# Model Paper Solution - BCS501

## 1a. Define Software Engineering and explain practices involved in software Engineering.

**:Software engineering** is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

Software engineering involves a range of practices aimed at ensuring the development of high-quality, reliable, and efficient software systems. Some key practices include:

**1. Requirements Analysis**

Purpose: Understand and define what the software system is supposed to do.

Activities:

- Gather requirements from stakeholders.
- Analyze and document functional and non-functional requirements.
- Ensure clear, precise, and testable requirements.

**2. System Design**

Purpose: Plan the architecture and design of the software.

Activities:

- Define software architecture and choose appropriate technologies.
- Create high-level system models (e.g., UML diagrams).
- Design user interfaces, databases, and communication protocols.
- Ensure scalability, performance, and security considerations.

**3. Coding/Implementation**

- Purpose: Write the actual code that implements the design.

Activities:

- Follow coding standards and best practices.
- Write modular, reusable, and maintainable code.
- Perform peer code reviews and refactor when necessary.

**4. Testing**

Purpose: Ensure the software meets the requirements and is free of defects.

Activities:

- Write unit tests, integration tests, and end-to-end tests.
- Perform manual testing and automated testing.
- Test for functionality, performance, security, and usability.

- Debug and fix issues discovered during testing.

**5. Documentation**

Purpose: Provide clear, understandable, and maintainable documentation.

Activities:

- Document code with inline comments.
- Create technical documentation for the system's architecture, design, and usage.
- Maintain user manuals, API documentation, and system requirements.

**6. Project Management**

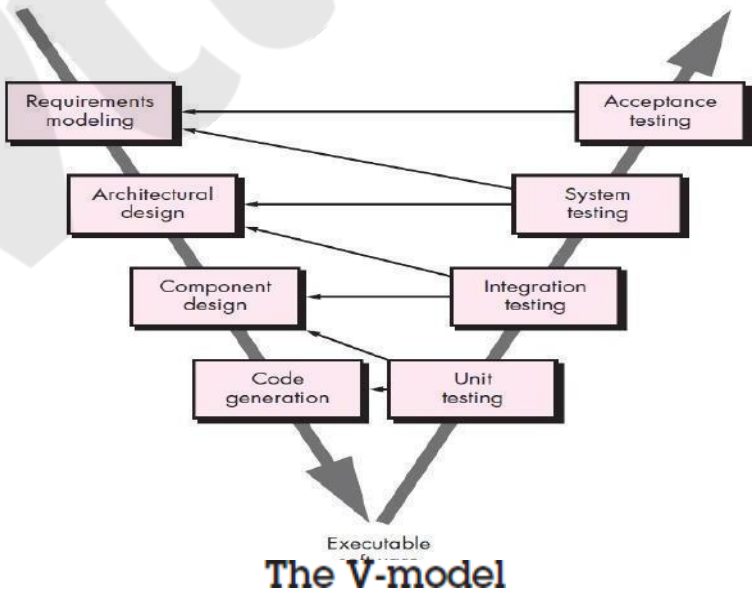
Purpose: Manage resources, timelines, and risks.

Activities:

- Plan sprints or milestones using methodologies like Agile, Scrum, or Waterfall.
- Track progress, allocate tasks, and address bottlenecks.
- Communicate with stakeholders and ensure project goals are aligned with business objectives.

## 1b. Demonstrate V- Model with a neat, labelled diagram

A variation in the representation of the waterfall model is called the ***V-model.*** Represented in following figure. The V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities.




The V-model

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. Once code has been generated, the team moves up the right side of the **V**, essentially performing a series of tests that validate each of the models created as the team moved down the left side. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

## 1c. Explain new Software Challenges and Software Myths

**New Software Challenges:**

- **Open-world Computing**: Creating software to allow machines of all sizes to communicate with each other across vast networks (Distributed computing—wireless networks)
- **Net Sourcing**: Architecting simple and sophisticated applications that benefit targeted end-user markets worldwide (the Web as a computing engine)
- **Open Source**: Distributing source code for computing applications so customers can make local modifications easily and reliably ( "free" source code open to the computing community)

### Software Myths

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

**1. "We can skip the planning phase and start coding immediately."**

**Reality**: Proper planning ensures the software meets user needs, aligns with business goals, and avoids costly mistakes. Without planning, projects can quickly veer off track.

**2. "Adding more developers to a late project will speed up development."**

**Reality**: Adding more people often increases coordination overhead and can slow progress, especially if the team needs to get up to speed on the project.

**3. "Once the software is written, it's done."**

**Reality:** Software requires ongoing maintenance, bug fixes, updates, and performance enhancements. It is never truly "finished."

**4. "The customer knows exactly what they want."**

**Reality**: Customers often can't fully articulate their needs upfront. Iterative feedback, user testing, and collaboration are necessary to refine and clarify requirements over time.

**5. "Testing is an optional activity."**

**Reality**: Comprehensive testing (unit, integration, system) is critical to ensure software reliability, security, and functionality. Skipping testing can lead to significant defects in production.

## 2a. With a neat diagram explain Incremental Process Models.

**Incremental Process Models**

The incremental model delivers a series of releases, called increments, that provide progressively more functionality for the customer as each increment is delivered.

The *incremental* model combines elements of linear and parallel process flows discussed in Section 1.7. The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow.

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.

When an incremental model is used, the first increment is often a *core product.* That is, basic requirements are addressed but many supplementary features remain undelivered. The core product is used by the customer. As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

Incremental development is particularly useful when **staffing is unavailable** for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks.

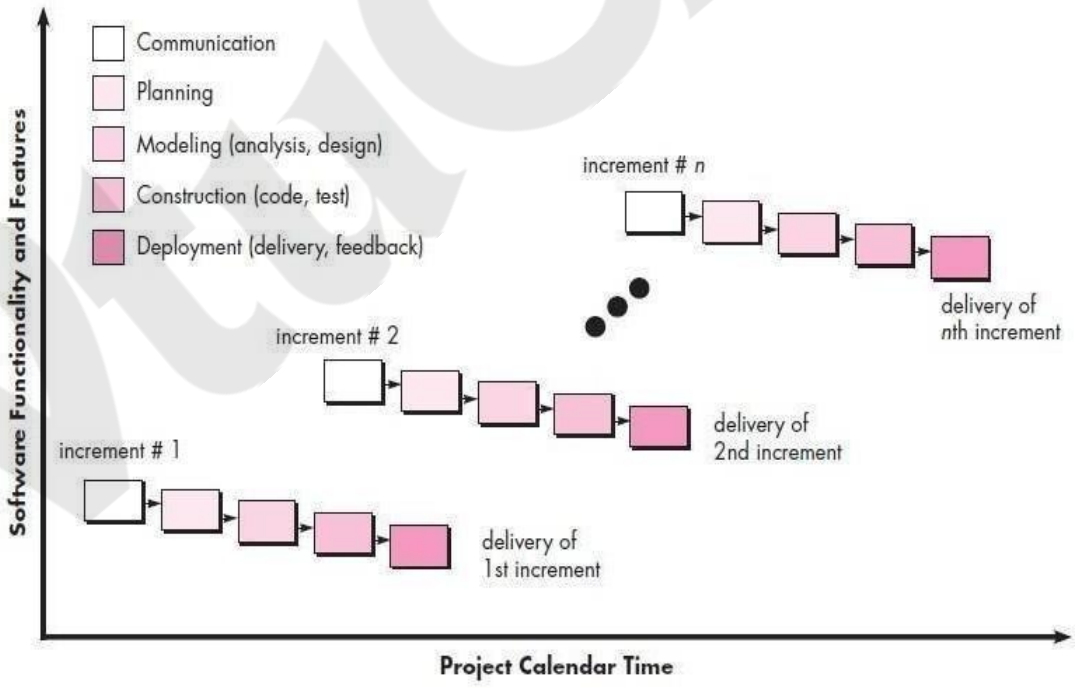



**Fig : Incremental Model**

## 2b. Briefly explain the Spiral Model with neat diagrams.

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

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

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced

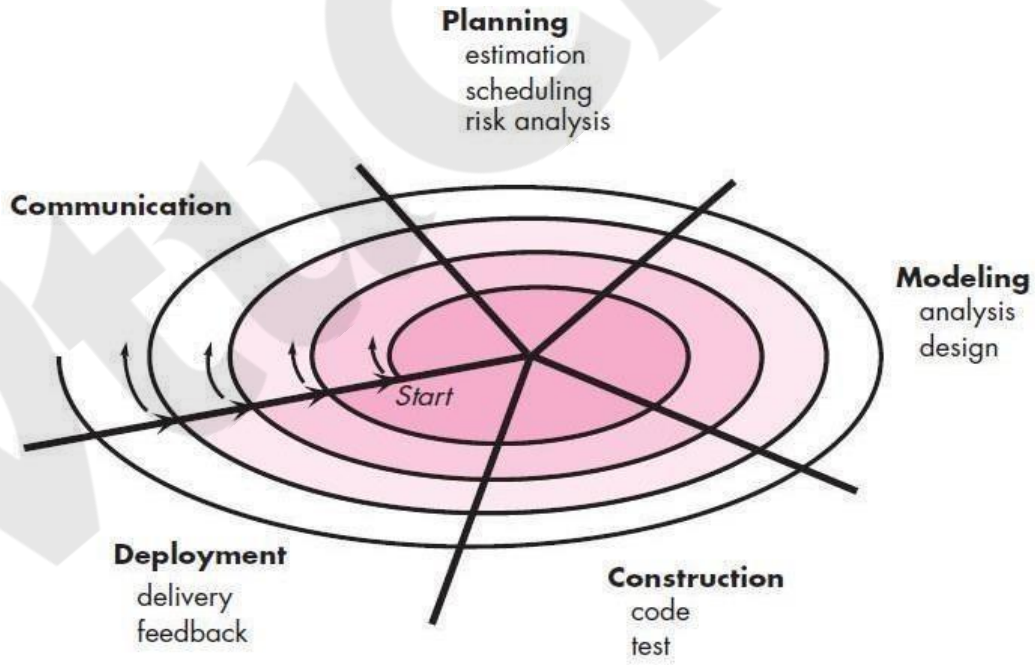



**Fig: The Spiral Model**

## 2c. Explain unique nature of Web Apps.

- **Network intensiveness.** A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).
- **Concurrency.** A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.
- **Unpredictable load.** The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.
- **Performance.** If a WebApp user must wait too long, he or she may decide to go elsewhere.
- **Availability.** Although expectation of 100 percent availability is un reasonable, users of popular WebApps often demand access on a 24/7/365 basis
- **Data driven.** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).
- **Content sensitive.** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.
- **Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously.
- **Immediacy.** Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time- to-market that can be a matter of a few days or weeks.

## 3a. Explain Negotiating requirements & Validating requirements

### NEGOTIATING REQUIREMENTS

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

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

1. Identification of the system or subsystem's key stakeholders.
2. Determination of the stakeholders' "win conditions."
3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned.
4. Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to subsequent software engineering activities.

### VALIDATING REQUIREMENTS

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

A review of the requirements model addresses the following questions:

- Is each requirement consistent with the overall objectives for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally,

a specific individual) noted for each requirement?

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

## 3b. Explain key steps are involved in establishing the foundational groundwork for a successful software engineering project.

ESTABLISHING THE GROUNDWORK

**Identifying Stakeholders:**

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

**Recognizing Multiple Viewpoints:**

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

**Working toward Collaboration:**

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

**Asking the First Questions**

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

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

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a

successful implementation and possible alternatives to custom software development.

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

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached? The final set of questions focuses on the effectiveness of the communication activity itself. Gause and Weinberg call these "**meta-questions**" and propose the following list:
- Are you the right person to answer these questions? Are your answers
- "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

These questions will help to "**break the ice**" and initiate the communication that is essential to successful elicitation. But a question-and-answer meeting format is not an approach that has been overwhelmingly successful.

## 3 c. Provide a brief overview of the key tasks involved in requirement engineering

: It encompasses **seven** distinct tasks: **inception, elicitation, elaboration, negotiation, specification, validation, and management**.

**Inception:** It establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

**Elicitation:** In this stage, proper information is extracted to prepare to document the

requirements. It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day- to-day basis.

- **Problems of scope.** The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
- **Problems of understanding.** The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or un testable.
- **Problems of volatility.** The requirements change over time.

**Elaboration:** The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information. Elaboration is driven by the creation and refinement of user scenarios that describe **how** the end user (and other actors) will interact with the system.

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

**Specification:** The term *specification* means **different things to different people**. A specification can be a written document, a set of graphical models, a formal

mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

**Validation:** The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies,

**Requirements management:** Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds. Many of these activities are identical to the software configuration management (SCM) techniques

## 4 a. Demonstrate access camera surveillance via Internet – display camera views function with a neat activity diagram

### Developing an Activity Diagram

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart,

An activity diagram uses:

- **Rounded rectangles** to imply a specific system function
- **Arrows** to represent flow through the system
- **Decision diamonds** to depict a branching decision.
- **Solid horizontal lines** to indicate that parallel activities are occurring.

A UML activity diagram represents the actions and decisions that occur as some function is performed.

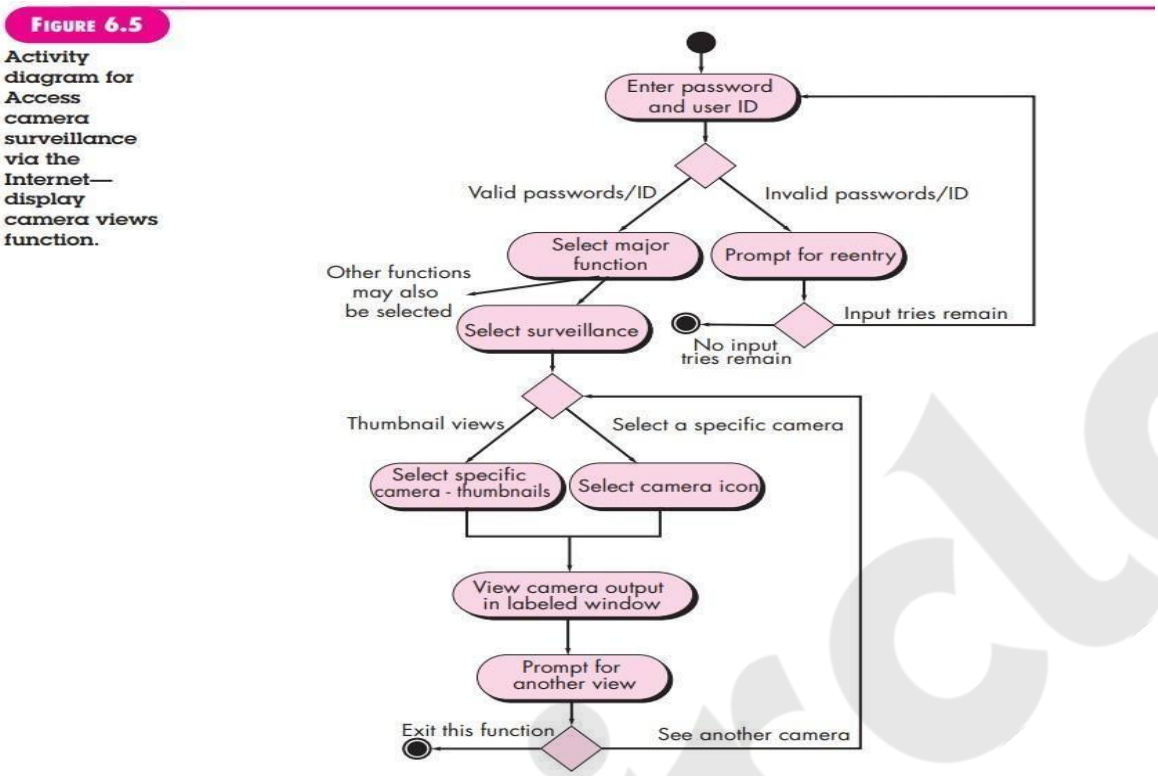



**FIGURE 6.5**
Activity diagram for Access camera surveillance via the Internet—display camera views function.

## 4b. Explain key steps involved in building a requirement model, and how do they contribute to the software development process?

Here are the key steps involved in building a requirements model and their contributions to the software development process:

1. **Requirements Elicitation**

Contribution: Ensures that all necessary information is gathered from stakeholders to define the system's needs, which drives the direction of the entire project.

2. **Requirements Analysis**

Contribution: Organizes and refines gathered information, resolving any conflicts or ambiguities, ensuring the requirements are realistic and feasible.

3. **Requirements Specification**

Contribution: Documents the requirements in a clear, structured way, providing a formalized blueprint for development and ensuring all stakeholders have a shared understanding.

4. **Requirements Validation**

Contribution: Verifies that the documented requirements are correct, complete, and aligned with the user's and business's needs, reducing the risk of miscommunication.

**5. Modeling the Requirements**

Contribution: Visualizes complex requirements using models (e.g., use cases, activity diagrams), making them easier to understand and communicate, ensuring clarity for the development team.

**6. Requirements Prioritization**

Contribution: Ranks the requirements based on importance and urgency, ensuring that the most critical features are developed first and helping manage project scope.

**7. Requirements Traceability**

Contribution: Tracks each requirement through design, implementation, and testing, ensuring all requirements are addressed and making it easier to manage changes.

**8. Managing Changes to Requirements**

Contribution: Provides a structured process for handling changes to requirements, ensuring that changes are controlled and do not disrupt the development timeline or scope.

## 4c. Demonstrate class-based modelling and identify the analysis classes

### CLASS-BASED MODELING

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

The elements of a class-based model include classes and objects, attributes, operations, class responsibility collaborator (CRC) models, collaboration diagrams, and packages

### Identifying Analysis Classes

We can begin to identify classes by examining the usage scenarios developed as part of the requirements model and performing a "**grammatical parse**" on the use cases developed for the system to be built.

*Analysis classes* manifest themselves in one of the following ways:

- **External entities** (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.

- **Things** (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
- ***Occurrences or events*** (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- **Roles** (e.g., manager, engineer, salesperson) played by people who interact with the system.
- **Organizational units** (e.g., division, group, team) that are relevant to an application.
- **Places** (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- **Structures** (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

## 5a. Explain Agile Process and its principles

### AGILE PROCESS

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

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable

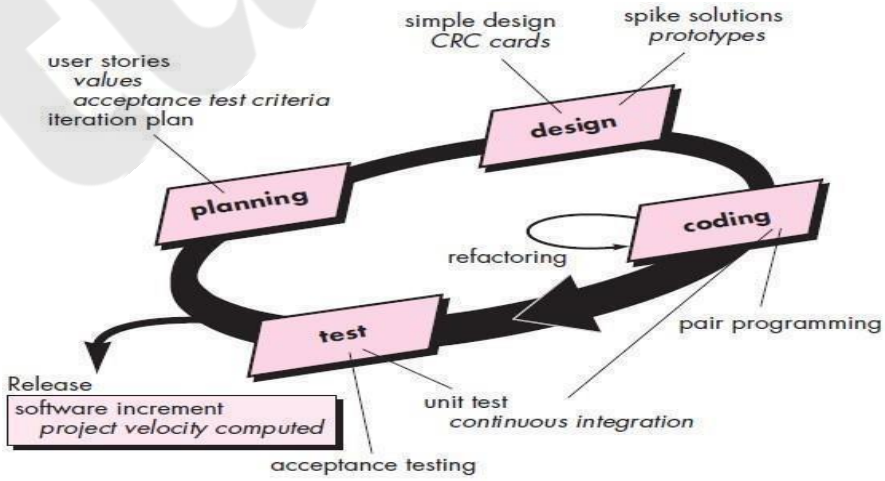
**Agility Principles**

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.

## 5b. Demonstrate Extreme Programming Process with a neat, labelled diagram



Key XP activities are

- **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.
- **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

  - **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 Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.
  - **Testing.** 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. This encourages a regression testing strategy whenever code is modified. 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. Wells states: "Fixing small problems every few hours takes less time than fixing huge problems just before the deadline."

## 5c. Write short notes on tool set for Agile Process

### Tool Set for Agile Process

Agile processes require tools that support collaboration, iteration, and continuous improvement. Below are some key categories and examples of tools commonly used:

**1. Project Management Tools**

- **Purpose**: To manage sprints, tasks, and backlogs efficiently.
- **Examples**: Jira, Trello, Asana, Monday.com.
- **Features**:
    - Sprint planning and tracking.
    - Kanban and Scrum boards.
    - Task assignment and prioritization.

**2. Communication and Collaboration Tools**

- **Purpose**: Facilitate team communication and knowledge sharing.
- **Examples**: Slack, Microsoft Teams, Zoom, Confluence.
- **Features**:
    - Real-time messaging and video conferencing.
    - Document sharing and version control.
    - Centralized discussion threads.

**3. Version Control Tools**

- **Purpose**: Manage source code and versioning in development.
- **Examples**: Git, GitHub, GitLab, Bitbucket.
- **Features**:
    - Code branching and merging.
    - Pull requests and code reviews.
    - Change tracking.

**4. Continuous Integration/Continuous Deployment (CI/CD) Tools**

- **Purpose**: Automate building, testing, and deploying code.
- **Examples**: Jenkins, Travis CI, Circle CI, GitHub Actions.
- **Features**:
    - Automated testing.
    - Deployment pipelines.
    - Integration with version control.

**5. Testing Tools**

- **Purpose**: Ensure quality through automated and manual testing.
- **Examples**: Selenium, JUnit, Postman, Cypress.
- **Features**:
    - Functional and regression testing.
    - API testing and load testing.
    - Reporting and debugging.

**6. Retrospective and Feedback Tools**

- **Purpose**: Reflect on team performance and gather feedback.
- **Examples**: Parabol, Retrium, Easy Retro.
- **Features**:
    - Retrospective templates.
    - Anonymous feedback collection.

- Action item tracking.

**7. Documentation and Knowledge Management Tools**

- **Purpose**: Document processes, decisions, and learnings.
- **Examples**: Confluence, Notion, Google Docs.
- **Features**:
  - Rich text editing.
  - Shared knowledge base.
  - Version history.

**8. Metrics and Reporting Tools**

- **Purpose**: Track team performance and project progress.
- **Examples**: Tableau, Power BI, Jira dashboards.
- **Features**:
  - Burn-down and burn-up charts.
  - Velocity tracking.
  - Customizable reports.

By leveraging these tools, Agile teams can enhance transparency, communication, and adaptability, ensuring successful project delivery.

## 6a. Explain the principles that guide each framework activity in the software engineering process

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

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

- **Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer. The intent is to understand stakeholders objectives for the project and to gather requirements that help define software features and functions.
- **Planning.** Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a

*software project plan*—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

- **Modeling.** Creation of models to help developers and customers understand the requires and software design
- **Construction.** This activity combines code generation and the testing that is required to uncover errors in the code.
- **Deployment.** The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

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

## 6 b. Describe software engineering knowledge and its core principles

**Software Engineering Knowledge**

Software engineering knowledge encompasses the principles, methods, tools, and practices used to design, develop, test, and maintain high-quality software systems efficiently and effectively. It combines insights from computer science, project management, quality assurance, and systems engineering to produce reliable, maintainable, and scalable software.

**Key Areas of Software Engineering Knowledge**

1. **Software Development Life Cycle (SDLC)**:
    - Frameworks for planning, creating, testing, and deploying software (e.g., Waterfall, Agile, DevOps).
2. **Software Design and Architecture**:
    - Techniques to design systems for scalability, modularity, and performance.
3. **Programming and Implementation**:
    - Proficient use of programming languages, frameworks, and tools.
4. **Testing and Quality Assurance**:
    - Processes for validating and verifying software functionality and performance.

## Core Principles of Software Engineering

### 1. Emphasis on Quality

- Deliver high-quality software by focusing on usability, reliability, performance, and security.
- Apply rigorous testing and validation techniques.

### 2. Systematic Approach

- Follow structured processes and methodologies to reduce complexity and improve predictability.

### 3. Abstraction

- Simplify complex systems by focusing on essential features and ignoring irrelevant details.
- Use models and designs to represent software components and their interactions.

### 4. Modularity

- Divide software into smaller, manageable, and reusable components.
- Facilitates easier maintenance, testing, and scalability.

### 5. Iterative Development

- Develop software incrementally, allowing for feedback and continuous improvement.
- Commonly used in Agile and iterative models.

### 6. Risk Management

- Identify, assess, and mitigate risks throughout the software lifecycle to prevent costly issues.

### 7. Maintainability

- Write clean, well-documented code to ensure software can be easily understood and updated.

### 8. User-Centric Design

- Focus on user needs and behaviors to create intuitive and functional software.
- Design systems to handle increasing workloads and maintain performance under stress.

### 10. Collaboration and Communication

- Encourage teamwork and effective communication among developers, stakeholders, and end-users.

### 11. Ethics and Responsibility

- Ensure the software respects privacy, adheres to legal requirements, and aligns with societal values.

**12. Continuous Learning**

- Stay updated with new technologies, practices, and tools to adapt to a rapidly evolving field.

By adhering to these principles, software engineering ensures that software is developed efficiently, meets user expectations, and remains sustainable over time.

## 6 c. Write short notes on other Agile software process.

## OTHER AGILE PROCESS MODELS

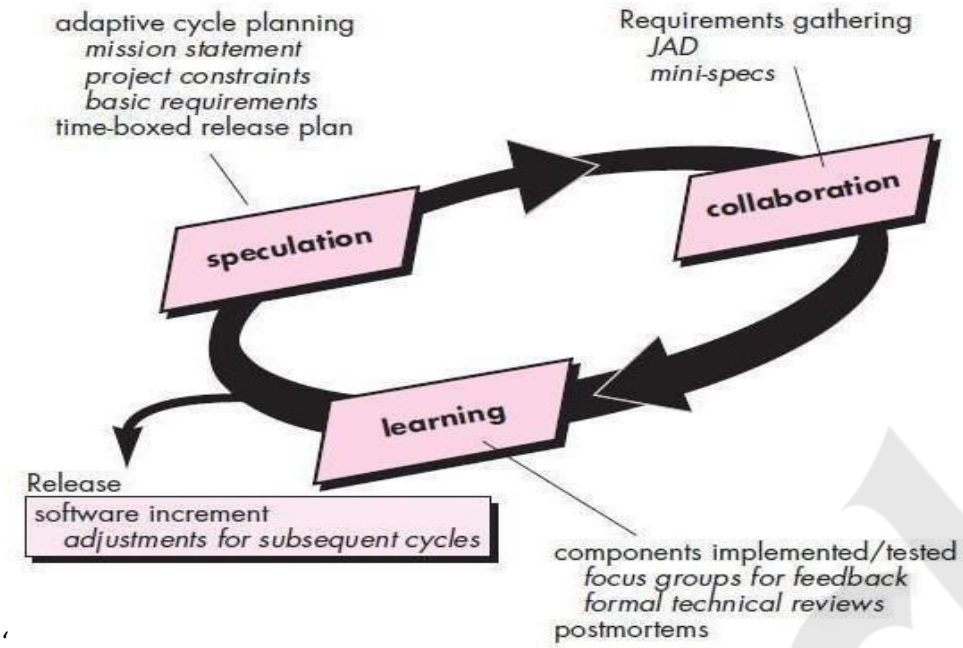
Other agile process models have been proposed and are in use across the industry. Among the most common are:

- Adaptive Software Development (ASD)
- Scrum
- Dynamic Systems Development Method (DSDM)
- Crystal
- Feature Drive Development (FDD)
- Lean Software Development (LSD)
- Agile Modeling (AM)
- Agile Unified Process (AUP)

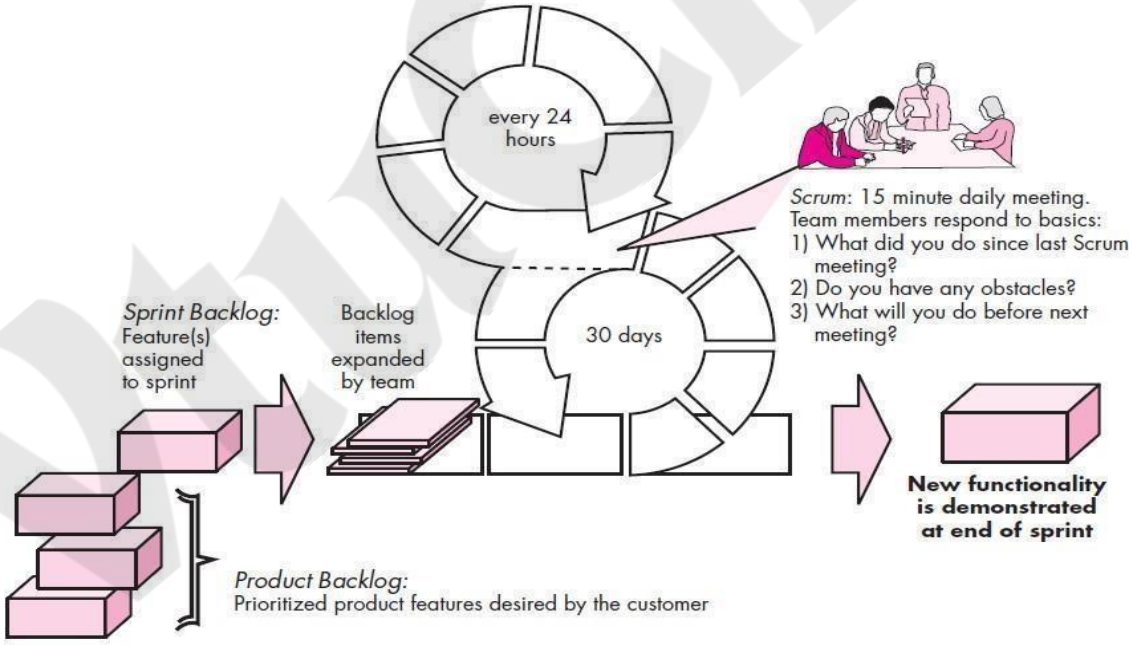
**Adaptive Software Development (ASD)**

*Adaptive Software Development* (ASD) has been proposed by Jim Highsmith as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization.

High smith argues that an agile, adaptive development approach based on collaboration is "as much a source of *order* in our complex interactions as discipline and engineering." He defines an ASD "life cycle" that incorporates three phases, speculation, collaboration, and learning.




## Scrum:




## Feature Drive Development (FDD)

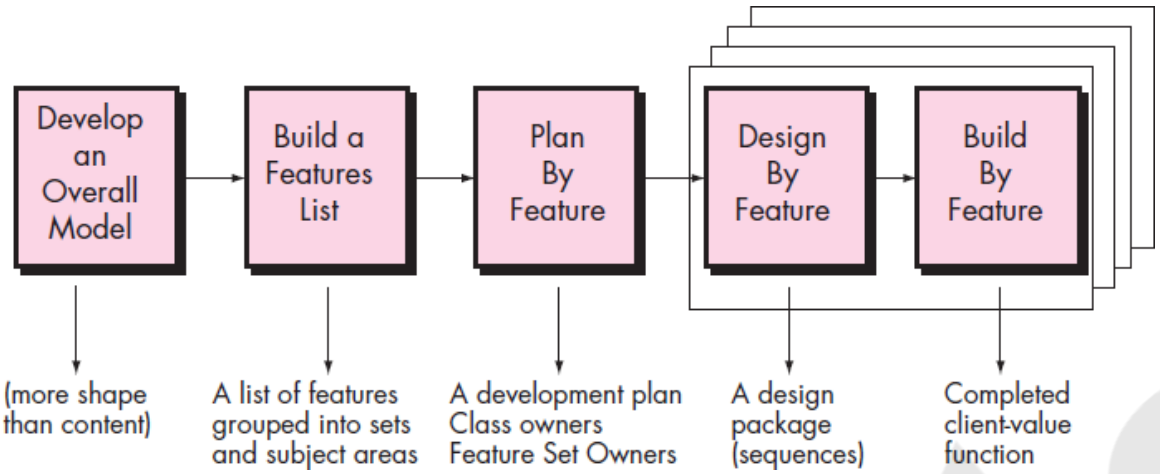



**Fig : Feature Driven Development (FDD)**

**Crystal**

Alistair Cockburn and Jim Highsmith created the *Crystal family of agile methods* in order to achieve a software development approach that puts a premium on "maneuverability" during what Cockburn characterizes as "a resource limited, cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game"

The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects. The intent is to allow agile teams to select the member of the crystal family that is most appropriate for their project and environment.

## 7 a. Describe Project Management life cycle

### PROJECT MANAGEMENT LIFE CYCLE

Software development life cycle denotes (SDLC) the stages through which a software is developed. In contrast to SDLC, the project management life cycle typically starts well before the software development activities start and continues for the entire duration of SDLC.

In Project Management process, the project manager carries out project initiation, planning, execution, monitoring, controlling and closing.

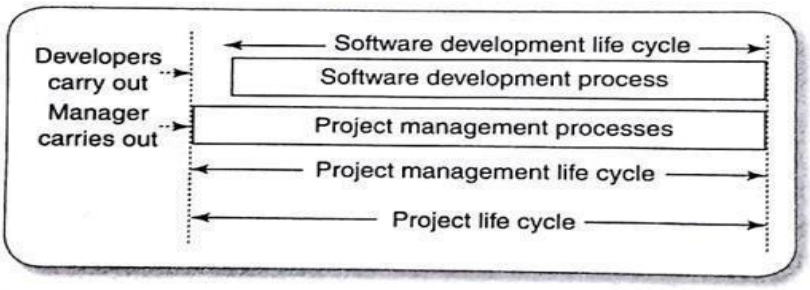



**FIGURE 1.7** Project management life cycle versus software development life cycle

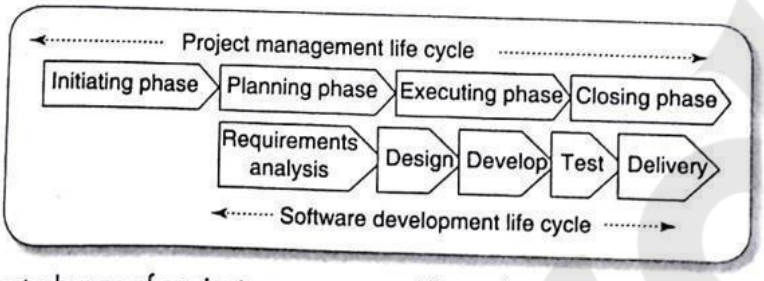



**FIGURE 1.8** Different phases of project management life cycle and software development life cycle

1. **Project Initiation**: The project initiation phase starts with project **concept development**. During concept development the different characteristics of the software to be developed are thoroughly understood, which includes, the scope of the project, the project constraints, the cost that would be incurred and the benefits that would accrue. Based on this understanding, a **feasibility study** is undertaken to determine the project would be financially and technically feasible.

   Based on feasibility study, the **business case** is developed. Once the top management agrees to the business case, the **project manager** is appointed, the **project charter** is written and finally **project team** is formed. This sets the ground for the manager to start the **project planning phase.**

W5HH Principle: Barry Boehm, summarized the questions that need to be asked and answered in order to have an understanding of these project characteristics.

- Why is the software being built?
- What will be done?
- When will it be done?
- Who is responsible for a function?

2. **Project Bidding:** Once the top management is convinced by the business case, the **project charter** is developed. For some categories of projects, it may be necessary to have formal bidding process to select suitable vendor based on some cost-

performance criteria. The different types of bidding techniques are:

- **Request for quotation(RFQ) :** An organization advertises an RFQ if it has good understanding of the project and the possible solutions.

3. **Project Planning:** An importance of the project initiation phase is the project charter. During the project planning the project manger carries out several processes and creates the following documents:
    - **Project plan:** This document identifies the project the project tasks and a schedule for the project tasks that assigns project resources and time frames to the tasks.
    - **Resource Plan:** It lists the resources , manpower and equipment that would be required to execute the project.
    - **Functional Plan:** It documents the plan for manpower, equipment and other costs.
    - **Quality Plan:** Plan of quality targets and control plans are included in this document.
    - **Project Execution:** In this phase the tasks are executed as per the project plan developed during the planning phase. Quality of the deliverables is ensured through execution of proper processes. Once all the deliverables are produced and accepted by the customer, the project execution phase completes and the project closure phase starts.

    - **Project Closure:** Project closure involves completing the release of all the required deliverables to the customer along with the necessary documentation. All the Project resources are released and supply agreements with the vendors are terminated and all the pending payments are completed. Finally, a postimplementation review is undertaken to analyze the project performance and to list the lessons for use in future projects.

## 7 b. Demonstrate activities covered by software project management with neat diagram

## ACTIVITIES COVERED BY SOFTWARE PROJECT MANAGEMENT:

The activities covered by Software Project management are diagrammatically illustrated as follows:

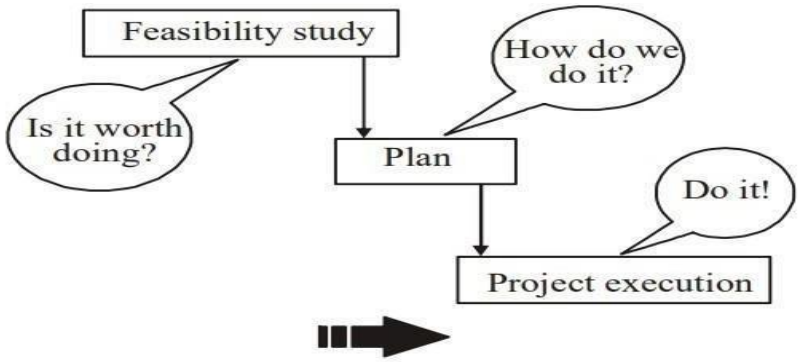



**Figure 1.1: The Feasibility Study / Plan / Execution Cycle**

## he Feasibility Study:

This is an investigation into whether a prospective project is worth starting that it has a valid *business case*. Information is gathered about the requirements of the proposed application. The probable developmental and operational costs, along with the value of the benefits of the new system, are estimated. The study could be part of a strategic planning exercise examining and prioritizing a range of potential software developments.

## Planning:

If the feasibility study produces results which indicate that the prospective project appears viable, planning of the project can take place. However, for a large project, we would not do all our detailed planning right at the beginning. We would formulate an outline plan for the whole project and a detailed one for the first stage. More detailed planning of the later stages would be done as they approached. This is because we would have more detailed and accurate information upon which to base our plans nearer to the start of the later stages.

## Project Execution:

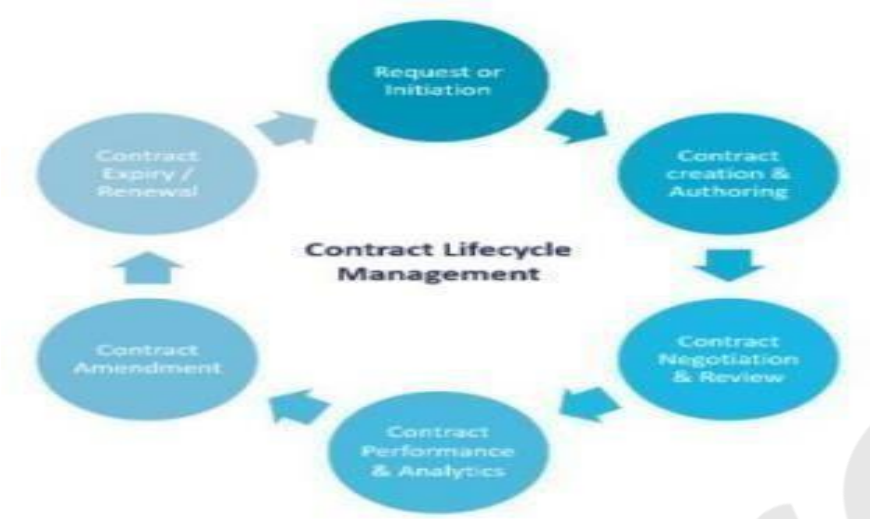
The project can now be executed. The execution of a project often contains *design* and *implementation* subphases. The same is illustrated in **Figure 1.2** which shows the typical sequence of software development activities recommended in the international standard ISO 12207.

## Requirements Analysis:

This starts with requirement elicitation or requirement gathering which establishes what the users require of the system that the project is to implement. Some work along these lines will almost certainly have been carried out when the project was evaluated,

but now the original information obtained needs to be updated and supplemented.

## 7 c. Explain various stages of contract management.




**Various Stages of Contract Management**

### Request and Creation:

**Request**: Identifying the need for a contract and gathering the necessary information to draft it.

**Creation**: Drafting the contract terms and conditions that align with the requirements and objectives of all parties involved.

**Example**: A software company needs to hire a third-party developer to work on a new project.

*The project manager identifies the need for a contract and gathers details about the scope of work, timelines, payment terms, and other specifics.*

### 1. Negotiation:

Parties involved discuss and negotiate the terms of the contract to reach a mutual agreement. This stage often involves revisions and adjustments.

**Example**: The software company and the third-party developer negotiate the terms. The developer might request more time or a higher payment, while the company might request milestones for progress checks.

## 2. Approval and Execution:

**Approval**: Obtaining necessary approvals from stakeholders and legal departments.

**Execution**: Signing the contract, making it a legally binding document.

**Example**: Once the terms are finalized, the contract is reviewed by both parties' legal teams. After approval, both the software company and the developer sign the contract.

## 3. Obligations and Performance:

Ensuring that all parties adhere to the terms and conditions agreed upon in the contract. Monitoring performance and compliance.

**Example**: The developer starts working on the project, adhering to the deadlines and deliverables specified in the contract. The software company provides the necessary resources and makes payments as per the contract.

## 4. Modification and Renewal:

Making necessary amendments if any changes occur during the contract period. Reviewing and renewing contracts as needed.

**Example**: Midway through the project, the software company requests additional features not covered in the original contract. An amendment is made to include these new features and adjust the payment terms accordingly. As the project nears completion, the company and developer may negotiate a renewal for ongoing maintenance.

## 5. Closure:

Completing all contractual obligations, ensuring all parties have met their requirements, and formally closing the contract.

**Example**: The developer finishes the project, and the software company conducts a final review to ensure all deliverables meet the agreed-upon standards. Once confirmed, the contract is closed, and a final payment is made.

## 8 a. Explain project risk evaluation.

**Project Risk Evaluation**

Project risk evaluation is the process of Analyzing identified risks to determine their potential impact on the project's objectives (such as scope, schedule, budget, and quality) and prioritizing them for mitigation or monitoring. This process is essential for proactive risk management and ensures that potential problems are addressed before they become critical.

---

**Steps in Project Risk Evaluation**

1. **Identify Risks**
    - Compile a comprehensive list of risks identified during brainstorming sessions, historical data analysis, or expert consultations.
2. **Assess Risk Probability**
    - Estimate the likelihood of each risk occurring using qualitative (e.g., high, medium, low) or quantitative (e.g., percentage probabilities) methods.
3. **Assess Risk Impact**
    - Evaluate the potential consequences of each risk on project objectives.
    - Consider impacts on:
        - Project timeline (delays).
        - Budget (cost overruns).
        - Deliverables (scope creep or reduced quality).
        - Stakeholders (satisfaction and reputation).
4. **Calculate Risk Severity (Risk Exposure)**
    - Combine probability and impact to prioritize risks.
    - **Risk Exposure Formula**:
      Risk Exposure=Probability of Occurrence×Impact\text{Risk Exposure} = \text{Probability of Occurrence} \times \text{Impact}
5. **Risk Categorization**
    - Group risks into categories (e.g., technical, financial, operational, external) for better understanding and targeted mitigation.
6. **Prioritize Risks**
    - Use tools like a **Risk Matrix** or **Heat Map** to rank risks based on their severity:
        - **High Priority**: Immediate action required.

- **Medium Priority**: Monitor and mitigate as necessary.
- **Low Priority**: Minimal attention required.

---

**Tools for Risk Evaluation**

1. **Risk Matrix**:
   - A visual tool that maps risks based on their probability and impact.
   - Example:
     - High Probability & High Impact → Critical Risk.
     - Low Probability & Low Impact → Insignificant Risk.
2. **SWOT Analysis**:
   - Identify strengths, weaknesses, opportunities, and threats related to risks.
3. **Risk Register**:
   - A document listing all risks, their evaluation details (probability, impact), and mitigation strategies.
4. **Monte Carlo Simulation**:
   - A quantitative tool that uses statistical modeling to assess the combined impact of multiple risks.

---

**Benefits of Risk Evaluation**

- **Proactive Mitigation**: Helps in developing targeted strategies to reduce the likelihood or impact of critical risks.
- **Resource Allocation**: Focuses resources on high-priority risks, optimizing time and cost.
- **Informed Decision-Making**: Enables stakeholders to make better decisions based on risk data.
- **Improved Project Success**: Reduces uncertainty and enhances the likelihood of meeting project objectives.

Risk evaluation is a continuous process, revisited throughout the project lifecycle to adapt to new risks and changing project conditions.

## 8 b. Explain cost benefits of evaluation techniques.

### Cost Benefits of Evaluation Techniques

Evaluation techniques help organizations assess and compare the costs and benefits of different strategies, projects, or decisions. These techniques ensure that resources are allocated efficiently, maximize returns, and minimize waste. Below are key cost-benefit evaluation techniques and their associated benefits.

---

### Key Cost-Benefit Evaluation Techniques

#### 1. Cost-Benefit Analysis (CBA)

- **Description**: Compares the total expected costs against the total expected benefits of a project to determine its feasibility.
- **Cost Benefits**:
    - Identifies high-value projects with a positive return on investment (ROI).
    - Prevents resource wastage by avoiding non-profitable projects.
    - Provides a clear monetary framework for decision-making.

#### 2. Return on Investment (ROI)

- **Description**: Measures the profitability of an investment by calculating the ratio of net benefits to costs.
- **Formula**: ROI=Net BenefitsCosts×100\text{ROI} = \frac{\text{Net Benefits}}{\text{Costs}} \times 100
- **Cost Benefits**:
    - Highlights projects with the highest financial returns.
    - Simplifies the comparison of multiple investment options.
    - Supports strategic allocation of funds to maximize profit.

#### 3. Payback Period

- **Description**: Determines the time required to recover the initial investment from the project's benefits.
- **Cost Benefits**:
    - Assists in selecting projects with quick cost recovery.
    - Reduces financial risk by focusing on short-term gains.
    - Useful for cash flow management.

#### 4. Net Present Value (NPV)

- **Description**: Calculates the present value of cash inflows and outflows over the project's life, considering the time value of money.
- **Formula**: NPV=∑(Cash In flowt(1+r)t)−Initial Investment\text{NPV} = \sum \left( \frac{\text{Cash Inflow}_t}{(1 + r)^t} \right) - \text{Initial Investment}
    - tt: Time period
    - rr: Discount rate
- **Cost Benefits**:
    - Highlights projects that deliver long-term profitability.
    - Accounts for inflation and the time value of money.
    - Facilitates comparison of projects with varying time horizons.

**5. Internal Rate of Return (IRR)**

- **Description**: Determines the discount rate at which the NPV of a project becomes zero.
- **Cost Benefits**:
    - Helps in evaluating the profitability and efficiency of investments.
    - Facilitates comparison of projects with different scales.
    - Indicates the breakeven discount rate.

**6. Economic Value Added (EVA)**

- **Description**: Measures the project's value creation by comparing net operating profit after taxes (NOPAT) to the cost of capital.
- **Cost Benefits**:
    - Identifies projects that exceed the cost of capital, ensuring value creation.
    - Highlights areas of inefficiency in cost management.

**7. Sensitivity Analysis**

- **Description**: Examines how changes in key assumptions (e.g., costs, benefits, discount rates) affect project outcomes.
- **Cost Benefits**:
    - Identifies risks associated with varying assumptions.
    - Helps in making robust decisions under uncertainty.

## 8 c. Differentiate between Software Projects V/S Other types of projects.

**Differences Between Software Projects and Other Types of Projects**

| Aspect | Software Projects | Other Types of Projects |
|---|---|---|
| **Nature of Deliverables** | Intangible deliverables like code, applications, and systems. | Tangible deliverables like buildings, products, or events. |
| **Requirements Definition** | Often ambiguous or incomplete; requires iterative clarification and validation. | Typically well-defined and measurable before the project begins. |
| **Development Process** | Iterative and incremental; Agile, Scrum, or Waterfall methods are common. | Linear or sequential; follows fixed phases like planning, execution, and closure. |
| **Measurement of Progress** | Progress is harder to measure; depends on completed functionality and quality. | Progress is easily measurable (e.g., percentage of physical construction completed). |
| **Resource Requirements** | Relies heavily on skilled human resources (developers, designers, testers). | Requires both skilled and unskilled labor, along with physical resources and materials. |
| **Risk and Uncertainty** | High due to intangible deliverables, evolving requirements, and technology challenges. | Moderate to low; risks are more predictable and based on physical constraints. |
| **Quality Control** | Includes code reviews, testing, and debugging; heavily relies on automation tools. | Includes inspections, physical testing, or adherence to standards like ISO. |
| **Change Management** | Changes are frequent and expected; managed through version control and agile practices. | Changes are costly and disruptive; require formal approvals and redesign efforts. |
| **Customer Involvement** | Requires ongoing customer involvement for feedback and requirement adjustments. | Limited customer involvement after initial requirements are finalized. |

| Aspect | Software Projects | Other Types of Projects |
|---|---|---|
| **Tools and Techniques** | Specialized tools like IDEs, CI/CD pipelines, version control, and testing frameworks. | Tools and machinery specific to the domain, such as construction equipment or event software. |
| **Cost Estimation** | Challenging; subject to variability due to uncertain requirements and technology changes. | Easier to estimate due to standardized processes and materials. |
| **Delivery Timeframe** | Deadlines are flexible; scope adjustments may extend timelines. | Deadlines are stricter; often tied to external events or physical constraints. |
| **Lifecycle** | Ongoing maintenance and updates after delivery (e.g., bug fixes, feature enhancements). | Minimal post-delivery involvement except for repairs or warranties. |

## 9 a. Define Software Quality and explain the place of software quality in project management

### THE PLACE OF SOFTWARE QUALITY IN PROJECT PLANNING

Quality will be of concern at all stages of project planning and execution, but will be particular interest at Stepwise framework .

**Step 1 : Identifying project scope and objectives** Some objective could relate to the quality of the application to be delivered.

**Step 2 : Identifying project infrastructure** Within this step activity 2.2 involves identifying installation standards and procedures. Some of these will almost certainly be about quality requirements.

**Step 3 : Analyze project characteristics.** In this activity the application to be implemented will be examined to see if it has any special quality requirements.

**Example**: Safety-critical applications might require additional activities such as n-version development, where multiple teams develop versions of the same software to cross-check outputs.

**Step 4 : Identify the product and activities of the project.** It is at that point the entry, exit and process requirement are identified for each activity.

**Step 8: Review and publicize plan.** At this stage the overall quality aspects of the project plan are reviewed.

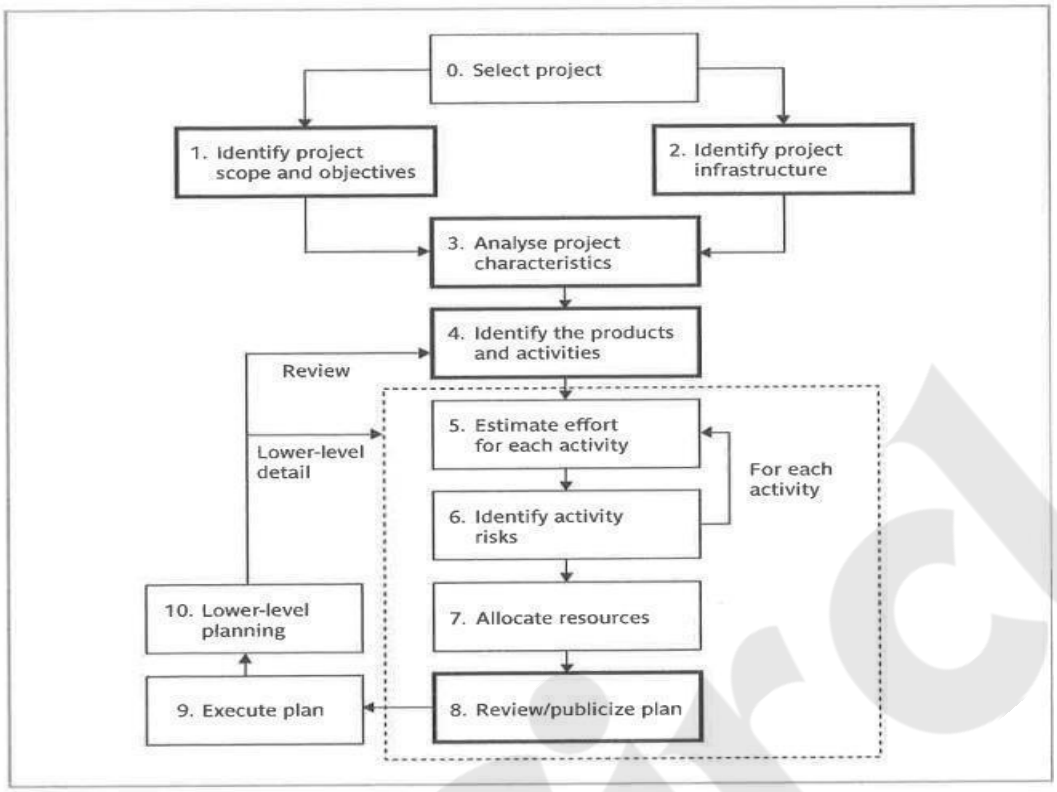



**FIGURE 13.1** The place of software quality in Step Wise

## 9 b. Explain Software Quality Models.

## SOFTWARE QUALITY MODELS

1. McCall's model.
2. Dromey's Model.
3. Boehm's Model.
4. ISO 9126 Model.

**1) McCall' Model:** McCall defined the quality of a software in terms of three broad parameters: its operational characteristics, how easy it is to fix defects and how easy it is to part it to different platforms. These three high-level quality attributes are defined based on the following eleven attributes of the software:


**Correctness:** The extent to which a software product satisfies its specifications.
**Reliability:** The probability of the software product working satisfactorily over a given duration.
**Efficiency:** The amount of computing resources required to perform the required functions.
**Integrity:** The extent to which the data of the software product remain valid.
**Usability:** The effort required to operate the software product.
**Maintainability:** The ease with which it is possible to locate and fix bugs in the software product.
**Flexibility:** The effort required to adapt the software product to changing requirements.
**Testability:** The effort required to test a software product to ensure that it performs its intended function.
**Portability:** The effort required to transfer the software product from one hardware or software system environment to another.
**Reusability:** The extent to which a software can be reused in other applications.
**Interoperability:** The effort required to integrate the software with other software.

**2. Dromey's model:** Dromey proposed that software product quality depends on four major high-level properties of the software: Correctness, internal characteristics, contextual characteristics and certain descriptive properties. Each of these high-level properties of a software product, in turn depends on several lower-level quality attributes.

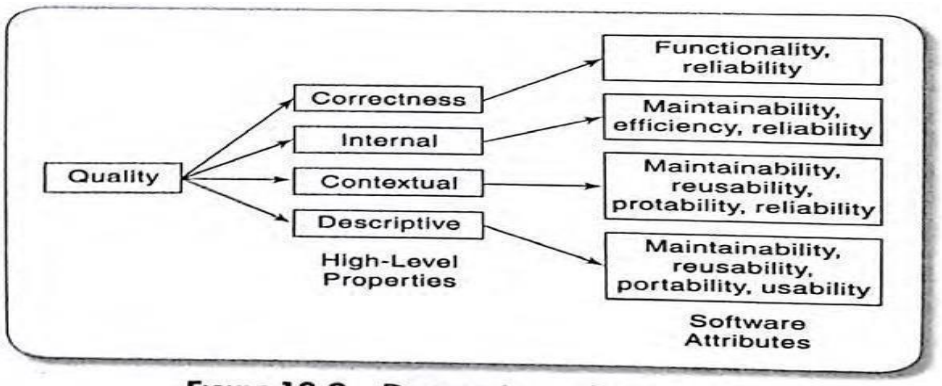



FIGURE 13.2 Dromey's quality model

3. **Boehm's Model:** Boehm's suggested that the quality of a software can be defined based on these high-level characteristics that are important for the users of the software. These three high-level characteristics are the following:

**As-is -utility:** How well (easily, reliably and efficiently) can it be used? **Maintainability:** How easy is to understand, modify and then retest the software? **Portability:** How difficult would it be to make the software in a changed environment?

Boehm's expressed these high-level product quality attributes in terms of several measurable product attributes.

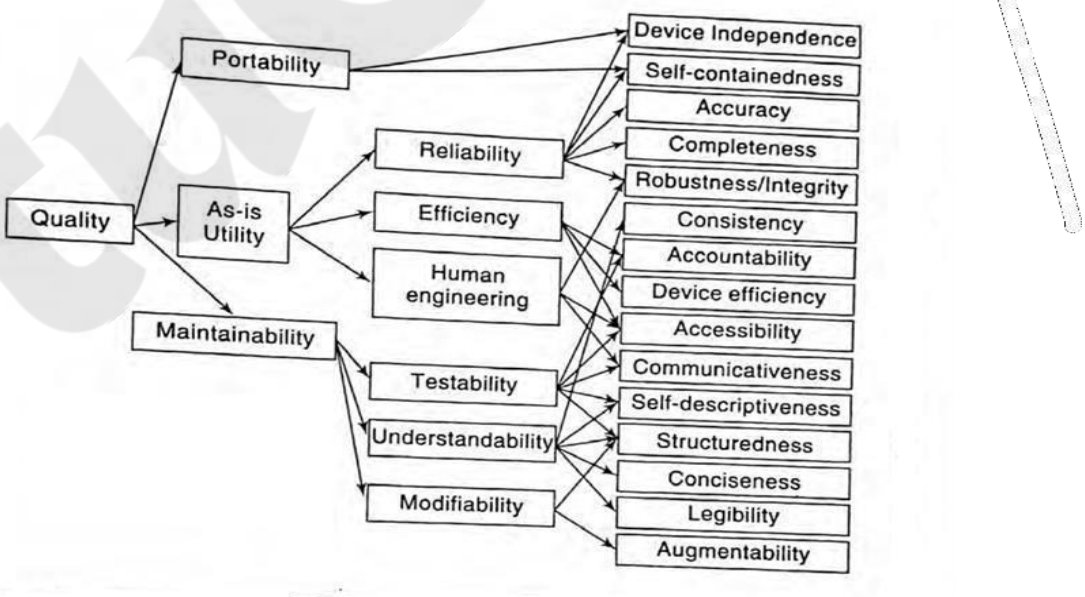



FIGURE 13.3 Boehm's quality model

## 9 c. Differentiate between Product and Process Quality Management.

| Aspect | Product Quality Management | Process Quality Management |
|---|---|---|
| **Definition** | Focuses on ensuring the final deliverable (product) meets quality standards and user expectations. | Focuses on ensuring the processes used to create the product are efficient and adhere to quality standards. |
| **Objective** | Deliver a high-quality product that satisfies customer requirements. | Establish and maintain efficient, consistent, and repeatable processes. |
| **Scope** | Concerned with the outcomes or results of the process (the product). | Concerned with the methods and workflows used to produce the outcomes. |
| **Measurement Criteria** | Quality is measured through functional testing, usability, performance, and reliability of the product. | Quality is measured through compliance with process standards, efficiency, and defect prevention. |
| **Techniques Used** | - Testing (unit, integration, system, acceptance). - Customer feedback. - Inspections and audits. | - Process audits and evaluations. - Statistical process control (SPC). - Process performance metrics (e.g., cycle time, defect rate). |
| **Standards** | Adheres to product-specific standards (e.g., ISO/IEC 25010 for software product quality). | Adheres to process standards (e.g., ISO 9001 for quality management systems, CMMI for software processes). |
| **Focus Area** | The end deliverable and its features. | The methods and procedures used to produce the deliverable. |
| **Responsibility** | Primarily the responsibility of product teams, testers, and quality assurance specialists. | Primarily the responsibility of process managers, process engineers, and quality control teams. |
| **Impact on Quality** | Directly impacts customer satisfaction and market acceptance of the product. | Indirectly impacts product quality by improving the efficiency and reliability of production processes. |
| **Examples** | - Ensuring software meets performance benchmarks. - Validating a product against customer requirements. | - Standardizing coding practices. - Establishing a consistent testing protocol. |

## 10 a. Describe decomposition techniques in software project estimation

**Decomposition Techniques in Software Project Estimation**

Decomposition techniques are used in software project estimation to break down a project into smaller, manageable components. This approach allows for more accurate and detailed estimation by analyzing individual components and aggregating their estimates. Decomposition techniques are widely used to estimate project size, effort, cost, and schedule.

---

**Types of Decomposition Techniques**

**1. Decomposition by Functionality**

- **Description**:
  - The project is broken down into functional modules or features.
  - Each module or feature is estimated separately.
- **Steps**:

1. Identify key functionalities or modules.
2. Break down each functionality into smaller sub-tasks.
3. Estimate effort, cost, or time for each sub-task.
4. Aggregate the estimates.

- **Advantages**:
  - Easy to understand for non-technical stakeholders.
  - Aligns with user requirements and business goals.
- **Example**:
  - For an e-commerce website, functionalities like "User Registration," "Product Search," and "Payment Processing" would be decomposed for estimation.

---

**2. Decomposition by Activities**

- **Description**:
  - The project is divided into activities based on the software development lifecycle (SDLC), such as requirements gathering, design, coding, testing, and deployment.
- **Steps**:

1. List all project activities.
2. Break down each activity into smaller tasks.

3. Estimate effort or cost for each task.
4. Sum up the estimates.

- **Advantages**:
  - Useful for projects with well-defined lifecycle stages.
  - Provides a clear understanding of the effort distribution across phases.
- **Example**:
  - A software project might be broken into "Requirement Analysis (40 hours)," "Design (60 hours)," "Development (120 hours)," etc.

---

**3. Work Breakdown Structure (WBS)**

- **Description**:
  - A hierarchical decomposition of the project into smaller components or tasks.
  - Tasks are grouped into work packages, which are then estimated.
- **Steps**:

1. Create a hierarchical tree structure of deliverables.
2. Break each deliverable into smaller work packages.
3. Estimate time, effort, or cost for each work package.

- **Advantages**:
  - Comprehensive and widely used in project management.
  - Facilitates detailed tracking and management.
- **Example**:
  - A WBS for a mobile app might include levels for "Frontend Development," "Backend Development," and "Testing," each with detailed sub-tasks.

---

**4. Decomposition by Technical Complexity**

- **Description**:
  - The project is divided based on technical layers or complexity, such as frontend, backend, database, and integration.
- **Steps**:

1. Identify technical layers or components.
2. Break down each component into smaller parts.

3. Estimate the effort required for each part.

- **Advantages**:
  - Helps focus on technically challenging areas.
  - Useful for projects with high technical complexity.
- **Example**:
  - For a web application, decomposition might include "Frontend UI Development (80 hours)," "API Integration (40 hours)," and "Database Design (60 hours)."

---

**5. Decomposition by Deliverables**

- **Description**:
  - The project is decomposed into deliverables or milestones.
  - Each deliverable is estimated separately.
- **Steps**:

1. Identify all project deliverables.
2. Break down each deliverable into tasks or sub-deliverables.
3. Estimate effort or cost for each task.
4. Aggregate the estimates.

- **Advantages**:
  - Keeps focus on project outcomes.
  - Simplifies tracking and progress monitoring.
- **Example**:
  - For a software product, deliverables might include "Prototype," "Beta Version," and "Final Release."

---

**Benefits of Decomposition Techniques**

- **Improved Accuracy**: Smaller components are easier to estimate accurately.
- **Enhanced Clarity**: Decomposition provides a clear understanding of project scope and requirements.
- **Risk Management**: Identifies high-risk areas during estimation.
- **Better Resource Allocation**: Enables precise allocation of resources to specific tasks or components.

**Challenges**

- **Overhead**: Decomposition can be time-consuming and requires expertise.
- **Inaccuracy in Aggregation**: Errors in estimating smaller components can add up.
- **Scope Creep**: If decomposition is not well-managed, it can lead to scope expansion.

Decomposition techniques are foundational in project estimation as they provide a systematic way to understand, estimate, and manage complex software projects.

## 10 b. Explain Empirical Estimation model

An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC or FP.11 Values for LOC or FP are estimated using the approach described in Sections 26.6.3 and 26.6.4. But instead of using the tables described in those sections, the resultant values for LOC or FP are plugged into the estimation model.

The empirical data that support most estimation models are derived from a limited sample of projects. For this reason, no estimation model is appropriate for all classes of software and in all development environments. Therefore, you should use the results obtained from such models judiciously.

An estimation model should be calibrated to reflect local conditions. The modelshould be tested by applying data collected from completed projects, plugging the data into the model, and then comparing actual to predicted results. If agreement is poor, the model must be tuned and retested before it can be used.

### 26.7.1 The Structure of Estimation Models

A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form [Mat94]

$$E = A + B \times (e_v)^C \qquad (26.3)$$

where $A$, $B$, and $C$ are empirically derived constants, $E$ is effort in person-months, and $e_v$ is the estimation variable (either LOC or FP). In addition to the relationship noted in Equation (26.3), the majority of estimation models have some form of project adjustment component that enables $E$ to be adjusted by other project characteristics (e.g., problem complexity, staff experience, development environment). Among the many LOC-oriented estimation models proposed in the literature are

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

FP-oriented models have also been proposed. These include

| | |
|---|---|
| $E = -91.4 + 0.355\ \text{FP}$ | Albrecht and Gaffney model |
| $E = -37 + 0.96\ \text{FP}$ | Kemerer model |
| $E = -12.88 + 0.405\ \text{FP}$ | Small project regression model |

A quick examination of these models indicates that each will yield a different result for the same values of LOC or FP. The implication is clear. Estimation models must be calibrated for local needs!

### 26.7.2 The COCOMO II Model

In his classic book on "software engineering economics," Barry Boehm [Boe81] introduced a hierarchy of software estimation models bearing the name COCOMO, for *COnstructive COst MOdel*. The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, called COCOMO II [Boe00]. Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the following areas:

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

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

### 26.7.3 The Software Equation

The *software equation* [Put92] is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project. The model has been derived from productivity data collected for over 4000 contemporary software projects. Based on these data, we derive an estimation model of the form

$$E = \frac{\text{LOC} \times B^{0.333}}{P^3} \times \frac{1}{t^4} \tag{26.4}$$

where

$E$ = effort in person-months or person-years
$t$ = project duration in months or years
$B$ = "special skills factor"[13]
$P$ = "productivity parameter" that reflects: overall process maturity and management practices, the extent to which good software engineering practices are used, the level of programming languages used, the state of the software environment, the skills and experience of the software team, and the complexity of the application

Typical values might be $P = 2000$ for development of real-time embedded software, $P = 10{,}000$ for telecommunication and systems software, and $P = 28{,}000$ for business systems applications. The productivity parameter can be derived for local conditions using historical data collected from past development efforts.

You should note that the software equation has two independent parameters: (1) an estimate of size (in LOC) and (2) an indication of project duration in calendar months or years.

## 10 c. Write a short notes on Observations on project estimations

### 26.5 Software Project Estimation

**Quote:**

"In an age of outsourcing and increased competition, the ability to estimate more accurately . . . has emerged as a critical success factor for many IT groups."

**Rob Thomsett**

Software cost and effort estimation will never be an exact science. Too many variables—human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it. However, software project estimation can be transformed from a black art to a series of systematic steps that provide estimates with acceptable risk. To achieve reliable cost and effort estimates, a number of options arise:

1. Delay estimation until late in the project (obviously, we can achieve 100 percent accurate estimates after the project is complete!).
2. Base estimates on similar projects that have already been completed.
3. Use relatively simple decomposition techniques to generate project cost and effort estimates.
4. Use one or more empirical models for software cost and effort estimation.

Unfortunately, the first option, however attractive, is not practical. Cost estimates must be provided up-front. However, you should recognize that the longer you wait, the more you know, and the more you know, the less likely you are to make serious errors in your estimates.

The second option can work reasonably well, if the current project is quite similar to past efforts and other project influences (e.g., the customer, business conditions, the software engineering environment, deadlines) are roughly equivalent. Unfortunately, past experience has not always been a good indicator of future results.

The remaining options are viable approaches to software project estimation. Ideally, the techniques noted for each option should be applied in tandem; each used as a cross-check for the other. Decomposition techniques take a divide-and-conquer approach to software project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion. Empirical estimation models can be used to complement decomposition techniques and offer a potentially valuable estimation approach in their own right. A model is based on experience (historical data) and takes the form

$$d = f(v_i)$$

where $d$ is one of a number of estimated values (e.g., effort, cost, project duration) and $v_i$ are selected independent parameters (e.g., estimated LOC or FP).

Automated estimation tools implement one or more decomposition techniques or empirical models and provide an attractive option for estimating. In such systems, the characteristics of the development organization (e.g., experience, environment) and the software to be developed are described. Cost and effort estimates are derived from these data.

Each of the viable software cost estimation options is only as good as the historical data used to seed the estimate. If no historical data exist, costing rests on a very shaky foundation. In Chapter 25, we examined the characteristics of some of the software metrics that provide the basis for historical estimation data.