UNIT V

Work Breakdown Structure (WBS): -

A Work Breakdown Structure (WBS) in software engineering is a hierarchical decomposition of a project into smaller, manageable components or tasks. The WBS organizes and defines the total scope of the project, breaking down the project deliverables and activities into smaller parts that can be more easily planned, managed, and assigned.

The primary goal of a WBS is to clarify the tasks involved in completing the project and to establish clear responsibilities, timelines, and resources required.

How a typical WBS is structured in software engineering:

(1). Top-Level Goal (Project Deliverable)

This is the overall project goal, such as delivering a new software application, updating a system, or launching a new feature.

(2). Major Phases or Deliverables

These could include:

Requirements Analysis – Gathering requirements, defining scope, and setting objectives.

Design – System architecture design, database schema design, UI/UX design, etc. UX (User Experience) design is about the overall feel and function of the product. It focuses on user satisfaction by improving the usability, accessibility, and desirability of the product.

Development – Writing and testing code for each feature or module.

Testing – Unit testing, integration testing, system testing, user acceptance testing (UAT).

Deployment and Release – Preparing production environments, deploying code, and making the product available.

Maintenance and Support – Post-release bug fixing, updates, and ongoing support.

(3). Tasks and Subtasks

Within each major phase, tasks are broken down further:

i. Requirements Analysis –

- Identify stakeholders
- Conduct interviews or workshops
- Document functional and non-functional requirements

ii. Design –

- Create system architecture
- Design user interfaces

- Develop a database schema

iii. Development –

1. Feature or Module Development

   Front-end Development:

   UI Implementation - Code user interfaces for each screen (e.g., login page, dashboard, etc).

   Component Development - Build reusable components (e.g., buttons, forms, navigation).

   Front-end Testing - Unit testing for each UI component.

   Back-end Development:

   API Development - Design and implement APIs for communication between front-end and back-end.

   Database Interactions - Implement functions to interact with the database.

   Data Validation and Error Handling - Set up data validation and error-handling mechanisms.

   Back-end Testing - Unit testing for back-end functions and services.

   Database Development:

   Schema Design and Optimization - Define tables, relationships, indexes, and constraints.

   Database Scripts - Write SQL scripts for database initialization, migration, and seeding. Database migration is the process of making changes to a database schema, like adding or modifying tables and columns etc. Database seeding is the process of populating a database with initial or sample data.

   Database Testing - Test queries, data retrieval, and data integrity constraints.

2. Integration

   - Front-End and Back-End Integration - Ensure proper communication between the client (front-end) and server (back-end).
   - API Endpoints Testing: Verify data flows correctly and endpoints respond accurately.
   - Authentication and Authorization: Develop and integrate authentication (e.g., login) and role-based authorization.

3. Security Implementation

   - User Authentication and Authorization - Implement secure user authentication and roles.
   - Data Encryption - Encrypt sensitive data.
   - Vulnerability Testing: Test for common vulnerabilities, such as SQL injection.

4. Code Review and Refactoring

- Peer Review - Conduct code reviews with team members to catch errors and improve quality.
- Refactoring - Refine and optimize code based on feedback from reviews.
- Documentation - Document code and write comments to make it easier for future maintenance.

5. Internal Testing and Debugging

- Unit Testing - Write unit tests for individual functions, modules, or classes.
- Integration Testing - Test how different modules and services work together.
- Performance Testing - Check for bottlenecks, memory leaks, and slow-loading functions.
- Automated Testing Scripts - Develop scripts to automate repetitive tests.

Work Packages –

The smallest units of work, known as work packages, are often assigned to specific team members. Work packages are detailed tasks or deliverables that are concrete and measurable.

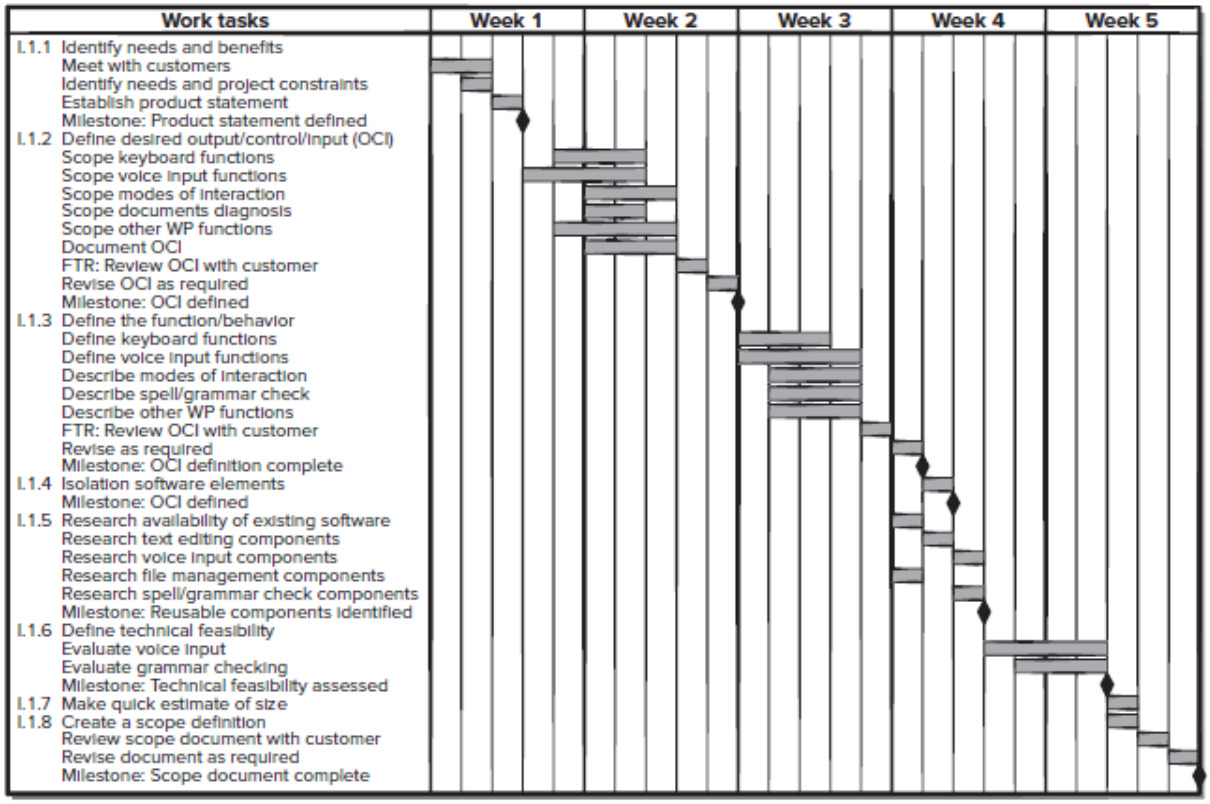
A set of tasks sometimes called the project work breakdown structure (WBS).

As a consequence of these inputs (WBS, Effort, duration, and start date) in automated tools, a time-line chart, also called a Gantt chart, is generated.

A time-line chart can be developed for the entire project. Alternatively, separate charts can be developed for each project function

Gantt Chart

FIGURE 25.6 An example time-line chart



Project Table

FIGURE 25.7 An example project table

Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1 Identify needs and benefits							Scoping will require more effort/time
Meet with customers	wk1, d1	wk1, d1	wk1, d2	wk1, d2	BLS	2 p-d	
Identify needs and project constraints	wk1, d2	wk1, d2	wk1, d2	wk1, d2	JPP	1 p-d	
Establish product statement	wk1, d3	wk1, d2	wk1, d3	wk1, d3	BLS/	1 p-d	
Milestone: Product statement defined	wk1, d3	wk1, d3	wk1, d3	wk1, d3			
I.1.2 Define desired output/control/input (OCI)							
Scope keyboard functions	wk1, d4	wk1, d4	wk2, d2		BLS	1.5 p-d	
Scope voice input functions	wk1, d3	wk1, d3	wk2, d2		JPP	2 p-d	
Scope modes of interaction	wk2, d1		wk2, d3		MLL	1 p-d	
Scope documents diagnosis	wk2, d1		wk2, d2		BLS	1.5 pd	
Scope other WP functions	wk1, d4	wk1, d4	wk2, d3		JPP	2 p-d	
Document OCI	wk2, d1		wk2, d3		MLL	3 p-d	
FTR: Review OCI with customer	wk2, d3		wk2, d3		all	3 p-d	
Revise OCI as required	wk2, d4		wk2, d4		all	3 p-d	
Milestone: OCI defined	wk2, d5		wk2, d5				
I.1.3 Define the Function/behavior							

Effort estimation models: -

Effort estimation models in software engineering are methods used to predict the amount of effort (usually measured in person-hours, person-months, or similar units) needed to complete a project. Accurately estimating effort is critical for project planning, budgeting, resource allocation, and setting realistic deadlines.

How effort is calculated with different estimation models in software engineering:

1. COCOMO (Constructive Cost Model)

Formula: Effort = a × (Size in KLOC)^{b}

where:

- Size is in terms of thousands of delivered lines of code (KLOC).
- a and b are constants that vary depending on the type of project (Organic, Semi-detached, or Embedded).

Organic: Simple, small projects with a well-understood problem domain and minimal innovation, often developed by small teams with good experience (e.g., business data processing).

Semi-Detached: Projects of intermediate complexity, involving mixed teams with various levels of experience and moderate levels of innovation (e.g., operating systems).

Embedded: Complex projects with strict requirements, often involving hardware constraints, or real-time systems that need specialized knowledge (e.g., military or aerospace systems).

Here are the typical values for a and b in the Basic COCOMO model:

Project Type	a	b
Organic	2.4	1.05
Semi-Detached	3.0	1.12
Embedded	3.6	1.20

Example: Let's assume an Organic project with:

- Size (KLOC) = 50
- a = 2.4 and b = 1.05

Solution: Using the formula:

Effort = $2.4 \times (50)^{1.05} = 2.4 \times 57.19 \approx 137.25$

The calculated effort is 137.25 person-months.

2. Expert Judgment Model

In this model, an expert provides an estimate based on their experience.

For example:

- Expert 1: 25 person-months
- Expert 2: 30 person-months
- Expert 3: 28 person-months

The average of these estimates gives a simple effort prediction:

Effort = (25 + 30 + 28)/3 = 27.67 person-months

The estimated effort is 27.67 person-months.

3. Analogy-Based Estimation

For analogy-based estimation, we look at similar past projects:

Past Project A (size: 40 KLOC, effort: 100 person-months)

Past Project B (size: 60 KLOC, effort: 150 person-months)

For a new project with an estimated size of 50 KLOC, we find estimated effort:

Estimated Effort (New Project) = Effort (Project A) +
(Size (New Project) - Size (Project A))/((Size (Project B) - Size (Project A)) ×
(Effort (Project B) - Effort (Project A))

Effort = 100 + (50 − 40)/(60 − 40) × (150 − 100)

= 100 + 10/20 × 50

= 100 + 25 = 125 person-months

So, the estimated effort is 125 person-months.

4. Machine Learning Model (Regression Example)

Let's assume we have a simple linear regression model based on historical data, where:

Effort = 5 + 3 × (Size in KLOC)

For a project of 30 KLOC:

Effort = 5 + 3 × 30 = 5 + 90 = 95 person-months

The estimated effort is 95 person-months.

5. Agile Estimation Technique (Story Points)

In agile, estimation is done in terms of Story Points. Suppose we assign Story Points based on complexity and have a velocity (team capacity) of 20 story points per sprint.

For a project with total 100 story points:

Sprints Needed = Total Story Points/Velocity = 100/20 = 5 sprints

If each sprint is 2 weeks:

Effort = 5×2 = 10 weeks

Converting to person-months, assuming a 4-week month:

Effort = 10/4 = 2.5 person-months

So, the estimated effort is 2.5 person-months.

6. Function Point Analysis (FPA)

FP-Based Estimation –

Decomposition for FP-based estimation focuses on information domain values rather than software functions.

Example of FP-Based Estimation:

You would estimate inputs, outputs, inquiries, files, and external interfaces for the CAD software.

Formula:

$FP_{estimated} = \text{count total} \times [0.65 + 0.01 \times \Sigma(F_i)]$

Count Total:

TABLE 25.1

Estimating information domain values

Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of external inputs	20	24	30	24	4	96 (24 x 4 = 96)
Number of external outputs	12	14	22	14	5	70 (14 x 5 = 70)
Number of external inquiries	16	20	28	20	5	100 (20 x 5 = 100)
Number of internal logical files	4	4	5	4	10	40 (4 x 10 = 40)
Number of external interface files	2	2	3	2	7	14 (2 x 7 = 14)
Count total						*320*

Information Domain values

TABLE 25.2 Estimating information domain values

Complexity Factor	Value
Backup and recovery	4
Data communications	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
Online data entry	4
Input transaction over multiple screens	5
Master files updated online	3
Information domain values complex	5
Internal processing complex	5
Code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5

Question: -

A software project has the following functional components –

External Inputs (EI): 10, all of medium complexity

External Outputs (EO): 8, all of high complexity

External Inquiries (EQ): 6, all of low complexity

Internal Logical Files (ILF): 5, all of high complexity

External Interface Files (EIF): 4, all of medium complexity

The complexity weights for function points are as follows:

Component	Low Complexity	Medium Complexity
External Inputs (EI)	3	4
External Outputs (EO)	4	5
External Inquiries (EQ)	3	4
Internal Logical Files (ILF)	7	10
External Interface Files (EIF)	5	7

Calculate the unadjusted function points (UFP) for the project.

Solution: -

To calculate the UFP, multiply the count of each component by its corresponding weight based on complexity.

External Inputs (EI):

Count: 10

Complexity: Medium

Weight: 4

Contribution = 10 × 4 = 40

External Outputs (EO):

Count: 8

Complexity: High

Weight: 7

Contribution = 8 × 7 = 56

External Inquiries (EQ):

Count: 6

Complexity: Low

Weight: 3

Contribution = 6 × 3 = 18

Internal Logical Files (ILF):

Count: 5

Complexity: High

Weight: 15

Contribution = 5 × 15 = 75

External Interface Files (EIF):

Count: 4

Complexity: Medium

Weight: 7

Contribution = 4 × 7 = 28

Now, add up the contributions:

UFP = 40 + 56 + 18 + 75 + 28 = 217

Answer: The unadjusted function points (UFP) for this project is 217.

Question: -

A software project has been analyzed and the following functional components have been identified:

Count Total (CT): The total count of identified function points is 80.

The project team has evaluated the following factors that can affect productivity, assigning values to each factor (F_i):

F_1 (Communication): 3

F_2 (Complexity of the process): 4

F_3 (Documentation): 2

F_4 (Performance): 1

F_5 (Experience): 5

F_6 (Technology): 3

F_7 (Development Environment): 2

F_8 (Support): 1

F_9 (Reusability): 4

F_{10} (Quality Requirements): 2

Calculate the Function Point (FP) estimated for the project using the formula:

$FP_{\text{estimated}}$ = count total × [0.65 + 0.01 × $\Sigma(F_i)$]

Solution: -

Calculate $\Sigma(F_i)$:

$\Sigma(F_i) = F_1 + F_2 + F_3 + F_4 + F_5 + F_6 + F_7 + F_8 + F_9 + F_{10}$

Substituting the values:

$\Sigma(F_i)$ = 3 + 4 + 2 + 1 + 5 + 3 + 2 + 1 + 4 + 2 = 27

Substitute into the Formula: Now, substitute the count total and Σ(Fi) into the formula:

$FP_{\text{estimated}}$ = 80 × [0.65 + 0.01 × 27]

$FP_{estimated} = 80 \times 0.92 = 73.6$

Question: -

Function Point Analysis calculates the effort based on the number and complexity of functional elements (e.g., inputs, outputs, inquiries). First, we calculate the Unadjusted Function Points (UFP) and then adjust based on environmental factors.

Suppose we have the following counts for different elements, with their respective weights:

Element Type	Count	Weight
External Inputs	20	4
External Outputs	10	5
Internal Logic	15	7

Solution: -

The UFP:

$UFP = (Count_1 \times Weight_1) + (Count_2 \times Weight_2) + (Count_3 \times Weight_3)$

$UFP = (20 \times 4) + (10 \times 5) + (15 \times 7)$

$UFP = 80 + 50 + 105 = 235$

If we apply an Adjustment Factor (CAF) of 1.2 (representing environmental conditions):

Adjusted Function Points = UFP × CAF = 235 × 1.2 = 282

To convert function points to effort, we assume Productivity Rate = 10 function points per person-month:

Effort = Adjusted Function Points/Productivity Rate

= 282/10=28.2 person-months

So, the estimated effort is 28.2 person-months.

CAF – CAF stands for Complexity Adjustment Factor.

The total score across all 14 characteristics forms the Total Degree of Influence (TDI)

$CAF=0.65+(0.01\times TDI)$ or $CAF=0.65+(0.01\times \Sigma(Fi))$

Scrum framework for software development: -

The Scrum framework is one of the most widely adopted Agile methodologies for managing and completing complex projects. It provides a structured approach to iterative development through defined roles, events, and artifacts.

Key Roles:

Product Owner:

Represents the stakeholders and the voice of the customer.

Responsible for defining and prioritizing the product backlog.

Ensures the team is working on the most valuable features.

Scrum Master:

Facilitates the Scrum process and ensures that the team adheres to Scrum practices.

Helps resolve impediments that the team encounters.

Acts as a coach for the team, ensuring that Agile principles are followed.

Development Team:

A cross-functional group of professionals (developers, testers, designers) responsible for delivering potentially shippable increments of the product. Self-organizing and responsible for managing their work.

Key Events:

Sprint: A time-boxed iteration, typically 2-4 weeks, during which a potentially shippable product increment is created. Consists of planning, execution, review, and retrospective phases.

Sprint Planning: A meeting held at the beginning of each sprint to determine what can be delivered in the upcoming sprint and how that work will be achieved.

The Product Owner presents the top items from the product backlog, and the team selects the items they can commit to.

Daily Scrum (Daily Standup): A short, daily meeting (usually 15 minutes) where the development team synchronizes their work and plans for the next 24 hours.

Team members answer three questions: What did I do yesterday? What will I do today? Are there any impediments in my way?

Sprint Review: Held at the end of the sprint to inspect the increment and adapt the product backlog if needed. The development team demonstrates what has been completed during the sprint. Stakeholders provide feedback.

Sprint Retrospective: A meeting held after the sprint review and before the next sprint planning to reflect on the sprint. The team discusses what went well, what didn't, and how processes can be improved.

Key Artifacts:

Product Backlog: An ordered list of everything that is known to be needed in the product. Managed by the Product Owner and continuously refined.

Sprint Backlog: A list of items selected from the product backlog to be completed in the current sprint, along with a plan for delivering the product increment and achieving the sprint goal.

Increment: The sum of all the product backlog items completed during a sprint and all previous sprints. Must be in a usable condition regardless of whether the Product Owner decides to release it.

Backlog Refinement: The Product Owner and development team regularly refine the product backlog to ensure that it is well-defined and prioritized.

Sprint Planning: At the start of each sprint, the team selects items from the product backlog to move to the sprint backlog and defines a sprint goal.

Daily Scrum: The team meets daily to discuss progress, plan the day's work, and identify any impediments.

Sprint Execution: The development team works on the tasks in the sprint backlog to create a potentially shippable product increment.

Sprint Review: At the end of the sprint, the team presents the increment to stakeholders for feedback.

Sprint Retrospective: The team reflects on the sprint and identifies improvements for the next sprint.

Kanban framework for software development: -

The Kanban framework is a visual process management system that helps teams visualize their work, limit work in progress, and maximize efficiency or flow. Unlike Scrum, which has predefined roles and events, Kanban is more flexible and can be applied to various existing workflows without necessitating drastic changes.

Key Principles:

Visualize Work: Use a Kanban board to represent work items and their status. This visualization helps the team understand the flow of work and identify bottlenecks.

Limit Work in Progress (WIP): Restrict the number of work items in progress at any stage of the workflow to ensure focus and efficiency.

Manage Flow: Continuously monitor and manage the flow of work items from start to finish to optimize throughput.

Make Process Policies Explicit: Clearly define and communicate the process policies so that everyone understands how work should be done.

Implement Feedback Loops: Regularly review and adapt processes based on feedback to improve continuously.

Improve Collaboratively, Evolve Experimentally: Use data and collaborative discussion to drive continuous improvement in small, incremental changes.

Key Components:

Kanban Board: The central tool in Kanban, which visually represents the workflow. It typically includes columns such as "To Do," "In Progress," and "Done," but can be customized to fit the team's process.

Cards: Represent individual work items or tasks. Each card includes information about the task, such as a description, assignee, due date, and other relevant details.

WIP Limits: Set maximum limits for the number of work items in each column (e.g., no more than 3 items in "In Progress") to prevent overloading and to focus on finishing tasks.

Visualize Workflow: Map out the entire workflow on the Kanban board, from initial request to delivery. Break down the workflow into distinct stages or columns.

Create and Move Cards: Each work item is represented by a card that moves across the board from left to right as it progresses through different stages.

Set WIP Limits: Establish WIP limits for each column to ensure that the team does not take on too much work at once, maintaining focus and productivity.

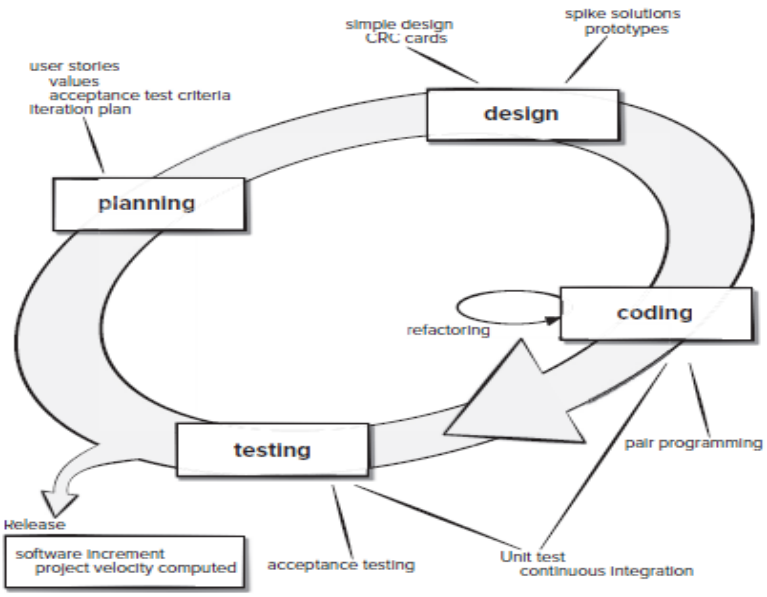
Monitor and Manage Flow: Regularly review the board to identify bottlenecks, track progress, and manage the flow of work. Ensure that work items are moving smoothly from one stage to the next.

Continuous Improvement: Use metrics such as lead time (time from task initiation to completion) and cycle time (time from start of work to completion) to identify areas for improvement. Hold regular meetings to discuss process improvements.

XP (Extreme Programming) Framework: -

Extreme Programming encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing.

FIGURE 3.3
The Extreme Programming process



Planning – The planning activity (also called the planning game) begins with a requirements activity called listening. 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 user 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.

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.

Once a basic commitment (agreement on stories to be included, delivery date, and other project matters) is made for a release, the XP team orders the stories that will be developed in one of three ways:

(1) All stories will be implemented immediately (within a few weeks).

(2) The stories with highest value will be moved up in the schedule and implemented first.

(3) The riskiest stories will be moved up in the schedule and implemented first.

After the first project release (also called a software increment) has been delivered, the XP team computes project velocity. Project velocity is the number of customer stories implemented during the first release.

Project velocity can then be used to help estimate delivery dates and schedule for subsequent releases. The XP team modifies its plans accordingly.

Design – XP design rigorously follows the "keep it simple" principle.

XP encourages the use of CRC cards (class-responsibility collaborator) as an effective mechanism for thinking about the software in an object-oriented context.

CRC cards identify and organize the object-oriented classes9 that are relevant to the current software increment.

CRC cards are the only design work product produced as part of the XP process.

If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. A central notion in XP is that design occurs both before and after coding commences. Refactoring—modifying/optimizing the code in a way that does not change the external behavior of the software.

Coding – After user 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.

A key concept during the coding activity is pair programming. XP recommends that two people work together at one computer to create code for a story. This provides a mechanism for real-time problem solving and real-time quality assurance.

Testing – The unit tests that are created should be implemented using a framework that enables them to be automated. This encourages implementing a regression testing strategy. XP acceptance tests, also called customer tests, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. They are derived from user stories that have been implemented as part of a software release.

Risk management and mitigation strategies in software projects: -

Risk management in software projects involves systematically identifying, analyzing, and addressing potential risks that could negatively impact the project's objectives, such as timelines, budget, quality, or scope.

Key Steps in Risk Management:

Risk Identification – Identify potential risks, such as technical difficulties, resource limitations, stakeholder changes, or dependency issues.
Tools like brainstorming sessions, checklists, and expert interviews can be used for this purpose.

Risk Analysis – Assess the likelihood and impact of each identified risk.
Use techniques like qualitative analysis (categorizing risks as high, medium, or low) or quantitative analysis (assigning numerical values to the probability and impact).

Risk Prioritization – Prioritize risks based on their severity to focus on the most critical ones first.
Create a risk matrix to visually represent which risks need immediate attention.

Mitigation Strategies:

Risk Avoidance – Modify project plans to eliminate the risk, such as changing the design or scope to bypass a known problem area.

Risk Reduction – Take proactive steps to minimize the likelihood or impact of the risk, like conducting extensive testing, using proven technologies, or adding redundancies.

Risk Transfer – Shift the risk to a third party, for example, by outsourcing a risky component, purchasing insurance, or using contractual agreements.

Risk Acceptance – Acknowledge the risk and decide to proceed without immediate action, but prepare a contingency plan to address it if it occurs.

Contingency Planning – Develop backup plans to handle risks that materialize, such as having additional resources on standby or alternative solutions for critical tasks.

Example Strategies in Action:

Technical Risk: If there is a risk that a new technology might not work as expected, a mitigation strategy could be to build a prototype early in the project.

Resource Risk: If a key team member might become unavailable, a strategy could include cross-training other team members.

Software documentation and knowledge management practices: -

Software documentation and knowledge management practices are essential for maintaining clarity, ensuring efficient team collaboration, and supporting future maintenance and scalability of software projects.

Software Documentation:

Documentation provides written records about various aspects of a software project.

It can be categorized as:

1. Requirements Documentation – Describes what the software should do, including functional and non-functional requirements.
Example: User stories, use cases, and detailed requirement specifications.

2. Design Documentation – Outlines the architecture, system design, and data models.
Includes diagrams like UML, flowcharts, and database schemas to explain system components and their interactions.

3. Technical Documentation – Describes how to build, deploy, and maintain the software.
Includes API documentation, code comments, setup instructions, and infrastructure details.

4. User Documentation – Guides end-users on how to use the software.
Includes manuals, help files, tutorials etc.

5. Maintenance Documentation – Provides information for future developers or teams to understand and modify the software.
Includes known issues, troubleshooting guides, and change logs.

Knowledge Management Practices:

Knowledge management ensures that information is captured, organized, and made accessible to all team members.

Effective practices include:

1. Centralized Repositories – Use version-controlled systems (e.g., GitHub) to store all documentation.

2. Knowledge Sharing Sessions – Conduct regular meetings like code reviews, technical talks, or sessions to share expertise and lessons learned.

3. Document Updates and Versioning – Keep documentation up-to-date with software changes.
Use version control for documentation to track changes and maintain historical records.

4. Code Commenting and Standards – Follow best practices for inline code comments and establish coding standards to make the codebase understandable.

5. Onboarding and Training – Develop comprehensive onboarding materials to help new team members get up to speed quickly.
Use knowledge base articles and interactive training sessions.

6. Feedback and Continuous Improvement – Gather feedback from team members on the usefulness of documentation.
Continuously improve documentation based on this feedback to ensure it meets the needs of the team.

Benefits:

Efficiency – Well-documented projects save time by making it easier to onboard new developers and maintain the software.

Collaboration – Knowledge is shared across the team, reducing dependency on individual members.

Future-Proofing – Proper documentation ensures that future changes can be made efficiently, even if the original developers are no longer available.