What is Singleton Design Pattern?

1. Ensures a class has only one instance and provides a global point of access to it.
2. A singleton is a class that only allows a single instance of itself to be created, and usually gives simple access to that instance.
3. Most commonly, singletons don't allow any parameters to be specified when creating the instance, since a second request of an instance with a different parameter could be problematic! (If the same instance should be accessed for all requests with the same parameter then the factory pattern is more appropriate.)

There are various ways to implement the Singleton Pattern in C#. The following are the common characteristics of a Singleton Pattern.

* A single constructor, that is private and parameterless.
* The class is sealed.
* A static variable that holds a reference to the single created instance, if any.
* A public static means of getting the reference to the single created instance, creating one if necessary.

Advantages of Singleton Pattern

The advantages of a Singleton Pattern are:

1. Singleton pattern can be implemented interfaces.
2. It can be also inherit from other classes.
3. It can be lazy loaded.
4. It has Static Initialization.
5. It can be extended into a factory pattern.
6. It helps to hide dependencies.
7. It provides a single point of access to a particular instance, so it is easy to maintain.

Disadvantages of Singleton Pattern

The disadvantages of a Singleton Pattern are:

1. Unit testing is more difficult (because it introduces a global state into an application).
2. This pattern reduces the potential for parallelism within a program, because to access the singleton in a multi-threaded system, an object must be serialized (by locking).

Singleton class vs. Static methods

The following conpares Singleton class vs. Static methods:

1. A Static Class cannot be extended whereas a singleton class can be extended.
2. A Static Class can still have instances (unwanted instances) whereas a singleton class prevents it.
3. A Static Class cannot be initialized with a STATE (parameter), whereas a singleton class can be.
4. A Static class is loaded automatically by the CLR when the program or namespace containing the class is loaded.

How to Implement Singleton Pattern in your code

There are many way to implement a Singleton Pattern in C#.

1. No Thread Safe Singleton.
2. Thread-Safety Singleton.
3. Thread-Safety Singleton using Double-Check Locking.
4. Thread-Safe Singleton without using locks and no lazy instantiation.
5. Fully lazy instantiation.
6. Using .NET 4's Lazy<T> type.

**1. No Thread Safe Singleton**

Explanation of the following code:

1. The following code is not thread-safe.
2. Two different threads could both have evaluated the test (if instance == null) and found it to be true, then both creates instances, which violates the singleton pattern.
3. Note that in fact the instance may already have been created before the expression is evaluated, but the memory model doesn't guarantee that the new value of instance will be seen by other threads unless suitable memory barriers have been passed.
4. **public** **sealed** **class** Singleton
5. {
6. //Private Constructor.
7. **private** Singleton()
8. {
9. }
10. **private** **static** Singleton instance = **null**;
11. **public** **static** Singleton Instance
12. {
13. **get**
14. {
15. **if** (instance == **null**)
16. {
17. instance = **new** Singleton();
18. }
19. **return** instance;
20. }
21. }
22. }

**2. Thread Safety Singleton**

Explanation of the following code:

1. This implementation is thread-safe.
2. In the following code, the thread is locked on a shared object and checks whether an instance has been created or not.
3. This takes care of the memory barrier issue and ensures that only one thread will create an instance.
4. For example: Since only one thread can be in that part of the code at a time, by the time the second thread enters it, the first thread will have created the instance, so the expression will evaluate to false.
5. The biggest problem with this is performance; performance suffers since a lock is required every time an instance is requested.
6. **public** **sealed** **class** Singleton
7. {
8. Singleton()
9. {
10. }
11. **private** **static** **readonly** **object** padlock = **new** **object**();
12. **private** **static** Singleton instance = **null**;
13. **public** **static** Singleton Instance
14. {
15. **get**
16. {
17. **lock** (padlock)
18. {
19. **if** (instance == **null**)
20. {
21. instance = **new** Singleton();
22. }
23. **return** instance;
24. }
25. }
26. }
27. }

**3. Thread Safety Singleton using Double Check Locking**

Explanation of the following code:

1. In the following code, the thread is locked on a shared object and checks whether an instance has been created or not with double checking.
2. **public** **sealed** **class** Singleton
3. {
4. Singleton()
5. {
6. }
7. **private** **static** **readonly** **object** padlock = **new** **object**();
8. **private** **static** Singleton instance = **null**;
9. **public** **static** Singleton Instance
10. {
11. **get**
12. {
13. **if** (instance == **null**)
14. {
15. **lock** (padlock)
16. {
17. **if** (instance == **null**)
18. {
19. instance = **new** Singleton();
20. }
21. }
22. }
23. **return** instance;
24. }
25. }
26. }

**Façade:**

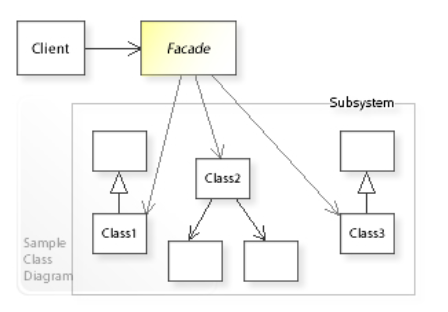
Facade is a **structural design pattern** that provides a simplified interface to a library, a framework, or any other complex set of classes.

If we try to understand this in simpler terms, then we can say that a room is a façade and just by looking at it from outside the door, one cannot predict what is inside the room and how the room is structured from inside. Thus, Façade is a general term for simplifying the outward appearance of a complex or large system.

In software terms, Facade pattern hides the complexities of the systems and provides a simple interface to the clients.

This pattern involves one wrapper class which contains a set of methods available for the client. This pattern is particularly used when a system is very complex or difficult to understand and when the system has multiple subsystems.

Let’s see the below UML diagram,

  
Image source: Wikipedia

Here, we can see that the client is calling the Façade class which interacts with multiple subsystems making it easier for the client to interact with them.

However, it is possible that façade may provide limited functionality in comparison to working with the subsystem directly, but it should include all those features which are actually required by the client.

For example, when someone calls the restaurant, suppose, for ordering pizza or some other food, then the operator on behalf of the restaurant gives the voice interface which is actually the façade for their customers.

Customers place their orders just by talking to the operator and they don’t need to bother about how they will prepare the pizza, what all operations will they perform, on what temperature they will cook, etc.

Similarly, in our code sample, we can see that the client is using the restaurant façade class to order pizza and bread of different types without directly interacting with the subclasses.

Now, it's time to dive into the real code.

This is the interface specific to the pizza.

1. **public** **interface** IPizza {
2. **void** GetVegPizza();
3. **void** GetNonVegPizza();
4. }

This is a pizza provider class which will get pizza for their clients. Here methods can have other private methods which client is not bothered about.

1. **public** **class** PizzaProvider: IPizza {
2. **public** **void** GetNonVegPizza() {
3. GetNonVegToppings();
4. Console.WriteLine("Getting Non Veg Pizza.");
5. }
6. **public** **void** GetVegPizza() {
7. Console.WriteLine("Getting Veg Pizza.");
8. }
9. **private** **void** GetNonVegToppings() {
10. Console.WriteLine("Getting Non Veg Pizza Toppings.");
11. }
12. }

Similarly, this is the interface specific for the bread.

1. **public** **interface** IBread {
2. **void** GetGarlicBread();
3. **void** GetCheesyGarlicBread();
4. }

And this is a bread provider class.

1. **public** **class** BreadProvider: IBread {
2. **public** **void** GetGarlicBread() {
3. Console.WriteLine("Getting Garlic Bread.");
4. }
5. **public** **void** GetCheesyGarlicBread() {
6. GetCheese();
7. Console.WriteLine("Getting Cheesy Garlic Bread.");
8. }
9. **private** **void** GetCheese() {
10. Console.WriteLine("Getting Cheese.");
11. }
12. }

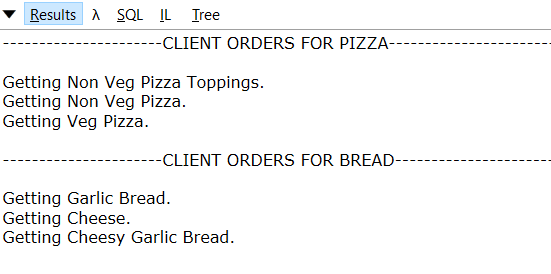
Below is the restaurant façade class, which will be used by the client to order different pizzas or breads.

1. **public** **class** RestaurantFacade {
2. **private** IPizza \_PizzaProvider;
3. **private** IBread \_BreadProvider;
4. **public** RestaurantFacade() {
5. \_PizzaProvider = **new** PizzaProvider();
6. \_BreadProvider = **new** BreadProvider();
7. }
8. **public** **void** GetNonVegPizza() {
9. \_PizzaProvider.GetNonVegPizza();
10. }
11. **public** **void** GetVegPizza() {
12. \_PizzaProvider.GetVegPizza();
13. }
14. **public** **void** GetGarlicBread() {
15. \_BreadProvider.GetGarlicBread();
16. }
17. **public** **void** GetCheesyGarlicBread() {
18. \_BreadProvider.GetCheesyGarlicBread();
19. }
20. }

Finally, below is the main method of our program,

1. **void** Main() {
2. Console.WriteLine("----------------------CLIENT ORDERS FOR PIZZA----------------------------\n");
3. var facadeForClient = **new** RestaurantFacade();
4. facadeForClient.GetNonVegPizza();
5. facadeForClient.GetVegPizza();
6. Console.WriteLine("\n----------------------CLIENT ORDERS FOR BREAD----------------------------\n");
7. facadeForClient.GetGarlicBread();
8. facadeForClient.GetCheesyGarlicBread();
9. }

**OUTPUT**



Now, let’s see when we should use this pattern and what could be the real-life scenarios:

**WHEN TO USE THIS PATTERN**

Use this pattern to simplify the problem when there are multiple complex subsystems and interacting with them individually is really difficult/cumbersome.

**REAL LIFE USE CASE**

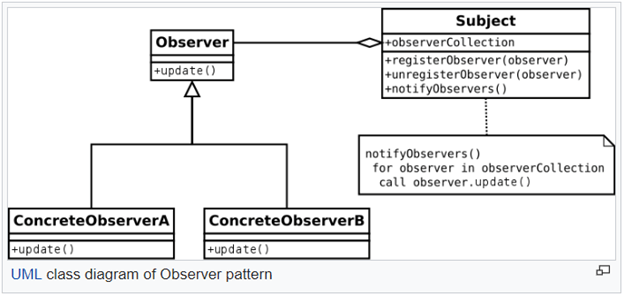
* The shopkeeper is a façade for all the items in the shop.
* Online travel portal is a façade for their customers for different holiday/travel packages.
* Customer care is a façade for their customers for different services.

**Observer Pattern:**

This pattern falls under the category of behavioral pattern. As the name suggests, it is based on the principle where one object observes another object.

So, in simple words, this pattern defines a one-to-many relationship between subject and observer objects so that if the subject changes its state, then all of the dependent observer objects will be notified of the change.

Let’s understand it in more details using the UML diagram, as shown below.

  
Image source: Wikipedia

**For example**

Consider a technical author like we have in C# Corner, who writes blogs, articles etc. So, if someone likes reading articles from a specific author, one can follow/subscribe the author for future articles. This is a real-life scenario of the Observer pattern.

In this case, all the subscribers who want to subscribe to the author are Observer Objects and the author is a Subject.

Now, let’s dive into the actual code to see how it works.

This is the interface which will be implemented by the subject.

1. **public** **interface** ISubject {
2. **void** registerObserver(Observer observer);
3. **void** unregisterObserver(Observer observer);
4. **void** notifyObservers();
5. }

This is the Subject class and as per our example mentioned above, the author will be the Subject. So here, we are assuming that the author has already written 1 article and those who will like the article can subscribe or unsubscribe (if they have already subscribed) accordingly.

Now, whenever the subject changes its state, all of its observers will be notified.

1. **public** **class** Subject: ISubject {
2. **private** List < Observer > Observers = **new** List < Observer > ();
3. **private** **int** articlesCount = 1;
4. **public** **int** Articles {
5. **get** {
6. **return** articlesCount;
7. }
8. **set** {
9. **if** (value > articlesCount) {
10. articlesCount++;
11. notifyObservers();
12. }
13. }
14. }
15. **public** **void** registerObserver(Observer observer) {
16. Observers.Add(observer);
17. }
18. **public** **void** unregisterObserver(Observer observer) {
19. Observers.Remove(observer);
20. }
21. **public** **void** notifyObservers() {
22. **foreach**(var observer **in** Observers) {
23. observer.Update();
24. }
25. }
26. }

This is the interface which will be implemented by all the Observer objects. As per our example, all the subscribers/followers for the author will be the Observer objects.

1. **public** **interface** IObserver {
2. **void** Update();
3. }

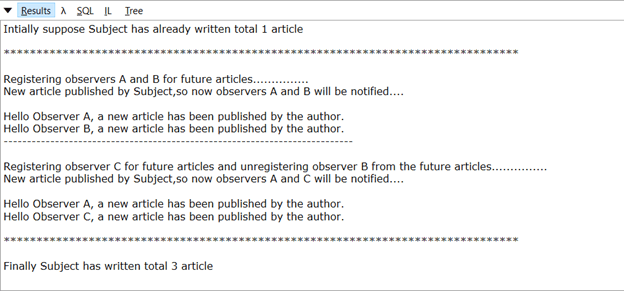
This is the Observer class which can be used to create different observers.

1. **public** **class** Observer: IObserver {
2. **public** **string** ObserverName;
3. **public** Observer(**string** name) {
4. ObserverName = name;
5. }
6. **public** **void** Update() {
7. //Observer can update his system accordingly
8. Console.WriteLine("Hello " + ObserverName + ", a new article has been published by the author.");
9. }
10. }

Finally, this is the main method which can be used for running this program.

1. **void** Main() {
2. var subject = **new** Subject();
3. var observerA = **new** Observer("Observer A");
4. var observerB = **new** Observer("Observer B");
5. var observerC = **new** Observer("Observer C");
6. Console.WriteLine("Intially suppose Subject has already written total " + subject.Articles + " article");
7. Console.WriteLine("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");
8. Console.WriteLine("Registering observers A and B for future articles...............");
9. subject.registerObserver(observerA);
10. subject.registerObserver(observerB);
11. Console.WriteLine("New article published by Subject,so now observers A and B will be notified....\n");
12. subject.Articles++;
13. Console.WriteLine("---------------------------------------------------------------------------\n");
14. Console.WriteLine("Registering observer C for future articles and unregistering observer B from the future articles...............");
15. subject.registerObserver(observerC);
16. subject.unregisterObserver(observerB);
17. Console.WriteLine("New article published by Subject,so now observers A and C will be notified....\n");
18. subject.Articles++;
19. Console.WriteLine("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");
20. Console.WriteLine("Finally Subject has written total " + subject.Articles + " article \n");
21. }

**OUTPUT**



**When should we use this**

We should use it when multiple objects are dependent on the state of one object.

**Some common Use Cases**

* Following someone on Instagram/Twitter and other such platforms.
* App users gets notified for the updates.
* Email Subscriptions and many more.

**Inversion of Control:**

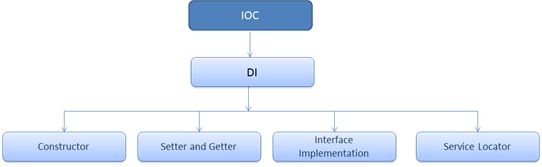
public class clsDAL  
{  
    private clsSqlServer \_sql;   
   
    public clsDAL()  
    {  
        \_sql = new clsSqlServer();   
    }  
}

Consider the above example, where we have a DAL class. The default constructor of the DAL class creates an object of the SqlServer class. That means the DAL Class is responsible for creating an object of the SqlServer class. So there is tight coupling between the DAL class and SqlServer class.  
  
Three main problems in the above code:

1. The DAL class is responsible for creating an object of the SqlServer class.
2. The SqlServer class is directly referenced in the DAL class.
3. The DAL class should be aware of the SqlServer class type.

Now we understand the problem. Now let's discuss the solution. The solution is to shift the object creation part from this class. We need to shift the object creation control from here i.e. Inversion of control.  
  
**Principles of IOC**

1. Main classes aggregating other classes should not depend on the direct implementation of the aggregated classes. Both the classes should depend on abstraction.
2. Abstraction should not depend on details, details should depend on abstraction.

**What is Dependency Injection?**  
  
Inversion of control is implemented by dependency injection because Inversion of control is a principle and dependency injection is a way of implementing IOC.  
  
**Ways of implementing IOC**  
  
Now let's discuss each method with an example.  
  
**Constructor Methodology**

public class clsDAL

{

    private ISql \_sql;    public clsDAL(ISql obj)

    {

         \_sql = obj;

    }

}

In this methodology we pass an object of SQL into the DAL class. Here in the above code you can see that there is a parameterized constructor in the DAL class. And the parameterized constructor accepts an object of SQL. The DAL class is not responsible for creating an object of SQL in this case. So there is no tight coupling between these classes. This method is not useful for the client who only can use a default constructor.  
  
**Setter and Getter**

public class clsDAL

{

    private ISql \_sql;    public Isql Sql

    {

        set

        {

             \_sql = value;

        }

    }

}

In this method we expose an object of SQL through the get/set methods of the DAL class. But it violates the encapsulation rule of OOP. Encapsulation means hiding internal details of an object. So here rather than hiding an object, we are exposing an object.  
  
**Interface Implementation**

interface ISqlDI

{

    void setConnection(ISql obj);

}

public class clsDAL : ISqlDI

{

    private ISql \_sql;    public void setConnection(ISql obj)

    {

        \_sql = obj;

    }

}

In the preceding code we have implemented an interface which has a setConnection method which sets the SQL object. And the DAL class implements a SQL interface. So with the help of the setConnection method the client can inject a SQL object in the DAL class.  
  
**Server Locator**  
  
static class LocateConnection  
{  
    public static ISql getConnection() { }  
}

 interface clsDAL

{

    private ISql \_sql;

    public clsDAL()

    {

        \_sql = LocateConnection.getConnection();

    }

}

In this method we create a static class and a static method inside this class. The DAL class calls this static method from its default constructor. So in this way the SQL object is injected into the DAL class.

## Dependency Injection in C#

Dependency Injection (DI) is a software design pattern. It allows us to develop loosely-coupled code. The intent of Dependency Injection is to make code maintainable. Dependency Injection helps to reduce the tight coupling among software components. Dependency Injection reduces the hard-coded dependencies among your classes by injecting those dependencies at run time instead of design time technically. This article explains how to implement Dependency Injection in C# and .NET code.

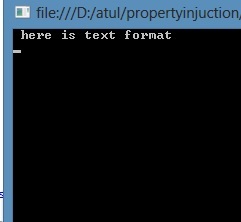
We have the following ways to implement Dependency Injection.

## Constructor Injection

This is the most commonly used dependency pattern in Object Oriented Programming. The constructor injection normally has only one parameterized constructor, so in this constructor dependency there is no default constructor and we need to pass the specified value at the time of object creation. We can use the injection component anywhere within the class. It addresses the most common scenario where a class requires one or more dependencies.

The following is an example:

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Threading.Tasks;
7. **namespace** propertyinjuction
8. {
9. **public** **interface** text
10. {
11. **void** print();
12. }
13. **class** format : text
14. {
15. **public** **void** print()
16. {
17. Console.WriteLine(" here is text format");
18. }
19. }
20. // constructor injection
21. **public** **class** constructorinjection
22. {
23. **private** text \_text;
24. **public** constructorinjection(text t1)
25. {
26. **this**.\_text = t1;
27. }
28. **public** **void** print()
29. {
30. \_text.print();
31. }
32. }
33. **class** constructor
34. {
35. **static** **void** Main(**string**[] args)
36. {
37. constructorinjection cs = **new** constructorinjection(**new** format());
38. cs.print();
39. Console.ReadKey();
40. }
41. }
42. }



By passing the services that implemented the text interface the builder assembled the dependencies.

## Property Injection

We use constructor injection, but there are some cases where I need a parameter-less constructor so we need to use property injection.

 The following is an example:

1. **public** **interface** INofificationAction
2. {
3. **void** ActOnNotification(**string** message);
4. }
5. **class** atul     {
6. INofificationAction task = **null**;
7. **public** **void** notify(INofificationAction  at ,**string** messages)
8. {
9. **this**.task = at;
10. task.ActOnNotification(messages);
11. }
12. }
13. **class** EventLogWriter : INofificationAction
14. {
15. **public** **void** ActOnNotification(**string** message)
16. {
17. // Write to event log here
18. }
19. }
20. **class** Program
21. {
22. **static** **void** Main(**string**[] args)
23. {
24. //services srv = new services();
25. //other oth = new other();
26. //oth.run();
27. //Console.WriteLine();
28. EventLogWriter elw = **new** EventLogWriter();
29. atul at = **new** atul();
30. at.notify(elw, "to logg");
31. Console.ReadKey();
32. }
33. }

You cannot control when the dependency is set at all, it can be changed at any point in the object's lifetime.

## Method Injection

In method injection we need to pass the dependency in the method only. The entire class does not need the dependency, just the one method. I have a class with a method that has a dependency. I do not want to use constructor injection because then I would be creating the dependent object every time this class is instantiated and most of the methods do not need this dependent object.

The following is an example:

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Threading.Tasks;
7. **namespace** propertyinjuction
8. {
9. **public** **interface** Iset
10. {
11. **void** print();
12. }
13. **public** **class** servic : Iset
14. {
15. **public** **void** print()
16. {
17. Console.WriteLine("print........");
18. }
19. }
20. **public** **class** client
21. {
22. **private** Iset \_set;
23. **public** **void** run(Iset serv)
24. {
25. **this**.\_set = serv;
26. Console.WriteLine("start");
27. **this**.\_set.print();
28. }
29. }
30. **class** method
31. {
32. **public** **static** **void** Main()
33. {
34. client cn = **new** client();
35. cn.run(**new** servic());
36. Console.ReadKey();
37. }
38. }
39. }

# **Abstract Factory Design Pattern In C#**

Abstract Factory pattern also falls under Creational Pattern of Gang of Four (GoF) Design Patterns.

**Background**

Before talking about Abstract Factory pattern, I just want to share a little about "Gang of Four (GoF)" to which the Abstract Factory pattern belongs.

## Who are the Gang of Four?

The Gang of Four are the authors of the book, "Design Patterns: Elements of Reusable Object-Oriented Software". This important book describes various development techniques and pitfalls in addition to providing 23 object-oriented programming design patterns. The four authors are Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

Now, let's move on to the Abstract Factory design pattern.

In this article, I have tried to cover the concept of Abstract Factory design pattern and the ways of implementing Abstract Factory design pattern.

**What is it?**

An interface for creating families of related or dependent objects without specifying their concrete classes. We can say it is just an object maker which can create more than one type of object.

The object it produces is known to the client only by that object's interface, not by the object's actual concrete implementation.

**When to use it?**

We use it when we have a requirement to create a set of related objects, or dependent objects which must be used together as families of objects. Concrete classes should be decoupled from clients.

## How does it differ from Factory Method?

First of all, both of them fall under Creational category and it means both will solve the problem relating to object creation. Factory Method and Abstract Factory design pattern are about creating objects.

## Factory Method Design Pattern

Here, we define an interface which will expose a method which will create objects for us. Return type of that method is never a concrete type; rather, it will be some interface (or may be an abstract class).

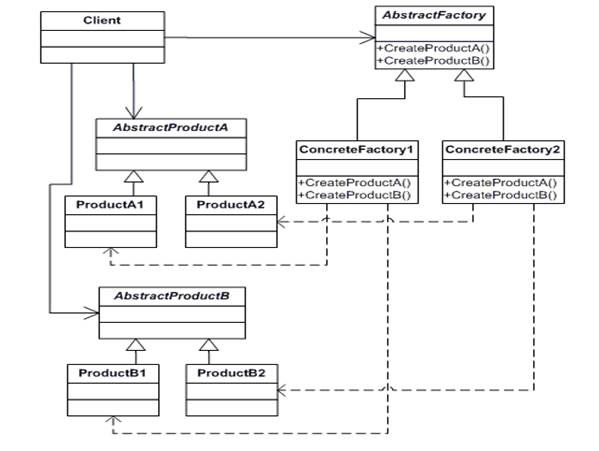
* Creates object through inheritance
* Produce only one product
* Implements code in the abstract creator that makes use of the concrete type that sub class produces

## Abstract Factory Design Pattern

Here, we define an interface which will create families of related or dependent objects. In simple words, interface will expose multiple methods each of which will create some object. Again, here method return types will be generic interfaces. All these objects will together become part of some important functionality.

* Creates object through composition
* Produce families of products
* Concrete factories implements factory method to create product

## UML Class Diagram

****

The classes and objects participating in the above UML class diagram are as follow.

1. AbstractFactory  
   This is an interface for operations which is used to create abstract product.
2. ConcreteFactory  
   This is a class which implements the AbstractFactory interface operations to create concrete products.
3. AbstractProduct  
   This declares an interface for a type of product object
4. Product  
   This defines a product object to be created by the corresponding concrete factory also implements the AbstractProduct interface
5. Client  
   This is a class which uses AbstractFactory and AbstractProduct interfaces to create a family of related objects.

Now, let’s understand this with a real world example,

The example here has an implementation of an Abstract Factory as an Interface IMobilePhone that has methods that can create a Smart Phone object and a Normal Phone object. The client codes against IMobilePhone and gets ISmartPhone and INormalPhone interfaces.

In case of "Nokia", it creates a family of Nokia objects (SmartPhone and NormalPhone) and in case of "Samsung", creates a family of Samsung objects (SmartPhone and NormalPhone).

The client doesn't care which object (Nokia SmartPhone and NormalPhone or Samsung SmartPhone and NormalPhone), IMobilePhone interface returns as it codes against ISmartPhone and INormalPhone interface.

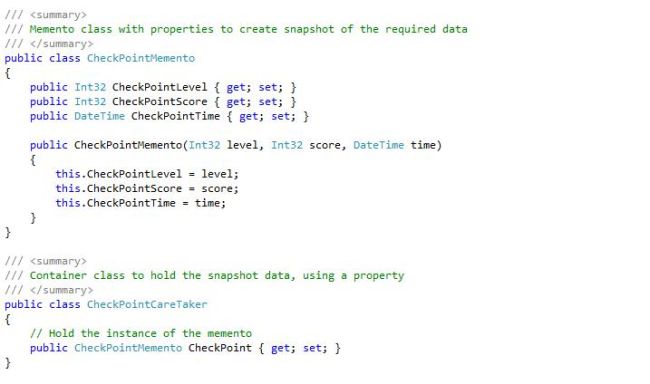
# **Memento Design Pattern Using C#:**

**What is the Memento Pattern?**  
According to the GoF's definition, what this pattern does is:  
  
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.  
  
**A Real world example**  
Let's use an example of a computer game, where we have a player who will need to cross levels 1 to 5 to complete the game and will score different scores at each level. We will create a checkpoint for this player at level 1, with some initial score at a specific instance of time, and store a snapshot of its data. Then we will update the score, level and time of the player. Now with some business rule, say the player looses a life, he will again need to start from that checkpoint. So when he starts again from the checkpoint, we will restore the snapshot data and proceed further accordingly. In order to do this, we will divide the system into the following components.

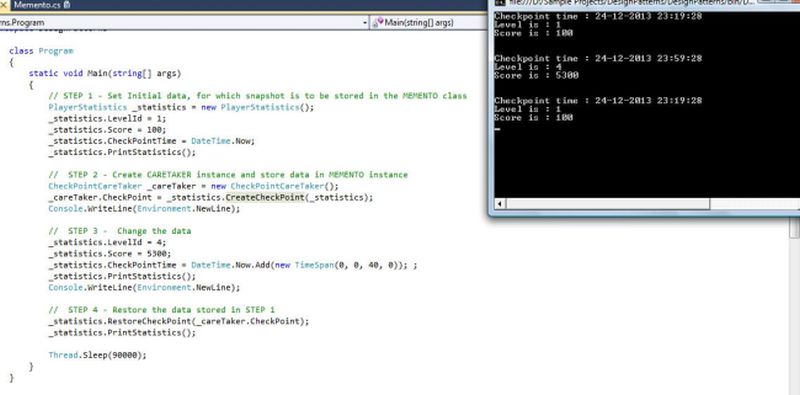
1. **Originator or MainClass**This is the main class of the system that maintains the data of the player, in other words its level, score and so on and for which we want to create a snapshot of the data. In our case, it will be a PlayerStatistics class, with 3 properties named Level, Score and CheckPointTime. These are the properties or the data of the player, for which we want to create a snapshot.
2. **Memento**This class is used to store the snapshot of the Originator class data. In our case, it will be the CheckPointMemento class, only with properties, for which we want to store the data, in other words for each of the properties that we want to have in the snapshot, we create a property in the memento class.
3. **CareTakerClass**This is like a container class, that holds the Memento class instance, that further holds the snapshot of the original data. In our case, it will be CheckPointCareTaker, and will hold the memento instance using a property defined in it.

So, to start with, we create the following classes:

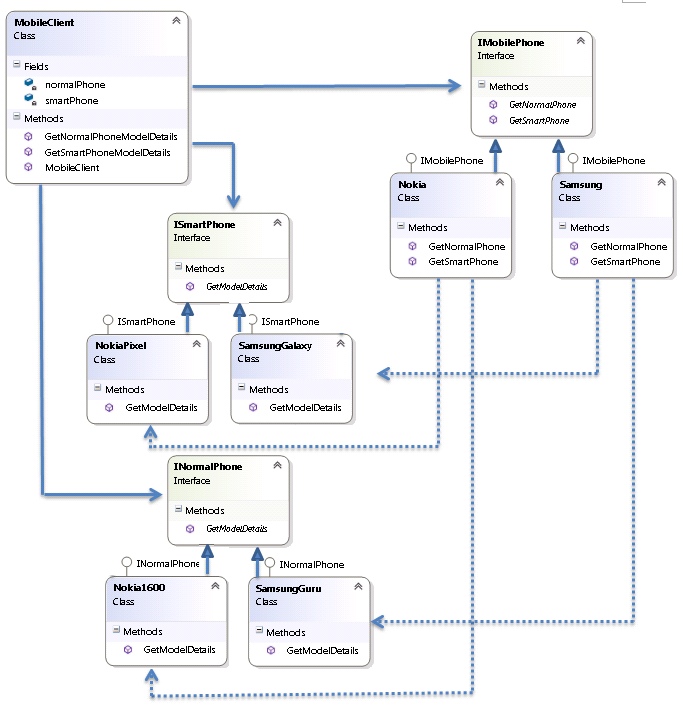
* A PlayerStatistics class and its properties.
* A CheckPointMemento class, with the properties for which we want to store the snapshot data of the PlayerStatistics class.
* A CheckPointCareTakerclass, that will act as a container for CheckPointMemento class.

See the classes below. These are the memento and caretaker classes.  
  
  
  
Now we will create the main class PlayerStatistics and an important point to note here is that the PlayerStatistics class will have two methods, apart from the properties to hold the player data. These methods are :

1. **CreateCheckPoint()**This method will be used to return the instance of the memento that is storing the data of which we would like to create a snapshot. The instance returned by this method will be added to the container or the CheckPointCareTaker class, by setting the property.
2. **RestoreCheckPoint()**This method will receive the memento class instance or you can say the snapshot data that we added at the checkpoint. The data received will then be returned to the original properties.

See the complete code for the Main/originator class, with the properties and the two methods explained above.  
  
  
  
Now we have set up the initial code. It's time to write the client code. So we start by setting some initial data for the player. Then we will store the data in a memento instance and add it to the caretaker container class. Further we will change the data and then finally retrieve the stored data from the memento instance. So our code will be like the following:  
  
  
  
**How is it working ?**  
Let's explain the client code now.

* In Step 1, we simply set the properties of the player data and print them.
* In Step 2, we call the CreateCheckPoint() method in the main class, and generate the instance of the memento class type. This instance is then assigned to the CheckPointCareTakerclass, that acts as a container here to hold the snapshot data.
* In Step 3, we simply update the data and print it, like we did in Step 1.
* In Step 4, we basically perform the undo operation and retrieve the original data that we had at the start. This is done by calling the RestoreCheckPoint method in the main class and passing it the data from the container or caretaker class. The data received is then set back to the properties and printed again.



**Who is what?**

The classes and objects participating in the above class diagram can be identified as shown below.

* AbstractFactory- IMobilePhone
* ConcreteFactory - Nokia, Samsung
* AbstractProduct- ISmartPhone, INormalPhone
* Product- NokiaPixel, Nokia1600, SamsungGalaxy, SamsungGuru
* Client- MobileClient

**Here are the code blocks for each participant**

1. AbstractFactory  
   1. namespace AbstractFactoryDesignPatternInCSharp
   2. {
   3. /// <summary>
   4. /// The 'AbstractFactory' interface.
   5. /// </summary>
   6. **interface** IMobilePhone
   7. {
   8. ISmartPhone GetSmartPhone();
   9. INormalPhone GetNormalPhone();
   10. }
   11. }
2. ConcreteFactoryNokia
   1. namespace AbstractFactoryDesignPatternInCSharp
   2. {
   3. /// <summary>
   4. /// The 'ConcreteFactory1' class.
   5. /// </summary>
   6. **class** Nokia : IMobilePhone
   7. {
   8. **public** ISmartPhone GetSmartPhone()
   9. {
   10. **return** **new** NokiaPixel();
   11. }
   13. **public** INormalPhone GetNormalPhone()
   14. {
   15. **return** **new** Nokia1600();
   16. }
   17. }
   18. }

Samsung

* 1. namespace AbstractFactoryDesignPatternInCSharp
  2. {
  3. /// <summary>
  4. /// The 'ConcreteFactory2' class.
  5. /// </summary>
  6. **class** Samsung : IMobilePhone
  7. {
  8. **public** ISmartPhone GetSmartPhone()
  9. {
  10. **return** **new** SamsungGalaxy();
  11. }
  13. **public** INormalPhone GetNormalPhone()
  14. {
  15. **return** **new** SamsungGuru();
  16. }
  17. }
  18. }

1. AbstractProductISmartPhone
   1. namespace AbstractFactoryDesignPatternInCSharp
   2. {
   3. /// <summary>
   4. /// The 'AbstractProductA' interface
   5. /// </summary>
   6. **interface** ISmartPhone
   7. {
   8. string GetModelDetails();
   9. }
   10. }

INormalPhone

* 1. namespace AbstractFactoryDesignPatternInCSharp
  2. {
  3. /// <summary>
  4. /// The 'AbstractProductB' interface
  5. /// </summary>
  6. **interface** INormalPhone
  7. {
  8. string GetModelDetails();
  9. }
  10. }

1. ProductNokiaPixel
   1. namespace AbstractFactoryDesignPatternInCSharp
   2. {
   3. /// <summary>
   4. /// The 'ProductA1' class
   5. /// </summary>
   6. **class** NokiaPixel : ISmartPhone
   7. {
   8. **public** string GetModelDetails()
   9. {
   10. **return** "Model: Nokia Pixel\nRAM: 3GB\nCamera: 8MP\n";
   11. }
   12. }
   13. }

SamsungGalaxy

* 1. **namespace** AbstractFactoryDesignPatternInCSharp
  2. {
  3. /// <summary>
  4. /// The 'ProductA2' class
  5. /// </summary>
  6. **class** SamsungGalaxy : ISmartPhone
  7. {
  8. **public** **string** GetModelDetails()
  9. {
  10. **return** "Model: Samsung Galaxy\nRAM: 2GB\nCamera: 13MP\n";
  11. }
  12. }
  13. }

Nokia1600

* 1. namespace AbstractFactoryDesignPatternInCSharp
  2. {
  3. /// <summary>
  4. /// The 'ProductB1' class
  5. /// </summary>
  6. **class** Nokia1600 : INormalPhone
  7. {
  8. **public** string GetModelDetails()
  9. {
  10. **return** "Model: Nokia 1600\nRAM: NA\nCamera: NA\n";
  11. }
  12. }
  13. }

SamsungGuru

* 1. namespace AbstractFactoryDesignPatternInCSharp
  2. {
  3. /// <summary>
  4. /// The 'ProductB2' class
  5. /// </summary>
  6. **class** SamsungGuru : INormalPhone
  7. {
  8. **public** string GetModelDetails()
  9. {
  10. **return** "Model: Samsung Guru\nRAM: NA\nCamera: NA\n";
  11. }
  12. }
  13. }

1. Client
   1. namespace AbstractFactoryDesignPatternInCSharp
   2. {
   3. /// <summary>
   4. /// The 'Client' class
   5. /// </summary>
   6. **class** MobileClient
   7. {
   8. ISmartPhone smartPhone;
   9. INormalPhone normalPhone;
   11. **public** Client(IMobilePhone factory)
   12. {
   13. smartPhone = factory.GetSmartPhone();
   14. normalPhone = factory.GetNormalPhone();
   15. }
   17. **public** string GetSmartPhoneModelDetails()
   18. {
   19. **return** smartPhone.GetModelDetails();
   20. }
   22. **public** string GetNormalPhoneModelDetails()
   23. {
   24. **return** normalPhone.GetModelDetails();
   25. }
   26. }
   27. }

Factory Pattern Client Demo

* 1. using System;
  3. namespace AbstractFactoryDesignPatternInCSharp
  4. {
  5. /// <summary>
  6. /// Abstract Factory Pattern Demo
  7. /// </summary>
  8. **class** Program
  9. {
  10. **static** **void** Main()
  11. {
  12. IMobilePhone nokiaMobilePhone = **new** Nokia();
  13. MobileClient nokiaClient = **new** MobileClient(nokiaMobilePhone);
  15. Console.WriteLine("\*\*\*\*\*\*\*\*\* NOKIA \*\*\*\*\*\*\*\*\*\*");
  16. Console.WriteLine(nokiaClient.GetSmartPhoneModelDetails());
  17. Console.WriteLine(nokiaClient.GetNormalPhoneModelDetails());
  19. IMobilePhone samsungMobilePhone = **new** Samsung();
  20. MobileClient samsungClient = **new** MobileClient(samsungMobilePhone);
  22. Console.WriteLine("\*\*\*\*\*\*\* SAMSUNG \*\*\*\*\*\*\*\*\*\*");
  23. Console.WriteLine(samsungClient.GetSmartPhoneModelDetails());
  24. Console.WriteLine(samsungClient.GetNormalPhoneModelDetails());
  26. Console.ReadKey();
  27. }
  28. }
  29. }

In Object Oriented Programming (OOP), SOLID is an acronym, introduced by Michael Feathers, for five design principles used to make software design more understandable, flexible, and maintainable. These principles are a subset of many principles promoted by Robert C. Martin.

## SOLID Principles

There are five SOLID principles:

1. Single Responsibility Principle (SRP)
2. Open Closed Principle (OCP)
3. Liskov Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

## Single Responsibility Principle (SRP)

**Definition:** A class should have only one reason to change.

In layman terminology, this means that a class should not be loaded with multiple responsibilities and a single responsibility should not be spread across multiple classes or mixed with other responsibilities. The reason is that more changes requested in the future, the more changes the class need to apply.

### **Understanding**

Single Responsibility Principle is one of the five SOLID principles which guide developers as they write code or design an application.

In simple terms, a module or class should have a very small piece of responsibility in the entire application. Or as it states, a class/module should have not more than one reason to change.

If a class has only a single responsibility, it is likely to be very robust. It’s easy to verify its working as per logic defined. And it’s easy to change in class as it has single responsibility.

The Single Responsibility Principle provides another benefit. Classes, software components and modules that have only one responsibility are much easier to explain, implement and understand than ones that give a solution for everything.

This also reduces number of bugs and improves development speed and most importantly makes developer’s life lot easier.

### **Implementation**

Let’s take a scenario of Garage service station functionality. It has 3 main functions; open gate, close gate and performing service. Below example violates SRP principle. The code below, violates SRP principle as it mixes open gate and close gate responsibilities with the main function of servicing of vehicle.

1. Public class GarageStation
2. {
3. Public void DoOpenGate()
4. {
5. *//Open the gate functinality*
6. }
7. Public void PerformService(Vehicle vehicle)
8. {
9. *//Check if garage is opened*
10. *//finish the vehicle service*
11. }
12. Public void DoCloseGate()
13. {
14. *//Close the gate functinality*
15. }
16. }

We can correctly apply SRP by refactoring of above code by introducing interface. A new interface called IGarageUtility is created and gate related methods are moved to different class called GarageStationUtility.

1. Public class GarageStation
2. {
3. IGarageUtility \_garageUtil;
4. Public GarageStation(IGarageUtility garageUtil)
5. {
6. this.\_garageUtil=garageUtil;
7. }
8. Public void OpenForService()
9. {
10. \_garageUtil.OpenGate();
11. }
12. Public void DoService()
13. {
14. *//Check if service station is opened and then*
15. *//finish the vehicle service*
16. }
17. Public void CloseGarage()
18. {
19. \_garageUtil.CloseGate();
20. }
21. }
22. Public class GarageStationUtility : IGarageUtility
23. {
24. Public void OpenGate()
25. {
26. *//Open the Garage for service*
27. }
28. Public void CloseGate()
29. {
30. *//Close the Garage functionlity*
31. }
32. }
33. public interface IGarageUtility
34. {
35. voidOpenGate();
36. voidCloseGate();
37. }

## Open Closed Principle (OCP)

**Definition:** Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Bertrand Meyer is generally credited for having originated the definition of open/closed principle in his book Object-Oriented Software Construction.

### **Understanding**

This principle suggests that the class should be easily extended but there is no need to change its core implementations.

The application or software should be flexible to change. How change management is implemented in a system has a significant impact on the success of that application/ software. The OCP states that the behaviors of the system can be extended without having to modify its existing implementation.

New features should be implemented using the new code, but not by changing existing code. The main benefit of adhering to OCP is that it potentially streamlines code maintenance and reduces the risk of breaking the existing implementation.

### **Implementation**

Let’s take an example of bank accounts like regular savings, salary saving, corporate etc. for different customers. As for each customer type, there are different rules and different interest rates. The code below violates OCP principle if the bank introduces a new Account type. Said code modifies this method for adding a new account type.

1. Public class Account
2. {
3. *// members and function declaration*
4. Public decimal CalcInt(AccType accType)
5. {
6. If(accType==”Regular”)*// savings*
7. {
8. Inte=bal\*0.4;
9. If(bal<1000)Inte-=bal\*0.2;
10. If(bal<50000)Inte+=amt\*0.4;
11. }
12. elseif(accType==”Salary”)*// salary savings*
13. {
14. inte=bal\*0.5;
15. }
16. elseif(accType==”Corporate”)*// Corporate*
17. {
18. inte=bal\*0.3;
19. }
20. }
21. }

We can apply OCP by using interface, abstract class, abstract methods and virtual methods when you want to extend functionality. Here I have used interface for example only but you can go as per your requirement.

1. Interface IAccount
2. {
3. *// members and function declaration, properties*
4. decimal CalcInt();
5. }
6. Public Class RegularSavingAcc : IAccount
7. {
8. *//regular savings account*
9. Public decimal CalcInt()
10. {
11. Inte=bal\*0.4;
12. If(bal<1000)inte-=bal\*0.2;
13. If(bal<50000)inte+=amt\*0.4;
14. }
15. }
16. Public Class SalarySavingAcc : IAccount
17. {*//Salary savings account*
18. Public decimal CalcInt()
19. {
20. Inte=bal\*0.5;
21. }
22. }
23. Public Class CorporateAcc : IAccount
24. {
25. Public decimal CalcInt()
26. {
27. Inte=bal\*0.3;
28. }
29. }

In the above code three new classes are created; regular saving account, SalarySavingAccount, and CorporateAccount, by extending them from IAccount.

This solves the problem of modification of class and by extending interface, we can extend functionality.

Above code is implementing both OCP and SRP principle, as each class has single is doing a single task and we are not modifying class and only doing an extension.

## Liskov Substitution Principle (LSP)

**Definition by Robert C. Martin:** Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

The **Liskov substitution principle (LSP)** is a definition of a subtyping relation, called **(strong) behavioral subtyping,** that was initially introduced by Barbara Liskov in a 1987 conference keynote address titled **Data abstraction and hierarchy.**

### **Understanding**

LSP states that the child class should be perfectly substitutable for their parent class. If class C is derived from P then C should be substitutable for P.

We can check using LSP that inheritance is applied correctly or not in our code.

LSP is a fundamental principle of SOLID Principles and states that if program or module is using base class then derived class should be able to extend their base class without changing their original implementation.

### **Implementation**

Let’s consider the code below where LSP is violated. We cannot simply substitute a Triangle, which results in printing shape of a triangle, with Circle.

1. namespaceDemo
2. {
3. publicclassProgram
4. {
5. staticvoidMain(string[]args)
6. {
7. Triangletriangle=newCircle();
8. Console.WriteLine(triangle.GetColor());
9. }
10. }
11. publicclassTriangle
12. {
13. publicvirtual string GetShape()
14. {
15. return" Triangle ";
16. }
17. }
18. publicclassCircle:Triangle
19. {
20. public override string GetShape()
21. {
22. return"Circle";
23. }
24. }
25. }

To correct above implementation, we need to refactor this code by introducing interface with method called GetShape.

1. namespaceDemo
2. {
3. classProgram
4. {
5. staticvoidMain(string[]args)
6. {
7. Shapeshape=newCircle();
8. Console.WriteLine(shape.GetShape());
9. shape =newTriangle();
10. Console.WriteLine(shape.GetShape());
11. }
12. }
13. public abstract classShape
14. {
15. public abstract string GetShape();
16. }
17. publicclassTriangle:Fruit
18. {
19. public override string GetShape()
20. {
21. return"Triangle";
22. }
23. }
24. publicclassCircle:Triangle
25. {
26. public override string GetShape()
27. {
28. return"Circle";
29. }
30. }
31. }

## Interface Segregation Principle (ISP)

**Definition:** No client should be forced to implement methods which it does not use, and the contracts should be broken down to thin ones.

### **Understanding**

Interface segregation principle is required to solve the design problem of the application. When all the tasks are done by a single class or in other words, one class is used in almost all the application classes then it has become a fat class with overburden. Inheriting such class will results in having sharing methods which are not relevant to derived classes but it’s there in the base class so that will inherit in the derived class.

Using ISP, we can create separate interfaces for each operation or requirement rather than having a single class to do the same work.

### **Implementation**

In below code, ISP is broken as process method is not required by OfflineOrder class but is forced to implement.

1. public interface IOrder
2. {
3. Void AddToCart();
4. Void CCProcess();
5. }
6. publicclassOnlineOrder:IOrder
7. {
8. publicvoidAddToCart()
9. {
10. *//Do Add to Cart*
11. }
12. publicvoidCCProcess()
13. {
14. *//process through credit card*
15. }
16. }
17. publicclassOfflineOrder:IOrder
18. {
19. publicvoidAddToCart()
20. {
21. *//Do Add to Cart*
22. }
23. publicvoidCCProcess()
24. {
25. *//Not required for Cash/ offline Order*
26. thrownewNotImplementedException();
27. }
28. }

We can resolve this violation by dividing IOrder Interface.

1. public interface IOrder
2. {
3. voidAddToCart();
4. }
5. public interface IOnlineOrder
6. {
7. voidCCProcess();
8. }
9. publicclassOnlineOrder:IOrder,IOnlineOrder
10. {
11. Public void AddToCart()
12. {
13. *//Do Add to Cart*
14. }
15. Public void CCProcess()
16. {
17. *//process through credit card*
18. }
19. }
20. Public class OfflineOrder:IOrder
21. {
22. publicvoidAddToCart()
23. {
24. *//Do Add to Cart*
25. }
26. }

## Dependency Inversion Principle (DIP)

This principle is about dependencies among components. The definition of DIP is given by Robert C. Martin is as follows:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

### **Understanding**

The principle says that high-level modules should depend on abstraction, not on the details, of low-level modules. In simple words, the principle says that there should not be a tight coupling among components of software and to avoid that, the components should depend on abstraction.

The terms Dependency Injection (DI) and Inversion of Control (IoC) are generally used as interchangeably to express the same design pattern. The pattern was initially called IoC, but Martin Fowler (known for designing the enterprise software) anticipated the name as DI because all frameworks or runtime invert the control in some way and he wanted to know which aspect of control was being inverted.

Inversion of Control (IoC) is a technique to implement the Dependency Inversion Principle in C#. Inversion of control can be implemented using either an abstract class or interface. The rule is that the lower level entities should join the contract to a single interface and the higher-level entities will use only entities that are implementing the interface. This technique removes the dependency between the entities.

#### Note:

In below implementation, I have used interface as a reference, but you can use abstract class or interface as per your requirement.

### **Implementation**

In below code, we have implemented DIP using IoC using injection constructor. There are different ways to implement Dependency injection. Here, I have use injection thru constructor but you inject the dependency into class's constructor (Constructor Injection), set property (Setter Injection), method (Method Injection), events, index properties, fields and basically any members of the class which are public.

1. public interface IAutomobile
2. {
3. voidIgnition();
4. voidStop();
5. }
7. publicclassJeep:IAutomobile
8. {
9. *#region IAutomobile Members*
10. publicvoidIgnition()
11. {
12. Console.WriteLine("Jeep start");
13. }
14. publicvoidStop()
15. {
16. Console.WriteLine("Jeep stopped.");
17. }
18. *#endregion*
19. }
20. publicclass SUV :IAutomobile
21. {
22. *#region IAutomobile Members*
23. publicvoidIgnition()
24. {
25. Console.WriteLine("SUV start");
26. }
27. publicvoidStop()
28. {
29. Console.WriteLine("SUV stopped.");
30. }
31. *#endregion*
32. }

35. publicclassAutomobileController
36. {
37. IAutomobile m\_Automobile;
38. publicAutomobileController(IAutomobile automobile)
39. {
40. this.m\_Automobile= automobile;
41. }
42. publicvoidIgnition()
43. {
44. m\_Automobile.Ignition();
45. }
46. publicvoidStop()
47. {
48. m\_Automobile.Stop();
49. }
50. }
51. classProgram
52. {
53. staticvoidMain(string[]args)
54. {
55. IAutomobile automobile =newJeep();
56. *//IAutomobile automobile = new SUV();*
57. AutomobileController automobileController=newAutomobileController(automobile);
58. automobile.Ignition();
59. automobile.Stop();
60. Console.Read();
61. }
62. }

In the above code, IAutomobile interface is in an abstraction layer and AutomobileController as the higher-level module. Here, we have integrated all in a single code but in real-world, each abstraction layer is a separate class with additional functionality. Here products are completely decoupled from the consumer using IAutomobile interface. The object is injected into the constructor of the AutomobileController class in reference to the interface IAutomobile. The constructor where the object gets injected is called injection constructor.

DI is a software design pattern that allows us to develop loosely coupled code. Using DI, we can reduce tight coupling between software components. DI also allows us to better accomplish future changes and other difficulties in our software. The purpose of DI is to make code sustainable.