Design Patterns

1 Builder Design pattern:

Builder Design Pattern is very useful when we are dealing with complex objects (i.e. implementing different objects with different construction ways). Let us consider a scenario where we need to build a vehicle, at that time one vehicle class with too many arguments is not enough for constructing all types of vehicle since vehicle may be two-wheeler or 4-wheeler or any other type. In that situations our code become very complex since we need vehicle which is very specific to user. even it is very hard to maintain order of the arguments. Even we don’t want to send all parameters in Object initialisation.

To overcome the above problem Builder pattern will very mush useful. In this builder class contain constructor for all required parameters and it have the methods for optional parameters. Then this builder object sent to our vehicle class so that only required vehicle can be created.

The main vehicle class only contains getters and private constructor which takes the builder object as parameter to build vehicle.

In the main method client will provide whatever vehicle he want with builder class for example Bike contains only engine, wheels as main parameters air bags and FM radio are optional for the bike so it will make them as optional and passes this builder to the Vehicle class there our vehicle class fixes whatever they want.

2 Command Design Pattern

Command Pattern is one of the Behavioural Design Pattern. Command design pattern is used to implement loose coupling in a request-response model.

In command pattern, the request is sent to the invoker and invoker pass it to the encapsulated command object.

Command object passes the request to the appropriate method of Receiver to perform the specific action.

The client program creates the receiver object and then attach it to the Command. Then it creates the invoker object and attach the command object to perform an action.

We can also perform undoable operation with command pattern by storing Actions along with some data in the queue so that we can perform undo and redo operations.

**Command** declares an interface for all commands, providing a simple **execute ()** method which asks the **Receiver** of the command to carry out an operation. The **Receiver** has the knowledge of what to do to carry out the request.  The **Invoker** holds a command and can get the **Command** to execute a request by calling the execute method. The **Client** creates **Concrete Comman**ds and sets a **Receiver** for the command. The **Concrete Command** defines a binding between the action and the receiver. When the **Invoker** calls execute the Concrete Command will run one or more actions on the Receiver.

Let us consider Universal Remote as an example for this command. We can use if else ladder we can perform on/off/volume up operations but Universal remote is capable of controlling many things like lights/TV/AC /Radio/DVD etc… in this scenario implementing all the command in if else is not so suitable, since in future if any new feature comes into picture we need to modify if else block this is the violation of open/closed rule.

To overcome this problem Command pattern will help us a lot. Here we encapsulate request as an object and then we use this object to perform respective operation and we can use this object form any other commands also.

Here in the above example

Client is **Remote** (responsible for creating different commands of different receivers and sending it to invoker) each command act as object and which will implement a interface.

Light/DVD/AC/Radio are **Receivers**

Invoker is responsible for only maintaining the queue and executing command(method) of respective Receivers. here invoker doesn’t know about which Receiver command Is executing. So it uses Receiver method inside command so that it perform specific action.

Disadvantage :

We need to create small classes for every command.

3 Singleton Design Pattern

This is the Creational Design pattern which will ensures that we can get only single instance for the class. This means it will provide globally single access.

The class which is act as singleton will contain private constructor and one synchronised public getInstance () method. This will ensure that only one global instance is created in entire application.

This will also ensure that Thread safety in the application i.e. only one thread will get instantiated at time.so that only one instance is used in all the cases.

It also used in Logger classes, Multithreading, Serialization etc.

In serialization and deserialization there is chance of getting multiple instances so to overcome that problem we can use readResolve method () .

4 Strategy Design Pattern

Strategy design Pattern is the used to make our code loosely coupled and class behaviour can be changed at runtime. This pattern comes under behaviour Design pattern.

In order to implement Strategy pattern, we need to follow some some steps

1.we need to create a interface which will be implemented by each behaviour (Concrete methods).

2.concrete classes must implement this interface.

3.create the demo class to see change in behaviour when it changes at runtime.

Let us consider a scenario of bird fly () abstract method in interface

There are different behaviours for different objects example flight has different way of flying and some birds can’t fly. So, in order to achieve this type of behaviour we can use strategy pattern

Different concrete objects implement interface fly method and then implements their own behaviour so that at runtime dynamically they can change behaviour.

This behaviour will makes the code loosely coupled and easy to implement new behaviour at any instance.

5 Chain of Responsibility Design pattern (Behavioural Design pattern)

In case of chain of responsibility pattern the change of receiving objects against one request i.e. one request can be handled by one or more receiver objects.

As name suggests it creates chain of receiver objects for request. It decouples sender and receiver of request based on type of request. In this pattern each receiver maintain reference to other receiver.

The best example for chain of responsibilities is ATM machine where we enter our required amount and it will process based on amount that we entered. Let us consider we entered amount of Rs 4550 then it will first checks for 2000 notes it will handled first next remaining balance is 550 it cannot be handled by 2000 class so it next it will be handled by other class called 500 class it will do the process for one time remaining balance can be handled by 50 class in this way chain of responsibilities design pattern used by many receivers. If no such handling class available then it will return please enter valid Amount to withdraw.

The main flaw in the chain of responsibility Design pattern is that there is a chance of broken chain in the concrete classes. This will happen when we forgot to give chain of handles in all concrete handlers but it we can overcome this problem by placing handling request in base class.

6 Observer Design Pattern (Behavioural Design Pattern)

This pattern used when there is one to many relationships between objects such as when one Object is modified then its dependent objects will be notified immediately.

Let us consider Flipkart as an example, In Flipkart when an item is not available, we have an option called **notify me.** When users click on notify, they will be intimated when the product is available. That means here product is an object and users are dependent objects of product whenever the state changes from Not-Available to available then all the users intimated with change that occurred on product.

In order to implement this pattern first we need to create an parent object interface with four abstract methods they are set-status, register, unregister, notify.

All the parent objects will implement the interface such that every observer of this objects will be notified when there is a change in these objects

And observers implement interface which only have one abstract method called update when ever state changes in the parent object then all the observers/dependents of that object are updated so that they will be informed.

More real time examples of Observer pattern are Amazon, YouTube, Online Courses etc.

7 Factory Design pattern (creation design pattern)

This pattern provides one of the best way to create an Object such that we can create objects without exposing creation logic to the client and we can refer to the newly created object with common interface.

Let us consider a scenario where we are dealing with multiple concrete classes. In order to access these concrete classes, we can use if else ladder.

But when we use if else ladder there are two main issues arises those are many new object creations scattered and second one is client knows which object is instantiated.

In order to over come these problems Factory design pattern will be used very much. To solve this we will create one factory class which is responsible for giving object when we pass the type so that all the new keywords are removed from client class then in order to make objects not aware of client we can create a interface such that every concrete class will implement this interface and in client class we will mention only the interface reference so that client never knows which object is created for which invoice.

8 Abstract Factory Design Pattern (Creational Design pattern)

Abstract factory is used to create other factories which are responsible for creating respective concrete products. This type of factory also known as factory of Factories.

In Abstract factory pattern, an interface is responsible for creating a factory of related objects without specifying their classes explicitly. Each generated factory from this factory can give objects as per the factory pattern.

Lets us consider an example of for this Abstract factory method. In the above example we have seen how Factory pattern works. So, let us consider two such factories 1. Shape Factory 2. Colour Factory.

These two Factories can generate their own respective classes such as (circle, rectangle, square for shape Factory) and (Red, Green, Blue for colour factory) now abstract factory is responsible to produce the factory based on input given by user. Here abstract factory acts as factory for shape and colour factory that’s why we called abstract factor as factory of factories.

9.Decorator Pattern (Structural pattern)

Decorator Pattern allows the user to add extra functionalities without changing object structure. That means it acts as a wrapper to the existing object.

This pattern provides a Decorator class which provides additional functionalities by wrapping original object.

Let us consider an example where we can use decorator pattern ex: pizza preparation

1. In Pizza preparation normal implementation of pizza is first we will prepare plain pizza and we have cost for pizza. These methods implemented by interface pizza
2. But when we want varieties in pizza i.e. if we want Tangy Cheese Pizza or any other pizza, we need to add extra functionalities for that pizza.
3. In order to prepare any pizza, we will first set a decorator-pizza class this will implement the same properties that were implemented by plain pizza but it contains one reference to the interface of pizza.
4. Whenever new topping needed for pizza, we will create a class with topping name and we implement the decorator pattern here along with abstract methods of interface we will get one more extra method that will just add toppings along with cost.
5. So whenever we want to prepare a pizza with different toppings then we can create small classes with topping name and add it to plain pizza. In this way we can use decorator pattern without effecting original object structure i.e. plain Pizza

Decorator pattern mainly useful when we want to add additional functions or remove functions dynamically.

10. Flyweight Design Pattern

Flyweight design pattern is primarily used to reduce the number of objects created and decreases memory utilisation and increases performance of a program.

This will mainly reduce the no of objects need to be created and improves structure of application. Flyweight pattern tries to reuse already existing similar kind objects by storing them in a map. So whenever new objects needed it will first check the ap whether that object exist in the map or not. If map contains the object then it will return the same object otherwise it will create new object in this way flyweight pattern reduce the no of objects created and increases the performance

This pattern mainly very useful when we create more than 1000 objects.

Example: let us consider a rectangle shape with 5 colours so when we want to create 500 rectangle with same colours then it will checks whether rectangle with same colour object exist or not then it will create object according to that.