <p>1. Inception Inception is a task where the requirement engineering asks a set of questions to establish a software process. In this task, it understands the problem and evaluates with the proper solution. It collaborates with the relationship between the customer and the developer. The developer and customer decide the overall scope and the nature of the question.</p> <p>2. Elicitation Elicitation means to find the requirements from anybody. The requirements are difficult because the following problems occur in elicitation.</p> <p>Problem of scope: The customer give the unnecessary technical detail rather than clarity of the overall system objective.</p> <p>Problem of understanding: Poor understanding between the customer and the developer regarding various aspect of the project like capability, limitation of the computing environment.</p>

Problem of volatility: In this problem, the requirements change from time to time and it is difficult while developing the project.

3. Elaboration

In this task, the information taken from user during inception and elaboration and are expanded and refined in elaboration.

Its main task is developing pure model of software using functions, feature and constraints of a software.

4. Negotiation

In negotiation task, a software engineer decides the how will the project be achieved with limited business resources.

To create rough guesses of development and access the impact of the requirement on the project cost and delivery time.

5. Specification

In this task, the requirement engineer constructs a final work product.

The work product is in the form of software requirement specification.

In this task, formalize the requirement of the proposed software such as informative, functional and behavioral.

The requirement are formalize in both graphical and textual formats.

6. Validation

The work product is built as an output of the requirement engineering and that is accessed for the quality through a validation step.

The formal technical reviews from the software engineer, customer and other stakeholders helps for the primary requirements validation mechanism.

7. Requirement management

It is a set of activities that help the project team to identify, control and track the requirements and changes can be made to the requirements at any time of the ongoing project.

These tasks start with the identification and assign a unique identifier to each of the requirement.

After finalizing the requirement traceability table is developed.

The examples of traceability table are the features, sources, dependencies, subsystems and interface of the requirement.

- b)
*
- The department of Public works for a large city has decided to develop a web-based Pothole Tracking and Repair System (PTRS). Assume suitable places, persons and attributes involved in PTRS. Draw a UML use case diagram for PTRS. Also develop an activity diagram for any one aspect of PTRS.

The department of public works for a large city has decided to develop a Web-based pothole tracking and repair system (PHTRS). A description follows: Citizens can log onto a website and report the location and severity of potholes. As potholes are reported they are logged within a "public works department repair system" and are assigned an identifying number, stored by street address, size (on a scale of 1 to 10), location (middle, curb, etc.), district (determined from street address), and repair priority (determined from the size of the pothole). Work order data are associated with each pothole and include pothole location and size, repair crew identifying number, number of people on crew, equipment assigned, hours applied to repair, hole status (work in progress, repaired, temporary repair, not repaired), amount of filler material used, and cost of repair (computed from hours applied, number of people, material and equipment used). Finally, a damage file is created to hold information about reported damage due to the pothole and includes citizen's name, address, phone number, type of damage, and dollar amount of damage. PHTRS is an online system; all queries are to be made interactivity.

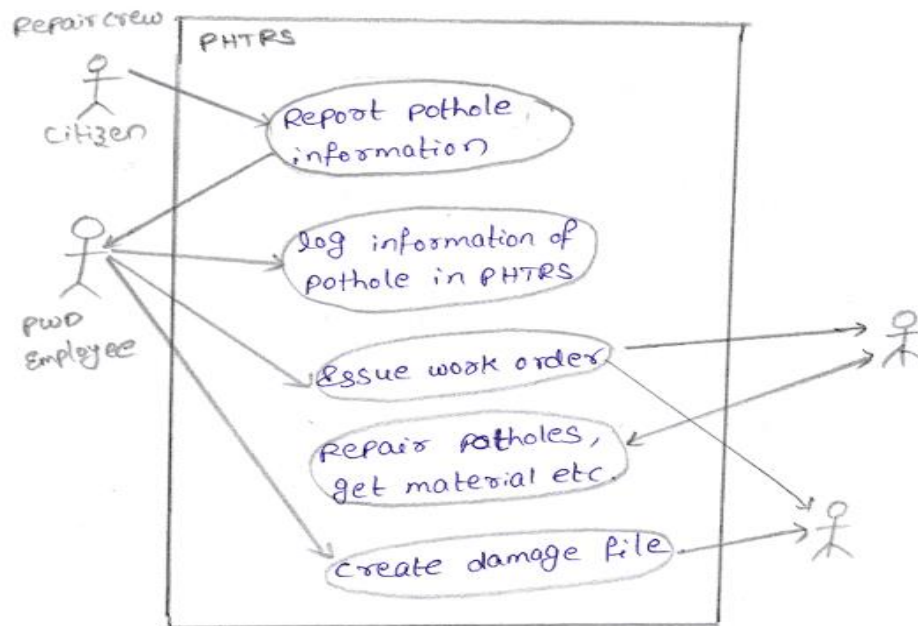
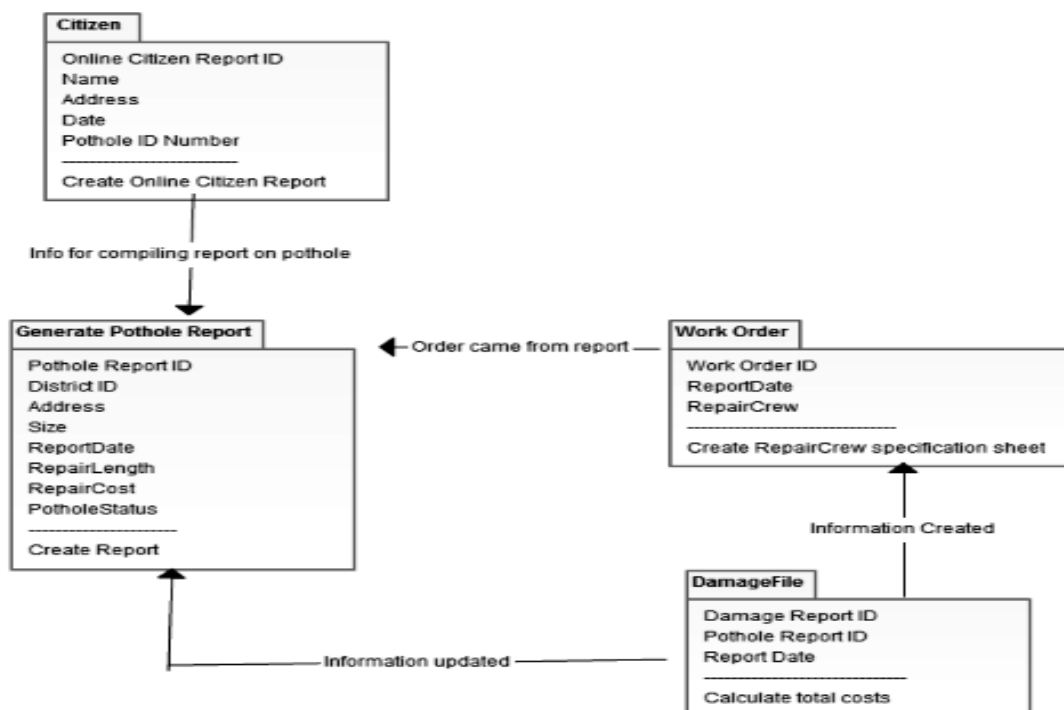


Fig 1:- UML Use case diagram for PHTRS



c) How to negotiate requirements? Describe with suitable examples.

Employee-to-Third-Party Negotiations

Depending on your job, you may be called upon to negotiate constructively with people outside of your company or firm. If you are a salesperson, this may involve negotiating favorable B2B or B2C contracts with clients. If you have purchasing responsibilities, you'll need to source and negotiate with vendors for cost-saving supply contracts. And, of course, if you are a lawyer or paralegal, negotiating with opposing counsel and court personnel is a given.

Even jobs such as teaching require a degree of, if not of negotiation, then its close relative, mediation. Teachers frequently structure learning contracts with their students, and parent communication often requires persuasive mediation skills. Examples of employee-to-third-party negotiations include:

- Negotiating with a customer over the price and terms of a sale
- Negotiating a legal settlement with an opposing attorney
- Negotiating service or supply agreements with vendors
- Mediating with students on lesson plan goals

3 A) Suggest who might be stakeholders in a Hospital management system. Explain why it is almost inevitable that the requirements of different stakeholders will conflict in some way.

Stakeholders as those entities that are integrally involved in the healthcare system and would be substantially affected by reforms to the system. The major stakeholders in the healthcare system are patients, physicians, employers, insurance companies, pharmaceutical firms and government.

As stakeholders are everyone who will be affected in some way, it's fairly hard to list them all, but some examples would be students, teachers, other faculty members relating to the records such as the billing department, government officials in charge of the financial aid given to students who are below a certain number of hours, people of that nature. The requirements that stakeholders have will conflict in various ways, as different people have different priorities, and some of the stakeholders who influence the design of a system more than others may not know exactly what they really want and need. In the student records system, the requirements of students may become thing along the lines of they want to see at any given time what classes are remaining in their curriculum, and the government officials may want to see at any given time the hours of the courses already taken. In an extremely simple records system, these would conflict with each other, and would need each to be built up onto meet the requirements of both parties.

- B) Using your knowledge of how a Bank ATM is used, develop a set of use-cases and a sequence diagram that could serve as a basis for understanding the requirements for a Bank ATM system.

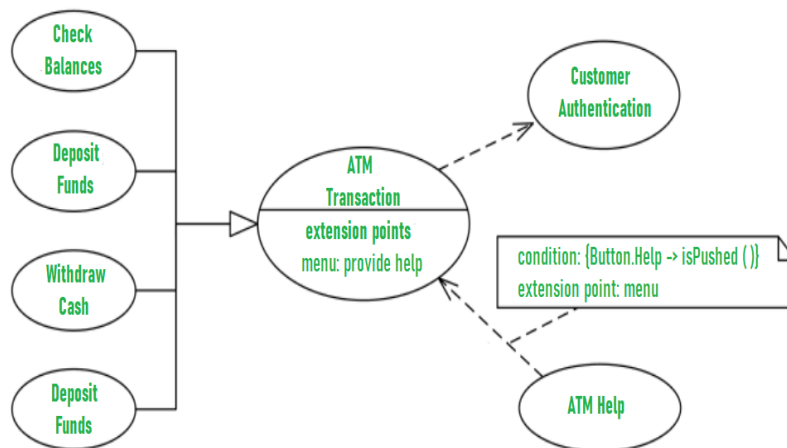
Automated Teller Machine (ATM) also known as ABM (Automated Banking Machine) is a banking system. This banking system allows customers or users to have access to financial transactions. These transactions can be done in public space without any need for a clerk, cashier, or bank teller. Working and description of the ATM can be explained with the help of the **Use Case Diagram**.

We will understand about designing the use case diagram for the ATM system. Some scenarios of the system are as follows.

- **Step-1:**

The user is authenticated when enters the plastic ATM card in a Bank ATM. Then enters the user name and PIN (Personal Identification Number). For every ATM transaction, a Customer Authentication use case is required and essential. So, it is shown as include relationship.

Example of use case diagram for Customer Authentication is shown below:

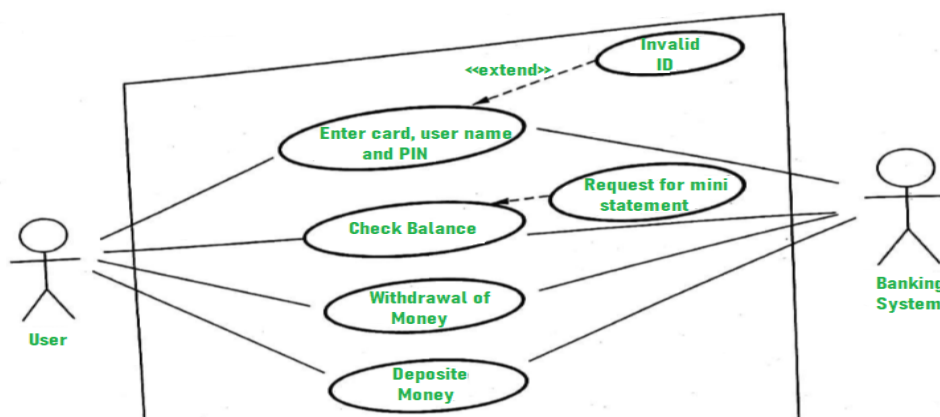


Use Case Diagram for Customer Authentication

- **Step-2:**

User checks the bank balance as well as also demands the mini statement about the bank balance if they want. Then the user withdraws the money as per their need. If they want to deposit some money, they can do it. After complete action, the user closes the session.

Example of the use case diagram for Bank ATM system is shown below:

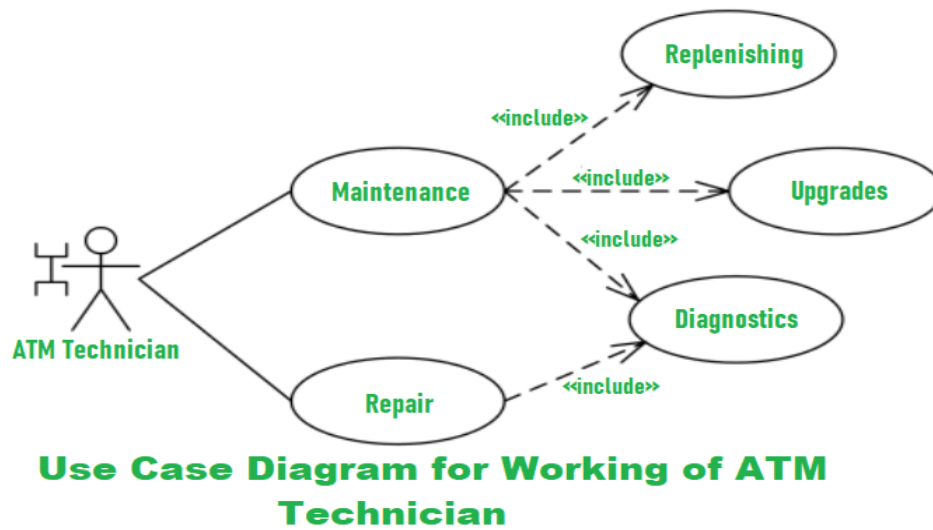


Use Case Diagram for Bank ATM System

• **Step-3:**

If there is any error or repair needed in Bank ATM, it is done by an ATM technician. ATM technician is responsible for the maintenance of the Bank ATM, upgrades for hardware, firmware or software, and on-site diagnosis.

Example of use case diagram for working of ATM technician is shown below:



- 4 A) Briefly explain the characteristic properties of Software Requirements Document and the structure of a requirements document as identified by the IEEE. Prepare IEEE standard SRS for Online Fashion Store.

Following are the characteristics of a good SRS document:

1. Correctness:

User review is used to ensure the correctness of requirements stated in the SRS. SRS is said to be correct if it covers all the requirements that are actually expected from the system.

2. Completeness:

Completeness of SRS indicates every sense of completion including the numbering of all the pages, resolving the to be determined parts to as much extent as possible as well as covering all the functional and non-functional requirements properly.

3. Consistency:

Requirements in SRS are said to be consistent if there are no conflicts between any set of requirements. Examples of conflict include differences in terminologies used at separate places, logical conflicts like time period of report generation, etc.

4. Unambiguousness:

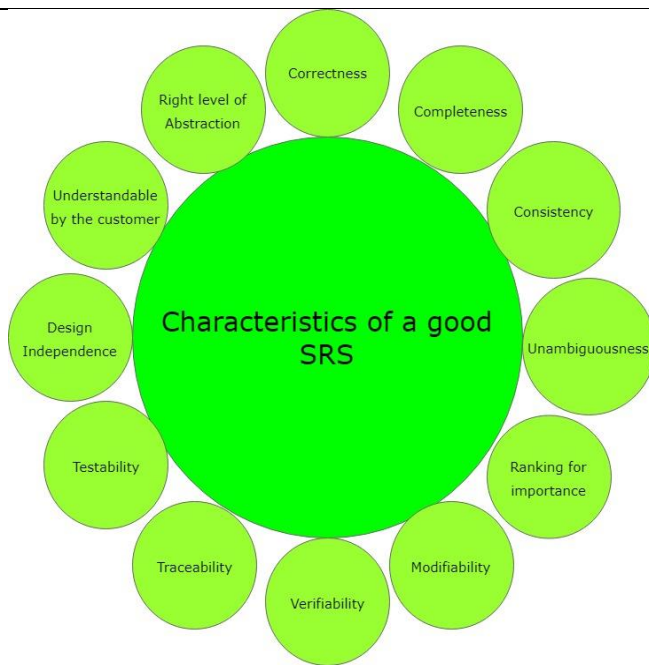
A SRS is said to be unambiguous if all the requirements stated have only 1 interpretation. Some of the ways to prevent unambiguousness include the use of modelling techniques like ER diagrams, proper reviews and buddy checks, etc.

5. Ranking for importance and stability:

There should a criterion to classify the requirements as less or more important or more specifically as desirable or essential. An identifier mark can be used with every requirement to indicate its rank or stability.

6. Modifiability:

SRS should be made as modifiable as possible and should be capable of easily accepting changes to the system to some extent. Modifications should be properly indexed and cross-referenced.



7. Verifiability:

A SRS is verifiable if there exists a specific technique to quantifiably measure the extent to which every requirement is met by the system. For example, a requirement stating that the system must be user-friendly is not verifiable and listing such requirements should be avoided.

8. Traceability:

One should be able to trace a requirement to design component and then to code segment in the program. Similarly, one should be able to trace a requirement to the corresponding test cases.

9. Design Independence:

There should be an option to choose from multiple design alternatives for the final system. More specifically, the SRS should not include any implementation details.

10. Testability:

A SRS should be written in such a way that it is easy to generate test cases and test plans from the document.

11. Understandable by the customer:

An end user maybe an expert in his/her specific domain but might not be an expert in computer science. Hence, the use of formal notations and symbols should be avoided to as much extent as possible. The language should be kept easy and clear.

12. Right level of abstraction:

If the SRS is written for the requirements phase, the details should be explained explicitly. Whereas, for a feasibility study, fewer details can be used. Hence, the level of abstraction varies according to the purpose of the SRS.

The most widely known requirements document standard is IEEE/ANSI 830-1998 (IEEE, 1998). This IEEE standard suggests the following structure for requirements documents:

1. Introduction

1.1 Purpose of the requirements document

1.2 Scope of the product

1.3 Definitions, acronyms and abbreviations

1.4 References

1.5 Overview of the remainder of the document

2. General description

2.1 Product perspective

2.2 Product functions

2.3 User characteristics

2.4 General constraints

2.5 Assumptions and dependencies

3. Specific requirements, covering functional, non-functional and interface requirements. This is obviously the most substantial part of the document but because of the wide variability in organisational practice, it is not appropriate to define a standard structure for this section. The requirements may document external interfaces, describe system functionality and performance, and specify logical database requirements, design constraints, emergent system properties and quality characteristics.

4. Appendices

5. Index

Although the IEEE standard is not ideal, it contains a great deal of good advice on how to write requirements and how to avoid problems. It is too general to be an organisational standard in its own right. It is a general framework that can be tailored and adapted to define a standard geared to the needs of a particular organisation.

https://www.academia.edu/29499527/SOFTWARE_REQUIREMENT_SPECIFICATION_FOR_ONLINE_FASHION_STORE

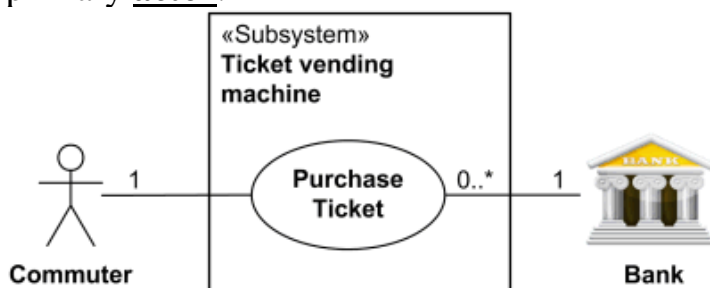
B) Analyze the use of UML diagrams in building the Analysis Model with a real time example.

A UML diagram is a diagram based on the UML (Unified Modeling Language) with the purpose of **visually representing a system** along with its main actors, roles, actions, artifacts or classes, in order to better understand, alter, maintain, or document information about the system.

Mainly, UML has been used as a general-purpose modeling language in the field of software engineering. However, it has now found its way into the documentation of several business processes or workflows. For example, activity diagrams, a type of UML diagram, can be used as a replacement for flowcharts. They provide both a more standardized way of modeling workflows as well as a wider range of features to improve readability and efficacy.

Ticket vending machine, i.e. vending machine that sells and produces tickets to commuters, is a **subject** of the example use case diagram. This kind of a machine is a combination of both hardware and software, and it is only a part of the whole system selling tickets to the customers. So we will use «**Subsystem**» stereotype.

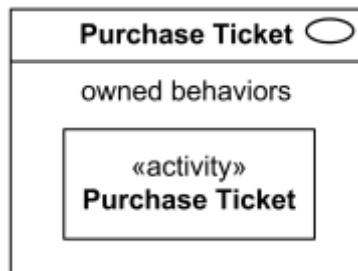
Ticket vending machine allows commuters to buy tickets. So **Commuter** is our primary **actor**.



Ticket vending machine provides Purchase Ticket use case for the Commuter and Bank actors.

The ultimate goal of the Commuter in relation to our ticket vending machine is to buy a ticket. So we have **Purchase Ticket use case**. Purchasing ticket might involve a bank, if payment is to be made using a debit or credit card. So we are also adding another actor - **Bank**. Both actors participating in the use case are connected to the use case by association.

Use case **behaviors** may be described in a natural language text (opaque behavior), which is current common practice, or by using UML **behavior diagrams**. UML tools should allow binding behaviors to the described use cases. Example of such binding of the Purchase Ticket use case to the behavior represented by activity is shown below using UML 2.5 notation.



5 a) Propose several software quality guidelines and attributes for a good design.

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews suggests 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.

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,2 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.

Quality Attributes.

- 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 the predictability 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 ability to organize and control elements of the software configuration, the ease with which a system can be installed, and the ease with which problems can be localized).

b) Briefly explain the taxonomy of architectural styles.

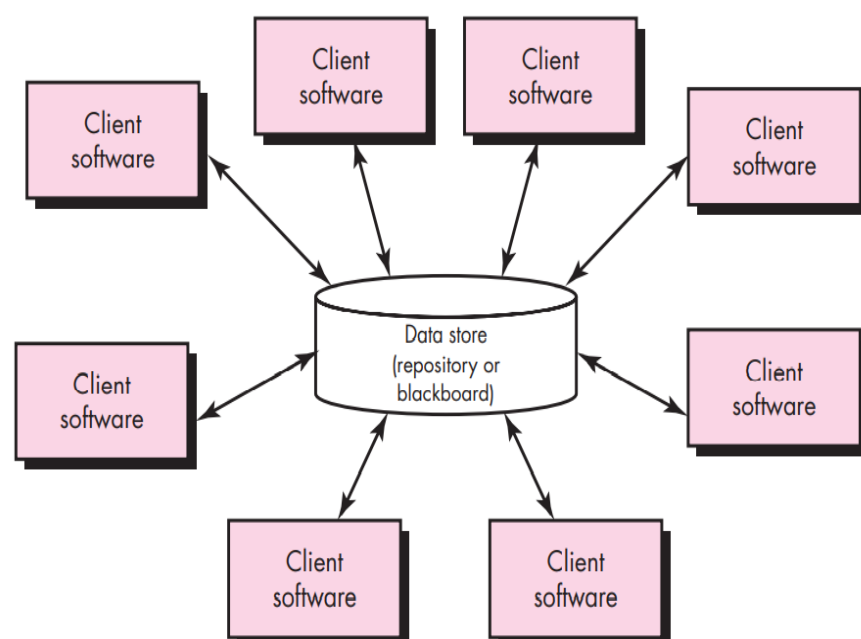
Taxonomy of Architectural styles:

Data centred architectures:

- A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete or modify the data present within the store.
- The figure illustrates a typical data centered style. The client software access a central repository. Variation of this approach are used to transform the repository into a blackboard when data related to client or data of interest for the client change the notifications to client software.
- This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.
- Data can be passed among clients using blackboard mechanism.

FIGURE 9.1

Data-centered
architecture

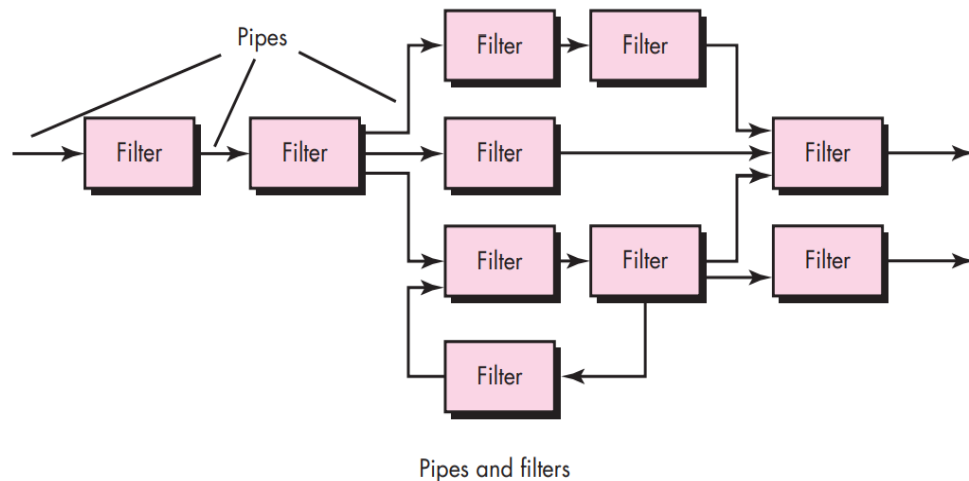


Data flow architectures:

- This kind of architecture is used when input data to be transformed into output data through a series of computational manipulative components.
- The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by pipes.
- Pipes are used to transmit data from one component to the next.
- Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don't require any knowledge of the working of neighboring filters.
- If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.

FIGURE 9.2

Data-flow architecture

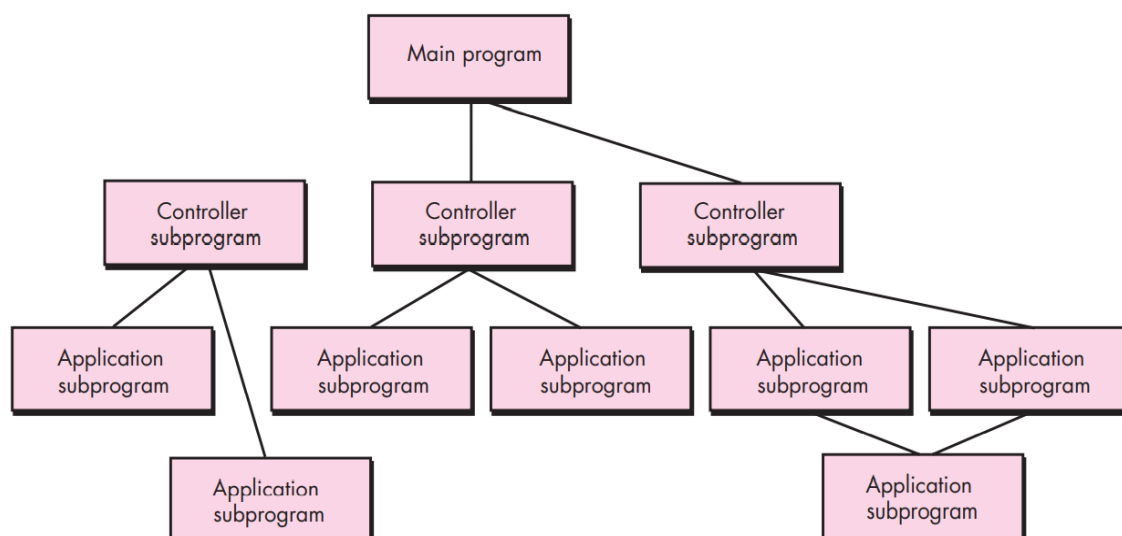


Call and Return architectures: It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.

- **Remote procedure call architecture:** This component is used to present in a main program or subprogram architecture distributed among multiple computers on a network.
- **Main program or Subprogram architectures:** The main program structure decomposes into a number of subprograms or functions into a control hierarchy. The main program contains a number of subprograms that can invoke other components.

FIGURE 9.3

Main program/subprogram architecture

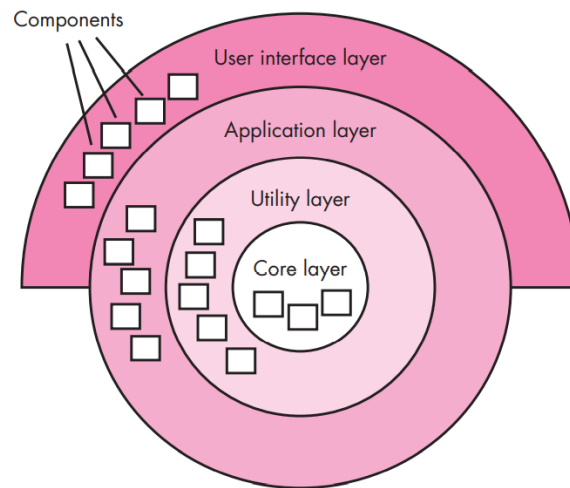


Object Oriented architecture: The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.

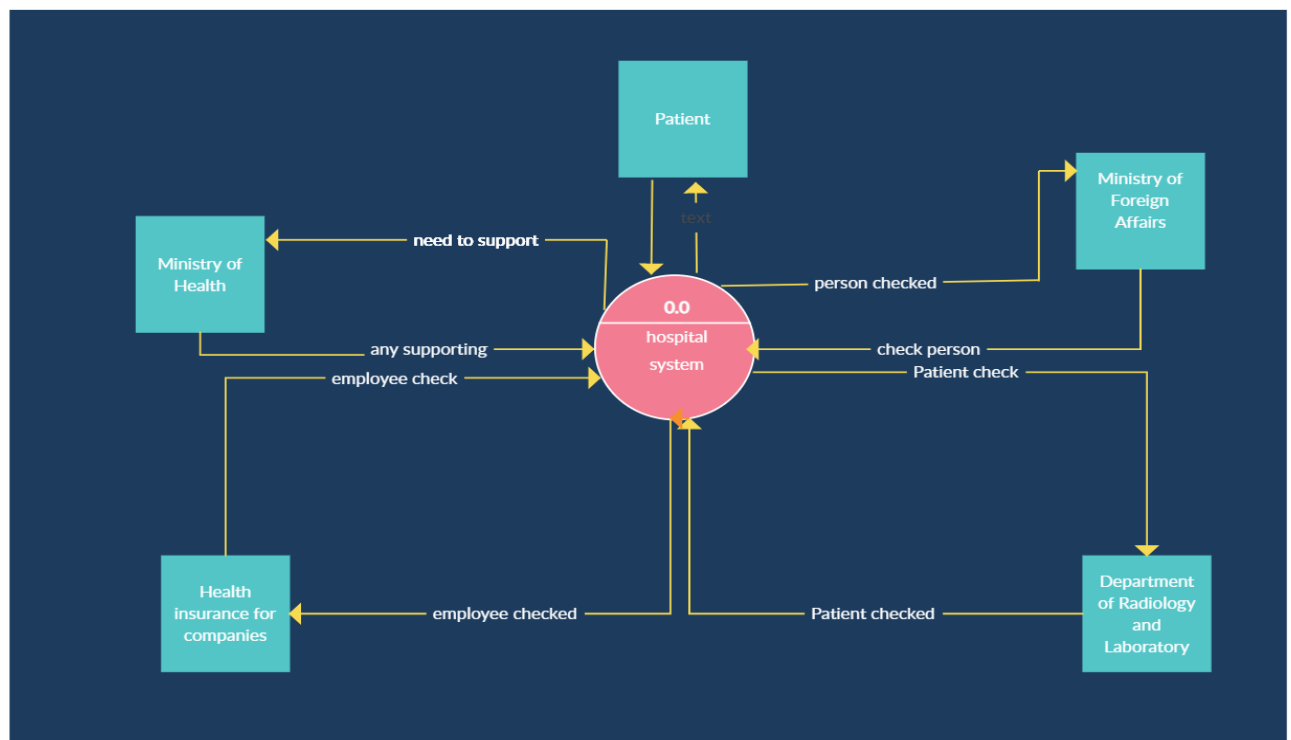
Layered architecture:

- A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively.
- At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing (communication and coordination with OS)
- Intermediate layers to utility services and application software functions.

FIGURE 9.4
Layered
architecture



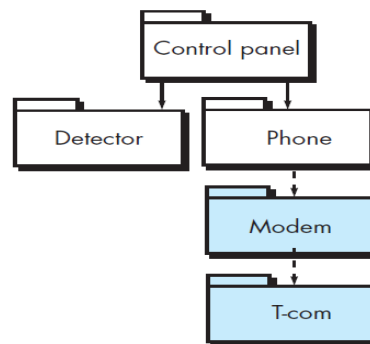
c) Draw the architectural context diagram for Hospital Management System.



6 a) Describe Cohesion and coupling with suitable examples.

Cohesion:

- Within the context of component-level design for object-oriented systems, *cohesion* implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.
- **Functional.** Exhibited primarily by operations, this level of cohesion occurs when a module performs one and only one computation and then returns a result.
- **Layer.** Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers
- **Communicational.** All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.
- Example: the Safe Home security function requirement to make an outgoing phone call if an alarm is sensed.



Coupling:

- *Coupling* is a qualitative measure of the degree to which classes are connected to one another.
- As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep coupling as low as is possible.
- *Class coupling* can manifest itself in a variety of ways. For example, *content coupling* occurs when one component “surreptitiously modifies data that is internal to another component”.
- *Control coupling* occurs when operation *A* () invokes operation *B* () and passes a control flag to *B*. The control flag then “directs” logical flow within *B*. The problem with this form of coupling is that an unrelated change in *B* can result in the necessity to change the meaning of the control flag that *A* passes. If this is overlooked, an error will result.
- *External coupling* occurs when a component communicates or collaborates with infrastructure components.

- b) Explain the following design concepts: design patterns, separation of concerns, refinement and refactoring.
Design Concepts:

- The **software design concept** simply means the idea or principle behind the design.
- It describes how you plan to solve the problem of designing software, the logic, or thinking behind how you will design software.
- It allows the software engineer to create the model of the system or software or product that is to be developed or built.
- The software design concept provides a supporting and essential structure or model for developing the right software.

Design Pattern:

- A design pattern describes a design structure that solves a particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.
- The intent of each design pattern is to provide a description that enables a designer to determine:
 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 or structurally different pattern.

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.
- A *concern* is a feature or behaviour that is specified as part of the requirements model for the software.
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.
- It 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.

Refinement:

- An application is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached.
- Refinement is actually a process of *elaboration*.
- The statement describes function or information conceptually but provides no indication of the internal workings of the function or the internal structure of the information

Refactoring:

- Refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behaviour.
- 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.

c) Briefly describe each of the four elements of the design model.

1. Data design elements

The data design element produced a model of data that represent a high level of abstraction.

This model is then more refined into more implementation specific representation which is processed by the computer based system.

The structure of data is the most important part of the software design.

2. Architectural design elements

The architecture design elements provides us overall view of the system.

The architectural design element is generally represented as a set of interconnected subsystem that are derived from analysis packages in the requirement model.

The architecture model is derived from following sources:

The information about the application domain to built the software.

Requirement model elements like data flow diagram or analysis classes, relationship and collaboration between them.

The architectural style and pattern as per availability.

3. Interface design elements

The interface design elements for software represents the information flow within it and out of the system.

They communicate between the components defined as part of architecture.

Following are the important elements of the interface design:

1. The user interface
2. The external interface to the other systems, networks etc.
3. The internal interface between various components.

4. Component level diagram elements

The component level design for software is similar to the set of detailed specification of each room in a house.

The component level design for the software completely describes the internal details of the each software component.

The processing of data structure occurs in a component and an interface which allows all the component operations.

In a context of object-oriented software engineering, a component shown in a UML diagram.

The UML diagram is used to represent the processing logic.

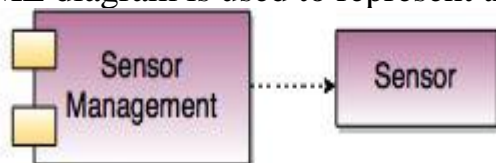


Fig. - UML component diagram for sensor management

5. Deployment level design elements

The deployment level design element shows the software functionality and subsystem that allocated in the physical computing environment which support the software.

Following figure shows three computing environment as shown. These are the personal computer, the CPI server and the Control panel.

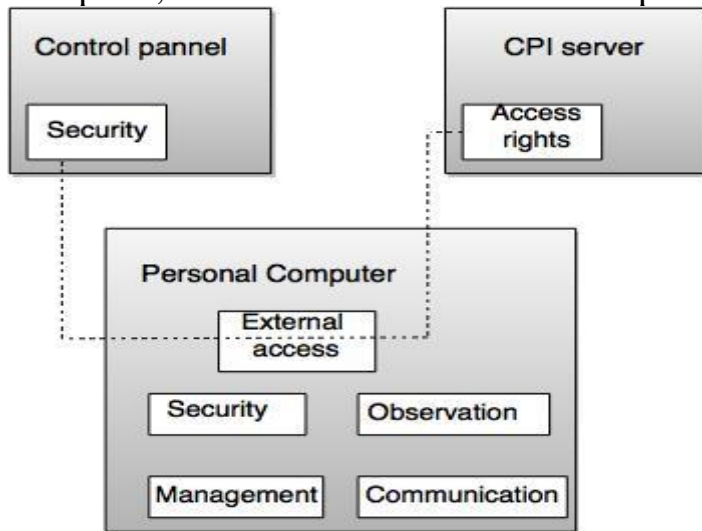


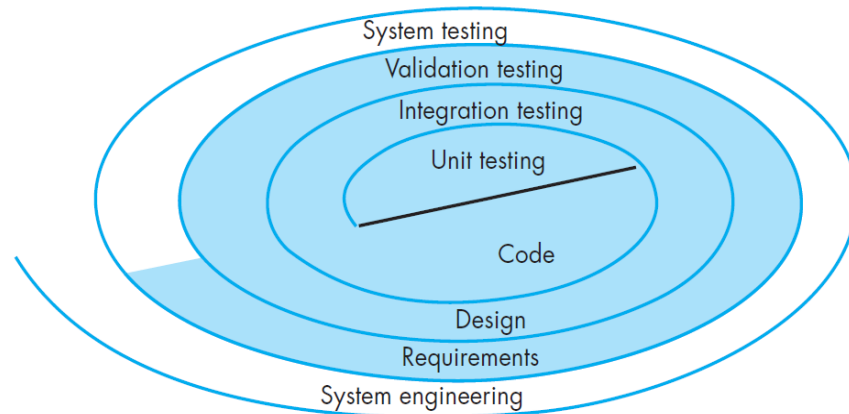
Fig. - Deployment level diagram

5	A) *	Explain several control styles used in software design process.
	B) *	<p>With Component based development, explain the CBSE activities in detail.</p> <p>Component Based Software Engineering (CBSE) is a process that focuses on the design and development of computer-based systems with the use of reusable software components.</p> <p>CBSE Framework Activities</p> <p>Framework activities of Component Based Software Engineering are as follows:-</p> <ol style="list-style-type: none"> 1. Component Qualification: This activity ensures that the system architecture define the requirements of the components for becoming a reusable component. Reusable components are generally identified through the traits in their interfaces. It means “the services that are given, and the means by which customers or consumers access these services ” are defined as a part of the component interface. 2. Component Adaptation: This activity ensures that the architecture defines the design conditions for all component and identifying their modes of connection. In some of the cases, existing reusable components may not be allowed to get used due to the architecture’s design rules and conditions. These components should adapt and meet the requirements of the architecture or refused and replaced by other, more suitable components. 3. Component Composition: This activity ensures that the Architectural style of the system integrates the software components and form a working system. By identifying connection and coordination mechanisms of the system, the architecture describes the composition of the end product.

		<p>4. Component Update: This activity ensures the updation of reusable components. Sometimes, updates are complicated due to inclusion of third party (the organization that developed the reusable component may be outside the immediate control of the software engineering organization accessing the component currently.).</p>
6	A) *	Compare and contrast shared repository and client – server system organization models.
	B)	<p>Explain the different types of Design classes used in System design process</p> <ul style="list-style-type: none"> • User interface classes define all abstractions that are necessary for human computer interaction (HCI). In many cases, HCI occurs within the context of a metaphor (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor. • Business domain classes are often refinements of the analysis classes defined earlier. The classes identify the 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.

7	a)	<p>Explain a strategic approach to software testing.</p> <p><u>Verification and validation:</u></p> <ul style="list-style-type: none"> ➤ <i>Verification</i> refers to the set of tasks that ensure that software correctly implements a specific function. ➤ <i>Validation</i> refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. ➤ Verification and validation include a wide array of SQA activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing <p><u>Organising for software testing:</u></p> <ul style="list-style-type: none"> ➤ The software developer is always responsible for testing the individual units of the program, ensuring that each performs the function or exhibits the behavior for which it was designed. ➤ The role of an <i>independent test group</i> (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present. ➤ The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. <p><u>Software Testing Strategy:</u></p>
---	----	--

- 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.
- Moving inward along the spiral, you come to design and finally to coding.
- 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 of the software as implemented in source code.
- ➔ Testing progresses by moving outward along the spiral to *integration testing*, where the focus is on design and the construction of the software architecture.
- ➔ Taking another turn outward on the spiral, you encounter *validation testing*, where requirements established as part of requirements modeling are validated against the software that has been constructed.
- ➔ Finally, you arrive at *system testing*, where the software and other system elements are tested as a whole.
- ➔ To test computer software, you spiral out along streamlines that broaden the scope of testing with each turn.
- ➔

Criteria for completion of testing:

- The *cleanroom software engineering* approach suggests statistical use techniques that execute a series of tests derived from a statistical sample of all possible program executions by all users from a targeted population.

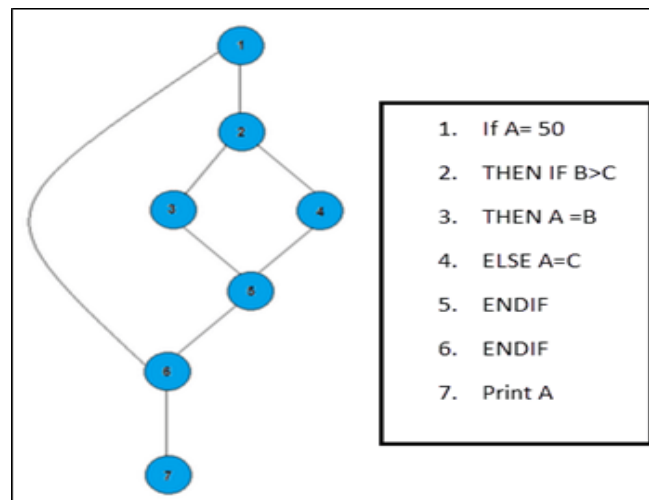
b) With suitable example, explain basis path testing in detail.

“The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.”

Steps for basis path testing:

- Draw a control graph (to determine different program paths)
- Calculate Cyclomatic complexity (metrics to determine the number of independent paths)
- Find a basis set of paths
- Generate test cases to exercise each path

Example:



- Basis path testing involves execution of all possible blocks in a program and achieves maximum path coverage with the least number of test cases.
- In the above example, we can see there are few conditional statements that is executed depending on what condition it suffice. Here there are 3 paths or condition that need to be tested to get the output,

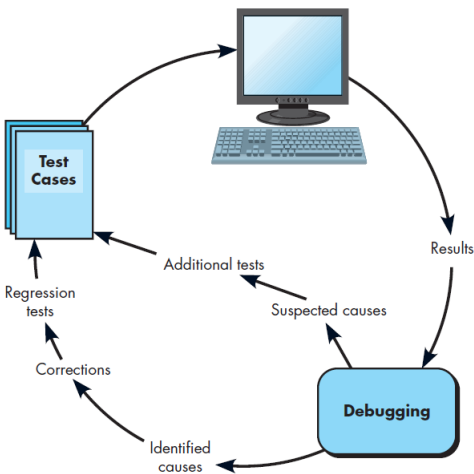
- **Path 1:** 1,2,3,5,6, 7
- **Path 2:** 1,2,4,5,6, 7
- **Path 3:** 1, 6, 7

c) Compare and contrast top-down and bottom-up integration testing strategies.

Top-Down integration testing	Bottom-up integration testing
Integration testing takes place from top to bottom i.e., it begins with the top-level module.	Integration testing takes place from bottom to top i.e., it begins with the lowest level module.
In this testing higher level modules are tested first and then lower-level modules and then the modules are integrated accordingly.	In this testing lower level modules are tested first and then top level modules and then the modules are integrated accordingly.
Here stubs are used to simulate the submodule. If the invoked submodule is not developed means stub works as a momentary replacement.	Here drivers are used to simulate the submodule. If the invoked submodule is not developed means driver works as a momentary replacement.
Beneficial if the significant defect occurs toward the top of the program.	Beneficial if the significant defect occurs toward the bottom of the program.
The main module is designed first and then the submodules are called from it.	The submodules are designed first and then are integrated into the main function.
It is implemented on structural-oriented programming.	It is implemented on object-oriented programming.
The complexity of this testing is simple.	The complexity of this testing is complex and data intensive.

		It works on big to small components. Stub modules must be produced	It works on small to big components. Driver modules must be produced.
--	--	---	--

8 a) With a neat diagram, describe the debugging process.



The debugging process begins with the execution of a test case.

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.

Few characteristics of bugs provide some clues:

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.
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.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real- time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.

The symptom may be due to causes that are distributed across a number of tasks running on different processors.

b) Will exhaustive testing guarantee that the program is 100 percent correct? Illustrate with suitable examples.

No, even exhaustive testing will not guarantee that the program is 100 percent correct. There are too many variables to consider.

Consider this...

Installation testing - did the program install according to the instructions?

Integration testing - did the program work with all of the other programs on the system

without interference, and did the installed modules of the program integrate and work with other installed modules?

Function testing - did each of the program functions work properly?

Unit testing - did the unit work as a standalone as designed, and did the unit work when placed in the overall process?

User Acceptance Testing - did the program fulfill all of the user requirements and work per the user design?

Performance testing - did the program perform to a level that was satisfactory and could it carry the volume load placed upon it?

While these are just the basic tests for an exhaustive testing scenario, you could keep testing beyond these tests using destructive methods, white box internal program testing, establish program exercises using automated scripts, etc.

The bottom line is... testing has to stop at some point in time. Either the time runs out that was allotted for testing, or you gain a confidence level that the program is going to work. (Of course, the more you test, the higher your confidence level).

I don't know anyone that would give a 100% confidence level that the program is 100% correct, (to do so is to invite people to prove you wrong and they will come back with all kinds of bugs you never even considered). However, you may be 95% confident that you found most all of the major bugs. Based upon this level of confidence, you would then place the program into production use - always expecting some unknown bug to be found.

c) Distinguish between white-box and black-box testing.

Black Box Testing	White Box Testing
It is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it.	It is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software.
It is mostly done by software testers.	It is mostly done by software developers.
No knowledge of implementation is needed.	Knowledge of implementation is required.
It can be referred as outer or external software testing.	It is the inner or the internal software testing.
It is functional test of the software.	It is structural test of the software.
This testing can be initiated on the basis of requirement specifications document.	This type of testing of software is started after detail design document.
No knowledge of programming is required.	It is mandatory to have knowledge of programming.
It is the behavior testing of the software.	It is the logic testing of the software.
It is applicable to the higher levels of testing of software.	It is generally applicable to the lower levels of software testing.
It is also called closed testing.	It is also called as clear box testing.
It is least time consuming.	It is most time consuming.
It is not suitable or preferred for algorithm testing.	It is suitable for algorithm testing.
Can be done by trial and error ways and methods.	Data domains along with inner or internal boundaries can be better tested.

		Example: search something on google by using keywords	Example: by input to check and verify loops
--	--	--	--

7

A)

Explain the following:

- i. Defect testing
- ii. Structural testing
- iii. Inspection checks
- iv. Path Testing
- v. Inspection team

i.

ii. **Structural testing** is the type of testing carried out to test the structure of code. It is also known as White Box testing or Glass Box testing. This type of testing requires knowledge of the code, so, it is mostly done by the developers. It is more concerned with how system does it rather than the functionality of the system. It provides more coverage to the testing. For ex, to test certain error message in an application, we need to test the trigger condition for it, but there must be many trigger for it. It is possible to miss out one while testing the requirements drafted in SRS. But using this testing, the trigger is most likely to be covered since structural testing aims to cover all the nodes and paths in the structure of code.

iii. An **inspection checklist** is simply an assurance that specific software product has been inspected. An inspection checklist should be developed by discussion with some experienced staff and as well as regularly updated as more experience is gained from inspection process. Guidebook generally includes checklist simply for various artifacts such as design documents, requirements, etc.

iv. **Path Testing** is a method that is used to design the test cases. In path testing method, the control flow graph of a program is designed to find a set of linearly independent paths of execution. In this method Cyclomatic Complexity is used to determine the number of linearly independent paths and then test cases are generated for each path.

It give complete branch coverage but achieves that without covering all possible paths of the control flow graph. McCabe's Cyclomatic Complexity is used in path testing. It is a structural testing method that uses the source code of a program to find every possible executable path.

v.

- **Author:** The person who created the work product being inspected.
- **Moderator:** This is the leader of the inspection. The moderator plans the inspection and coordinates it.

- **Reader:** The person reading through the documents, one item at a time. The other inspectors then point out defects.
- **Recorder/Scribe:** The person that documents the defects that are found during the inspection.
- **Inspector:** The person that examines the work product to identify possible defects.

B) Identify the different Test Strategies for Conventional Software and explain any one strategy in detail.

Following are the four strategies for conventional software:

- 1) Unit testing
- 2) Integration testing
- 3) Regression testing
- 4) Smoke testing

1) Unit testing

Unit testing focus on the smallest unit of software design, i.e module or software component.

Test strategy conducted on each module interface to access the flow of input and output. The local data structure is accessible to verify integrity during execution.

Boundary conditions are tested.

In which all error handling paths are tested.

An Independent path is tested.

Following figure shows the unit testing:

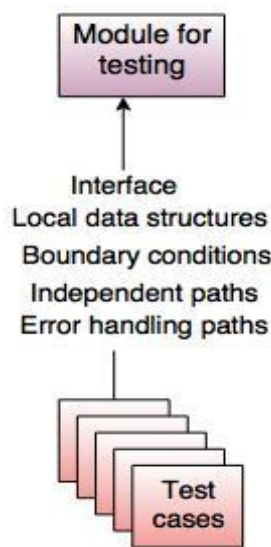


Fig. - Unit test

2) Integration testing

Integration testing is used for the construction of software architecture.

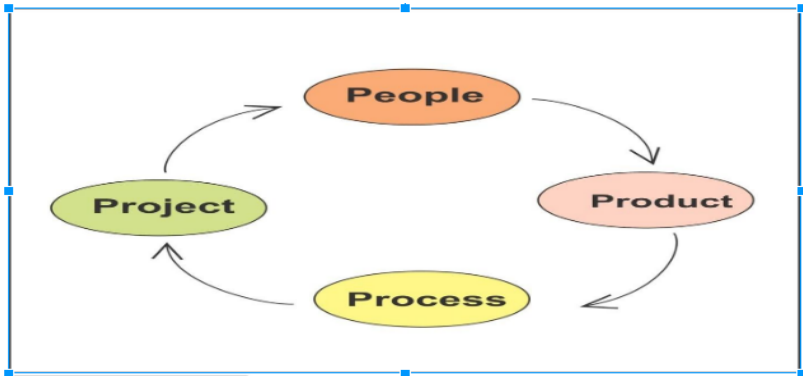
There are two approaches of incremental testing are:

- i) Non incremental integration testing
- ii) Incremental integration testing

i) Non incremental integration testing

Combines all the components in advanced.

		<p>A set of error is occurred then the correction is difficult because isolation cause is complex.</p> <p>ii) Incremental integration testing The programs are built and tested in small increments. The errors are easier to correct and isolate. Interfaces are fully tested and applied for a systematic test approach to it.</p> <p>3) Regression testing</p> <p>In regression testing the software architecture changes every time when a new module is added as part of integration testing.</p> <p>4)Smoke Testing :</p> <ul style="list-style-type: none"> Smoke testing is an integration testing approach that is commonly used when product software is developed Smoke testing is performed by developers before releasing the build to the testing team and after releasing the build to the testing team it is performed by testers whether to accept the build for further testing or not. It is designed as a pacing (Speedy) mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.
8	A) *	Explain McCabe's path testing for a simple flow graph.
	B) *	With Black box testing for Boundary Value Analysis, find test cases for a function of two variables.

9	a)	<p>Briefly explain the management spectrum in software project management.</p> <p>The management spectrum describes the management of a software project or how to make a project successful. It focuses on the four P's; people, product, process and project. Here, the manager of the project has to control all these P's to have a smooth flow in the project progress and to reach the goal.</p>  <pre> graph TD Project([Project]) --> People([People]) People --> Product([Product]) Product --> Process([Process]) Process --> Project </pre> <p>The four P's of management spectrum has been described briefly in below.</p> <p>The People:</p> <p>People of a project includes from manager to developer, from customer to end user. But mainly people of a project highlight the developers. It is so important to have highly skilled and motivated developers that the Software Engineering Institute has developed a</p>
---	----	---

People Management Capability Maturity Model (PM-CMM), “to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability”. Organizations that achieve high levels of maturity in the people management area have a higher likelihood of implementing effective software engineering practices.

The Product:

Product is any software that has to be developed. To develop successfully, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable and accurate estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks or a manageable project schedule that provides a meaningful indication of progress.

The Process:

A software process provides the framework from which a comprehensive plan for software development can be established. A number of different tasks sets— tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

The Project:

Here, the manager has to do some job. The project includes all and everything of the total development process and to avoid project failure the manager has to take some steps, has to be concerned about some common warnings etc.

b) How to establish a software metrics program? Describe with various steps and goals.

To establish a software metrics program, these are the steps:

1. Identify your business goals.
2. Identify what you want to know or learn.
3. Identify your sub goals.
4. Identify the entities and attributes related to your sub goals.
5. Formalize your measurement goals.
6. Identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals.
7. Identify the data elements that you will collect to construct the indicators that help answer your questions.
8. Define the measures to be used, and make these definitions operational.
9. Identify the actions that you will take to implement the measures.
10. Prepare a plan for implementing the measures.

Software supports business functions, differentiates computer-based systems or products, or acts as a product in itself, goals defined for the business can almost always be traced downward to specific goals at the software engineering level. For example, consider the Safe Home product.

Working as a team, software engineering and business managers develop a list of prioritized business goals:

1. Improve our customers’ satisfaction with our products.

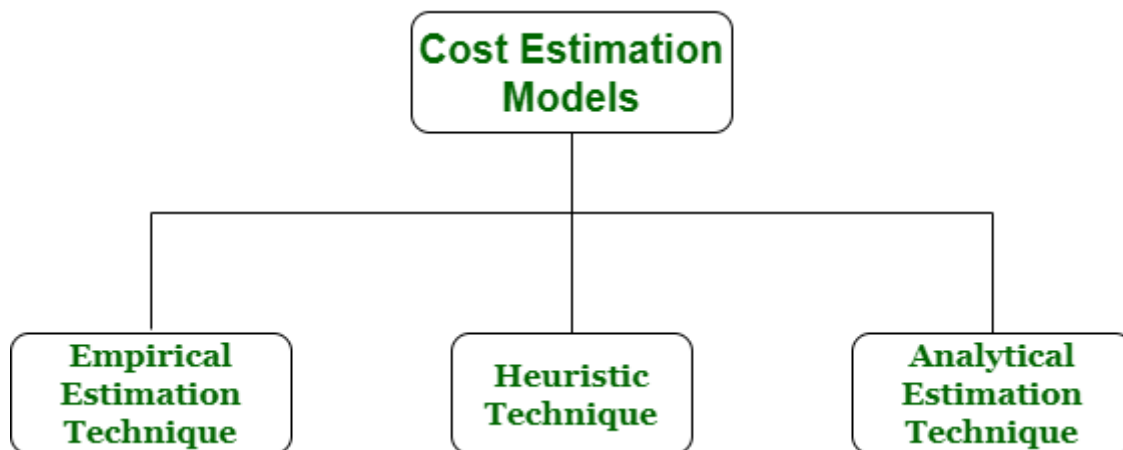
2. Make our products easier to use.
3. Reduce the time it takes us to get a new product to market.
4. Make support for our products easier.
5. Improve our overall profitability.

c)
*

Describe Empirical estimation models used during estimation of software projects.

Cost estimation simply means a technique that is used to find out the cost estimates. The cost estimate is the financial spend that is done on the efforts to develop and test software in Software Engineering. Cost estimation models are some mathematical algorithms or parametric equations that are used to estimate the cost of a product or a project.

Various techniques or models are available for cost estimation, also known as Cost Estimation Models as shown below :



1. **Empirical Estimation Technique –**

Empirical estimation is a technique or model in which empirically derived formulas are used for predicting the data that are a required and essential part of the software project planning step. These techniques are usually based on the data that is collected previously from a project and also based on some guesses, prior experience with the development of similar types of projects, and assumptions. It uses the size of the software to estimate the effort.

In this technique, an educated guess of project parameters is made. Hence, these models are based on common sense. However, as there are many activities involved in empirical estimation techniques, this technique is formalized. For example Delphi technique and Expert Judgement technique.

2. **Heuristic Technique –**

Heuristic word is derived from a Greek word that means “to discover”. The heuristic technique is a technique or model that is used for solving problems, learning, or discovery in the practical methods which are used for achieving immediate goals. These techniques are flexible and simple for taking quick decisions through shortcuts and good enough calculations, most probably when working with complex data. But the decisions that are made using this technique are necessary to be optimal.

In this technique, the relationship among different project parameters is expressed using mathematical equations. The popular heuristic technique is given by Constructive Cost Model (COCOMO). This technique is also used to increase or speed up the analysis and investment decisions.

3. **Analytical Estimation Technique –**

Analytical estimation is a type of technique that is used to measure work. In this technique, firstly the task is divided or broken down into its basic component operations or elements for analyzing. Second, if the standard time is available from some other source, then these sources are applied to each element or component of work.

		<p>Third, if there is no such time available, then the work is estimated based on the experience of the work. In this technique, results are derived by making certain basic assumptions about the project. Hence, the analytical estimation technique has some scientific basis. <u>Halstead's</u> software science is based on an analytical estimation model.</p>
10	a) *	<p>The decisions made by senior management can have a significant impact on the effectiveness of a software engineering team. Provide five examples to illustrate that this is true.</p>
	b)	<p>Briefly explain any four software metrics used for software measurement.</p> <p>Source code metrics</p> <p>These are measurements of the source code that make up all your software. Source code is the fundamental building block of which your software is made, so measuring it is key to making sure your code is high-caliber. (Not to mention there is almost always room for improvement.) Look closely enough at even your best source code, and you might spot a few areas that you can optimize for even better performance.</p> <p>When measuring source code quality make sure you're looking at the number of lines of code you have, which will ensure that you have the appropriate amount of code and it's no more complex than it needs to be. Another thing to track is how compliant each line of code is with the programming languages' standard usage rules. Equally important is to track the percentage of comments within the code, which will tell you how much maintenance the program will require. The less comments, the more problems when you decide to change or upgrade. Other things to include in your measurements is code duplications and unit test coverage, which will tell you how smoothly your product will run (and at when are you likely to encounter issues).</p> <p>Development metrics</p> <p>These metrics measure the <u>custom software development</u> process itself. Gather development metrics to look for ways to make your operations more efficient and reduce incidents of software errors.</p> <p>Measuring number of defects within the code and time to fix them tells you a lot about the development process itself. Start by tallying up the number of defects that appear in the code and note the time it takes to fix them. If any defects have to be fixed multiple time then there might be a misunderstanding of requirements or a skills gap – which is important to address as soon as possible.</p> <p>Testing metrics</p> <p>These metrics help you evaluate how functional your product is. (And we're assuming you want it very functional for your customers.)</p> <p>There are two major testing metrics. One of them is “test coverage” that collects data about which parts of the software program are executed when it runs a test. The second part is a test of the testing itself. It's called “defect removal efficiency,” and it checks your success rate for spotting and removing defects.</p> <p>The more you measure, the more you know about your <u>software product</u>, the more likely you are able to improve it. Automating the measurement process is the best way to</p>

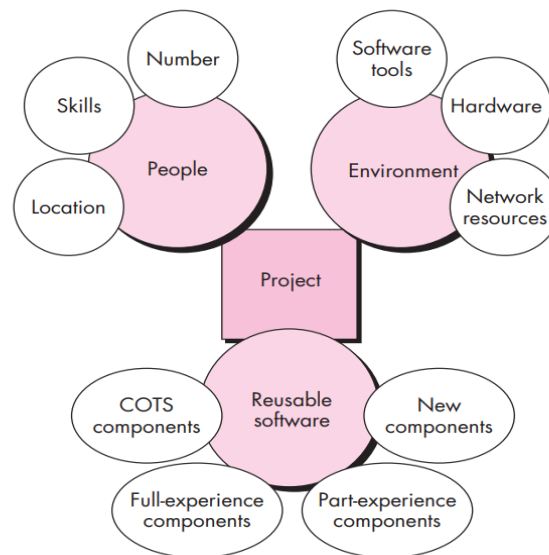
measure software quality – it's not the easiest thing, or the cheapest, but it'll save you tons of cost down the line.

- c) List and explain the three major categories of software engineering resources.

Figure 26.1 depicts the three major categories of software engineering resources—people, reusable software components, and the development environment (hardware and software tools). Each resource is specified with four characteristics: description of the resource, a statement of availability, time when the resource will be required, and duration of time that the resource will be applied.

FIGURE 26.1

Project
resources



Human Resources

The planner begins by evaluating software scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client-server) are specified. For relatively small projects (a few person-months), a single individual may perform all software engineering tasks, consulting with specialists as required. For larger projects, the software team may be geographically dispersed across a number of different locations. Hence, the location of each human resource is specified. The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made. Techniques for estimating effort are discussed later in this chapter.

Reusable Software

Resources Component-based software engineering (CBSE)⁴ emphasizes reusability—that is, the creation and reuse of software building blocks. Such building blocks, often called components, must be cataloged for easy reference, standardized for easy application, and validated for easy integration. Bennatan [Ben00] suggests four software resource categories that should be considered as planning proceeds: Off-the-shelf components. Existing software that can be acquired from a third party or from a past project. COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated. Full-experience components. Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore, modifications required for full-experience components will be relatively low

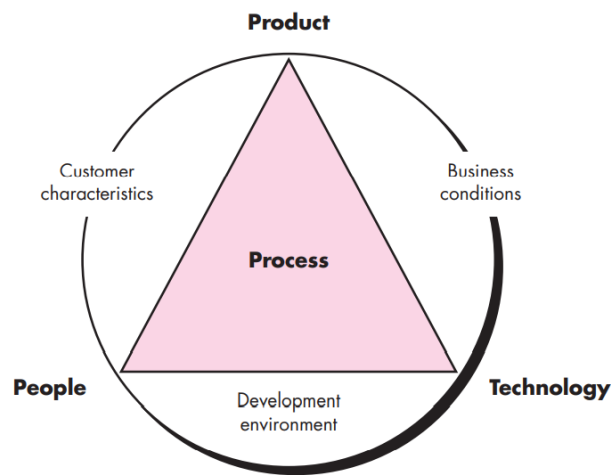
		<p>risk. Partial-experience components. Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications required for partial-experience components have a fair degree of risk. New components. Software components must be built by the software team specifically for the needs of the current project. Ironically, reusable software components are often neglected during planning, only to become a paramount concern later in the software process. It is better to specify software resource requirements early. In this way technical evaluation of the alternatives can be conducted and timely acquisition can occur.</p> <p>Environmental Resources</p> <p>The environment that supports a software project, often called the software engineering environment (SEE), incorporates hardware and software. Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice.⁵ Because most software organizations have multiple constituencies that require access to the SEE, you must prescribe the time window required for hardware and software and verify that these resources will be available. When a computer-based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware elements being developed by other engineering teams. For example, software for a robotic device used within a manufacturing cell may require a specific robot (e.g., a robotic welder) as part of the validation test step; a software project for advanced page layout may need a high-speed digital printing system at some point during development. Each hardware element must be specified as part of planning.</p>
--	--	--

9	A) *	<p>Discuss the importance of determinants for software quality and organizational effectiveness with a neat diagram.</p> <p>The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement. But before I discuss software metrics and their impact on software process improvement, it is important to note that process is only one of a number of “controllable factors in improving software quality and organizational performance”.</p>
---	---------	---

FIGURE 25.1

Determinants
for software
quality and
organizational
effectiveness.

Source: Adapted from
[Pau94].



Referring to Figure 25.1, process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance. The skill and motivation of people has been shown to be the single most influential factor in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e., the software engineering methods and tools) that populates the process also has an impact. In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., integrated software tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication and collaboration). You can only measure the efficacy of a software process indirectly. That is, you derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end users, work products delivered (productivity), human effort expended, calendar time expended, schedule conformance, and other measures. You can also derive process metrics by measuring the characteristics of specific software engineering tasks. For example, you might measure the effort and time spent performing the umbrella activities and the generic software engineering activities

- B) Describe the major challenges faced during finalizing project resources in estimating the software projects.

In a blog series on Project estimation, this is the first part in which we will look at some common challenges associated with Project estimation.

Poor design: While no one wants to admit it, poor design is time and again the root-cause that plays spoilsport even when the best efforts are taken. Often, people conjure up a tight design on the basis of a requirements document, that is misconstrued to be frozen and do not show the forethought to make the design scalable. A poor design results in unnecessary code tweaking and heavy-duty maintenance applying pressure on schedules.

Not splitting the tasks enough: Most projects have a WBS (Work breakdown structure), but sometimes they are not broken enough to be conceptualised with clarity. Each task unit should be equivalent to an Agile story point. Lack of this usually implies that there is no proper basis for estimation and then well, it slips out into a subjective guess.

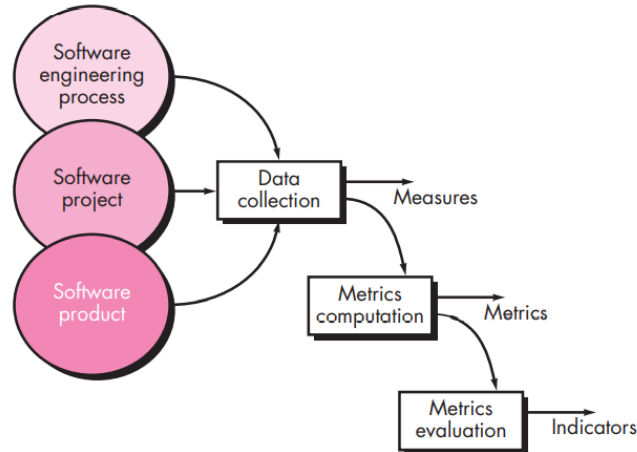
Not factoring the dependencies right: Often, an external dependency or a decision point is missed out causing the project to suffer, this is termed as “coordination neglect”. For example, a vendor product’s license may be about to expire or a new

		<p>version may be up for release which will require for your product to be tested with the newer version. A good understanding of mandatory, discretionary and external dependencies can help you plan the schedule well. Not to forget the dependencies on people resourcing and allowance for vacation and sick leave.</p> <p>How much buffer is the right amount? What is the right buffer to pad once you arrived at an estimate? This is a common challenge for Project Managers and there is no simple formula here. For example, you may have observed that new programmers usually provide aggressive estimates for the fear of being perceived as incompetent. Then they end up working long hours to finish the task. Although 20% padding is usually done, the best figure is arrived considering the people’s skillsets and complexity of the project.</p> <p>Top to bottom scheduling: This is a practical problem one needs to deal with. Instead of doing bottoms-up estimation, most projects start with – “I need this done in 6 months” and then a work breakdown is done where the task estimates are retrofitted inside these 6 months. It is okay to have a high level guideline, but it’s a dangerous trend if the management exerts pressure to submit unrealistic estimates.</p> <p>The risk of analogous estimation: Often, project estimates are done based on an expert judgment or from past projects’ experience. While picking an analogy and mapping the estimate might seem like an intuitive thing to do, it’s often risky because of the numerous variables in a project and the unique elements and dependencies, the people involved and their skillset, diverse tools and technologies adopted and the infrastructure and resources in place.</p> <p>Ignoring team capacity: There is a lot of debate about what unit, estimates need to be provided in – should we measure complexity, time or effort? Irrespective of what unit is followed, many Project Managers tend to ignore considering their team’s capacity. For example, imagine how different people take different time to cook the same dish. It seems obvious that different people would take different time to code, but when we draw estimates, we come up with a standard effort estimate. This inherently adds a risk layer which is going to make the estimate unreliable.</p> <p>In conclusion, it’s important to recognise that project estimation is not a frozen piece of work. As the name says, it’s just ‘estimation’, the actual effort is certain to vary. So, after a few weeks of coding, get a sense of your team capacity and factor that in. Of course, frequent revisions are not a good sign, but if your schedule is rigid since the beginning, you are not reading into the project well.</p>
10	A) *	<p>Illustrate the concept of integrating metrics within the software process using collection process.</p> <p>The majority of software developers still do not measure, and sadly, most have little desire to begin.</p> <p>By establishing a metrics baseline, benefits can be obtained at the process, project, and product (technical) levels. Yet the information that is collected need not be fundamentally different. The same metrics can serve many masters. The metrics baseline consists of data collected from past software development projects and can be as simple as the table presented in Figure 25.2 or as complex as a comprehensive database containing dozens of project measures and the metrics derived from them. To be an effective aid in process improvement and/or cost and effort estimation, baseline data must have the following attributes:</p>

- (1) data must be reasonably accurate—“guestimates” about past projects are to be avoided,
- (2) data should be collected for as many projects as possible,
- (3) measures must be consistent (for example, a line of code must be interpreted consistently across all projects for which data are collected),
- (4) applications should be similar to work that is to be estimated—it makes little sense to use a baseline for batch information systems work to estimate a real-time, embedded application.

FIGURE 25.3

Software
metrics collec-
tion process



B) Effective Software Project Management focuses on four P's, Justify the statement with a simple analogy.

*

Effective software project management focuses on the four P's: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the project.