**Categorising Design Patterns:**

* **Creational Patterns: During creation of a class and instantiate them. Best templates 😊**
* **Structural patterns: Now we have many classes, then organize or structure classes**
* **Behavioural Patterns: How classes will interact with each other**

**C# corner refe: https://www.c-sharpcorner.com/UploadFile/bd5be5/design-patterns-in-net/**

**Define:It’s a template, A reusable solution to common software design problems.**

Class files of the codes are added from now on. We just call these form program files.

**Creational patterns**

**Factory Patterns: (Like factory for creating 😉)**

**A factory is an object which is used for creating other objects**

Think of a client and server. Server is a vehicle and clients asks what kind of Vehicle is needed.  
Client makes a call everytime and gets a response from server.  
  
So for all the new calls or new type of calls the client make to server, server should be in such a position to handle any such requests and give a response.  
It should not be the thing that client has to make a change at their end.   
This way both client and server is “Loosely coupled”. And no issue with client’s code recompile, redeploy.  
  
Eg:

Vehicle base class

Car class derived has a CreateVehicle method

Bike class derived has a CreateVehicle method

Normal case, client calls car class or bike class depending on the requirement, but then it has to add if else condition.

Factory method, create a new factory class that client calls and handle any such logic there. This way we can loosely couple it.

Eg: In controller we add some logic like factory before calling the service layer or the DB layer, making loosely coupled.

From Lehman’s point of view, we can say that a factory is a place where products are created. In order words, we can say that it is a centralized place for creating products. Later, based on the order received, the appropriate product is delivered by the factory. For example, a car factory can produce different types of cars. If you are ordering a car to the car factory, then based on your requirements or specifications, the factory will create the appropriate car and then delivered that car to you.

**Abstract Factory Pattern**

**The Abstract Factory Design Pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes**.

**Called the Factory of Factories.**

More of how we can Keep an abstract of the Multiple same Abstract factories.

And then How we call them. All these are covered in the code

**Prototype Pattern**

**Singleton Pattern**

Decides how we create objects of a class 🡪 Single instance of the class.

Eg: single instance of DB connection or config managers or loggers, db conn, caching

Consider eg of logger

We can also create a static variable counter so I will know how many times a class is instantiated.

There are now ways to make a class singleton:

1. Making constructor as private ()
2. Create instance of class in a variable as static
3. When users call the class, always use the same instance

But above way is not thread safe and so there might be more than one instance of a class.

By handling multi threading we can make class instantiate singleinstance

So when first instance is called, we wait for the thread execution. Then second instance is called and so it does not instantiate again.

\*Now locks are expensive when it comes to call them everytime, so what we do is make a “double check”. Before making new instance, we check a condition is the class is instantiated or not?

The following are the implementation guidelines for using the singleton design pattern in C#.

1. You need to declare a constructor that should be **private** and **parameterless**. This is required because it is not allowed the class to be instantiated from outside the class. It only instantiates from within the class.
2. The class should be declared as **sealed** which will ensure that it cannot be inherited. This is going to be useful when you are dealing with the nested class. We will discuss this scenario with an example in our upcoming article.
3. You need to create a **private static variable** that is going to hold a reference to the single created instance of the class if any.
4. You also need to create a **public static property/method** which will return the single-created instance of the singleton class. This method or property first check if an instance of the singleton class is available or not. If the singleton instance is available, then it returns that singleton instance otherwise it will create an instance and then return that instance.

**Disadvantage**:

Parallelism

Unit test could be tough.

**Builder Design Pattern**

When Complex objects/configurations, we make builder design pattern.

Basic eg is when you have a constructor with a lot of fields to be initialized, why not initialise it with a class params.

Also, we can add an interface based class so we can only pass the required class to another constructor which only requires some set of variables only.

Public NormalClass

{

private string \_name;

Private string \_id;

Public NormalClass(int Id, string Name)

{

\_id = Id, \_name = Name;

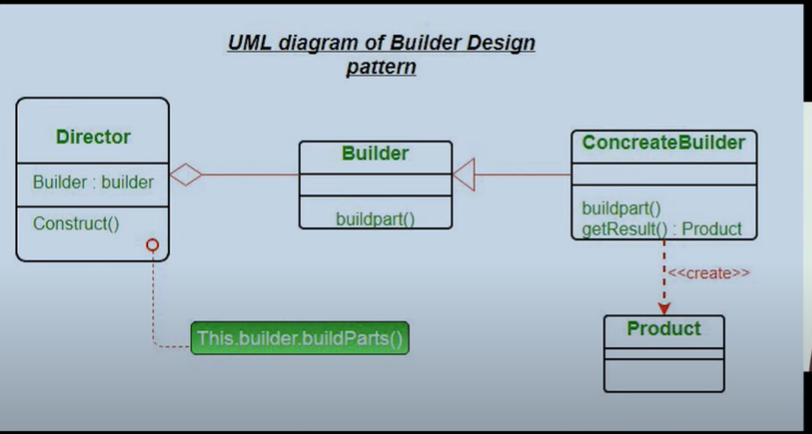
}

}

In the above example, observe we are passing params to constructor, we can increase in future and so will be tough. Need to implement Builder pattern. Solution is in the project.

Sometimes, we can also just pass a class into a constructor and make it easy but in real world problems, its better to make it very easy using design patterns.

It’s very convenient to use, when an entity has a lot of constructors and becomes very hard for external user of entity to decide which one to use.



Eg2: You have a HTML which you update again and again to add and modify it. But why add multiple add update, just add a HTML builder and then call it at once. This will reduce multiple lines and calls.

**Prototype Design Pattern**

**Prototype Design Pattern specifies the kind of objects to create using a prototypical instance, and create new objects by copying this prototype**“.

In C#, when we try to copy one object to another object using the assignment (=) operator, then both objects will share the same memory address. And the reason is the assignment operator (=) copies the reference, not the object except when there is a value type field. This operator will always copy the reference, not the actual object when working with the Reference type.

Eg;

Employee emp1 = new Employee**()**;

emp1.Name = "Anurag";

emp1.Department = "IT";

// Creating another Instance of Employee with Existing Employee Instance using Assignment Operator

// Both emp1 and emp2 share the same memory location as = Operator uses Call By Reference Mechanism

Employee emp2 = emp1;

// Changing the Name Property Value of emp2 instance,

// it will also change the Name Property Value of emp1 instance

emp2.Name = "Pranaya";

This issue of cloning can be resolved by Object Cloning technique:

To do so, C# provides one method called **MemberwiseClone** which will create a new complete copy of the object having a different memory

Timeline

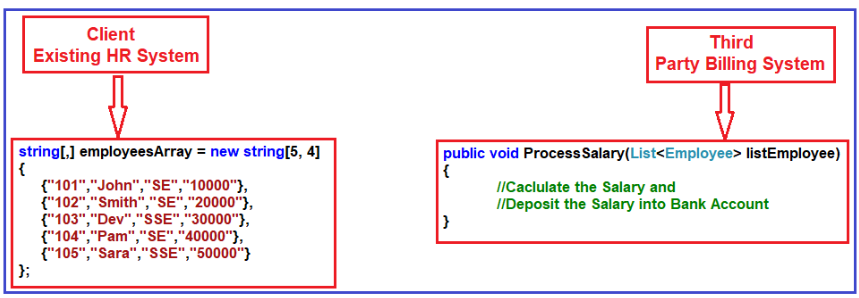
Description automatically generated

**Structural Design Patterns**

**Adapter Design Pattern**

The Adapter Design Pattern in C# involves a single class called Adapter which is responsible for communication between two independent or incompatible interfaces. So, in simple words, we can say that the Adapter Design Pattern helps two incompatible interfaces to work together.

* Mainly used when there are incompatibility issues between systems.
* Can be used to Allow a system to use classes of another system that is incompatible with it.



But if you look at the HR system, the employee information is stored in the form of a string array and the ProcessSalary method of the Third Party Billing System wants data in List<Employee>.

Now the adapter class is implemented like this:

Text

Description automatically generated

*You can see that there is an intermediate method which takes string types as the input. And then it will convert the strings type to the employees list types which is compatible with the destination Method type.*

*This is how we tackle the incompatibility issues.*

*THE ABOVE IS EG OF object ADAPTER DESIGN PATTERN. NOW WE CAN HANDLE IT WITH class ADAPTER DESIGN PATTERNS. In this we will inherit the Adaptee class to use the destination method directly to make it easy to call (instead of creating an object again).*

**Facade Design Pattern**

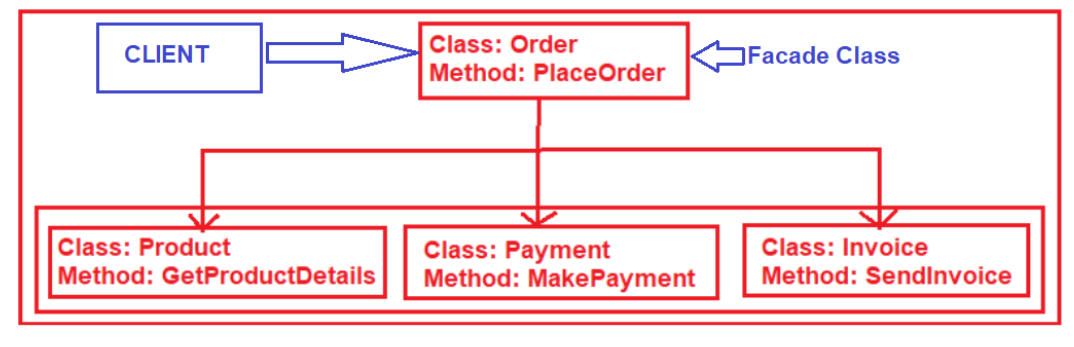
As per the GOF definition, **Facade Design Pattern states that you need to provide a unified interface to a set of interfaces in a subsystem. The Facade Design Pattern defines a higher-level interface that makes the subsystem easier to use**.

In simple words, we can say that the Facade Design Pattern is used to hide the complexities of a system and provides an easy-to-use interface to the client using which the client can access the system.

We need to use the Facade Design Pattern in C# when a system is very complex or very difficult to understand.

It also helps to move the unwanted dependencies to one place

So Façade class will help us reduce complexity and so it will take responsibility of doing the complex task in order. And client will call Façade class only to use the methods, then internally façade class handles the complexities.



Here is Façade class:

Text

Description automatically generated

**Decorator Pattern:**

OR **extending functionality dynamically.**

The Decorator Design Pattern in C# allows us to dynamically add new functionalities to an existing object without altering or modifying its structure

A decorator is an object that adds features to another object.

Diagram

Description automatically generated

Here Draw Method was just a normal object. Now to make any modification to this existing object, we create a decorator class and add new functionalities to it. Eg: SetColor()

When to use?

1. We want to add new functionalities to existing objects **dynamically**,
2. A class definition may be hidden or otherwise unavailable for subclasses.
3. Useful when Sealed classes

More code is in the Solution explorer.

Note: Decorator design pattern is like extending a features of a base functionality. We can also have another option to use interface extension feature.  
But difference here is that the code is called dynamically with the help of instance of base class.

More like you don’t want to modify class but you want to use modified class when it is dynamically calling.

**Behavioural Design Patterns**

**Iterator Design Pattern:**

This is more like iterating through each element in list or Arrays or any collection

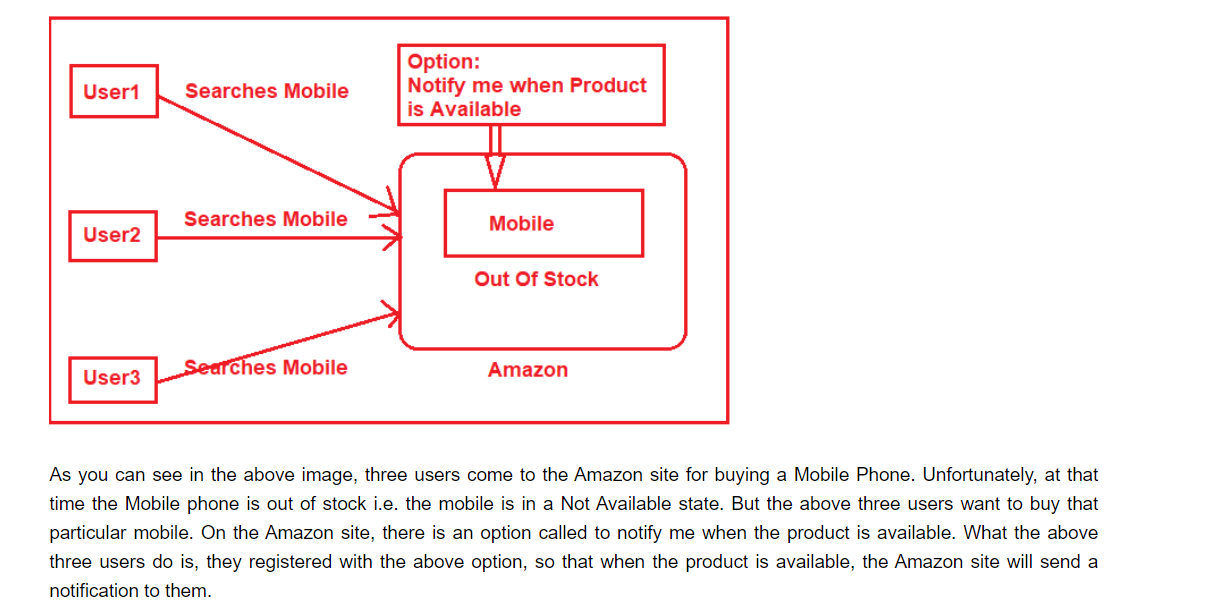
**Observer Design Pattern**

When you want to notify multiple systems when a change happened.

Also a publisher is publishing and then notifying all the Subscribers

Or, a subject is there and all the observers are observing it

According to GoF, the Observer design Pattern should “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically”.



**Bridge Design Pattern**

**Composite Design Pattern**

## ****Chain of Responsibility****