### **DotNet Design Patterns**

### **(Intermediate)**

1. **Prototype Pattern** :(Cloning/zeroxing of the object - By Val)

This pattern falls under the category of **creational design pattern**. This pattern gives us a way to create a new object from the existing instance of the object i.e. we clone the existing object with its data. Cloning can be done in two way :

1. Shallow cloning
2. Deep cloning

**Following are the scenarios where we use prototype pattern :**

* You want to instantiate classes at run time, for example, by dynamic loading.
* Avoid building a class hierarchy of factories that parallels the class hierarchy of products.
* When new instantiations of class can have one of only a few different combinations of state.
* New objects are going to be clones of existing objects.
* Avoid subclasses of an object creator in the client application i.e. Abstract Factory Pattern.
* Avoid resource intensive object instantiation and initialisation logic.

**Advantages of the Prototype Design Pattern :**

* Reduce the time complexity to creating resource consuming objects by using the prototype pattern.
* Reduces the sub-classing.
* Enables adding and removing objects at run time.
* Enables configuring application classes dynamically.

1. **Adapter Pattern** :

This pattern falls under the category of structural design pattern.This pattern is used when two incompatible interfaces are involved i.e it is used like a bridge between incompatible interfaces.

Ex. Memory card reader is used between memory card and laptop

**Following are the scenarios where we use adaptor pattern :**

* This pattern is used when the class has no interface or its interface is incompatible with the rest of the code
* We should use the Adapter when we want to reuse existing classes from our project but they lack a common functionality. By using the Adapter pattern in this case, we do not need to extend each class separately and create a redundant code

**Below are the participants of adapter design pattern**

* **ITarget :** This is an interface which is used by the client to achieve its functionality/request.
* **Adaptee :** This is a class which has the functionality, required by the client. However, its interface is not compatible with the client.
* **Adapter :** This is a class which implements the ITarget interface and inherits the Adaptee class. It is responsible for communication between Client and Adaptee.
* **Client :** This is a class which interacts with a type that implements the ITarget interface. However, the communication class called adaptee, is not compatible with the client

1. **Observer Pattern :**

This pattern falls under the category of Behavioral pattern. This pattern defines one to many dependency between the objects so that when one object changes state, all its dependents are notified and updated automatically.Two important term **subject** and **observer** are used in this pattern.

* **Subject :** This is the object which holds the value and takes the responsibility in notifying the observer when the value is changed. The subject could be database change, property change etc.
* **Observer :**  This is the object listening to the subject’s changes. Basically it will be having its own updating/calculating routine that runs when get notified.

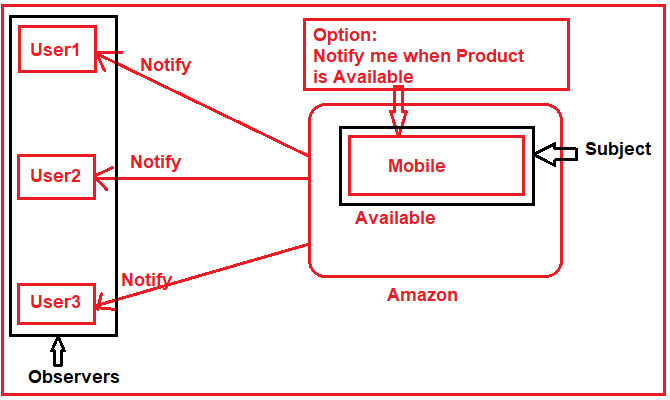
**Following are the scenarios where we use observer pattern :**

* Changes in the state of an object need to be notified to a set of dependent objects, not all of them.
* Notification capability is required.
* The object sending the notification does not need to know about the receivers objects.

**Real Time Example of Observer Pattern :**

As you can see in the above image, three users come to the Amazon site for buy a Mobile Phone. Unfortunately, at that time the Mobile phone is out of stock i.e. the mobile is in Not Available state. But the above three users want to buy that particular mobile. In the Amazon site, there is an option called to notify me when the product is available. What the above three users do is, they registered with the above option, so that when the product is available, the Amazon site will send a notification to them.

After a few days, the Product is available, and so the status is changed from Out Of Stock to available. So, what Amazon will do is send notifications to all the users who are registered. As we already discussed the Observer Design Pattern has two main components i.e. the Subject and the Observer. In our examples, the Mobile is the Subject and three users (i.e. User1, User2, and User3) are the Observers. For better understanding please have a look at the following image.



1. **Mediator Pattern :**

This pattern falls under the category of Behavioral pattern. This pattern defines an object that encapsulates how a set of objects interacts i.e a centralized class is used for interaction between two classes. It will receive input from one class and pass to another class for whom it was intended and the response from the other class will be returned to the first class from where the process started. So it acts as a request-response system between the classes, thus resulting in a loosely coupled system.**The Mediator object acts as the communication center for all objects**. That means when an object needs to communicate to another object, then it does not call the other object directly, instead, it calls the mediator object and it is the responsibility of the mediator object to route the message to the destination object.

**When to use this pattern :**

* Communication between multiple objects is well defined but potentially complex.
* When too many relationships exist and a common point of control or communication is required.
* Some objects can be grouped and customized based on behaviors.

**Below are the participants of adapter design pattern**

* **Mediator :** It is an interface and it defines all possible interactions between the colleagues.
* **ConcreteMediator :** It is a class that implements the Mediator interface and coordinates communication between colleague objects.
* **Colleague :** (Participant) These are abstract classes and these abstract classes are going to be implemented by Concrete Colleague classes.
* **ConcreteColleage1 / ConcreteColleage2 :** These are classes and implement the Colleague interface. If a concrete colleague (let say ConcreteColleage1) wants to communicate with another concrete colleague (let say ConcreteColleage2), they will not communicate directly instead they will communicate via the ConcreteMediator.

**Real Time Example of Mediator Pattern : Air Traffic Control (ATC) system**

Suppose Flight 101 wants to land at a particular terminal in the Airport. Then what the Flight Pilot will do is he will communicate with the ATC Mediator and say he wants to land Flight 101 at the particular airport terminal. Then what the ATC Mediator will do is, he will check whether any flight is there at that particular terminal or not. If no flight is there, then what the ATC mediator will do is it will send a message to Pilots of other flights saying that Flight 101 is going to land and you should not land at that particular terminal. Then the ATC mediator sends a message to the Flight 101 pilot and says you can land your flight at the particular airport terminal. Once the Flight 101 pilot gets the confirmation message from the ATC Mediator then he will land the flight at that particular terminal.



1. **Bridge Pattern :**

This pattern falls under the category of Structural pattern. This pattern divides huge class or business logics into separate classes hierarchies that can be implemented independently.This is useful when dealing with cross platform apps, supporting multiple types of database server or working with several API providers of a certain kind such as cloud platform, social networks etc.

**Real Time Example of Bridge Design Pattern:**

In the Bridge Design Pattern, there are two layers. The first layer is the Abstraction layer and the second layer is the Implementation Layer. If I do make any changes in the Implementation Layer, then it won’t affect the Abstraction Layer. Similarly, if I made any changes in the Abstraction Layer, then it won’t affect the Implementation layer. On the left-hand side, you can see the abstraction. Suppose, you want to turn on the TV or turn off the TV, then what you can do here is, you can use the Remote Control to turn On/Off the TV. The implementation will be done by the original TV implementer. So, in this case, Samsung TV or Sony TV will implement the turn On or turn Off functionality. So, the abstraction will use one of the implementers to turn on or turn off the TV.

