**Lab 3: Tìm hiểu và cài đặt nhóm mẫu Structural (tt)(5 tiết)**

Yêu cầu:

* Sinh viên đọc hiểu rõ mục đích, ý nghĩa và áp dụng ứng dụng của nhóm mẫu cấu trúc.
* Sử dụng Visual Studio cài đặc nhóm mẫu trên.
* Nộp bài báo cáo: Mỗi parttern hãy lấy 2 ví dụ thể hiện bằng sơ đồ lớp (Class diagram)

**Structural Patterns:**

|  |  |
| --- | --- |
| [**Chain of Resp.**](http://www.dofactory.com/Patterns/PatternChain.aspx) | A way of passing a request between a chain of objects |
| [**Command**](http://www.dofactory.com/Patterns/PatternCommand.aspx) | Encapsulate a command request as an object |
| [**Interpreter**](http://www.dofactory.com/Patterns/PatternInterpreter.aspx) | A way to include language elements in a program |
| [**Iterator**](http://www.dofactory.com/Patterns/PatternIterator.aspx) | Sequentially access the elements of a collection |
| [**Mediator**](http://www.dofactory.com/Patterns/PatternMediator.aspx) | Defines simplified communication between classes |
| [**Memento**](http://www.dofactory.com/Patterns/PatternMemento.aspx) | Capture and restore an object's internal state |
| [**Observer**](http://www.dofactory.com/Patterns/PatternObserver.aspx) | A way of notifying change to a number of classes |
| [**State**](http://www.dofactory.com/Patterns/PatternState.aspx) | Alter an object's behavior when its state changes |
| [**Strategy**](http://www.dofactory.com/Patterns/PatternStrategy.aspx) | Encapsulates an algorithm inside a class |
| [**Template Method**](http://www.dofactory.com/Patterns/PatternTemplate.aspx) | Defer the exact steps of an algorithm to a subclass |
| [**Visitor**](http://www.dofactory.com/Patterns/PatternVisitor.aspx) | Defines a new operation to a class without change |

1. [**Command**](http://www.dofactory.com/Patterns/PatternCommand.aspx)

**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**

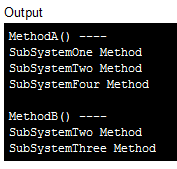
     The classes and/or objects participating in this pattern are:

* **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.

**Sample code in C#**

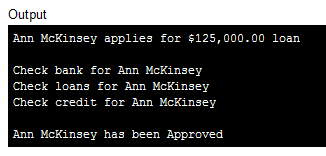
This structural code demonstrates the Facade pattern which provides a simplified and uniform interface to a large subsystem of classes.

|  |
| --- |
| // 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();      }    }  } |



This real-world code demonstrates the Facade pattern as a MortgageApplication object which provides a simplified interface to a large subsystem of classes measuring the creditworthyness of an applicant.

|  |
| --- |
| // Facade pattern -- Real World example |
| using System;    namespace DoFactory.GangOfFour.Facade.RealWorld  {    /// <summary>    /// MainApp startup class for Real-World    /// Facade Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      static void Main()      {        // Facade        Mortgage mortgage = new Mortgage();          // Evaluate mortgage eligibility for customer        Customer customer = new Customer("Ann McKinsey");        bool eligible = mortgage.IsEligible(customer, 125000);          Console.WriteLine("\n" + customer.Name +            " has been " + (eligible ? "Approved" : "Rejected"));          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Subsystem ClassA' class    /// </summary>    class Bank    {      public bool HasSufficientSavings(Customer c, int amount)      {        Console.WriteLine("Check bank for " + c.Name);        return true;      }    }      /// <summary>    /// The 'Subsystem ClassB' class    /// </summary>    class Credit    {      public bool HasGoodCredit(Customer c)      {        Console.WriteLine("Check credit for " + c.Name);        return true;      }    }      /// <summary>    /// The 'Subsystem ClassC' class    /// </summary>    class Loan    {      public bool HasNoBadLoans(Customer c)      {        Console.WriteLine("Check loans for " + c.Name);        return true;      }    }      /// <summary>    /// Customer class    /// </summary>    class Customer    {      private string \_name;        // Constructor      public Customer(string name)      {        this.\_name = name;      }        // Gets the name      public string Name      {        get { return \_name; }      }    }      /// <summary>    /// The 'Facade' class    /// </summary>    class Mortgage    {      private Bank \_bank = new Bank();      private Loan \_loan = new Loan();      private Credit \_credit = new Credit();        public bool IsEligible(Customer cust, int amount)      {        Console.WriteLine("{0} applies for {1:C} loan\n",          cust.Name, amount);          bool eligible = true;          // Check creditworthyness of applicant        if (!\_bank.HasSufficientSavings(cust, amount))        {          eligible = false;        }        else if (!\_loan.HasNoBadLoans(cust))        {          eligible = false;        }        else if (!\_credit.HasGoodCredit(cust))        {          eligible = false;        }          return eligible;      }    }  } |

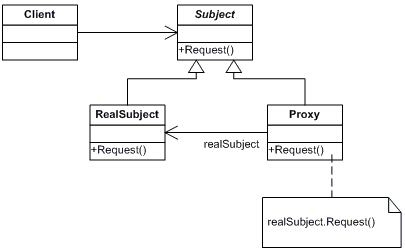


1. [**Proxy**](http://www.dofactory.com/Patterns/PatternProxy.aspx)

**Definition**

|  |  |
| --- | --- |
| http://www.dofactory.com/Images/pixel.gif | Provide a surrogate or placeholder for another object to control access to it. |

### UML class diagram



**Participants**

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

* **Proxy**   **(MathProxy)**
  + maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  + provides an interface identical to Subject's so that a proxy can be substituted for for the real subject.
  + controls access to the real subject and may be responsible for creating and deleting it.
  + other responsibilites depend on the kind of proxy:
    - *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
    - *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real images's extent.
    - *protection proxies* check that the caller has the access permissions required to perform a request.
* **Subject**   **(IMath)**
  + defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
* **RealSubject**   **(Math)**
  + defines the real object that the proxy represents.

**Sample code in C#**

This structural code demonstrates the Proxy pattern which provides a representative object (proxy) that controls access to another similar object.

|  |
| --- |
| // Proxy pattern -- Structural example |
| using System;    namespace DoFactory.GangOfFour.Proxy.Structural  {    /// <summary>    /// MainApp startup class for Structural    /// Proxy Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      static void Main()      {        // Create proxy and request a service        Proxy proxy = new Proxy();        proxy.Request();          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Subject' abstract class    /// </summary>    abstract class Subject    {      public abstract void Request();    }      /// <summary>    /// The 'RealSubject' class    /// </summary>    class RealSubject : Subject    {      public override void Request()      {        Console.WriteLine("Called RealSubject.Request()");      }    }      /// <summary>    /// The 'Proxy' class    /// </summary>    class Proxy : Subject    {      private RealSubject \_realSubject;        public override void Request()      {        // Use 'lazy initialization'        if (\_realSubject == null)        {          \_realSubject = new RealSubject();        }          \_realSubject.Request();      }    }  } |



This real-world code demonstrates the Proxy pattern for a Math object represented by a MathProxy object.

|  |
| --- |
| // Proxy pattern -- Real World example |
| using System;    namespace DoFactory.GangOfFour.Proxy.RealWorld  {    /// <summary>    /// MainApp startup class for Real-World    /// Proxy Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      static void Main()      {        // Create math proxy        MathProxy proxy = new MathProxy();          // Do the math        Console.WriteLine("4 + 2 = " + proxy.Add(4, 2));        Console.WriteLine("4 - 2 = " + proxy.Sub(4, 2));        Console.WriteLine("4 \* 2 = " + proxy.Mul(4, 2));        Console.WriteLine("4 / 2 = " + proxy.Div(4, 2));          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Subject interface    /// </summary>    public interface IMath    {      double Add(double x, double y);      double Sub(double x, double y);      double Mul(double x, double y);      double Div(double x, double y);    }      /// <summary>    /// The 'RealSubject' class    /// </summary>    class Math : IMath    {      public double Add(double x, double y) { return x + y; }      public double Sub(double x, double y) { return x - y; }      public double Mul(double x, double y) { return x \* y; }      public double Div(double x, double y) { return x / y; }    }      /// <summary>    /// The 'Proxy Object' class    /// </summary>    class MathProxy : IMath    {      private Math \_math = new Math();        public double Add(double x, double y)      {        return \_math.Add(x, y);      }      public double Sub(double x, double y)      {        return \_math.Sub(x, y);      }      public double Mul(double x, double y)      {        return \_math.Mul(x, y);      }      public double Div(double x, double y)      {        return \_math.Div(x, y);      }    }  } |

