**DESIGN PATTERNS**

* Design Patterns (DP) tells us how to structure classes and objects to solve certain problem.
* A pattern describes a proven solution to a recurring design problem.
* A pattern is a solution to a problem in a context.
* The context is the situation in which the pattern applies. This should be a recurring situation.

Why Use Patterns?

They have been proven and are reusable.

Types of Design Pattern

1. **CREATIONAL**

Involves object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Ex: Singleton, Factory Method, Abstract Factory, Builder, Prototype

1. **Singleton Pattern**

Ensure that only one instance of a class is created.

Provide a global point of access to the object.

Ex: Service Locator

1. **Factory Pattern**

Creates objects without exposing the instantiation logic to the client.

Refers to the newly created object through a common interface

1. **BEHAVIORAL**

Is concerned with how classes and objects interact and distribute responsibility.

Ex: Iterator, Observer

1. **Iterator Pattern**

Access contents of a collection without exposing its internal structure.

Support multiple simultaneous traversals of a collection.

Provide a uniform interface for traversing different collection.

1. **Observer Pattern**

Defines a **one-to-many dependency between objects** so that when one object changes state, all of its dependents are notified and updated automatically.

1. **STRUCTURAL**

Let’s you compose classes or objects into larger structures.

Ex: Decorative, Façade, Composite, Adapter, Proxy.

**Decorative DP**

The intent of this pattern is to add additional responsibilities dynamically to an object.

The decorator pattern applies when there is a need to dynamically add as well as remove responsibilities to a class, and when sub-classing would be impossible due to the large number of subclasses that could result.

Ex:

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

br.readLine()

**Façade Pattern**

Provides a unified interface to a set of interfaces in a subsystem.

**Open Close Principle (OCP)**

Software entities like classes, modules and functions should be open for extension but closed for modifications.

The Open Close Principle states that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code.

**Liskov's Substitution Principle(LSP)**

Derived types must be completely substitutable for their base types.

Yes, Liskov Substitution Principle is about inheritance, but about well-designed inheritance.

You should be able to substitute subclass object for base class without things going wrong.