What is a Design Pattern?

A design pattern is a reusable and general solution to a common software design problem.

It's a template or blueprint for solving a recurring problem in a specific context within software development.

Design patterns provide a way to create well-structured, maintainable, and efficient code by promoting best practices and proven solutions to common programming challenges.

Some characteristics of design patterns:

- Design patterns encapsulate solutions to common problems, making it easier to reuse and apply them in different parts of your software or in different projects.

- Design patterns often represent best practices in software design. They have been refined and proven over time by experienced developers.

- Design patterns give developers a common vocabulary to communicate about design decisions and code structure, making it easier to collaborate and understand each other's code.

- They promote flexibility and adaptability in software design. By following a design pattern, it becomes easier to make changes or extend the codebase without introducing excessive complexity.

- Design patterns can help create software architectures that are scalable and can handle growth and complexity effectively.

Design patterns are like tried-and-true solutions to common problems in software design. Think of them as recipes for building software in a smart and efficient way. Instead of reinventing the wheel, you can follow these recipes to solve specific challenges in a proven and reusable manner.

Imagine you're cooking, and you want to make a sandwich. You could follow a recipe that tells you how to assemble the ingredients (bread, cheese, meat, etc.) to create a delicious sandwich. In software development, design patterns are similar recipes, but for creating software components and structures.

The most common 3 categories of design patterns are –

1. **Creational Patterns**: These patterns focus on object creation mechanisms, providing ways to create objects in a manner that is flexible, efficient, and consistent. Examples include the Singleton, Factory Method, and Builder patterns.

2. **Structural Patterns**: These patterns deal with object composition and structure. They help define how objects can be combined to form larger structures while keeping the system flexible and easy to maintain. Examples include the Adapter, Decorator, and Composite patterns.

3. **Behavioral Patterns**: These patterns address how objects interact and communicate with each other. They define the responsibilities and collaboration between objects, making the system more flexible and easy to understand. Examples include the Observer, Strategy, and Command patterns.

Creational design patterns are a category of design patterns that deal with object creation mechanisms, trying to abstract/hide the process of object instantiation. They help in controlling which classes to instantiate and how to create objects, providing flexibility and control over the instantiation process. Creational patterns often hide the specifics of object creation, making the system more independent of the way its objects are created, composed, and represented. Here are some common creational design patterns:

Singleton Pattern:

Classified as one of the most known Creational Patterns.

The Singleton Pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. It's used when you want to limit the instantiation of a class to a single object that can coordinate actions across a system.

- Ensure that a class has only one instance.

- Provide a global point of access to that instance.

- Protects the instance from being overridden.

- **Private Constructor** - has a default private constructor to prevent other objects from using the new operator with the singleton class.

- The static creation method acts as a constructor under the hood, this method calls the private constructor to create an object and saves it in a static field (this is for getting the singleton instance).

Code ----------

```
public class HelpDesk {
    private static HelpDesk helpDesk; // Static variable holding the only instance
    private HelpDesk() {
        // Private constructor so no one can create another one with "new"
    }

    public static HelpDesk getInstance() {
        if (helpDesk == null) {
            helpDesk = new HelpDesk();  // Create only if it doesn’t exist yet (lazy initialization)
        }
        return helpDesk;
    }

    public void getService() {
        System.out.println("How can I help you?");
    }
}
```

```
//These red parts are how codes usually should be, but in this code these red lines are not correct

public class Student{

        HelpDesk hd = new HelpDesk();

        hd.getService();

}

public class Teacher{

        HelpDesk hd = new HelpDesk();

        hd.getService();

}

Public class Student{

        HelpDesk hd = HelpDesk.getInstance();

        hd.getService();

}


Public class Teacher {

        HelpDesk hd = HelpDesk.getInstance(); \\ here getInstance is a static method, we can call it with the class name (HelpDesk).

        hd.getService();

}
```

Lazy Instance: Make use of lazy initialization of the class.

- Its creation is deferred until it is first used.
- Avoid wasteful compilation, reduce program memory, and improve performance.

Advantages:

- Ensures a single point of control for a specific resource or functionality.

- Lazy initialization: The instance is created only when it's first requested, which can save resources.

Drawbacks:

- Can make the code less testable if it tightly couples components to the singleton instance.

- Overuse can lead to hidden dependencies and make code harder to understand.

Examples of Singleton Pattern Usage:

- **Settings Manager**: One place to manage app settings.
- **Database Pool**: One object handles all DB connections.

Lecture

by

Maliha Bushra Hoque (MBH)