**Lab 1: Tìm hiểu và cài đặt nhóm mẫu Creation (5 tiết)**

Yêu cầu:

* Sinh viên đọc hiểu rõ mục đích, ý nghĩa và áp dụng ứng dụng của nhóm mẫu Creation.
* Sử dụng Visual Studio cài đặc nhóm mẫu trên.
* Nộp bài sau buổi thực hành: Bài tóm tắt nội dung (2 trang word)+ chương trình

**Creational Patterns:**

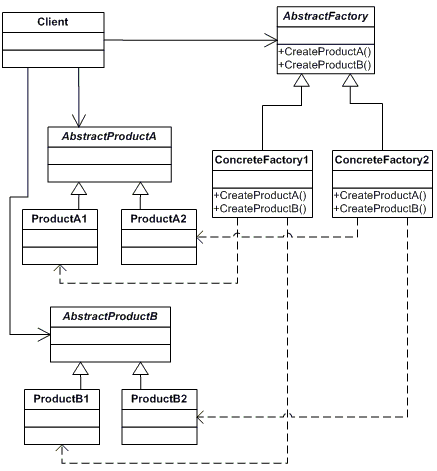
|  |  |
| --- | --- |
| **Abstract Factory** | Creates an instance of several families of classes |
| [**Builder**](http://www.dofactory.com/Patterns/PatternBuilder.aspx) | Separates object construction from its representation |
| [**Factory Method**](http://www.dofactory.com/Patterns/PatternFactory.aspx) | Creates an instance of several derived classes |
| [**Prototype**](http://www.dofactory.com/Patterns/PatternPrototype.aspx) | A fully initialized instance to be copied or cloned |
| **Singleton** | A class of which only a single instance can exist |

1. **Abstract Factory**

**definition**

|  |  |
| --- | --- |
| http://www.dofactory.com/Images/pixel.gif | Provide an interface for creating families of related or dependent objects without specifying their concrete classes. |

### UML class diagram



**Participants**

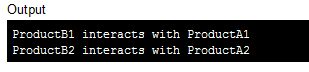
    The classes and/or objects participating in this pattern are:

* **AbstractFactory**  **(ContinentFactory)**
  + Declares an interface for operations that create abstract products
* **ConcreteFactory**   **(AfricaFactory, AmericaFactory)**
  + Implements the operations to create concrete product objects
* **AbstractProduct**   **(Herbivore, Carnivore)**
  + Declares an interface for a type of product object
* **Product**  **(Wildebeest, Lion, Bison, Wolf)**
  + Defines a product object to be created by the corresponding concrete factory
  + Implements the AbstractProduct interface
* **Client**  **(AnimalWorld)**
  + Uses interfaces declared by AbstractFactory and AbstractProduct classes

### Sample code in C#

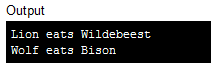
This structural code demonstrates the Abstract Factory pattern creating parallel hierarchies of objects. Object creation has been abstracted and there is no need for hard-coded class names in the client code.

|  |
| --- |
| // Abstract Factory pattern -- Structural example |
| using System;    namespace DoFactory.GangOfFour.Abstract.Structural  {    /// <summary>    /// MainApp startup class for Structural    /// Abstract Factory Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      public static void Main()      {        // Abstract factory #1        AbstractFactory factory1 = new ConcreteFactory1();        Client client1 = new Client(factory1);        client1.Run();          // Abstract factory #2        AbstractFactory factory2 = new ConcreteFactory2();        Client client2 = new Client(factory2);        client2.Run();          // Wait for user input        Console.ReadKey();      }    }      /// <summary>    /// The 'AbstractFactory' abstract class    /// </summary>    abstract class AbstractFactory    {      public abstract AbstractProductA CreateProductA();      public abstract AbstractProductB CreateProductB();    }        /// <summary>    /// The 'ConcreteFactory1' class    /// </summary>    class ConcreteFactory1 : AbstractFactory    {      public override AbstractProductA CreateProductA()      {        return new ProductA1();      }      public override AbstractProductB CreateProductB()      {        return new ProductB1();      }    }      /// <summary>    /// The 'ConcreteFactory2' class    /// </summary>    class ConcreteFactory2 : AbstractFactory    {      public override AbstractProductA CreateProductA()      {        return new ProductA2();      }      public override AbstractProductB CreateProductB()      {        return new ProductB2();      }    }      /// <summary>    /// The 'AbstractProductA' abstract class    /// </summary>    abstract class AbstractProductA    {    }      /// <summary>    /// The 'AbstractProductB' abstract class    /// </summary>    abstract class AbstractProductB    {      public abstract void Interact(AbstractProductA a);    }        /// <summary>    /// The 'ProductA1' class    /// </summary>    class ProductA1 : AbstractProductA    {    }      /// <summary>    /// The 'ProductB1' class    /// </summary>    class ProductB1 : AbstractProductB    {      public override void Interact(AbstractProductA a)      {        Console.WriteLine(this.GetType().Name + " interacts with " + a.GetType().Name);      }    }      /// <summary>    /// The 'ProductA2' class    /// </summary>    class ProductA2 : AbstractProductA    {    }      /// <summary>    /// The 'ProductB2' class    /// </summary>    class ProductB2 : AbstractProductB    {      public override void Interact(AbstractProductA a)      {        Console.WriteLine(this.GetType().Name + " interacts with " + a.GetType().Name);      }    }      /// <summary>    /// The 'Client' class. Interaction environment for the products.    /// </summary>    class Client    {      private AbstractProductA \_abstractProductA;      private AbstractProductB \_abstractProductB;        // Constructor      public Client(AbstractFactory factory)      {        \_abstractProductB = factory.CreateProductB();        \_abstractProductA = factory.CreateProductA();      }        public void Run()      {        \_abstractProductB.Interact(\_abstractProductA);      }    }  } |



This real-world code demonstrates the creation of different animal worlds for a computer game using different factories. Although the animals created by the Continent factories are different, the interactions among the animals remain the same.

|  |
| --- |
| // Abstract Factory pattern -- Real World example |
| using System;    namespace DoFactory.GangOfFour.Abstract.RealWorld  {    /// <summary>    /// MainApp startup class for Real-World    /// Abstract Factory Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      public static void Main()      {        // Create and run the African animal world        ContinentFactory africa = new AfricaFactory();        AnimalWorld world = new AnimalWorld(africa);        world.RunFoodChain();          // Create and run the American animal world        ContinentFactory america = new AmericaFactory();        world = new AnimalWorld(america);        world.RunFoodChain();          // Wait for user input        Console.ReadKey();      }    }        /// <summary>    /// The 'AbstractFactory' abstract class    /// </summary>    abstract class ContinentFactory    {      public abstract Herbivore CreateHerbivore();      public abstract Carnivore CreateCarnivore();    }      /// <summary>    /// The 'ConcreteFactory1' class    /// </summary>    class AfricaFactory : ContinentFactory    {      public override Herbivore CreateHerbivore()      {        return new Wildebeest();      }      public override Carnivore CreateCarnivore()      {        return new Lion();      }    }      /// <summary>    /// The 'ConcreteFactory2' class    /// </summary>    class AmericaFactory : ContinentFactory    {      public override Herbivore CreateHerbivore()      {        return new Bison();      }      public override Carnivore CreateCarnivore()      {        return new Wolf();      }    }      /// <summary>    /// The 'AbstractProductA' abstract class    /// </summary>    abstract class Herbivore    {    }      /// <summary>    /// The 'AbstractProductB' abstract class    /// </summary>    abstract class Carnivore    {      public abstract void Eat(Herbivore h);    }      /// <summary>    /// The 'ProductA1' class    /// </summary>    class Wildebeest : Herbivore    {    }      /// <summary>    /// The 'ProductB1' class    /// </summary>    class Lion : Carnivore    {      public override void Eat(Herbivore h)      {        // Eat Wildebeest        Console.WriteLine(this.GetType().Name + " eats " + h.GetType().Name);      }    }      /// <summary>    /// The 'ProductA2' class    /// </summary>    class Bison : Herbivore    {    }      /// <summary>    /// The 'ProductB2' class    /// </summary>    class Wolf : Carnivore    {      public override void Eat(Herbivore h)      {        // Eat Bison        Console.WriteLine(this.GetType().Name + " eats " + h.GetType().Name);      }    }      /// <summary>    /// The 'Client' class    /// </summary>    class AnimalWorld    {      private Herbivore \_herbivore;      private Carnivore \_carnivore;        // Constructor      public AnimalWorld(ContinentFactory factory)      {        \_carnivore = factory.CreateCarnivore();        \_herbivore = factory.CreateHerbivore();      }        public void RunFoodChain()      {        \_carnivore.Eat(\_herbivore);      }    }  } |

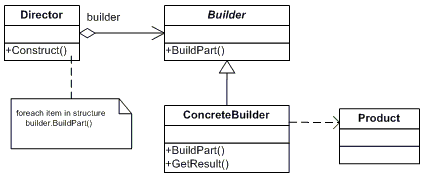


1. [**Builder**](http://www.dofactory.com/Patterns/PatternBuilder.aspx)

**Definition**

|  |  |
| --- | --- |
| http://www.dofactory.com/Images/pixel.gif | Separate the construction of a complex object from its representation so that the same construction process can create different representations. |

### UML class diagram



**Participants**

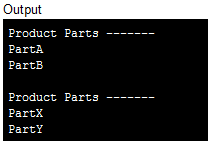
    The classes and/or objects participating in this pattern are:

* **Builder**  **(VehicleBuilder)**
  + Specifies an abstract interface for creating parts of a Product object
* **ConcreteBuilder**  **(MotorCycleBuilder, CarBuilder, ScooterBuilder)**
  + Constructs and assembles parts of the product by implementing the Builder interface
  + Defines and keeps track of the representation it creates
  + Provides an interface for retrieving the product
* **Director**  **(Shop)**
  + Constructs an object using the Builder interface
* **Product**  **(Vehicle)**
  + Represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled
  + Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

### Sample code in C#

This structural code demonstrates the Builder pattern in which complex objects are created in a step-by-step fashion. The construction process can create different object representations and provides a high level of control over the assembly of the objects.

|  |
| --- |
| // Builder pattern -- Structural example |
| using System;  using System.Collections.Generic;    namespace DoFactory.GangOfFour.Builder.Structural  {    /// <summary>    /// MainApp startup class for Structural    /// Builder Design Pattern.    /// </summary>    public class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      public static void Main()      {        // Create director and builders        Director director = new Director();          Builder b1 = new ConcreteBuilder1();        Builder b2 = new ConcreteBuilder2();          // Construct two products        director.Construct(b1);        Product p1 = b1.GetResult();        p1.Show();          director.Construct(b2);        Product p2 = b2.GetResult();        p2.Show();          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Director' class    /// </summary>    class Director    {      // Builder uses a complex series of steps      public void Construct(Builder builder)      {        builder.BuildPartA();        builder.BuildPartB();      }    }      /// <summary>    /// The 'Builder' abstract class    /// </summary>    abstract class Builder    {      public abstract void BuildPartA();      public abstract void BuildPartB();      public abstract Product GetResult();    }      /// <summary>    /// The 'ConcreteBuilder1' class    /// </summary>    class ConcreteBuilder1 : Builder    {      private Product \_product = new Product();        public override void BuildPartA()      {        \_product.Add("PartA");      }        public override void BuildPartB()      {        \_product.Add("PartB");      }        public override Product GetResult()      {        return \_product;      }    }      /// <summary>    /// The 'ConcreteBuilder2' class    /// </summary>    class ConcreteBuilder2 : Builder    {      private Product \_product = new Product();        public override void BuildPartA()      {        \_product.Add("PartX");      }        public override void BuildPartB()      {        \_product.Add("PartY");      }        public override Product GetResult()      {        return \_product;      }    }      /// <summary>    /// The 'Product' class    /// </summary>    class Product    {      private List<string> \_parts = new List<string>();        public void Add(string part)      {        \_parts.Add(part);      }        public void Show()      {        Console.WriteLine("\nProduct Parts -------");        foreach (string part in \_parts)          Console.WriteLine(part);      }    }  } |

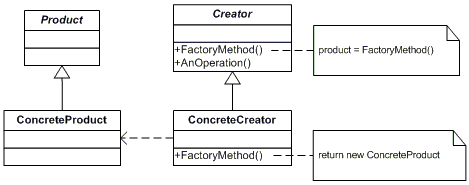


1. [**Factory Method**](http://www.dofactory.com/Patterns/PatternFactory.aspx)

**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**

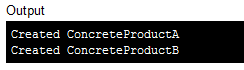
    The classes and/or objects participating in this pattern are:

* **Product**  **(Page)**
  + Defines the interface of objects the factory method creates
* **ConcreteProduct**  **(SkillsPage, EducationPage, ExperiencePage)**
  + Implements the Product interface
* **Creator**  **(Document)**
  + 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#

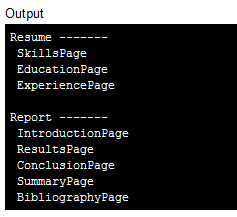
This structural code demonstrates the Factory method offering great flexibility in creating different objects. The Abstract class may provide a default object, but each subclass can instantiate an extended version of the object.

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



This real-world code demonstrates the Factory method offering flexibility in creating different documents. The derived Document classes Report and Resume instantiate extended versions of the Document class. Here, the Factory Method is called in the constructor of the Document base class.

|  |
| --- |
| // Factory Method pattern -- Real World example |
| using System;  using System.Collections.Generic;    namespace DoFactory.GangOfFour.Factory.RealWorld  {    /// <summary>    /// MainApp startup class for Real-World    /// Factory Method Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      static void Main()      {        // Note: constructors call Factory Method        Document[] documents = new Document[2];          documents[0] = new Resume();        documents[1] = new Report();          // Display document pages        foreach (Document document in documents)        {          Console.WriteLine("\n" + document.GetType().Name + "--");          foreach (Page page in document.Pages)          {            Console.WriteLine(" " + page.GetType().Name);          }        }          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Product' abstract class    /// </summary>    abstract class Page    {    }      /// <summary>    /// A 'ConcreteProduct' class    /// </summary>    class SkillsPage : Page    {    }      /// <summary>    /// A 'ConcreteProduct' class    /// </summary>    class EducationPage : Page    {    }      /// <summary>    /// A 'ConcreteProduct' class    /// </summary>    class ExperiencePage : Page    {    }      /// <summary>    /// A 'ConcreteProduct' class    /// </summary>    class IntroductionPage : Page    {    }      /// <summary>    /// A 'ConcreteProduct' class    /// </summary>    class ResultsPage : Page    {    }      /// <summary>    /// A 'ConcreteProduct' class    /// </summary>    class ConclusionPage : Page    {    }      /// <summary>    /// A 'ConcreteProduct' class    /// </summary>    class SummaryPage : Page    {    }      /// <summary>    /// A 'ConcreteProduct' class    /// </summary>    class BibliographyPage : Page    {    }      /// <summary>    /// The 'Creator' abstract class    /// </summary>    abstract class Document    {      private List<Page> \_pages = new List<Page>();        // Constructor calls abstract Factory method      public Document()      {        this.CreatePages();      }        public List<Page> Pages      {        get { return \_pages; }      }        // Factory Method      public abstract void CreatePages();    }      /// <summary>    /// A 'ConcreteCreator' class    /// </summary>    class Resume : Document    {      // Factory Method implementation      public override void CreatePages()      {        Pages.Add(new SkillsPage());        Pages.Add(new EducationPage());        Pages.Add(new ExperiencePage());      }    }      /// <summary>    /// A 'ConcreteCreator' class    /// </summary>    class Report : Document    {      // Factory Method implementation      public override void CreatePages()      {        Pages.Add(new IntroductionPage());        Pages.Add(new ResultsPage());        Pages.Add(new ConclusionPage());        Pages.Add(new SummaryPage());        Pages.Add(new BibliographyPage());      }    }  } |



1. [**Prototype**](http://www.dofactory.com/Patterns/PatternPrototype.aspx)

**Definition**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| http://www.dofactory.com/Images/pixel.gif | Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype. UML class diagram http://www.dofactory.com/Patterns/Diagrams/prototype.gif  **Participants**      The classes and/or objects participating in this pattern are:   * **Prototype**  **(ColorPrototype)**   + Declares an interface for cloning itself * **ConcretePrototype**  **(Color)**   + Implements an operation for cloning itself * **Client**  **(ColorManager)**   + Creates a new object by asking a prototype to clone itself  Sample code in C# This structural code demonstrates the Prototype pattern in which new objects are created by copying pre-existing objects (prototypes) of the same class.   |  | | --- | | // Prototype pattern -- Structural example | | using System;    namespace DoFactory.GangOfFour.Prototype.Structural  {    /// <summary>    /// MainApp startup class for Structural    /// Prototype Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      static void Main()      {        // Create two instances and clone each          ConcretePrototype1 p1 = new ConcretePrototype1("I");        ConcretePrototype1 c1 = (ConcretePrototype1)p1.Clone();        Console.WriteLine("Cloned: {0}", c1.Id);          ConcretePrototype2 p2 = new ConcretePrototype2("II");        ConcretePrototype2 c2 = (ConcretePrototype2)p2.Clone();        Console.WriteLine("Cloned: {0}", c2.Id);          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Prototype' abstract class    /// </summary>    abstract class Prototype    {      private string \_id;        // Constructor      public Prototype(string id)      {        this.\_id = id;      }        // Gets id      public string Id      {        get { return \_id; }      }        public abstract Prototype Clone();    }      /// <summary>    /// A 'ConcretePrototype' class    /// </summary>    class ConcretePrototype1 : Prototype    {      // Constructor      public ConcretePrototype1(string id)        : base(id)      {      }        // Returns a shallow copy      public override Prototype Clone()      {        return (Prototype)this.MemberwiseClone();      }    }      /// <summary>    /// A 'ConcretePrototype' class    /// </summary>    class ConcretePrototype2 : Prototype    {      // Constructor      public ConcretePrototype2(string id)        : base(id)      {      }        // Returns a shallow copy      public override Prototype Clone()      {        return (Prototype)this.MemberwiseClone();      }    }  } |     This real-world code demonstrates the Prototype pattern in which new Color objects are created by copying pre-existing, user-defined Colors of the same type.   |  | | --- | | // Prototype pattern -- Real World example | | using System;  using System.Collections.Generic;    namespace DoFactory.GangOfFour.Prototype.RealWorld  {    /// <summary>    /// MainApp startup class for Real-World    /// Prototype Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      static void Main()      {        ColorManager colormanager = new ColorManager();          // Initialize with standard colors        colormanager["red"] = new Color(255, 0, 0);        colormanager["green"] = new Color(0, 255, 0);        colormanager["blue"] = new Color(0, 0, 255);          // User adds personalized colors        colormanager["angry"] = new Color(255, 54, 0);        colormanager["peace"] = new Color(128, 211, 128);        colormanager["flame"] = new Color(211, 34, 20);          // User clones selected colors        Color color1 = colormanager["red"].Clone() as Color;        Color color2 = colormanager["peace"].Clone() as Color;        Color color3 = colormanager["flame"].Clone() as Color;          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Prototype' abstract class    /// </summary>    abstract class ColorPrototype    {      public abstract ColorPrototype Clone();    }      /// <summary>    /// The 'ConcretePrototype' class    /// </summary>    class Color : ColorPrototype    {      private int \_red;      private int \_green;      private int \_blue;        // Constructor      public Color(int red, int green, int blue)      {        this.\_red = red;        this.\_green = green;        this.\_blue = blue;      }        // Create a shallow copy      public override ColorPrototype Clone()      {        Console.WriteLine(          "Cloning color RGB: {0,3},{1,3},{2,3}",          \_red, \_green, \_blue);          return this.MemberwiseClone() as ColorPrototype;      }    }      /// <summary>    /// Prototype manager    /// </summary>    class ColorManager    {      private Dictionary<string, ColorPrototype> \_colors =        new Dictionary<string, ColorPrototype>();        // Indexer      public ColorPrototype this[string key]      {        get { return \_colors[key]; }        set { \_colors.Add(key, value); }      }    }  } | |

1. **Singleton**

**Definition**

|  |  |  |  |
| --- | --- | --- | --- |
| http://www.dofactory.com/Images/pixel.gif | Ensure a class has only one instance and provide a global point of access to it. UML class diagram http://www.dofactory.com/Patterns/Diagrams/singleton.gif  **Participants**      The classes and/or objects participating in this pattern are:   * **Singleton**   **(LoadBalancer)**   + Defines an Instance operation that lets clients access its unique instance. Instance is a class operation.   + Responsible for creating and maintaining its own unique instance.  Sample code in C# This structural code demonstrates the Singleton pattern which assures only a single instance (the singleton) of the class can be created.   |  | | --- | | // Singleton pattern -- Structural example | | using System;    namespace DoFactory.GangOfFour.Singleton.Structural  {    /// <summary>    /// MainApp startup class for Structural    /// Singleton Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      static void Main()      {        // Constructor is protected -- cannot use new        Singleton s1 = Singleton.Instance();        Singleton s2 = Singleton.Instance();          // Test for same instance        if (s1 == s2)        {          Console.WriteLine("Objects are the same instance");        }          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Singleton' class    /// </summary>    class Singleton    {      private static Singleton \_instance;        // Constructor is 'protected'      protected Singleton()      {      }        public static Singleton Instance()      {        // Uses lazy initialization.        // Note: this is not thread safe.        if (\_instance == null)        {          \_instance = new Singleton();        }          return \_instance;      }    }  } |     This real-world code demonstrates the Singleton pattern as a LoadBalancing object. Only a single instance (the singleton) of the class can be created because servers may dynamically come on- or off-line and every request must go throught the one object that has knowledge about the state of the (web) farm. |
| // Singleton pattern -- Real World example | |
| using System;  using System.Collections.Generic;  using System.Threading;    namespace DoFactory.GangOfFour.Singleton.RealWorld  {    /// <summary>    /// MainApp startup class for Real-World    /// Singleton Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      static void Main()      {        LoadBalancer b1 = LoadBalancer.GetLoadBalancer();        LoadBalancer b2 = LoadBalancer.GetLoadBalancer();        LoadBalancer b3 = LoadBalancer.GetLoadBalancer();        LoadBalancer b4 = LoadBalancer.GetLoadBalancer();          // Same instance?        if (b1 == b2 && b2 == b3 && b3 == b4)        {          Console.WriteLine("Same instance\n");        }          // Load balance 15 server requests        LoadBalancer balancer = LoadBalancer.GetLoadBalancer();        for (int i = 0; i < 15; i++)        {          string server = balancer.Server;          Console.WriteLine("Dispatch Request to: " + server);        }          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Singleton' class    /// </summary>    class LoadBalancer    {      private static LoadBalancer \_instance;      private List<string> \_servers = new List<string>();      private Random \_random = new Random();        // Lock synchronization object      private static object syncLock = new object();        // Constructor (protected)      protected LoadBalancer()      {        // List of available servers        \_servers.Add("ServerI");        \_servers.Add("ServerII");        \_servers.Add("ServerIII");        \_servers.Add("ServerIV");        \_servers.Add("ServerV");      }        public static LoadBalancer GetLoadBalancer()      {        // Support multithreaded applications through        // 'Double checked locking' pattern which (once        // the instance exists) avoids locking each        // time the method is invoked        if (\_instance == null)        {          lock (syncLock)          {            if (\_instance == null)            {              \_instance = new LoadBalancer();            }          }        }          return \_instance;      }        // Simple, but effective random load balancer      public string Server      {        get        {          int r = \_random.Next(\_servers.Count);          return \_servers[r].ToString();        }      }    }  } | |

