# DESIGN PATTERN IN C#

# Facade

### definition

|  |  |
| --- | --- |
| http://www.dofactory.com/Images/pixel.gif | Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use. |

### UML class diagram



### participants

    The classes and/or objects participating in this pattern are:

* **Facade**   **(MortgageApplication)**
  + knows which subsystem classes are responsible for a request.
  + delegates client requests to appropriate subsystem objects.
* **Subsystem classes**   **(Bank, Credit, Loan)**
  + implement subsystem functionality.
  + handle work assigned by the Facade object.
  + have no knowledge of the facade and keep no reference to it.

[http://www.dofactory.com/Images/up.gifreturn to top](http://www.dofactory.com/Patterns/PatternFacade.aspx)

### sample code in C#

This structural code demonstrates the Facade pattern which provides a simplified and uniform interface to a large subsystem of classes.

**sample code in C#**

This structural code demonstrates the Facade pattern which provides a simplified and uniform interface to a large subsystem of classes.

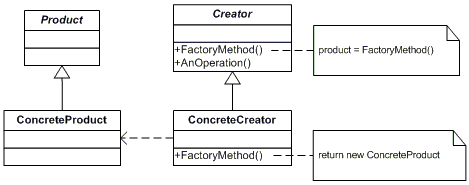
|  |
| --- |
| // Facade pattern -- Structural example |
| using System;    namespace DoFactory.GangOfFour.Facade.Structural  {    /// <summary>    /// MainApp startup class for Structural    /// Facade Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      public static void Main()      {        Facade facade = new Facade();          facade.MethodA();        facade.MethodB();          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Subsystem ClassA' class    /// </summary>    class SubSystemOne    {      public void MethodOne()      {        Console.WriteLine(" SubSystemOne Method");      }    }      /// <summary>    /// The 'Subsystem ClassB' class    /// </summary>    class SubSystemTwo    {      public void MethodTwo()      {        Console.WriteLine(" SubSystemTwo Method");      }    }      /// <summary>    /// The 'Subsystem ClassC' class    /// </summary>    class SubSystemThree    {      public void MethodThree()      {        Console.WriteLine(" SubSystemThree Method");      }    }      /// <summary>    /// The 'Subsystem ClassD' class    /// </summary>    class SubSystemFour    {      public void MethodFour()      {        Console.WriteLine(" SubSystemFour Method");      }    }      /// <summary>    /// The 'Facade' class    /// </summary>    class Facade    {      private SubSystemOne \_one;      private SubSystemTwo \_two;      private SubSystemThree \_three;      private SubSystemFour \_four;        public Facade()      {        \_one = new SubSystemOne();        \_two = new SubSystemTwo();        \_three = new SubSystemThree();        \_four = new SubSystemFour();      }        public void MethodA()      {        Console.WriteLine("\nMethodA() ---- ");        \_one.MethodOne();        \_two.MethodTwo();        \_four.MethodFour();      }        public void MethodB()      {        Console.WriteLine("\nMethodB() ---- ");        \_two.MethodTwo();        \_three.MethodThree();      }    }  } |
| Output  MethodA() ---- SubSystemOne Method SubSystemTwo Method SubSystemFour Method  MethodB() ---- SubSystemTwo Method SubSystemThree Method |

1. Factory Method Design Pattern

|  |  |
| --- | --- |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Definition  |  |  | | --- | --- | | http://www.dofactory.com/Images/pixel.gif | Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. | |  |

### UML class diagram



### participants

    The classes and/or objects participating in this pattern are:

* **Product**  **(Page) ( it holds static method)**
  + defines the interface of objects the factory method creates
* **ConcreteProduct**  **(SkillsPage, EducationPage, ExperiencePage)( non abstract)**
  + implements the Product interface
* **Creator**  **(Document)(user)**
  + declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  + may call the factory method to create a Product object.
* **ConcreteCreator**  **(Report, Resume)** 
  + overrides the factory method to return an instance of a ConcreteProduct.

### sample code in C#

This structural code demonstrates the Factory method offering great flexibility in creating different objects. The Abstract class may provide a default object, but each subclass can instantiate an extended version of the object.

EXAMPLE1:

|  |
| --- |
| // Factory Method pattern -- Structural example |
| using System;    namespace DoFactory.GangOfFour.Factory.Structural  {    /// <summary>    /// MainApp startup class for Structural    /// Factory Method Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      static void Main()      {        // An array of creators        Creator[] creators = new Creator[2];          creators[0] = new ConcreteCreatorA();        creators[1] = new ConcreteCreatorB();          // Iterate over creators and create products        foreach (Creator creator in creators)        {          Product product = creator.FactoryMethod();          Console.WriteLine("Created {0}",            product.GetType().Name);        }          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Product' abstract class    /// </summary>    abstract class Product    {    }      /// <summary>    /// A 'ConcreteProduct' class    /// </summary>    class ConcreteProductA : Product    {    }      /// <summary>    /// A 'ConcreteProduct' class    /// </summary>    class ConcreteProductB : Product    {    }      /// <summary>    /// The 'Creator' abstract class    /// </summary>    abstract class Creator    {      public abstract Product FactoryMethod();    }      /// <summary>    /// A 'ConcreteCreator' class    /// </summary>    class ConcreteCreatorA : Creator    {      public override Product FactoryMethod()      {        return new ConcreteProductA();      }    }      /// <summary>    /// A 'ConcreteCreator' class    /// </summary>    class ConcreteCreatorB : Creator    {      public override Product FactoryMethod()      {        return new ConcreteProductB();      }    }  } |
| Output  Created ConcreteProductA Created ConcreteProductB |

EXAMPLE 2:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace facade\_vehicle

{

public interface Factory

{

void drive(int miles);

}

class Scooter : Factory

{

public void drive(int miles)

{

Console.WriteLine("This scooter drives" + miles + " km ");

}

}

class Car : Factory

{

public void drive(int miles)

{

Console.WriteLine("This car dives " + miles + " km ");

}

}

public abstract class VehicleFactory

{

public abstract Factory Getvehicle(string vehicle);

}

public class ConcreteVehicleFactory : VehicleFactory

{

public override Factory Getvehicle(string vehicle)

{

switch (vehicle)

{

case "Scooter":

return new Scooter();

case "Car":

return new Car();

default:

throw new ApplicationException(string.Format("vehicl {0} cant be created",vehicle));

}

}

}

class Program

{

static void Main(string[] args)

{

VehicleFactory vf = new ConcreteVehicleFactory();

Factory scooter = vf.Getvehicle("Scooter");

scooter.drive(45);

Factory car = vf.Getvehicle("Car");

car.drive(78);

Console.ReadKey();

}

}

}

**OUTPUT:**

**This scooter drives 45 km**

**This car drives 78 km**

# Command Design Pattern

|  |  |
| --- | --- |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| definition  |  |  | | --- | --- | | http://www.dofactory.com/Images/pixel.gif | Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. | |  |

### UML class diagram



### participants

    The classes and/or objects participating in this pattern are:

* **Command**  **(Command)**
  + declares an interface for executing an operation
* **ConcreteCommand**  **(CalculatorCommand)**
  + defines a binding between a Receiver object and an action
  + implements Execute by invoking the corresponding operation(s) on Receiver
* **Client**  **(CommandApp)**
  + creates a ConcreteCommand object and sets its receiver
* **Invoker**  **(User)**
  + asks the command to carry out the request
* **Receiver**  **(Calculator)**
  + knows how to perform the operations associated with carrying out the request.

### sample code in C#

This structural code demonstrates the Command pattern which stores requests as objects allowing clients to execute or playback the requests.

|  |
| --- |
| // Command pattern -- Structural example |
| using System;    namespace DoFactory.GangOfFour.Command.Structural  {    /// <summary>    /// MainApp startup class for Structural    /// Command Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      static void Main()      {        // Create receiver, command, and invoker        Receiver receiver = new Receiver();        Command command = new ConcreteCommand(receiver);        Invoker invoker = new Invoker();          // Set and execute command        invoker.SetCommand(command);        invoker.ExecuteCommand();          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Command' abstract class    /// </summary>    abstract class Command    {      protected Receiver receiver;        // Constructor      public Command(Receiver receiver)      {        this.receiver = receiver;      }        public abstract void Execute();    }      /// <summary>    /// The 'ConcreteCommand' class    /// </summary>    class ConcreteCommand : Command    {      // Constructor      public ConcreteCommand(Receiver receiver) :        base(receiver)      {      }        public override void Execute()      {        receiver.Action();      }    }      /// <summary>    /// The 'Receiver' class    /// </summary>    class Receiver    {      public void Action()      {        Console.WriteLine("Called Receiver.Action()");      }    }      /// <summary>    /// The 'Invoker' class    /// </summary>    class Invoker    {      private Command \_command;        public void SetCommand(Command command)      {        this.\_command = command;      }        public void ExecuteCommand()      {        \_command.Execute();      } |
| }  } |
| Output  Called Receiver.Action() |