UTT
UNIVERSIDAD TECNOLÓGICA DE TIJUANA
GOBIERNO DE BAJA CALIFORNIA

TEMA:

Selection of Design Patterns

PRESENTADO POR:

Paredes Nevarez Alexis Omar

GRUPO:

10B

MATERIA:

Software Development Process Management

PROFESOR:

Ray Brunett Parra Galaviz

FECHA:

07/10/2025

Understanding Design Patterns

Design patterns are reusable solutions to recurring problems in software design. They are not finished designs but templates that provide guidance on how to solve specific design issues. Introduced by the "Gang of Four" (GoF) in their seminal book Design Patterns: Elements of Reusable Object-Oriented Software, these patterns are categorized into three main types:

1. **Creational Patterns**: Focus on object creation mechanisms, ensuring objects are created in a manner suitable for the situation. Examples include:
    - **Singleton**: Ensures a class has only one instance.
    - **Factory Method**: Defines an interface for creating objects but lets subclasses alter the type of objects created.
    - **Builder**: Separates the construction of a complex object from its representation.
2. **Structural Patterns**: Deal with the composition of classes and objects to form larger structures while keeping systems flexible. Examples include:
    - **Adapter**: Allows incompatible interfaces to work together.
    - **Composite**: Composes objects into tree structures to represent part-whole hierarchies.
    - **Decorator**: Adds behavior to individual objects dynamically.
3. **Behavioral Patterns**: Focus on communication and responsibility between objects. Examples include:
    - **Observer**: Defines a one-to-many dependency, so when one object changes state, all dependents are notified.
    - **Strategy**: Encapsulates algorithms, allowing them to be swapped interchangeably.
    - **Command**: Encapsulates a request as an object, enabling parameterization and queuing of requests.

How to Select the Right Pattern

1. Analyze the Problem Context

Start by clearly defining the problem you are solving. Consider:

- **Nature of the problem**: Is it related to object creation, structure, or behavior?
- **System requirements**: Are there specific performance, scalability, or maintainability constraints?
- **Stakeholder needs**: What are the expectations of end-users, clients, or team members?

2. Map the Problem to a Pattern Category

- Use **Creational Patterns** if your issue involves object instantiation, such as ensuring proper resource allocation or controlling object creation.
- Choose **Structural Patterns** if your focus is on class composition or integrating different subsystems.
- Opt for **Behavioral Patterns** if the problem involves communication, workflows, or algorithms.

3. Evaluate Pattern Suitability

- Study pattern intent: Match the goals of the pattern with your problem.
- Assess consequences: Evaluate how a pattern impacts performance, complexity, and scalability.
- Prototype solutions: Test the pattern in a smaller context to identify potential issues.

4. Consider Future Proofing

- Ensure the chosen pattern allows for easy maintenance and scalability.
- Avoid patterns that add unnecessary complexity or make the codebase harder to understand.

Common Challenges in Selecting Patterns

- **Pattern Overuse**: Over-enthusiasm for patterns can lead to overengineering and unnecessary complexity.
- **Misapplication**: Using a pattern without fully understanding its context or consequences can result in inefficiency.

- **Lack of Experience**: Teams unfamiliar with patterns may struggle to implement or adapt them effectively.

Example Scenarios for Pattern Use

Scenario 1: Simplifying Object Creation

- **Problem**: A system needs to create multiple types of objects, but the exact type is determined at runtime.
- **Solution**: Use the **Factory Method** pattern to delegate the responsibility of object instantiation to subclasses.

Scenario 2: Adding Features Dynamically

- **Problem**: New behaviors must be added to existing objects without modifying their code.
- **Solution**: Apply the **Decorator** pattern to extend object functionality at runtime.
-

Scenario 3: Handling Event Notifications

- **Problem**: An application must notify multiple components when a change occurs.
- **Solution**: Implement the **Observer** pattern to create a publish-subscribe mechanism.

Benefits of Selecting the Right Pattern

- **Improved Code Reusability**: Patterns standardize solutions, making them reusable across projects.
- **Enhanced Maintainability**: Clear design structures make the codebase easier to modify and extend.
- **Better Communication**: Patterns provide a shared vocabulary for developers, simplifying discussions and decision-making.