# Revision Purpose SE CT:03

| | |
|---|---|
| Created by | Ⓑ Borhan |
| Last edited time | @May 13, 2025 7:50 PM |
| Tag | |

## Lecture 10: Risk Management in Software Projects

### Introduction

Risk management is a critical component of software project planning, aimed at identifying, analyzing, and controlling risks to ensure project success. A **risk** is an uncertain future event with a probability of occurrence and potential for loss, impacting time, budget, performance, or project outcomes.

### Sources of Risk

Risks in software projects arise from various sources, including:

- **Misunderstanding Customer Requirements**: Incorrect or unclear interpretation of client needs.
- **Uncontrolled Requirement Changes**: Frequent or poorly managed changes to requirements.
- **Unrealistic Promises**: Overcommitting to clients beyond project capabilities.
- **Misjudging New Methodologies**: Overestimating the benefits or underestimating the challenges of new tools or processes.
- **Poor Software Design**: Underestimating the robustness or extensibility of the design.
- **Team Effectiveness Miscalculation**: Overestimating team collaboration or productivity.
- **Inaccurate Budget Estimation**: Underestimating costs or resources needed.

### Risk Identification

Risk identification involves detecting potential risks early to minimize their impact. Techniques include:

- **Brainstorming**: Collaborative sessions to identify potential risks.
- **SWOT Analysis**: Evaluating strengths, weaknesses, opportunities, and threats.
- **Causal Mapping**: Mapping cause-and-effect relationships to uncover risks.
- **Flowcharting**: Visualizing processes to identify risk points.

#### Types of Risks

1. **Technology Risks**: Issues with hardware or software used in development.
2. **People Risks**: Challenges related to team members, such as turnover or skill gaps.
3. **Organizational Risks**: Problems stemming from the organizational environment.
4. **Tools Risks**: Issues with development tools or support software.
5. **Requirement Risks**: Risks from changing or mismanaged customer requirements.
6. **Estimation Risks**: Errors in estimating resources, time, or costs.

### Risk Analysis and Prioritization

Risk analysis assesses identified risks by:

1. **Identifying Problems**: Determining the root causes of risks.
2. **Estimating Probability**: Calculating the likelihood of occurrence.
3. **Assessing Impact**: Evaluating the potential effect on the project.

### Probability Categories

- **Very Low (0–10%)**: Tolerable risk with no significant harm.
- **Low (10–25%)**: Minor effect on the project.
- **Moderate (25–50%)**: Impacts project timeline.
- **High (50–75%)**: Affects timeline and budget.
- **Very High (>75%)**: Severe impact on output, time, budget, and performance.

## Risk Control

Risk control involves planning, monitoring, and resolving risks to achieve desired project outcomes.

### 1. Risk Planning

Risk planning develops strategies to mitigate significant risks. Methods include:

- **Avoid the Risk**: Modify requirements, reduce scope, or offer incentives to retain staff.
- **Transfer the Risk**: Outsource risky components or purchase insurance.
- **Reduce the Risk**: Plan for potential losses, such as recruiting additional staff to cover turnover.

### 2. Risk Monitoring

Risk monitoring is an ongoing process to track project progress and evaluate risks:

- Continuously assess assumptions about risks.
- Identify changes in risk probability or impact.
- Take corrective actions as needed to keep risks under control.

### 3. Risk Resolution

Risk resolution ensures risks are managed within acceptable levels:

- Depends on accurate risk identification, analysis, and planning.
- Requires prompt and effective responses to emerging issues.
- Keeps the project on track by addressing risks as they arise.

## RMMM Plan

The **Risk Mitigation, Monitoring, and Management (RMMM)** Plan is a structured approach integrated into the overall project plan. It documents risks using a **Risk Information Sheet (RIS)**, which includes:

- Risk ID, date, probability, impact, description, avoidance strategies, monitoring actions, management plan, and current status.
- Managed via a database for easy creation, prioritization, searching, and analysis.

### Example: Late Project Delivery

**Risk**: Project delivery exceeds the deadline.

1. **Mitigation**:
   - Estimate development time as 20 days but quote 30 days to the client for buffer.
   - Implement precautionary measures before development starts.
2. **Monitoring**:
   - Create a project schedule with clear start and end dates.
   - Track progress within the 20–30-day window.
3. **Management**:
   - If the deadline is missed, negotiate with the client for extra time or offer additional features to maintain satisfaction.

## Risk Mitigation

Risk mitigation is a proactive approach to avoid risks before they occur. Steps include:

1. Communicate with staff to identify potential risks.
2. Eliminate causes of risks (e.g., unclear requirements).
3. Develop policies to ensure project continuity.
4. Regularly review and control project documents.
5. Conduct timely reviews to accelerate progress.

## Risk Management

Risk management is a reactive approach applied after risks materialize:

- Assumes mitigation efforts failed, and the risk has occurred.
- Handled by the project manager to resolve issues.
- Effective mitigation simplifies management (e.g., sufficient staff, clear documentation, and shared knowledge ease onboarding of new team members).

### Example Scenario

**Risk**: High staff turnover.

- **Mitigation Success**: Sufficient additional staff, comprehensive documentation, and shared knowledge ensure new employees can quickly adapt.
- **Management**: Project manager leverages these resources to maintain development continuity.

## Drawbacks of RMMM

While effective, the RMMM approach has limitations:

- **Additional Costs**: Implementing RMMM increases project expenses.
- **Time-Intensive**: Requires significant time for planning and execution.
- **Complex for Large Projects**: RMMM can become a project in itself.
- **No Guarantee**: Risks may still emerge post-delivery, and RMMM does not ensure a risk-free project.

## Conclusion

Effective risk management in software projects involves identifying, analyzing, and controlling risks through proactive mitigation and reactive management. The RMMM Plan provides a structured framework to document and address risks, but it requires careful planning and resources. By understanding risk sources, types, and control strategies, project managers can enhance the likelihood of successful project outcomes.

# Lecture 11 : Design Concepts

## Introduction

Software design is the process of transforming user requirements into a form suitable for coding and implementation. As the first step in the Software Development Life Cycle (SDLC), it shifts focus from the problem domain to the solution domain, specifying how to fulfill requirements outlined in the Software Requirements Specification (SRS). Software design is typically performed by software design engineers or UI/UX designers.

### Mitch Kapor’s Software Design Manifesto

Mitch Kapor, creator of Lotus 1-2-3, emphasized three qualities of good software design:

- **Firmness**: The software should be free of bugs that impair functionality.

- **Commodity**: The software should meet its intended purpose.
- **Delight**: The user experience should be pleasurable.

## Objectives of Software Design

A well-designed software system should achieve:

- **Correctness**: Accurately meet the specified requirements.
- **Completeness**: Include all necessary components, such as data structures, modules, and interfaces.
- **Efficiency**: Optimize resource usage.
- **Flexibility**: Adapt to changing needs.
- **Consistency**: Maintain uniformity across the design.
- **Maintainability**: Be simple enough for other designers to maintain.

## Software Quality Attributes (FURPS)

The FURPS model defines key quality attributes for software design:

- **Functionality**: Evaluates the feature set, capabilities, generality of functions, and system security.
- **Usability**: Considers human factors, aesthetics, consistency, and documentation.
- **Reliability**: Measures failure frequency, output accuracy, mean-time-to-failure (MTTF), recovery ability, and predictability.
- **Performance**: Assesses processing speed, response time, resource consumption, throughput, and efficiency.
- **Supportability**: Encompasses extensibility, adaptability, serviceability, testability, compatibility, configurability, ease of installation, and problem localization (collectively contributing to maintainability).

## Software Quality Guidelines

To ensure high-quality design, the following guidelines should be followed:

- Use recognizable **architectural styles or patterns** and components with good design characteristics.
- Enable **evolutionary implementation**, allowing incremental development (though smaller systems may use linear design).
- Ensure **modularity** by logically partitioning software into elements or subsystems.
- Provide distinct representations of **data, architecture, interfaces, and components**.
- Select **data structures** based on recognizable patterns suitable for implementation.
- Design **components** with independent functional characteristics.
- Create **interfaces** that reduce complexity between components and external systems.
- Derive the design using a **repeatable method** driven by requirements analysis.
- Represent the design with **notation** that clearly communicates its meaning.

## Software Design Process

The software design process translates the analysis model (requirements) into a design model (solution) and consists of three levels:

1. **Interface Design**:
    - Focuses on interactions between the system and users/devices.
    - Uses scenario-based and behavioral diagrams (e.g., use case diagrams).
    - Does not address internal structure.
2. **Architectural Design**:
    - Defines major system components, their responsibilities, properties, interfaces, and interactions.

- Uses class-based and flow-based diagrams (e.g., class diagrams, data flow diagrams).

3. **Detailed Design**:
    - Specifies internal elements of major components, including properties, relationships, processing, algorithms, and data structures.

The design must:

- Implement all explicit and implicit requirements from the analysis model.
- Be a readable, understandable guide for coders, testers, and support teams.
- Provide a complete picture of the software, addressing data, functional, and behavioral domains from an implementation perspective.

## Fundamental Design Concepts

Software design concepts provide the principles and logic behind creating effective software designs. These concepts form a supporting structure for development.

### 1. Abstraction

Abstraction hides unnecessary implementation details, showing only essential information to users.

- **Procedural Abstraction**: Divides subprograms into hidden and visible groups of functionalities (e.g., a function like open() hides the details of the enter algorithm).
- **Data Abstraction**: Represents data objects while hiding manipulation details (e.g., a stack's Push(), Pop(), Top(), and Empty() methods hide internal data structure implementation).
- **Example**: A door object might expose attributes like manufacturer, model, type, and swing direction while hiding internal data structures.

### 2. Architecture

Architecture defines the overall structure of program modules and their interactions, providing conceptual integrity.

- **Structural Properties**: Defines components (e.g., modules, objects) and their interactions (e.g., method invocations).
- **Extra-Functional Properties**: Addresses performance, capacity, reliability, security, and adaptability requirements.
- **Families of Systems**: Leverages reusable architectural patterns for similar systems.
- **Reference**: Shaw and Garlan [SHA95a].

### 3. Design Patterns

Design patterns are reusable solutions to common design problems.

- **Pattern Template**:
    - **Name**: A concise, expressive name.
    - **Intent**: Describes the pattern's purpose.
    - **Also-Known-As**: Lists synonyms.
    - **Motivation**: Provides a problem example.
    - **Applicability**: Specifies relevant design situations.
    - **Structure**: Describes required classes.
    - **Participants**: Outlines class responsibilities.
    - **Collaborations**: Details participant interactions.
    - **Consequences**: Discusses trade-offs and design forces.
    - **Related Patterns**: References related patterns.

### 4. Separation of Concerns

Separation of concerns divides complex problems into smaller, independently solvable pieces.

- Each concern represents a feature or behavior from the requirements model.
- Reduces effort and time by making problems more manageable.

### 5. Modularity

Modularity divides a system into smaller, manageable parts (modules) to reduce complexity.

- Modules are integrated to meet software requirements.
- **Benefits**: Makes software intellectually manageable, reduces development costs, and improves understanding [Mye78].
- **Contrast**: Monolithic software (single module) is difficult to understand due to numerous control paths, variables, and complexity.

### 6. Information Hiding

Information hiding ensures modules only share necessary information, hiding internal data structures and algorithms.

- **Benefits**:
  - Reduces side effects.
  - Limits the global impact of local design decisions.
  - Encourages controlled interfaces.
  - Discourages global data usage.
  - Promotes encapsulation, enhancing design quality.

### 7. Functional Independence

Functional independence is achieved through modules with single-minded functions and minimal interaction (low coupling, high cohesion).

- **Cohesion**: The degree to which a module performs a single task with little interaction with others (high cohesion is desirable).
- **Coupling**: The degree of interdependence between modules (low coupling is desirable).
- **Benefits of High Cohesion and Low Coupling**:
  - **Readability**: Modules are easier to understand.
  - **Maintainability**: Changes in one module have minimal impact on others.
  - **Modularity**: Simplifies module development.
  - **Scalability**: Facilitates adding or removing modules.
  - **Testability**: Simplifies testing and debugging.
  - **Reusability**: Modules can be reused in other systems.
  - **Reliability**: Improves overall system quality.

### 8. Refinement

Refinement is a top-down approach that elaborates procedural details hierarchically until programming language statements are reached.

- **Example**: The task “open door” is refined into steps like “walk to door,” “reach for knob,” “turn knob clockwise,” and so on, until all details are specified.

### 9. Aspects

Aspects represent cross-cutting concerns—requirements that affect multiple parts of the design.

- **Definition**: Requirement A cross-cuts requirement B if B cannot be satisfied without considering A [Ros04].
- **Example**: In the SafeHomeAssured.com WebApp:
  - Requirement A: Access camera surveillance via the internet.
  - Requirement B: Validate registered users before access.

- B cross-cuts A because user validation (B) must be implemented across all functions, including camera access (A).

### 10. Refactoring

Refactoring improves a software system's internal structure without altering its external behavior [FOW99].

- **Purpose**: Eliminates redundancy, unused elements, inefficient algorithms, or inappropriate data structures to enhance design quality.

## Object-Oriented (OO) Design Concepts

OO design leverages principles to create flexible, reusable designs:

- **Design Classes**:
  - **Entity Classes**: Refined from analysis classes to represent data and behavior.
  - **Boundary Classes**: Manage user interfaces (e.g., screens, reports) and represent entity objects to users.
  - **Controller Classes**: Handle creation/update of entity objects, instantiation of boundary objects, complex communication, and data validation.
- **Inheritance**: Subclasses inherit all responsibilities of their superclass.
- **Messages**: Stimulate behavior in receiving objects.
- **Polymorphism**: Allows objects to be treated as instances of their parent class, reducing effort to extend designs.

## Design Model Elements

The design model transforms the analysis model into a detailed implementation plan, encompassing:

1. **Data Elements**:
   - Data structures derived from the data model.
   - Database architecture for persistent storage.
2. **Architectural Elements**:
   - Derived from the application domain, analysis classes, and architectural patterns/styles [Sha96].
   - Define relationships, collaborations, and behaviors for design realizations.
3. **Interface Elements**:
   - User interfaces (UI).
   - External interfaces to other systems, devices, or networks.
   - Internal interfaces between design components.
4. **Component Elements**:
   - Detailed specifications of software components, including algorithms and processing logic.
5. **Deployment Elements**:
   - Define how software components are deployed across hardware or network environments.

## Conclusion

Software design bridges user requirements and implementation, ensuring correctness, efficiency, and maintainability. By adhering to quality guidelines, leveraging fundamental concepts (e.g., abstraction, modularity, functional independence), and applying OO principles, designers create robust, scalable systems. The design process—interface, architectural, and detailed design—transforms analysis models into actionable plans, supported by a comprehensive design model.

## Reference

Pressman, R. S. (2010). *Software Engineering: A Practitioner's Approach, 7th Edition*. McGraw-Hill.

# Lecture 11: Software Project Management and Scheduling

## Introduction

Software project management involves planning, organizing, and controlling resources to deliver a software product that meets customer requirements within time and budget constraints. Effective project management focuses on scheduling, cost estimation, and risk management to ensure project success. This guide covers key concepts, techniques, and models for managing and scheduling software projects, optimized for exam preparation.

## Project Management Spectrum

Effective software project management revolves around four key components, known as the **Four P's**:

### 1. People

- **Importance**: Human resources are the most critical factor in project success.
- **Selection**: Choose individuals with the right skills and talents.
- **Roles and Responsibilities**:
  - **Senior Manager**: Defines business issues and influences the project.
  - **Project Manager**: Plans, motivates, organizes, and controls project activities; possesses problem-solving and team management skills.
  - **Software Engineer**: Delivers technical expertise.
  - **Customer**: Specifies requirements.
  - **End Users**: Interact with the final software product.

### 2. Product

- **Definition**: The software product is the ultimate deliverable of the project.
- **Planning Requirements**:
  - Establish objectives and scope.
  - Identify alternative solutions.
  - Define technical and management constraints.
- **Importance**: Accurate product definition enables realistic cost estimation, risk identification, and scheduling.

### 3. Process

- **Definition**: A methodology that outlines steps to complete the project as per requirements.
- **Importance**: A clear process ensures team members know their tasks and timelines, increasing the likelihood of meeting project goals.
- **Phases**:
  - Documentation
  - Designing
  - Implementation
  - Software Configuration Management
  - Deployment
  - Interaction

### 4. Project

- **Definition**: Encompasses requirement analysis, development, delivery, maintenance, and updates.
- **Project Manager's Role**:

- Guides the team to achieve objectives.
- Resolves issues, monitors costs, and ensures adherence to deadlines.
- Manages activities to prevent project failure.

## Boehm's W5HH Principle

Barry Boehm's W5HH (Why, What, When, Who, Where, How, How Much) principle provides a framework for efficient project management by answering key questions:

1. **Why is the system being developed?**
   - Identifies business reasons and problem statements to justify the project's cost and time.
2. **What activities are needed?**
   - Defines key tasks required by the customer to establish a project schedule.
3. **When will it be completed?**
   - Specifies start and end dates for tasks to meet project goals.
4. **Who is responsible for each activity?**
   - Assigns roles and responsibilities to team members.
5. **Where are they organizationally located?**
   - Clarifies that responsibilities may extend beyond the software team to customers, users, and stakeholders.
6. **How will the job be done technically and managerially?**
   - Defines technical and management strategies once the product scope is established.
7. **How much of each resource is needed?**
   - Estimates resources required to complete the project within budget and requirements.

## Software Measurements and Metrics

Software measurements and metrics quantify attributes of the software product or process to aid decision-making and project success.

### Software Measurements

- **Definition**: Indicators of size, quantity, or dimension of a product or process attribute.
- **Categories**:
  - **Direct Measures**: Cost, effort, lines of code (LOC), execution speed, error count.
  - **Indirect Measures**: Functionality, quality, complexity, reliability, maintainability.

### Software Metrics

- **Definition**: Formulas or measures for software process and product aspects (e.g., performance, productivity).
- **Categories**:
  - **Product Metrics**: Measure size, complexity, quality, and reliability.
  - **Process Metrics**: Measure fault rates, testing defect patterns, and operation times.
  - **Project Metrics**: Measure developer count, cost, scheduling, and productivity.

### Principles of Software Measurement

1. **Formulation**: Derive appropriate measures and metrics.
2. **Collection**: Gather data to compute metrics.
3. **Analysis**: Apply mathematical tools to compute metrics.
4. **Interpretation**: Evaluate metrics to gain insights into quality.
5. **Feedback**: Provide recommendations to the software team based on metric analysis.

## Size Metrics

Size metrics estimate the software's size, critical for cost and effort estimation.

### 1. Lines of Code (LOC)

- **Definition**: Counts executable code lines, excluding comments and blank lines.
- **Use**: Estimates program size and compares programmer productivity.
- **Example**:

```
//Import header file
#include <iostream>
int main() {
    int num = 10;
    //Logic of even number
    if (num % 2 == 0) {
        cout << "It is even number";
    }
    return 0;
}
```

  - Total LOC = 9 (executable lines only).
- **Advantages**:
  - Widely used for cost estimation.
  - Easy to estimate efforts.
- **Disadvantages**:
  - Cannot measure specification size.
  - Poor design may inflate LOC.
  - Language-dependent.
  - Difficult for users to understand.
- **Example Table**:

| Project | LOC | Cost (SR) | Efforts (Persons/Month) | Documents | Errors |
|---|---|---|---|---|---|
| A | 10,000 | 110 | 18 | 365 | 39 |
| B | 12,000 | 115 | 20 | 370 | 45 |
| C | 15,400 | 130 | 25 | 400 | 32 |

### 2. Function Points (FP)

- **Definition**: Measures functionality by counting functions in the application.
- **Attributes**:
  - External Inputs (EI): Input screens, tables.
  - External Outputs (EO): Output screens, reports.
  - External Inquiries (EQ): Prompts, interrupts.
  - Internal Logical Files (ILF): Databases, directories.
  - External Interface Files (EIF): Shared databases, routines.
- **Calculation**:
  - Count each attribute, assign weights, and compute Unadjusted Function Count (UFC).
  - Apply Complexity Adjustment Factor (CAF) to get FP.
- **Example**:

| Information Domain | Optimistic | Likely | Pessimistic | Est. Count | Weight | FP Count |
|---|---|---|---|---|---|---|
| # of Inputs | 22 | 26 | 30 | 26 | 4 | 104 |
| # of Outputs | 16 | 18 | 20 | 18 | 5 | 90 |
| # of Inquiries | 16 | 21 | 26 | 21 | 4 | 84 |
| # of Files | 4 | 5 | 6 | 5 | 10 | 50 |
| # of External Interfaces | 1 | 2 | 3 | 2 | 7 | 14 |
| **UFC** | | | | | | 342 |
| **CAF** | | | | | | 1.17 |
| **FP** | | | | | | 400 |

- **Advantages**:
  - Requires detailed specifications.
  - Not restricted to code.
  - Language-independent.
- **Disadvantages**:
  - Ignores quality issues.
  - Subjective counting relies on estimation.

## Software Project Estimation

Software project estimation predicts the time, effort, and cost required to complete a project, critical for preventing project failure.

### Responsible Persons

- Software Manager
- Cognizant Engineers
- Software Estimators

### Factors Affecting Estimation

1. **Cost**: Ensure sufficient funds to avoid project failure.
2. **Time**: Estimate overall duration and task timings to manage client expectations.
3. **Size and Scope**: Identify all tasks to ensure adequate materials and expertise.
4. **Risk**: Predict potential events and their severity to create risk management plans.
5. **Resources**: Ensure availability of tools, people, hardware, and software.

### Steps of Project Estimation

1. **Estimate Project Size**:
   - Use LOC or FP, based on customer requirements, SRS, and system design documents.
   - Methods: Estimation by analogy (past projects) or dividing the system into subsystems.
2. **Estimate Efforts**:
   - Calculate person-hours or person-months for activities (design, coding, testing, documentation).
   - Methods:
     - Use historical organizational data for similar projects.
     - Apply algorithmic models (e.g., COCOMO) for unique projects.
3. **Estimate Project Schedule**:
   - Define the Work Breakdown Structure (WBS) to assign tasks, start/end dates, and team members.

- Convert efforts to calendar months using:**Schedule (months) = 3.0 * (man-months)^(1/3)**(The constant 3.0 varies by organization).

4. **Estimate Project Cost**:
   - Include labor costs (effort hours * labor rate) and other expenses (hardware, software, travel, training, office space).
   - Use specific labor rates for different roles for accuracy.

## Decomposition Techniques

Decomposition techniques break down the project into manageable parts for accurate estimation.

### 1. Software Sizing

- **Challenge**: Accurately estimate the product size.
- **Factors for Accuracy**:
  - Correct size estimation.
  - Translation of size into effort, time, and cost.
  - Reflection of team abilities in the plan.
  - Stability of requirements and environment.
- **Approaches**:
  - **Fuzzy Logic Sizing**: Uses application type and historical data.
  - **Function Point Sizing**: Estimates based on information domain characteristics.
  - **Standard Component Sizing**: Estimates size of subsystems, modules, or screens using historical data.
  - **Change Sizing**: Estimates modifications to existing software (reuse, add, change, delete code).

### 2. Problem-Based Estimation

- **Method**: Uses LOC or FP to size software elements and project costs/efforts.
- **Steps**:
  - Size each element using LOC or FP.
  - Use historical data to project costs and efforts.
  - Compute expected value:**S = (S_opt + 4S_m + S_pess) / 6**(S = size, S_opt = optimistic, S_m = most likely, S_pess = pessimistic).
- **Example**:

| Function | Estimated LOC |
|---|---|
| User Interface (UICF) | 2,300 |
| 2D Geometric Analysis | 5,300 |
| 3D Geometric Analysis | 6,800 |
| Database Management | 3,350 |
| Graphics Display | 4,950 |
| Peripheral Control | 2,100 |
| Design Analysis | 8,400 |
| **Total** | **33,200** |

### 3. Process-Based Estimation

- **Method**: Decomposes the process into tasks and estimates effort for each.
- **Steps**:
  - Identify software functions and process activities.
  - Estimate effort (person-months) for each activity per function.

    - Apply labor rates to calculate costs (senior staff may have higher rates).
- **Importance**: Ensures detailed task-level estimation for accuracy.

## Software Cost Estimation

Software cost estimation predicts the approximate cost before development begins, considering multiple factors.

### Factors for Project Budgeting

- Specification and scope
- Location
- Duration
- Team efforts
- Resources

### Tools and Techniques

1. **Expert Judgment**:
    - Leverages experienced professionals' insights for similar projects.
2. **Analogous Estimation**:
    - Uses historical data from similar projects (scope, budget, size).
    - Cost-effective but less accurate, used in early phases.
3. **Parametric Estimation**:
    - Uses statistical models to estimate man-hours based on past project data.
4. **Bottom-Up Estimation**:
    - Divides project into work packages (WBS), estimates each, and sums for total cost.
    - Time-consuming but highly accurate.
5. **Three-Point Estimation (PERT)**:
    - Uses optimistic, most likely, and pessimistic estimates to handle uncertainties.
6. **Reserve Analysis**:
    - Allocates reserve budget for unforeseen events, approved by sponsors.
7. **Cost of Quality**:
    - Includes costs to prevent and address failures during and after the project.
8. **Vendor Bid Analysis**:
    - Compares multiple vendor bids to estimate project cost.

### Typical Problems

1. **Complexity**: Large projects are hard to estimate accurately, especially early on.
2. **Inexperience**: Estimators may lack sufficient experience.
3. **Bias**: Tendency to underestimate, especially by senior professionals.
4. **Oversights**: Forgetting integration and testing costs in large projects.
5. **Management Pressure**: Demanding precise estimates for bids or funding.

## COCOMO Model

The **Constructive Cost Model (COCOMO)**, developed by Barry Boehm in 1981, estimates effort, cost, and schedule based on software size (KLOC).

### Software Project Types

1. **Organic**:
   - Small, simple projects (2–50 KLOC).
   - Small, experienced team; well-understood problem.
   - Examples: Simple inventory or data processing systems.
2. **Semidetached**:
   - Medium-sized projects (50–300 KLOC) with mixed requirements.
   - Mixed team experience; some known/unknown modules.
   - Examples: Database management, complex inventory systems.
3. **Embedded**:
   - Large, complex projects (>300 KLOC) with fixed requirements.
   - Large team, often less experienced; high complexity.
   - Examples: ATMs, air traffic control, banking software.

## Types of COCOMO Models

1. **Basic COCOMO**:
   - Static model for quick, rough estimates based on KLOC.
   - **Formulas**:
     - Effort (E) = a * (KLOC)^b Man-Months (MM)
     - Scheduled Time (D) = c * (E)^d Months
   - **Constants**:

| Project Type | a | b | c | d |
|---|---|---|---|---|
| Organic | 2.4 | 1.05 | 2.5 | 0.38 |
| Semidetached | 3.0 | 1.12 | 2.5 | 0.35 |
| Embedded | 3.6 | 1.20 | 2.5 | 0.32 |

   - **Example** (Semidetached, 300 KLOC):
     - E = 3.0 * (300)^1.12 = 1784.42 MM
     - D = 2.5 * (1784.42)^0.35 = 34.35 Months
     - Persons = E / D = 1784.42 / 34.35 ≈ 52
2. **Intermediate COCOMO**:
   - Enhances accuracy by including cost drivers (product, hardware, resource, project parameters).
   - **Formulas**:
     - Effort (E) = a * (KLOC)^b * EAF MM
     - Scheduled Time (D) = c * (E)^d Months
   - **Effort Adjustment Factor (EAF)**: Product of cost driver values (ideal = 1).
   - **Example** (Semidetached, 300 KLOC, very high application experience = 0.82, very low programming experience = 1.14):
     - EAF = 0.82 * 1.14 = 0.9348
     - E = 3.0 * (300)^1.12 * 0.9348 = 1668.07 MM
     - D = 2.5 * (1668.07)^0.35 = 33.55 Months
3. **Detailed COCOMO**:
   - Applies Basic/Intermediate COCOMO to each software engineering phase (planning, system design, detailed design, coding/testing, integration, cost model).
   - Divides software into modules, estimates effort per module, and sums for total effort.

- **Example** (Distributed MIS System):
  - Database (Semidetached)
  - GUI (Organic)
  - Communication (Embedded)
  - Estimate costs separately and sum for total cost.

### Advantages of COCOMO

- Systematic estimation at different development stages.
- Identifies key cost and effort factors.
- Leverages historical project data.
- Easy to implement with various factors.

### Disadvantages of COCOMO

- Ignores requirements, customer skills, and hardware issues.
- Limits accuracy due to assumptions and averages.
- Heavily time-dependent.
- Assumes size (KLOC) is the primary cost driver, which may not always apply.

## Conclusion

Software project management and scheduling require careful planning of people, product, process, and project activities. Boehm's W5HH principle guides objective setting, while measurements and metrics (LOC, FP) quantify progress. Estimation techniques (decomposition, COCOMO) predict time, effort, and cost, addressing risks and resource needs. Understanding these concepts ensures effective project execution and is critical for exam success.

# Lecture 12 : Software Testing

## Introduction

Software testing is a critical process in software development to ensure a product meets customer requirements, is defect-free, and performs reliably. It identifies errors, gaps, or missing requirements by evaluating attributes like reliability, scalability, portability, reusability, and usability. This guide covers software testing principles, test cases, white box and black box testing, unit testing, integration testing, and comparisons, optimized for exam preparation.

## Software Testing Overview

- **Definition**: A method to verify that a software product matches expected requirements and is free of defects.
- **Purpose**:
  - Identify errors, gaps, or missing requirements.
  - Ensure the product meets customer needs and performs reliably.
  - Prevent failures that could lead to dangerous situations.
- **Importance**: Mandatory to avoid deploying faulty software to end users.
- **Process**:
  - Conducted at every phase of the Software Development Life Cycle (SDLC).
  - Performed by software testers, developers, project managers, and end users.

## Principles of Software Testing

The following principles guide effective software testing:

1. **Testing Shows the Presence of Defects**:
   - Aims to identify defects that could cause product failure.
   - Cannot guarantee 100% error-free software but reduces undiscovered defects.
   - Requires well-designed test cases to maximize defect detection.
2. **Exhaustive Testing is Impossible**:
   - Testing all modules and features exhaustively is impractical.
   - Use risk analysis and prioritize critical modules to deliver on schedule.
3. **Early Testing**:
   - Involve testers from the requirement gathering phase to understand the product deeply.
   - Early defect detection saves time and cost compared to late-stage fixes.
4. **Defect Clustering**:
   - Most defects (80%) are found in a small portion (20%) of code (Pareto's 80-20 Rule).
   - Focusing on high-defect areas may miss bugs in other modules.
5. **Pesticide Paradox**:
   - Repeated use of the same test cases fails to uncover new defects.
   - Regularly update test cases to cover different software parts and find new bugs.
6. **Testing is Context-Dependent**:
   - Testing varies by project type (e.g., e-commerce, banking, commercial websites).
   - Different products require tailored test cases based on their features and requirements.
7. **Absence of Errors Fallacy**:
   - A bug-free application may still be unusable if it fails to meet user needs.
   - Testing must ensure both defect-free code and alignment with client requirements.

## Test Cases

- **Definition**: A set of actions or conditions to compare expected and actual results, verifying software functionality against customer requirements.
- **Components**:
  - **Test Scenario ID**: Identifies the test scenario (e.g., Login-1).
  - **Test Case ID**: Unique identifier for the test case (e.g., Login-1A).
  - **Description**: Details the test purpose (e.g., positive login test).
  - **Priority**: Importance level (e.g., High).
  - **Pre-Requisite**: Conditions required (e.g., valid user account).
  - **Post-Requisite**: Actions after testing (e.g., none).
  - **Execution Steps**: Actions, inputs, expected outputs, actual outputs, browser, and result.
- **Example** (Login Test Case):

| S.No | Action | Inputs | Expected Output | Actual Output | Browser | Result |
|---|---|---|---|---|---|---|
| 1 | Launch application | https://www.facebook.com/ | Facebook home | Facebook home | IE-11 | Pass |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | Enter email & password, hit login | Email: test@xyz.com, Password | Login success | Login success | IE-11 | Pass |

## White Box Testing

- **Definition**: Testing that analyzes the internal code structure, design, and functionality, requiring knowledge of the software's programming.
- **Also Known As**: Clear box, open box, transparent box, code-based, or glass box testing.
- **Performed By**: Software developers.
- **Testing Levels**: Unit testing and integration testing.
- **Tools**: EclEmma, NUnit, PyUnit, HtmlUnit, CppUnit.
- **Verification Areas**:
  - Internal code security.
  - Poorly structured code paths.
  - Input flow through code.
  - Loops, decision conditions, and statements.
  - Individual functions and modules.
  - Expected outputs.

### Techniques

1. **Path Coverage/Testing**:
   - Tests all possible paths from entry to exit based on the program's control flow.
   - Example: Functions A → B → C → D → E → F → H → I.
2. **Loop Testing**:
   - Verifies simple, nested, and concatenated loops (e.g., while, for, do-while).
   - Checks loop conditions and termination.
   - Example: `while (condition) { statement(s); }`.
3. **Branch Coverage/Condition Testing**:
   - Tests logical conditions (true/false) for if and else branches.
   - Ensures all decision points are covered.
4. **Statement Coverage**:
   - Executes every code statement at least once to verify functionality.
   - Example:

```
Prints(int a, int b) {
   int result = a + b;
   if (result > 0)
      Print("Positive", result);
   else
      Print("Negative", result);
}
```

     - Tests ensure lines 1–6 are executed.

### Advantages

- Optimizes code by identifying hidden errors.
- Easily automated test cases.

- Early error detection improves code quality.
- Covers most code paths.

### Disadvantages

- Complex, expensive, and time-consuming.
- Requires skilled programmers.
- Cannot detect missing functionalities.
- Code redesign requires rewriting test cases.

## Black Box Testing

- **Definition**: Testing functionalities without knowledge of internal code structure, focusing on inputs and outputs.
- **Also Known As**: Behavioral, functional, or closed box testing.
- **Performed By**: Software testers.
- **Testing Levels**: System and acceptance testing.
- **Tools**: QTP, Selenium, LoadRunner, JMeter.
- **Types**:
  - **Functional Testing**: Tests features and functionalities.
  - **Non-Functional Testing**: Tests performance, usability, scalability, etc.
  - **Regression Testing**: Ensures new changes do not affect existing functionalities.

### Techniques

1. **Equivalence Partitioning**:
   - Divides input values into classes with similar outcomes.
   - Tests one value per class to reduce test cases.
   - Example: Age (18–60) → Invalid: ≤17, Valid: 18–60, Invalid: ≥61.
2. **Boundary Value Analysis (BVA)**:
   - Tests boundary values (upper/lower limits) of input ranges.
   - Example: Age (18–60) → Test: 17 (invalid), 18, 19, 59, 60 (valid), 61 (invalid).
3. **Decision Table Testing**:
   - Captures input combinations and system behavior in a table.
   - Example (Gmail Login):

| Email (C1) | Password (C2) | Expected Result |
|---|---|---|
| True | True | Account Page |
| True | False | Incorrect password |
| False | True | Incorrect email |
| False | False | Incorrect email |

4. **Error Guessing**:
   - Uses tester experience to identify problematic areas.
   - Examples: Divide by zero, null values, empty submit, invalid file uploads.
5. **State Transition Testing**:
   - Tests behavior for different inputs to the same function.
   - Example: Limited login attempts (e.g., lock account after 3 failed tries).
6. **All Pairs Testing**:

- Tests discrete combinations of inputs (e.g., checkboxes, radio buttons).
- Reduces test cases for combinatorial inputs.

### Advantages

- No programming knowledge required.
- Efficient for large systems.
- Tests from the user's perspective.
- Identifies specification ambiguities.

### Disadvantages

- Difficult to design test cases without code knowledge.
- Cannot detect control structure errors.
- Exhaustive input testing is time-consuming.

### Black Box vs. White Box Testing

| Aspect | Black Box Testing | White Box Testing |
|---|---|---|
| **Knowledge** | No internal structure knowledge; focuses on input/output. | Knows internal structure and code. |
| **Also Known As** | Functional, data-driven, closed-box, behavioral. | Structural, glass box, code-based, transparent. |
| **Programming Knowledge** | Minimal required. | Complete knowledge required. |
| **Testing Levels** | System, acceptance testing. | Unit, integration testing. |
| **Performed By** | Software testers. | Software developers. |
| **Time** | Less time-consuming. | More time-consuming. |
| **Basis** | External expectations. | Internal code workings. |
| **Testing Approach** | Trial and error; tests data domains. | Tests internal boundaries and code paths. |
| **Example** | Search on Google. | Verify loops with input keywords. |

## Unit Testing

- **Definition**: The first level of testing, where individual units (functions, methods, modules, or objects) are tested in isolation.
- **Also Known As**: Component testing.
- **Performed By**: Software developers.
- **Testing Technique**: White box testing.
- **SDLC Phase**: Coding phase.
- **Tools**: JUnit, NUnit, PHPUnit, EMMA, JMockit.

### Purpose

- Verify code correctness and enable quick changes.
- Test every function and procedure.
- Fix bugs early to save costs.
- Aid documentation and code reuse.
- Improve software efficiency.

### Unit Testing in Object-Oriented Context

- Tests packages, classes, methods, subclasses, and attributes (public, private, protected).
- Example: Test individual modules (e.g., login, search, payment) in a project.

### Advantages

- Modular testing without waiting for other components.
- Focuses on unit functionality.
- Early issue detection improves quality.
- Enhances development efficiency.

### Disadvantages

- Time-consuming to create and maintain test cases.
- Limited to individual units, not interactions.
- Requires ongoing maintenance with code changes.

## Integration Testing

- **Definition**: The second level of testing, where integrated modules are tested as a group to verify communication and functionality.
- **Also Known As**: Thread testing, string testing.
- **Performed By**: Developers and testers.
- **Goal**: Ensure correctness and interaction among modules.
- **Tools**: Selenium, PyTest, JUnit, Jasmine, Steam, Mockito.

### Purpose

- Verify differing programming logic across modules.
- Check database interactions.
- Address untested requirement changes.
- Detect module incompatibilities.
- Ensure hardware-software compatibility.

### Example

- **Gmail Application**:
  - User 1: Logs in, composes, and sends mail to User 2; saves to Draft or Sent Items.
  - User 2: Logs in, checks Inbox, verifies mail receipt, replies if needed, logs out.

### Types

1. **Incremental Integration Testing**:
   - Modules are integrated and tested one by one in ascending order.
   - Tests data flow and function correctness.
   - **Subtypes**:
     - **Top-Down**:
       - Starts with top-level modules, moving downward.
       - Uses **stubs** (dummy programs) for missing lower modules.
       - Prioritizes critical modules for early flaw detection.
     - **Bottom-Up**:
       - Starts with lowest modules, moving upward.
       - Uses **drivers** (dummy programs) for missing higher modules.
       - Allows simultaneous testing of subsystems.
   - **Example** (Flipkart Application):

    - Flow: Login → Home → Search → Add to Cart → Payment → Logout.

2. **Non-Incremental Integration Testing (Big Bang Testing)**:
    - Integrates and tests all modules at once after individual testing.
    - Suitable for smaller systems.
    - Difficult to pinpoint errors due to lack of parent-child hierarchy.

### Incremental vs. Non-Incremental Testing

| Aspect | Incremental Testing | Non-Incremental Testing |
|---|---|---|
| **Integration Approach** | Tests modules gradually. | Tests all modules at once. |
| **Planning** | Requires step-by-step planning. | Simpler, one-time planning. |
| **Resource Efficiency** | Uses more resources (separate tests). | Uses fewer resources (single test). |
| **Issue Detection** | Early detection with progressive testing. | Late detection due to bulk testing. |
| **Complexity** | Manages smaller pieces, less complex. | More complex due to testing everything. |

## Conclusion

Software testing ensures a product is reliable, functional, and meets user needs. Key principles guide defect detection, while test cases verify functionality. White box testing examines code structure, black box testing focuses on functionality, unit testing isolates components, and integration testing verifies module interactions. Understanding these concepts, techniques, and examples is critical for exam success and effective software development.