UTT

UNIVERSIDAD TECNOLÓGICA DE TIJUANA

GOBIERNO DE BAJA CALIFORNIA

TEMA:

Design Patterns in Software Development

BY:

Derian Omar Tarango Mendez

GROUP:

10 B

COURSE:

Software development process management

PROFESSOR:

Ray Brunett Parra Galaviz

Tijuana, Baja California, 10 of january 2025

Design Patterns in Software Development

1. Introduction

The selection of design patterns is a critical step in software development. It ensures that proven solutions are applied to common problems, enhancing code quality, reusability, and maintainability. This document provides guidance on selecting appropriate design patterns based on specific scenarios and requirements.

---

2. Importance of Design Pattern Selection

1. **Problem Solving**: Design patterns provide standardized solutions to recurring challenges.
2. **Code Reusability**: Promotes modular and reusable code components.
3. **Improved Communication**: Offers a shared language for developers.
4. **Scalability**: Helps systems evolve to handle growing complexity.

---

3. Criteria for Selecting Design Patterns

3.1 Nature of the Problem

- **Creational Patterns**: If the challenge involves object creation, patterns like Singleton or Factory Method are suitable.
- **Structural Patterns**: For organizing code or relationships, consider Adapter or Composite.
- **Behavioral Patterns**: Use patterns like Observer or Strategy to manage object communication.

3.2 System Requirements

- **Scalability**: Patterns like Proxy or Flyweight manage resource usage effectively.
- **Flexibility**: Use Decorator or Strategy for interchangeable behaviors.
- **Security**: Apply patterns like Proxy to control access.

3.3 Team Expertise

- Familiarity with patterns ensures proper implementation.
- Simpler patterns like Singleton are easier for beginner teams.

3.4 Performance Impact

- Evaluate if the pattern adds overhead or optimizes performance.
- Example: Avoid overuse of Observer to prevent performance bottlenecks.

---

4. Example Scenarios and Pattern Selection

4.1 Scenario: Dynamic Behavior Changes

- **Challenge**: A payment system requires different algorithms for processing payments.
- **Solution**: Use the **Strategy Pattern** to define interchangeable payment algorithms.

4.2 Scenario: Limited Resource Management

- **Challenge**: A graphics rendering engine requires efficient memory usage.
- **Solution**: Apply the **Flyweight Pattern** to reuse common objects.

4.3 Scenario: Simplifying Complex Subsystems

- **Challenge**: A media application integrates audio, video, and playback controls.
- **Solution**: Use the **Facade Pattern** to simplify subsystem interaction.

---

5. Best Practices

1. **Understand the Problem**: Avoid over-engineering; only use patterns when necessary.
2. **Combine Patterns**: Mix patterns like Factory Method and Singleton for complex problems.
3. **Prioritize Readability**: Ensure the pattern improves clarity rather than complicating the code.
4. **Document Decisions**: Keep track of why and where patterns are applied.