**DESIGN PATTERNS**

**DESIGN PATTERNS:**

🡪 standardized and reusable solution to a specific software design problem

🡪 3 Categories

**1. Creational Design Pattern** 🡪 Creational patterns focus on the process of object creation, providing mechanisms to create objects in a flexible and efficient manner.

**2. Structural Design Pattern** 🡪 Structural patterns deal with the composition of classes and objects to create larger structures or entities.

**3. Behavioral Design Pattern** 🡪 Behavioral patterns deal with object collaboration, communication, and responsibility distribution.

**CREATIONAL DESIGN PATTERN:**

* **Singleton Pattern** **🡪** Ensures a class has only one instance and provides a global point of access to that instance.
* **Factory Method Pattern 🡪** Defines an interface for creating objects, allowing subclasses to decide which class to instantiate.
* **Abstract Factory Pattern** **🡪** Provides an interface for creating families of related or dependent objects without specifying concrete classes.
* **Builder Pattern** **🡪** Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
* **Prototype Pattern** **🡪** Specifies the kind of objects to create using a prototypical instance and create new objects by copying this prototype.

**STRUCTURAL DESIGN PATTERN:**

* **Adapter Pattern** **🡪** Converts the interface of a class into another interface that clients expect.
* **Bridge Pattern** **🡪** Decouples an abstraction from its implementation, allowing both to change independently.
* **Composite Pattern** **🡪** Composes objects into tree structures to represent part-whole hierarchies.
* **Decorator Pattern** **🡪** Attaches additional responsibilities to an object dynamically.
* **Facade Pattern** **🡪** Provides a simplified interface to a set of interfaces in a subsystem.
* **Flyweight Pattern** **🡪** Uses sharing to support large numbers of fine-grained objects efficiently.
* **Proxy Pattern** **🡪** Provides a surrogate or placeholder for another object to control access to it.

**BEHAVIORAL DESIGN PATTERN:**

* **Chain of Responsibility Pattern🡪** Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
* **Command Pattern** **🡪** Encapsulates a request as an object, allowing parameterization and queuing of requests.
* **Interpreter Pattern** **🡪** Provides a way to evaluate language grammar or expressions.
* **Iterator Pattern** **🡪** Provides a way to access the elements of an aggregate object sequentially without exposing its representation.
* **Mediator Pattern** **🡪** Defines an object that encapsulates how a set of objects interact, reducing direct dependencies.
* **Memento Pattern** **🡪** Captures an object's internal state to allow restoring it later.
* **Observer Pattern** **🡪** Defines a dependency between objects, so when one changes state, its dependents are notified and updated.
* **State Pattern** **🡪** Allows an object to change its behavior when its internal state changes.
* **Strategy Pattern** **🡪** Defines a family of algorithms, encapsulates each, and makes them interchangeable.
* **Template Method Pattern** **🡪** Defines the structure of an algorithm in a superclass and lets subclasses provide specific implementations for certain steps.
* **Visitor Pattern** **🡪** Represents an operation to be performed on elements of an object structure without changing their classes.