1. **Design Patterns:**

Definition: - design patterns are time tested solution for architecture problems, time tested practices of OOP

Understandable – it helps people learn object-oriented thinking in much better manner.

## GOF Design Patterns

The 23 Design patterns are defined by the Gang of Four programmers. These 23 patterns are divided into three groups depending on the nature of the design problem they intend to solve.

## Creational Design Patterns

These patterns deal with the process of objects creation in such a way that they can be decoupled from their implementing system. This provides more flexibility in deciding which objects need to be created for a given use case/ scenario. There are as follows:

* 1. [Factory Method](http://www.dotnettricks.com/learn/designpatterns/factory-method-design-pattern-dotnet) : Create instances of derived classes
  2. [Abstract Factory](http://www.dotnettricks.com/learn/designpatterns/abstract-factory-design-pattern-dotnet): Create instances of several classes belonging to different families
  3. [Builder](http://www.dotnettricks.com/learn/designpatterns/builder-design-pattern-dotnet) : Separates an object construction from its representation
  4. [Prototype](http://www.dotnettricks.com/learn/designpatterns/prototype-design-pattern-dotnet) : Create a duplicate object or clone of the object
  5. [Singleton](http://www.dotnettricks.com/learn/designpatterns/singleton-design-pattern-dotnet) : Ensures that a class can have only one instance

## Structural Design Patterns

These patterns deal with the composition of objects structures. The concept of inheritance is used to compose interfaces and define various ways to compose objects for obtaining new functionalities. There are as follows:

* 1. [Adapter](http://www.dotnettricks.com/learn/designpatterns/adapter-design-pattern-dotnet) : Match interfaces of different classes
  2. [Bridge](http://www.dotnettricks.com/learn/designpatterns/bridge-design-pattern-dotnet) : Separates an object’s abstraction from its implementation
  3. [Composite](http://www.dotnettricks.com/learn/designpatterns/composite-design-pattern-dotnet) : A tree structure of simple and composite objects
  4. [Decorator](http://www.dotnettricks.com/learn/mvc/file-upload-with-strongly-typed-view-and-model-validation) : Add responsibilities to objects dynamically
  5. [Façade](http://www.dotnettricks.com/learn/designpatterns/facade-design-pattern-dotnet) : A single class that represents an entire complex system
  6. [Flyweight](http://www.dotnettricks.com/learn/designpatterns/flyweight-design-pattern-dotnet) : Minimize memory usage by sharing as much data as possible with similar objects
  7. [Proxy](http://www.dotnettricks.com/learn/designpatterns/proxy-design-pattern-dotnet) : Provides a surrogate object, which references to other object

## Behavioral Design Patterns

These patterns deal with the process of communication, managing relationships, and responsibilities between objects. There are as follows:

* 1. [Chain of Responsibility](http://www.dotnettricks.com/learn/designpatterns/chain-of-responsibility-design-pattern-dotnet): Passes a request among a list or chain of objects.
  2. [Command](http://www.dotnettricks.com/learn/designpatterns/command-design-pattern-dotnet): Wraps a request under an object as a command and passed to invoker object.
  3. [Interpreter](http://www.dotnettricks.com/learn/designpatterns/interpreter-design-pattern-c-sharp): Implements an expression interface to interpret a particular context.
  4. [Iterator](http://www.dotnettricks.com/learn/designpatterns/iterator-design-pattern-c-sharp): Provides a way to access the elements of a collection object in sequential manner without knowing its underlying structure.
  5. [Mediator](http://www.dotnettricks.com/learn/designpatterns/mediator-design-pattern-c-sharp): Allows multiple objects to communicate with each other’s without knowing each other’s structure.
  6. [Memento](http://www.dotnettricks.com/learn/designpatterns/memento-design-pattern-c-sharp): Capture the current state of an object and store it in such a manner that it can be restored at a later time without breaking the rules of encapsulation.
  7. [Observer](http://www.dotnettricks.com/learn/designpatterns/observer-design-pattern-c-sharp): Allows an object (subject) to publish changes to its state and other objects (observer) that depend upon that object are automatically notified of any changes to the subject's state.
  8. [State](http://www.dotnettricks.com/learn/designpatterns/state-design-pattern-c-sharp): Alters the behavior of an object when it’s internal state changes.
  9. [Strategy](http://www.dotnettricks.com/learn/designpatterns/strategy-design-pattern-c-sharp): Allows a client to choose an algorithm from a family of algorithms at run-time and gives it a simple way to access it.
  10. [Visitor](http://www.dotnettricks.com/learn/designpatterns/visitor-design-pattern-c-sharp): Creates and performs new operations onto a set of objects without changing the object structure or classes.
  11. [Template Method](http://www.dotnettricks.com/learn/designpatterns/template-method-design-pattern-c-sharp): Defines the basic steps of an algorithm and allow the implementation of the individual steps to be changed.

|  |  |  |
| --- | --- | --- |
| **OOP Phase** | **Design pattern category** | **Example** |
| Template/class creation problem | Structural design pattern | Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, and Proxy. |
| Instantiation problems | Creational design pattern | Creational design patterns are the Factory Method, Abstract Factory, Builder, Singleton, Object Pool, and Prototype. |
| Runtime problems | Behavioral design pattern | Behavioral patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor |

Class new instance – use it

Polymorphism – fundamental thing to achieve decoupling.

Simple Factory pattern – it helps centralize object creation and thus decoupling achieved – don’t use direct obj creation in app

RIP – Replace IF with polymorphism - enum of customer type based on input type input in string return obj of customer

Lazy loading pattern – load objects on demand

* 1. Differences:
* Design pattern – code or pseudo code - eg.: Factory, iterator, singleton
* Architecture pattern – block diagrams – view of diagram – eg.: MVC, MVP, MVVM
* Architecture style – principles – eg.: REST(representational state transfer – http protocol crud operations), SOA(service oriented architctr)
  1. Differences:
* Design pattern – code or pseudo code
  1. **Unity:**

**Inversion of Control (IoC)** is a design principle (although, some people refer to it as a pattern). As the name suggests, it is used to invert different kinds of controls in object-oriented design to achieve loose coupling. Here, controls refer to any additional responsibilities a class has, other than its main responsibility. This include control over the flow of an application, and control over the flow of an object creation or dependent object creation and binding.

* 1. Dependency Injection (DI) is a design pattern used to implement IoC.

Constructor Injection: In the constructor injection, the injector supplies the service (dependency) through the client class constructor.

public CustomerBusinessLogic(ICustomerDataAccess custDataAccess)

{

\_dataAccess = custDataAccess;

}

* + 1. Property Injection: In the property injection (aka the Setter Injection), the injector supplies the dependency through a public property of the client class.

public ICustomerDataAccess DataAccess { get; set; }

* + 1. Method Injection: In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.

public void SetDependency(ICustomerDataAccess customerDataAccess)

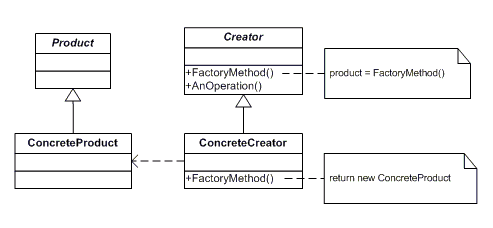
{

\_dataAccess = customerDataAccess;

}

# **Factory Method Design Pattern**

The factory design pattern in C# is used to replace class constructors, abstracting the process of object generation so that the type of the object instantiated can be determined at run-time. In this article, you will learn how to implement Factory Method Design Pattern In C# and .NET.

UML Class Diagram  


The classes and objects participating in the above UML class diagram are as follows:

1. *Product*  
   This defines the interface of objects the factory method creates
2. *ConcreteProduct*  
   This is a class which implements the Product interface.
3. *Creator*  
   This is an abstract class and declares the factory method, which returns an object of type Product.

This may also define a default implementation of the factory method that returns a default ConcreteProduct object.

This may call the factory method to create a Product object.

1. *ConcreteCreator*  
   This is a class which implements the Creator class and overrides the factory method to return an instance of a ConcreteProduct.

### **Who is what?**

The classes and objects participating in the above class diagram can be identified as follows:

1. Product - CreditCard
2. ConcreteProduct- MoneyBackCreditCard, TitaniumCreditCard, PlatinumCreditCard
3. Creator- CardFactory
4. ConcreteCreator- MoneyBackCardFactory, TitaniumCardFactory, PlatinumCardFactory
5. using System;
7. namespace FactoryMethodDesignPatternInCSharp
8. {
9. /// <summary>
10. /// Factory Pattern Demo
11. /// </summary>
12. **public** **class** ClientApplication
13. {
14. **static** **void** Main()
15. {
16. CardFactory factory = **null**;
17. Console.Write("Enter the card type you would like to visit: ");
18. string car = Console.ReadLine();
20. **switch** (car.ToLower())
21. {
22. **case** "moneyback":
23. factory = **new** MoneyBackFactory(50000, 0);
24. **break**;
25. **case** "titanium":
26. factory = **new** TitaniumFactory(100000, 500);
27. **break**;
28. **case** "platinum":
29. factory = **new** PlatinumFactory(500000, 1000);
30. **break**;
31. **default**:
32. **break**;
33. }
35. CreditCard creditCard = factory.GetCreditCard();
36. Console.WriteLine("\nYour card details are below : \n");
37. Console.WriteLine("Card Type: {0}\nCredit Limit: {1}\nAnnual Charge: {2}",
38. creditCard.CardType, creditCard.CreditLimit, creditCard.AnnualCharge);
39. Console.ReadKey();
40. }
41. }

**public class AirConditioner**

**{**

**private readonly Dictionary<Actions, AirConditionerFactory> \_factories;**

**public AirConditioner()**

**{**

**\_factories = new Dictionary<Actions, AirConditionerFactory>();**

**foreach (Actions action in Enum.GetValues(typeof(Actions)))**

**{**

**var factory = (AirConditionerFactory)Activator.CreateInstance(Type.GetType("FactoryMethod." + Enum.GetName(typeof(Actions), action) + "Factory"));**

**\_factories.Add(action, factory);**

**}**

**}**

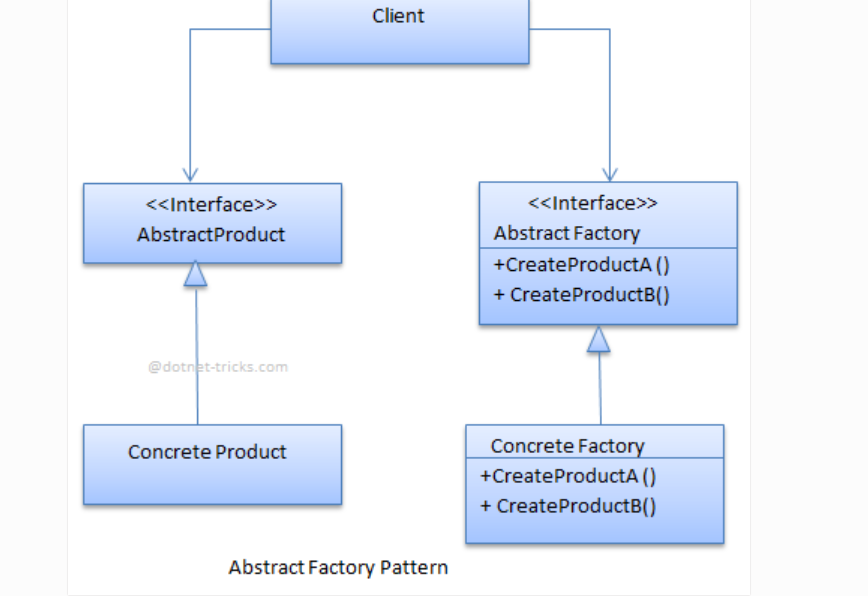
**}**

# **Abstract Factory Design Pattern**

* + Under Creational Pattern

**Abstract Factory** use Factory design pattern for creating objects. It may also use Builder design pattern and prototype design pattern for creating objects. It completely depends upon your implementation for creating objects.

Abstract Factory patterns act a super-factory which creates other factories. This pattern is also called a Factory of factories. In Abstract Factory pattern an interface is responsible for creating a set of related objects, or dependent objects without specifying their concrete classes.



The classes, interfaces, and objects in the above UML class diagram are as follows:

### **AbstractFactory**

This is an interface which is used to create abstract product

### **ConcreteFactory**

This is a class which implements the AbstractFactory interface to create concrete products.

### **AbstractProduct**

This is an interface which declares a type of product.

### **ConcreteProduct**

This is a class which implements the AbstractProduct interface to create a product.

### **Client**

This is a class which uses AbstractFactory and AbstractProduct interfaces to create a family of related objects.

public interface AbstractFactory

{

AbstractProductA CreateProductA();

AbstractProductB CreateProductB();

}

public class ConcreteFactoryA : AbstractFactory

{

public AbstractProductA CreateProductA()

{

return new ProductA1();

}

public AbstractProductB CreateProductB()

{

return new ProductB1();

}

}

public class ConcreteFactoryB : AbstractFactory

{

public AbstractProductA CreateProductA()

{

return new ProductA2();

}

public AbstractProductB CreateProductB()

{

return new ProductB2();

}

}

public interface AbstractProductA { }

public class ProductA1 : AbstractProductA { }

public class ProductA2 : AbstractProductA { }

public interface AbstractProductB { }

public class ProductB1 : AbstractProductB { }

public class ProductB2 : AbstractProductB { }

public class Client

{

private AbstractProductA \_productA;

private AbstractProductB \_productB;

public Client(AbstractFactory factory)

{

\_productA = factory.CreateProductA();

\_productB = factory.CreateProductB();

}

}

The classes, interfaces, and objects in the above class diagram can be identified as follows:

1. **VehicleFactory** - AbstractFactory interface
2. **HondaFactory & HeroFactory**- Concrete Factories
3. **Bike & Scooter**- AbstractProduct interface
4. **Regular Bike, Sports Bike, Regular Scooter & Scooty**- Concrete Products
5. **VehicleClient**- Client

/// <summary>

/// Abstract Factory Pattern Demo

/// </summary>

class Program

{

static void Main(string[] args)

{

VehicleFactory honda = new HondaFactory();

VehicleClient hondaclient = new VehicleClient(honda, "Regular");

Console.WriteLine("\*\*\*\*\*\*\* Honda \*\*\*\*\*\*\*\*\*\*");

Console.WriteLine(hondaclient.GetBikeName());

Console.WriteLine(hondaclient.GetScooterName());

hondaclient = new VehicleClient(honda, "Sports");

Console.WriteLine(hondaclient.GetBikeName());

Console.WriteLine(hondaclient.GetScooterName());

VehicleFactory hero = new HeroFactory();

VehicleClient heroclient = new VehicleClient(hero, "Regular");

Console.WriteLine("\*\*\*\*\*\*\* Hero \*\*\*\*\*\*\*\*\*\*");

Console.WriteLine(heroclient.GetBikeName());

Console.WriteLine(heroclient.GetScooterName());

heroclient = new VehicleClient(hero, "Sports");

Console.WriteLine(heroclient.GetBikeName());

Console.WriteLine(heroclient.GetScooterName());

Console.ReadKey();

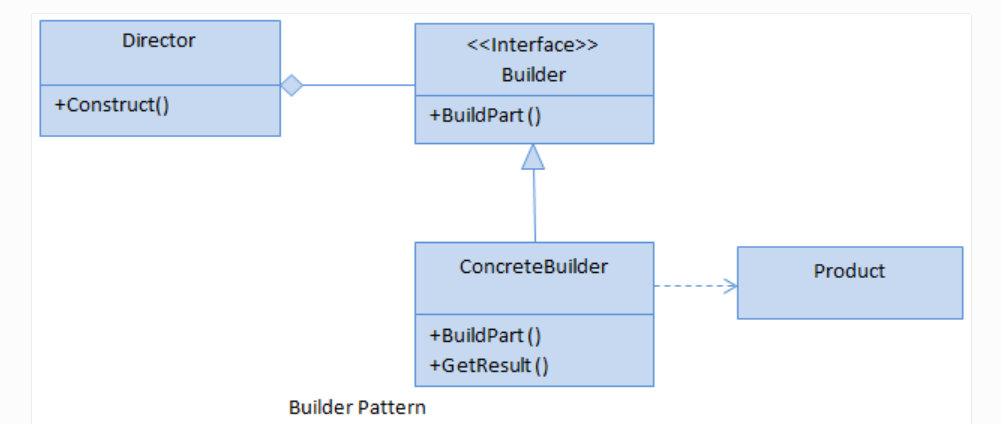
}

}

# **Builder Design Pattern**

**Builder pattern** builds a complex object by using a step by step approach. Builder interface defines the steps to build the final object. This builder is independent of the objects creation process. A class that is known as Director, controls the object creation process.

Moreover, **builder pattern** describes a way to separate an object from its construction. The same construction method can create a different representation of the object.



The classes, interfaces, and objects in the above UML class diagram are as follows:

### **Builder**

This is an interface which is used to define all the steps to create a product

### **ConcreteBuilder**

This is a class which implements the Builder interface to create a complex product.

### **Product**

This is a class which defines the parts of the complex object which are to be generated by the builder pattern.

### **Director**

This is a class which is used to construct an object using the Builder interface.

/// <summary>

/// The 'Builder' interface

/// </summary>

public interface IVehicleBuilder

{

void SetModel();

void SetEngine();

void SetTransmission();

void SetBody();

void SetAccessories();

Vehicle GetVehicle();

}

/// <summary>

/// The 'ConcreteBuilder1' class

/// </summary>

public class HeroBuilder : IVehicleBuilder

{

Vehicle objVehicle = new Vehicle();

public void SetModel()

{

objVehicle.Model = "Hero";

}

public void SetEngine()

{

objVehicle.Engine = "4 Stroke";

}

public void SetTransmission()

{

objVehicle.Transmission = "120 km/hr";

}

public void SetBody()

{

objVehicle.Body = "Plastic";

}

public void SetAccessories()

{

objVehicle.Accessories.Add("Seat Cover");

objVehicle.Accessories.Add("Rear Mirror");

}

public Vehicle GetVehicle()

{

return objVehicle;

}

}

/// <summary>

/// The 'ConcreteBuilder2' class

/// </summary>

public class HondaBuilder : IVehicleBuilder

{

Vehicle objVehicle = new Vehicle();

public void SetModel()

{

objVehicle.Model = "Honda";

}

public void SetEngine()

{

objVehicle.Engine = "4 Stroke";

}

public void SetTransmission()

{

objVehicle.Transmission = "125 Km/hr";

}

public void SetBody()

{

objVehicle.Body = "Plastic";

}

public void SetAccessories()

{

objVehicle.Accessories.Add("Seat Cover");

objVehicle.Accessories.Add("Rear Mirror");

objVehicle.Accessories.Add("Helmet");

}

public Vehicle GetVehicle()

{

return objVehicle;

}

}

/// <summary>

/// The 'Product' class

/// </summary>

public class Vehicle

{

public string Model { get; set; }

public string Engine { get; set; }

public string Transmission { get; set; }

public string Body { get; set; }

public List<string> Accessories { get; set; }

public Vehicle()

{

Accessories = new List<string>();

}

public void ShowInfo()

{

Console.WriteLine("Model: {0}", Model);

Console.WriteLine("Engine: {0}", Engine);

Console.WriteLine("Body: {0}", Body);

Console.WriteLine("Transmission: {0}", Transmission);

Console.WriteLine("Accessories:");

foreach (var accessory in Accessories)

{

Console.WriteLine("\t{0}", accessory);

}

}

}

/// <summary>

/// The 'Director' class

/// </summary>

public class VehicleCreator

{

private readonly IVehicleBuilder objBuilder;

public VehicleCreator(IVehicleBuilder builder)

{

objBuilder = builder;

}

public void CreateVehicle()

{

objBuilder.SetModel();

objBuilder.SetEngine();

objBuilder.SetBody();

objBuilder.SetTransmission();

objBuilder.SetAccessories();

}

public Vehicle GetVehicle()

{

return objBuilder.GetVehicle();

}

}

/// <summary>

/// Builder Design Pattern Demo

/// </summary>

class Program

{

static void Main(string[] args)

{

var vehicleCreator = new VehicleCreator(new HeroBuilder());

vehicleCreator.CreateVehicle();

var vehicle = vehicleCreator.GetVehicle();

vehicle.ShowInfo();

Console.WriteLine("---------------------------------------------");

vehicleCreator = new VehicleCreator(new HondaBuilder());

vehicleCreator.CreateVehicle();

vehicle = vehicleCreator.GetVehicle();

vehicle.ShowInfo();

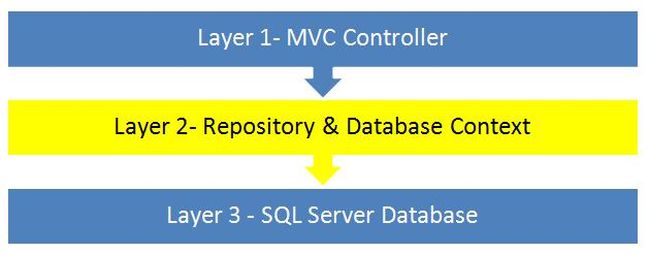
Console.ReadKey();

}

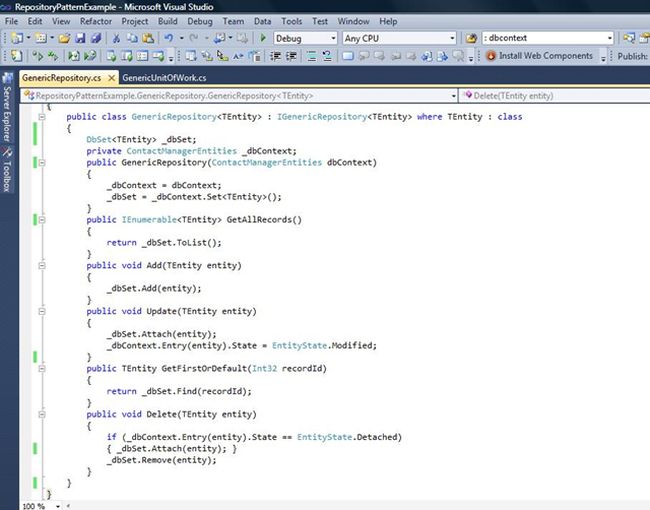
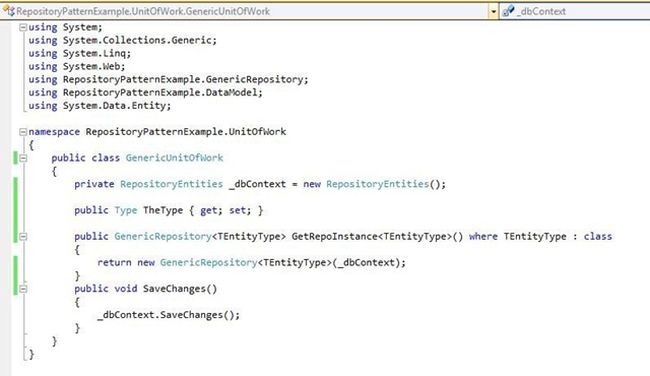
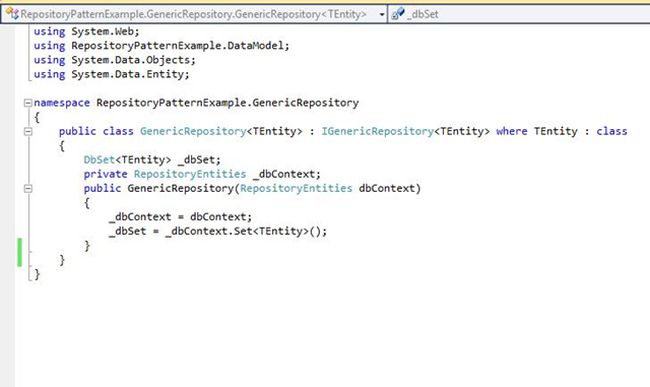
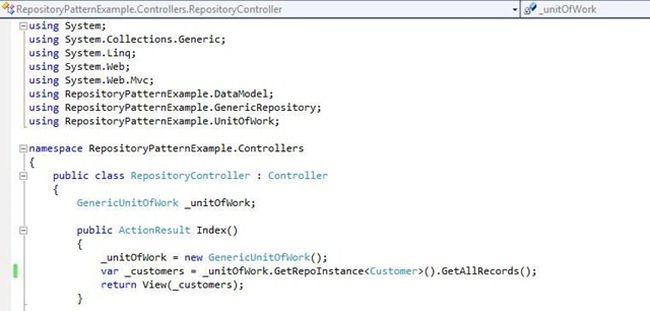
}

# **Repository Pattern**

the repository pattern can be implemented, to separate the layers. Our purpose will be to separate the controller and the data access layer (database context) using an intermediate layer, in other words repository layer, for communication between the two.



A generic repository is a generic class, with basic CRUD methods in it (and of course other methods can be added as needed). This class and its member functions can be used for any entity of the database. This means, if we have entities Customers and Orders, this single generic class can be used for both of them.

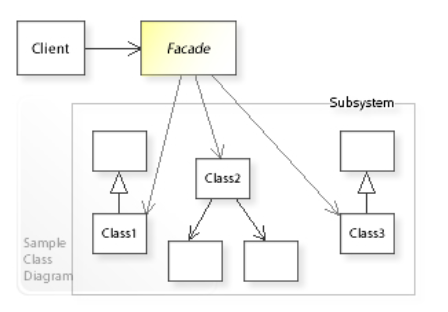
.  
  
:  
  
:  
  
  
And, our controller method to fetch the records will look like the following:  
  
:  
  


# **Facade Design Pattern**

Facade Pattern is used in hiding complexity of large systems and provide simpler interfaces,

A room is a façade and just by looking at it from outside the door, one can not 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.Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

UML diagram,



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.

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. }

**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 – example - Redbus app**

* 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.

# **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.)

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.

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. // Bad code! Do not use!
5. **public** **sealed** **class** Singleton
6. {
7. //Private Constructor.
8. **private** Singleton()
9. {
10. }
11. **private** **static** Singleton instance = **null**;
12. **public** **static** Singleton Instance
13. {
14. **get**
15. {
16. **if** (instance == **null**)
17. {
18. instance = **new** Singleton();
19. }
20. **return** instance;
21. }
22. }
23. }

**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. }

**4. Thread Safe Singleton without using locks and no lazy instantiation**

Explanation of the following code:

1. The preceding implementation looks like very simple code.
2. This type of implementation has a static constructor, so it executes only once per Application Domain.
3. It is not as lazy as the other implementation.
4. **public** **sealed** **class** Singleton
5. {
6. **private** **static** **readonly** Singleton instance = **new** Singleton();
7. // Explicit static constructor to tell C# compiler
8. // not to mark type as beforefieldinit
9. **static** Singleton()
10. {
11. }
12. **private** Singleton()
13. {
14. }
15. **public** **static** Singleton Instance
16. {
17. **get**
18. {
19. **return** instance;
20. }
21. }
22. }

**5. Fully lazy instantiation**

Explanation of the following code:

1. Here, instantiation is triggered by the first reference to the static member of the nested class, that only occurs in Instance.
2. This means the implementation is fully lazy, but has all the performance benefits of the previous ones.
3. Note that although nested classes have access to the enclosing class's private members, the reverse is not true, hence the need for instance to be internal here.
4. That doesn't raise any other problems, though, as the class itself is private.
5. The code is more complicated in order to make the instantiation lazy.
6. **public** **sealed** **class** Singleton
7. {
8. **private** **static** **readonly** Singleton instance = **new** Singleton();
9. // Explicit static constructor to tell C# compiler
10. // not to mark type as beforefieldinit
11. **static** Singleton()
12. {
13. }
14. **private** Singleton()
15. {
16. }
17. **public** **static** Singleton Instance
18. {
19. **get**
20. {
21. **return** instance;
22. }
23. }
24. }

**6. Using .NET 4's Lazy<T> type**

Explanation of the following code:

1. If you're using .NET 4 (or higher) then you can use the System.Lazy<T> type to make the laziness really simple.
2. All you need to do is pass a delegate to the constructor that calls the Singleton constructor, which is done most easily with a lambda expression.
3. It also allows you to check whether or not the instance has been created with the IsValueCreated property.
4. **public** **sealed** **class** Singleton
5. {
6. **private** Singleton()
7. {
8. }
9. **private** **static** **readonly** Lazy<Singleton> lazy = **new** Lazy<Singleton>(() => **new** Singleton());
10. **public** **static** Singleton Instance
11. {
12. **get**
13. {
14. **return** lazy.Value;
15. }
16. }

}