Design Patterns

design patterns are patterns that make things easier, such as Creative Design Patterns which these design patterns are about creating instances of a class. This pattern can be further divided into class creation patterns and object creation patterns.

Structural design patterns.

All of these design patterns are related to the composition of classes and objects. Structural class creation patterns use inheritance to create interfaces. **Adapter**:Map interfaces of different classes,**Bridge**:Separates an object's interface from its implementation.

Behavioral Design Patterns

All of these design patterns are related to the interaction of class objects. Behavioral patterns are those patterns that are most specifically related to communication between objects.

SOLID Principles

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. **I**nterface Segregation
5. **D**ependency Inversion

While these concepts may seem daunting, they can be easily understood with some simple code examples. In the following sections, we'll take a deep dive into these principles, with a quick Java example to illustrate each one.

Single Responsibility

Let's begin with the single responsibility principle. As we might expect, this principle states that **a class should only have one responsibility. Furthermore, it should only have one reason to change.**

How does this principle help us to build better software? Let's see a few of its benefits:

1. **Testing** – A class with one responsibility will have far fewer test cases.
2. **Lower coupling** – Less functionality in a single class will have fewer dependencies.
3. **Organization** – Smaller, well-organized classes are easier to search than monolithic ones.

Open for Extension, Closed for Modification

t's now time for the O in SOLID, known as the **open-closed principle.** Simply put, **classes should be open for extension but closed for modification. In doing so, we stop ourselves from modifying existing code and causing potential new bugs** in an otherwise happy application.

Of course, the **one exception to the rule is when fixing bugs in existing code**

Liskov Substitution

Next on our list is Liskov substitution, which is arguably the most complex of the five principles. Simply put, **if class *A* is a subtype of class *B*, we should be able to replace *B* with *A* without disrupting the behavior of our program.** By throwing a car without an engine into the mix, we are inherently changing the behavior of our program. This is **a blatant violation of Liskov substitution and is a bit harder to fix than our previous two principles.**

Interface Segregation

The I in SOLID stands for interface segregation, and it simply means that **larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.**

Dependency Inversion

The principle of dependency inversion refers to the decoupling of software modules. This way, instead of high-level modules depending on low-level modules, both will depend on abstractions.