**Design Patterns**

* **Factory Method**
* This method encapsulates the creation of objects, letting the subclass decide which objects to create.
* **Abstract Factory**
* **Abstract Factory** is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.
* **Builder**
* **Builder** is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.
* **Prototype**
* **Prototype** is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.
* **Singleton**
* Guarantee that there is only one instance of the singleton object within an application.
* This pattern is commonly used to ensure that only one instance of the particular object exists in memory. This is particularly useful for objects that coordinate access to shared resources such as data stores.
* **Adapter**
* **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.
* **Bridge**
* **Bridge** is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.
* **Composite**
* **Composite** is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.
* **Decorator**
* Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
* **Façade**
* **Facade** is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.
* **Flyweight**
* **Flyweight** is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.
* **Proxy**
* **Proxy** is a structural design pattern that lets you provide a substitute or placeholder for another object.
* A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.
* **Chain of Responsibilities**
* **Chain of Responsibility** is a behavioural design pattern that lets you pass requests along a chain of handlers.
* Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.
* **Command**
* **Command** is a behavioural design pattern that turns a request into a stand-alone object that contains all information about the request.
* This transformation lets you pass requests as a method-arguments, delay or queue a request’s execution, and support undoable operations.
* **State**
* **State** is a behavioural design pattern that lets an object alter its behaviour when its internal state changes.
* It appears as if the object changed its class.
* **Memento**
* Memento design pattern is used when we want to save the state of an object so that we can restore later on.
* Memento pattern is used to implement this in such a way that the saved state data of the object is not accessible outside of the object, this protects the integrity of saved state data
* **Iterator**
* **Iterator** is a behavioural design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.)
* **Strategy**
* **Strategy** is a behavioural design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
* **Template Method**
* **Template Method** is a behavioural design pattern that defines the skeleton of an algorithm in the superclass but let’s subclasses override specific steps of the algorithm without changing its structure.
* **Observer**
* **Observer** is a behavioural design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.
* **Mediator**
* **Mediator** is a behavioural design pattern that lets you reduce chaotic dependencies between objects.
* The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.
* **Visitor**
* The visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying the structures. It is one way to follow the open/closed principle.
* **Interpreter**
* Interpreter pattern is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.