Unit-III: System Design

1. What is Software Design?

Software design is a process where software requirements are transformed into a blueprint that guides coding and implementation. It focuses on how to meet requirements efficiently and lays the groundwork for development.

Key Components:

- **Architecture**: Defines the high-level structure.
- **Modules**: Independent components or building blocks.
- **Interfaces**: Connections between modules.

---

2. Importance of Software Design

- **Improves Maintainability**: Well-structured designs are easier to debug and update.
- **Enhances Scalability**: Systems can adapt to new requirements.
- **Improves Collaboration**: Serves as a shared blueprint for developers and stakeholders.
- **Cost Efficiency**: Early problem detection reduces rework during later stages.

---

3. Objectives of Design

1. **Correctness**: Meets functional and non-functional requirements.
2. **Efficiency**: Minimizes resources like memory and processing power.
3. **Modularity**: Breaks the system into manageable, cohesive parts.
4. **Flexibility**: Allows for future enhancements.
5. **Maintainability**: Simplifies debugging and updates.

---

4. Comparison of Good and Bad Design

Good Design	Bad Design
Modular with clear responsibilities	Overlapping or unclear module roles
Low coupling and high cohesion	High coupling and low cohesion
Well-documented and understandable	Poor documentation, hard to follow
Flexible for future changes	Rigid and hard to modify
Efficient use of resources	Resource-heavy and inefficient

---

5. Design Framework

A **design framework** outlines the steps to create a software blueprint:

1. **Requirement Analysis**: Understand user needs.
2. **Problem Partitioning**: Divide the system into smaller, manageable parts.
3. **Abstraction**: Focus on essential details and omit unnecessary complexity.
4. **Verification**: Ensure the design meets all requirements.
5. **Iteration**: Refine the design for clarity and feasibility.

Diagram: Design Framework

```
[Requirements Analysis]
         ↓
[Problem Partitioning] → [Abstraction]
         ↓
 [Design Verification] → [Iteration]
```

---

6. Problem Partitioning

Large systems are broken into smaller, manageable sub-systems or modules. This helps:

- Simplify complex problems.
- Enable parallel development.
- Isolate faults for easier debugging.

---

7. Abstraction

Abstraction hides unnecessary details while focusing on essential features.

- **Control Abstraction**: Simplifies control flow (e.g., function calls).
- **Data Abstraction**: Represents data logically (e.g., using classes in OOP).

---

8. Top-Down and Bottom-Up Design

- **Top-Down Design**:
  - Start with the overall system.
  - Break it into sub-systems and smaller components.
  - Example: Define overall app logic → Write individual functions.
- **Bottom-Up Design**:
  - Begin with detailed modules or components.
  - Integrate to form a complete system.
  - Example: Build utility libraries first → Assemble into a full program.

Diagram: Top-Down vs. Bottom-Up Design

```
Top-Down Design:
[Main System]
     ↓
[Sub-System A] → [Sub-System B]
     ↓                ↓
[Modules]      [Modules]

Bottom-Up Design:
[Modules]
     ↑
[Sub-System A] → [Sub-System B]
     ↑                ↑
[Main System]
```

---

9. Cohesion and Coupling

- **Cohesion**: Measures how closely related the tasks within a module are.
  - High cohesion means the module focuses on a single responsibility.
- **Coupling**: Refers to the interdependency between modules.
  - Low coupling is ideal for reducing complexity.

Diagram: High Cohesion and Low Coupling

```
[Module A] → Independent
[Module B] → Independent
(Modules interact only via defined interfaces)
```

Unit-IV: Coding and Testing

---

1. Coding

Top-Down vs. Bottom-Up Coding

- **Top-Down**: Code begins with the main logic and progressively fills in the details.
- **Bottom-Up**: Starts with utility components and integrates them into larger functions.

Structured Programming

Uses a clear flow of control:

1. **Sequence**: Instructions are executed in order.
2. **Selection**: Decision-making (e.g., `if-else`).
3. **Iteration**: Loops for repetitive tasks (e.g., `for`, `while`).

Information Hiding

Restricts access to certain parts of a program to ensure security and maintainability.

- Example: Using `private` methods in classes to protect data.

Programming Style

1. Use meaningful variable and function names.
2. Follow consistent indentation and formatting.
3. Write comments to clarify logic.

---

2. Testing

Levels of Testing

1. **Unit Testing**:
   - Focus: Individual functions or modules.
   - Example: Testing a login function independently.
2. **Integration Testing**:
   - Focus: Interaction between modules.
   - Approaches:
     - **Top-Down Integration Testing**: Start from the top-level module.
     - **Bottom-Up Integration Testing**: Start from low-level modules.
3. **System Testing**:
   - Focus: Entire system tested as a whole.
4. **Acceptance Testing**:
   - Focus: Validating against user requirements (Alpha/Beta).

Diagram: Levels of Testing

```
[Unit Testing]
      ↓
[Integration Testing]
      ↓
[System Testing]
      ↓
[Acceptance Testing]
```

Functional and Structural Testing

- **Functional Testing**:
  - Validates software against requirements.
  - Example: Black-box testing.
- **Structural Testing**:
  - Examines code logic.
  - Example: White-box testing.

Test Plan and Test Case Specification

- **Test Plan**: Outlines testing scope, schedule, and objectives.
- **Test Case Specification**: Details inputs, steps, and expected outcomes for testing.

Reliability Assessment

Measures the likelihood of software performing without failure for a specified period.

Software Testing Strategies

1. **Verification**: Confirms the system is built as per design.
2. **Validation**: Ensures the system meets user needs.

Alpha and Beta Testing

- **Alpha Testing**: Conducted by developers or internal teams.
- **Beta Testing**: Performed by end-users in a real-world environment.
