Design Pattern

Instructor: Mehroze Khan

Introduction

- Designing object-oriented software is hard, and designing reusable object-oriented software is even harder.
- It takes a long time for novices to learn what good object-oriented design is all about. Experienced designers evidently know something inexperienced ones don't.
- Expert designers find recurring patterns of classes and communicating objects in many object-oriented systems and can apply them immediately to design problems without having to rediscover them.
- Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems.
- Recording experience in designing object-oriented software as design patterns.
- Design patterns make it easier to reuse successful designs and architectures.

What is a Design Pattern?

- Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".
- In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design.
- A design pattern isn't a finished design that can be transformed directly into code.
- It is a description or **template** for how to solve a problem that can be used in many different situations.
- You can’t just find a pattern and copy it into your program, the way you can with off-the-shelf functions or libraries.
- The pattern is not a specific piece of code, but a general concept for solving a particular problem.
- You can follow the pattern details and implement a solution that suits the realities of your own program.

What is a Design Pattern?

Four essential elements of a pattern::

- The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
- The **problem** describes when to apply the pattern.
- The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations
- The **consequences** are the results and trade-offs of applying the pattern.

The design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

Benefits

Design patterns offer several benefits in software development. Here are some of the key advantages:

1. **Reusability**:

   - Design patterns provide tested, proven development paradigms. Developers can use these patterns to solve recurring problems without having to recreate solutions each time.

   - Reusing design patterns helps in preventing subtle issues that can cause major problems and improves the overall software quality.

2. **Maintainability**:

   - Design patterns make the code more maintainable. They provide a clear and modular structure, making it easier for developers to understand and modify existing code.

   - When a specific part of the system needs to be changed, developers can focus on that particular part without affecting the entire codebase.

3. **Scalability**:

   - Design patterns help in creating scalable solutions. They provide a foundation for building scalable and flexible software systems, making it easier to adapt to changing requirements and handle increased complexity.

4. **Abstraction**:

   - Design patterns promote abstraction by providing a high-level view of the system. This allows developers to focus on essential details while hiding unnecessary complexities.

   - Abstraction makes it easier to communicate and collaborate with other team members, as well as understand the overall architecture of the software.

Benefits

5. **Encapsulation**:

- Many design patterns promote encapsulation, which is the bundling of data and methods that operate on the data into a single unit or class. This helps in organizing code and prevents unintended interference from external components.

6. **Flexibility**:

- Design patterns make the code more flexible and adaptable to change. By following well-established patterns, developers can more easily incorporate new features or modify existing ones without disrupting the entire system.

7. **Readability**:

- Design patterns improve code readability. Since many developers are familiar with common design patterns, using them in code makes it easier for others to understand the structure and functionality of the software.

8. **Collaboration**:

- Design patterns provide a common vocabulary for developers. This facilitates communication and collaboration among team members, making it easier to discuss and share design ideas and solutions.

9. **Performance Optimization:**

- Some design patterns help in optimizing the performance of the software.

10**. Proven Solutions:**

- Design patterns encapsulate best practices and proven solutions to common problems. Leveraging these patterns can save development time and reduce the likelihood of introducing errors.

Drawbacks

While design patterns offer numerous benefits, it's important to be aware of potential drawbacks and considerations:

1. **Complexity**:

   - Overusing design patterns can lead to overly complex and convoluted code. Not every problem requires a design pattern and applying them unnecessarily can make the code harder to understand and maintain.

2. **Rigidity**:

   - Design patterns, if applied too rigidly, may make the system inflexible to change. In some cases, adhering strictly to a pattern may hinder the ability to adapt the software to evolving requirements.

3. **Applicability**:

   - Not all design patterns are suitable for every situation. Choosing the wrong pattern for a particular problem can lead to unnecessary complexity and reduced code efficiency.

4. **Overhead**:

   - Introducing design patterns can introduce additional layers of abstraction and indirection, which may result in some performance overhead. In performance-critical applications, this can be a concern.

Drawbacks

5. **Anti-patterns:**

- Misusing or misunderstanding design patterns can lead to the creation of **anti-patterns—solutions that seem to solve a problem but introduce more issues**. This can happen when a pattern is applied without a clear understanding of its intent.

6. **Code Bloat:**

- In some cases, design patterns may lead to code bloat, where the structure of the pattern adds more lines of code than a simpler, more direct solution would. This can impact readability and maintenance.

7. **Lack of Creativity:**

- Relying too heavily on design patterns might stifle creativity in problem-solving. Developers may become accustomed to applying known patterns without considering innovative or more efficient alternatives.

Catalog of Design Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Purpose

- The first criterion, called **purpose**, reflects what a pattern does.
- Patterns can have either **creational**, **structural**, or **behavioral** purpose.

There are three main groups of patterns:

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

Scope

- The second criterion, called **scope,** specifies whether the pattern applies primarily to classes or to objects.
- **Class patterns** deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are **static**—fixed at compile-time.
- **Object patterns** deal with object relationships, which can be changed at run-time and are more **dynamic**.

Reference

- Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Pearson, 1995.