**DESIGN PATTERNS**

Design patterns are recurring solutions to software design problems you find again and again in real-world application development. Patterns are about design and interaction of objects, as well as providing a communication platform concerning elegant, reusable solutions to commonly encountered programming challenges.

The Gang of Four (GoF) patterns are generally considered the foundation for all other patterns. They are categorized in three groups: Creational, Structural, and Behavioral. Here you will find information on these important patterns.

To give you a head start, the C# source code is provided in 2 forms: 'structural' and 'real-world'. Structural code uses type names as defined in the pattern definition and UML diagrams. Real-world code provides real-world programming situations where you may use these patterns.

|  |  |
| --- | --- |
| **Creational Patterns** | |
| [Abstract Factory](http://www.dofactory.com/Patterns/PatternAbstract.aspx) | 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](http://www.dofactory.com/Patterns/PatternSingleton.aspx) | A class of which only a single instance can exist |

|  |  |
| --- | --- |
| **Structural Patterns** | |
| [Adapter](http://www.dofactory.com/Patterns/PatternAdapter.aspx) | Match interfaces of different classes |
| [Bridge](http://www.dofactory.com/Patterns/PatternBridge.aspx) | Separates an object’s interface from its implementation |
| [Composite](http://www.dofactory.com/Patterns/PatternComposite.aspx) | A tree structure of simple and composite objects |
| [Decorator](http://www.dofactory.com/Patterns/PatternDecorator.aspx) | Add responsibilities to objects dynamically |
| [Facade](http://www.dofactory.com/Patterns/PatternFacade.aspx) | A single class that represents an entire subsystem |
| [Flyweight](http://www.dofactory.com/Patterns/PatternFlyweight.aspx) | A fine-grained instance used for efficient sharing |
| [Proxy](http://www.dofactory.com/Patterns/PatternProxy.aspx) | An object representing another object |

|  |  |
| --- | --- |
| **Behavioral 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 |

|  |
| --- |
| **Creational Patterns** |

**1. Abstract Factory**

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

**2. Builder**

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

**3. Factory Method**

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**4. Prototype**

Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

**5. Singleton**

Ensure a class has only one instance and provide a global point of access to it.

|  |
| --- |
| **Structural Patterns** |

**1. Adapter**

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

**2. Bridge**

Decouple an abstraction from its implementation so that the two can vary independently.

**3. Composite**

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

**4. Decorator**

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

**5. Facade**

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.

**6. Flyweight**

Use sharing to support large numbers of fine-grained objects efficiently.

**7. Proxy**

Provide a surrogate or placeholder for another object to control access to it.

|  |
| --- |
| **Behavioral Patterns** |

**1. Chain of Responsibility**

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

**2. Command**

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

**3. Interpreter**

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

**4. Iterator**

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**5. Mediator**

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

**6. Memento**

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

**7. Observer**

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**8. State**

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

**9. Strategy**

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

**10. Template Method**

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

**11. Visitor**

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

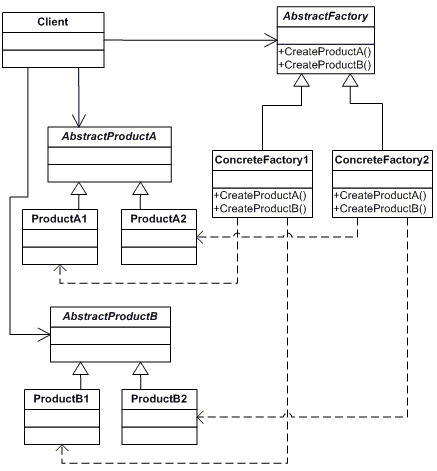
|  |
| --- |
| **Creational Patterns** |

**Abstract Factory**

### definition

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 {   // MainApp test application     class MainApp   {     public static void Main()     {       // Abstract factory #1        AbstractFactory factory1 = new ConcreteFactory1();       Client c1 = new Client(factory1);       c1.Run();        // Abstract factory #2        AbstractFactory factory2 = new ConcreteFactory2();       Client c2 = new Client(factory2);       c2.Run();        // Wait for user input        Console.Read();     }   }    // "AbstractFactory"     abstract class AbstractFactory   {     public abstract AbstractProductA CreateProductA();     public abstract AbstractProductB CreateProductB();   }    // "ConcreteFactory1"     class ConcreteFactory1 : AbstractFactory   {     public override AbstractProductA CreateProductA()     {       return new ProductA1();     }     public override AbstractProductB CreateProductB()     {       return new ProductB1();     }   }    // "ConcreteFactory2"     class ConcreteFactory2 : AbstractFactory   {     public override AbstractProductA CreateProductA()     {       return new ProductA2();     }     public override AbstractProductB CreateProductB()     {       return new ProductB2();     }   }    // "AbstractProductA"     abstract class AbstractProductA   {   }    // "AbstractProductB"     abstract class AbstractProductB   {     public abstract void Interact(AbstractProductA a);   }    // "ProductA1"     class ProductA1 : AbstractProductA   {   }    // "ProductB1"     class ProductB1 : AbstractProductB   {     public override void Interact(AbstractProductA a)     {       Console.WriteLine(this.GetType().Name +          " interacts with " + a.GetType().Name);     }   }    // "ProductA2"     class ProductA2 : AbstractProductA   {   }    // "ProductB2"     class ProductB2 : AbstractProductB   {     public override void Interact(AbstractProductA a)     {       Console.WriteLine(this.GetType().Name +          " interacts with " + a.GetType().Name);     }   }    // "Client" - the interaction environment of the products     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);     }   } } |
| Output  ProductB1 interacts with ProductA1 ProductB2 interacts with ProductA2 |

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.

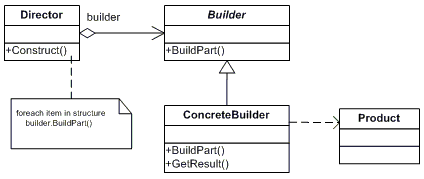
|  |
| --- |
| // Abstract Factory pattern -- Real World example |
| using System;  namespace DoFactory.GangOfFour.Abstract.RealWorld {   // MainApp test application     class MainApp   {     public static void Main()     {       // Create and run the Africa animal world        ContinentFactory africa = new AfricaFactory();       AnimalWorld world = new AnimalWorld(africa);       world.RunFoodChain();        // Create and run the America animal world        ContinentFactory america = new AmericaFactory();       world = new AnimalWorld(america);       world.RunFoodChain();        // Wait for user input        Console.Read();     }   }    // "AbstractFactory"     abstract class ContinentFactory   {     public abstract Herbivore CreateHerbivore();     public abstract Carnivore CreateCarnivore();   }    // "ConcreteFactory1"     class AfricaFactory : ContinentFactory   {     public override Herbivore CreateHerbivore()     {       return new Wildebeest();     }     public override Carnivore CreateCarnivore()     {       return new Lion();     }   }    // "ConcreteFactory2"     class AmericaFactory : ContinentFactory   {     public override Herbivore CreateHerbivore()     {       return new Bison();     }     public override Carnivore CreateCarnivore()     {       return new Wolf();     }   }    // "AbstractProductA"     abstract class Herbivore   {   }    // "AbstractProductB"     abstract class Carnivore   {     public abstract void Eat(Herbivore h);   }    // "ProductA1"     class Wildebeest : Herbivore   {   }    // "ProductB1"     class Lion : Carnivore   {     public override void Eat(Herbivore h)     {       // Eat Wildebeest        Console.WriteLine(this.GetType().Name +          " eats " + h.GetType().Name);     }   }    // "ProductA2"     class Bison : Herbivore   {   }    // "ProductB2"     class Wolf : Carnivore   {     public override void Eat(Herbivore h)     {       // Eat Bison        Console.WriteLine(this.GetType().Name +          " eats " + h.GetType().Name);     }   }    // "Client"     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);     }   } } |
| Output  Lion eats Wildebeest |

**Builder**

### definition

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;  namespace DoFactory.GangOfFour.Builder.Structural {   // MainApp test application     public class MainApp   {     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.Read();     }   }    // "Director"     class Director   {     // Builder uses a complex series of steps      public void Construct(Builder builder)     {       builder.BuildPartA();       builder.BuildPartB();     }   }    // "Builder"     abstract class Builder   {     public abstract void BuildPartA();     public abstract void BuildPartB();     public abstract Product GetResult();   }    // "ConcreteBuilder1"     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;     }   }    // "ConcreteBuilder2"     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;     }   }    // "Product"     class Product   {     ArrayList parts = new ArrayList();      public void Add(string part)     {       parts.Add(part);     }      public void Show()     {       Console.WriteLine("\nProduct Parts -------");       foreach (string part in parts)         Console.WriteLine(part);     }   } } |
| Output  Product Parts ------- PartA PartB  Product Parts ------- PartX PartY |

This real-world code demonstates the Builder pattern in which different vehicles are assembled in a step-by-step fashion. The Shop uses VehicleBuilders to construct a variety of Vehicles in a series of sequential steps.

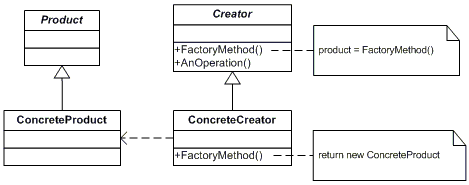
|  |
| --- |
| // Builder pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Builder.RealWorld {   // MainApp test application     public class MainApp   {     public static void Main()     {       // Create shop with vehicle builders        Shop shop = new Shop();       VehicleBuilder b1 = new ScooterBuilder();       VehicleBuilder b2 = new CarBuilder();       VehicleBuilder b3 = new MotorCycleBuilder();        // Construct and display vehicles        shop.Construct(b1);       b1.Vehicle.Show();        shop.Construct(b2);       b2.Vehicle.Show();        shop.Construct(b3);       b3.Vehicle.Show();        // Wait for user        Console.Read();     }   }    // "Director"     class Shop   {     // Builder uses a complex series of steps      public void Construct(VehicleBuilder vehicleBuilder)     {       vehicleBuilder.BuildFrame();       vehicleBuilder.BuildEngine();       vehicleBuilder.BuildWheels();       vehicleBuilder.BuildDoors();     }   }    // "Builder"     abstract class VehicleBuilder   {     protected Vehicle vehicle;      // Property      public Vehicle Vehicle     {       get{ return vehicle; }     }      public abstract void BuildFrame();     public abstract void BuildEngine();     public abstract void BuildWheels();     public abstract void BuildDoors();   }    // "ConcreteBuilder1"     class MotorCycleBuilder : VehicleBuilder   {     public override void BuildFrame()     {       vehicle = new Vehicle("MotorCycle");       vehicle["frame"] = "MotorCycle Frame";     }      public override void BuildEngine()     {       vehicle["engine"] = "500 cc";     }      public override void BuildWheels()     {       vehicle["wheels"] = "2";     }      public override void BuildDoors()     {       vehicle["doors"] = "0";     }   }    // "ConcreteBuilder2"     class CarBuilder : VehicleBuilder   {     public override void BuildFrame()     {       vehicle = new Vehicle("Car");       vehicle["frame"] = "Car Frame";     }      public override void BuildEngine()     {       vehicle["engine"] = "2500 cc";     }      public override void BuildWheels()     {       vehicle["wheels"] = "4";     }      public override void BuildDoors()     {       vehicle["doors"] = "4";     }   }    // "ConcreteBuilder3"     class ScooterBuilder : VehicleBuilder   {     public override void BuildFrame()     {       vehicle = new Vehicle("Scooter");       vehicle["frame"] = "Scooter Frame";     }      public override void BuildEngine()     {       vehicle["engine"] = "50 cc";     }      public override void BuildWheels()     {       vehicle["wheels"] = "2";     }      public override void BuildDoors()     {       vehicle["doors"] = "0";     }   }    // "Product"     class Vehicle   {     private string type;     private Hashtable parts = new Hashtable();      // Constructor      public Vehicle(string type)     {       this.type = type;     }      // Indexer (i.e. smart array)      public object this[string key]     {       get{ return parts[key]; }       set{ parts[key] = value; }     }      public void Show()     {       Console.WriteLine("\n---------------------------");       Console.WriteLine("Vehicle Type: {0}", type);       Console.WriteLine(" Frame : {0}", parts["frame"]);       Console.WriteLine(" Engine : {0}", parts["engine"]);       Console.WriteLine(" #Wheels: {0}", parts["wheels"]);       Console.WriteLine(" #Doors : {0}", parts["doors"]);     }   } } |
| Output  --------------------------- Vehicle Type: Scooter  Frame  : Scooter Frame  Engine : none  #Wheels: 2  #Doors : 0  --------------------------- Vehicle Type: Car  Frame  : Car Frame  Engine : 2500 cc  #Wheels: 4  #Doors : 4  --------------------------- Vehicle Type: MotorCycle  Frame  : MotorCycle Frame  Engine : 500 cc  #Wheels: 2  #Doors : 0 |

**Factory Method**

### definition

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; using System.Collections;  namespace DoFactory.GangOfFour.Factory.Structural {    // MainApp test application     class MainApp   {     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.Read();     }   }    // "Product"     abstract class Product   {   }    // "ConcreteProductA"     class ConcreteProductA : Product   {   }    // "ConcreteProductB"     class ConcreteProductB : Product   {   }    // "Creator"     abstract class Creator   {     public abstract Product FactoryMethod();   }    // "ConcreteCreator"     class ConcreteCreatorA : Creator   {     public override Product FactoryMethod()     {       return new ConcreteProductA();     }   }    // "ConcreteCreator"     class ConcreteCreatorB : Creator   {     public override Product FactoryMethod()     {       return new ConcreteProductB();     }   } } |
| Output  Created ConcreteProductA Created 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.

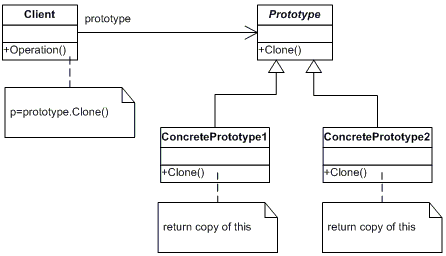
|  |
| --- |
| // Factory Method pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Factory.RealWorld {    // MainApp test application     class MainApp   {     static void Main()     {       // Note: constructors call Factory Method        Document[] 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.Read();         }   }    // "Product"     abstract class Page   {   }    // "ConcreteProduct"     class SkillsPage : Page   {   }    // "ConcreteProduct"     class EducationPage : Page   {   }    // "ConcreteProduct"     class ExperiencePage : Page   {   }    // "ConcreteProduct"     class IntroductionPage : Page   {   }    // "ConcreteProduct"     class ResultsPage : Page   {   }    // "ConcreteProduct"     class ConclusionPage : Page   {   }    // "ConcreteProduct"     class SummaryPage : Page   {   }    // "ConcreteProduct"     class BibliographyPage : Page   {   }    // "Creator"     abstract class Document   {     private ArrayList pages = new ArrayList();      // Constructor calls abstract Factory method      public Document()     {       this.CreatePages();     }      public ArrayList Pages     {       get{ return pages; }     }      // Factory Method      public abstract void CreatePages();   }    // "ConcreteCreator"     class Resume : Document   {     // Factory Method implementation      public override void CreatePages()     {       Pages.Add(new SkillsPage());       Pages.Add(new EducationPage());       Pages.Add(new ExperiencePage());     }   }    // "ConcreteCreator"     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());     }   } } |
| Output  Resume -------  SkillsPage  EducationPage  ExperiencePage  Report -------  IntroductionPage  ResultsPage  ConclusionPage  SummaryPage  BibliographyPage |

**Prototype**

### definition

Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

### UML class diagram



### 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 {      // MainApp test application     class MainApp   {          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.Read();     }   }    // "Prototype"     abstract class Prototype   {     private string id;      // Constructor      public Prototype(string id)     {       this.id = id;     }      // Property      public string Id     {       get{ return id; }     }      public abstract Prototype Clone();   }    // "ConcretePrototype1"     class ConcretePrototype1 : Prototype   {     // Constructor      public ConcretePrototype1(string id) : base(id)      {     }      public override Prototype Clone()     {       // Shallow copy        return (Prototype)this.MemberwiseClone();     }   }    // "ConcretePrototype2"     class ConcretePrototype2 : Prototype   {     // Constructor      public ConcretePrototype2(string id) : base(id)      {     }      public override Prototype Clone()     {       // Shallow copy        return (Prototype)this.MemberwiseClone();     }   } } |
| Output  Cloned: I Cloned: II |

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.

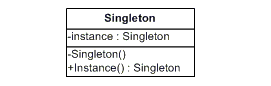
|  |
| --- |
| // Prototype pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Prototype.RealWorld {      // MainApp test application     class MainApp   {     static void Main()     {       ColorManager colormanager = new ColorManager();        // 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);        Color color;        // User uses selected colors        string name = "red";       color = colormanager[name].Clone() as Color;        name = "peace";       color = colormanager[name].Clone() as Color;        name = "flame";       color = colormanager[name].Clone() as Color;        // Wait for user        Console.Read();     }   }    // "Prototype"     abstract class ColorPrototype   {     public abstract ColorPrototype Clone();   }    // "ConcretePrototype"     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;     }   }    // Prototype manager     class ColorManager   {     Hashtable colors = new Hashtable();      // Indexer      public ColorPrototype this[string name]     {       get       {          return colors[name] as ColorPrototype;        }       set       {          colors.Add(name, value);        }     }   } } |
| Output  Cloning color RGB: 255,  0,  0 Cloning color RGB: 128,211,128 Cloning color RGB: 211, 34, 20 |

**Singleton**

### definition

Ensure a class has only one instance and provide a global point of access to it.

### UML class diagram



### 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 {      // MainApp test application     class MainApp   {          static void Main()     {       // Constructor is protected -- cannot use new        Singleton s1 = Singleton.Instance();       Singleton s2 = Singleton.Instance();        if (s1 == s2)       {         Console.WriteLine("Objects are the same instance");       }        // Wait for user        Console.Read();     }   }    // "Singleton"     class Singleton   {     private static Singleton instance;      // Note: Constructor is 'protected'      protected Singleton()      {     }      public static Singleton Instance()     {       // Use 'Lazy initialization'        if (instance == null)       {         instance = new Singleton();       }        return instance;     }   } } |
| Output  Objects are the same 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; using System.Threading;  namespace DoFactory.GangOfFour.Singleton.RealWorld {      // MainApp test application     class MainApp   {     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");       }        // All are the same instance -- use b1 arbitrarily        // Load balance 15 server requests        for (int i = 0; i < 15; i++)       {         Console.WriteLine(b1.Server);       }        // Wait for user        Console.Read();         }   }    // "Singleton"     class LoadBalancer   {     private static LoadBalancer instance;     private ArrayList servers = new ArrayList();      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();       }     }   } } |
| Output  Same instance  ServerIII ServerII ServerI ServerII ServerI ServerIII ServerI ServerIII ServerIV ServerII ServerII ServerIII ServerIV ServerII ServerIV |

|  |
| --- |
| // Singleton pattern -- .NET optimized  using System;  using System.Collections.Generic;  namespace DoFactory.GangOfFour.Singleton.NETOptimized  {  /// <summary>  /// MainApp startup class for .NET optimized  /// 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();  // Confirm these are the same instance  if (b1 == b2 && b2 == b3 && b3 == b4)  {  Console.WriteLine("Same instance\n");  }    // Next, load balance 15 requests for a server  LoadBalancer balancer = LoadBalancer.GetLoadBalancer();  for (int i = 0; i < 15; i++)  {  string serverName = balancer.NextServer.Name;  Console.WriteLine("Dispatch request to: " + serverName);  }  // Wait for user  Console.ReadKey();  }  }    /// <summary>  /// The 'Singleton' class  /// </summary>  sealed class LoadBalancer  {  // Static members are 'eagerly initialized', that is,  // immediately when class is loaded for the first time.  // .NET guarantees thread safety for static initialization  private static readonly LoadBalancer \_instance = new LoadBalancer();    // Type-safe generic list of servers  private List<Server> \_servers;  private Random \_random = new Random();  // Note: constructor is 'private'  private LoadBalancer()  {  // Load list of available servers  \_servers = new List<Server>  {  new Server{ Name = "ServerI", IP = "120.14.220.18" },  new Server{ Name = "ServerII", IP = "120.14.220.19" },  new Server{ Name = "ServerIII", IP = "120.14.220.20" },  new Server{ Name = "ServerIV", IP = "120.14.220.21" },  new Server{ Name = "ServerV", IP = "120.14.220.22" },  };  }    public static LoadBalancer GetLoadBalancer()  {  return \_instance;  }    // Simple, but effective load balancer  public Server NextServer  {  get  {  int r = \_random.Next(\_servers.Count);  return \_servers[r];  }  }  }    /// <summary>  /// Represents a server machine  /// </summary>  class Server  {  // Gets or sets server name  public string Name { get; set; }  // Gets or sets server IP address  public string IP { get; set; }  }  } |
|  |

Output



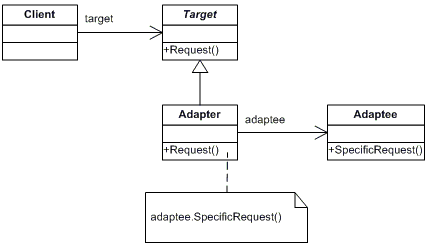
|  |
| --- |
| **Structural Patterns** |

**Adapter**

### definition

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

### UML class diagram



### participants

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

* **Target**   **(ChemicalCompound)**
  + defines the domain-specific interface that Client uses.
* **Adapter**   **(Compound)**
  + adapts the interface Adaptee to the Target interface.
* **Adaptee**   **(ChemicalDatabank)**
  + defines an existing interface that needs adapting.
* **Client**   **(AdapterApp)**
  + collaborates with objects conforming to the Target interface.

### sample code in C#

This structural code demonstrates the Adapter pattern which maps the interface of one class onto another so that they can work together. These incompatible classes may come from different libraries or frameworks.

|  |
| --- |
| // Adapter pattern -- Structural example |
| using System;  namespace DoFactory.GangOfFour.Adapter.Structural {    // Mainapp test application     class MainApp   {     static void Main()     {       // Create adapter and place a request        Target target = new Adapter();       target.Request();        // Wait for user        Console.Read();     }   }    // "Target"     class Target   {     public virtual void Request()     {       Console.WriteLine("Called Target Request()");     }   }    // "Adapter"     class Adapter : Target   {     private Adaptee adaptee = new Adaptee();      public override void Request()     {       // Possibly do some other work        // and then call SpecificRequest        adaptee.SpecificRequest();     }   }    // "Adaptee"     class Adaptee   {     public void SpecificRequest()     {       Console.WriteLine("Called SpecificRequest()");     }   } } |
| Output  Called SpecificRequest() |

This real-world code demonstrates the use of a legacy chemical databank. Chemical compound objects access the databank through an Adapter interface.

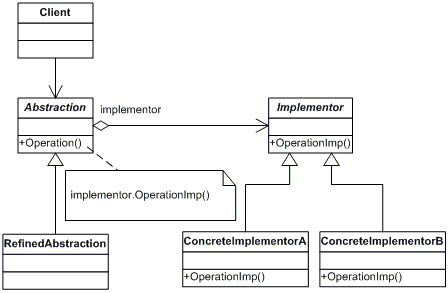
|  |
| --- |
| // Adapter pattern -- Real World example |
| using System;  namespace DoFactory.GangOfFour.Adapter.RealWorld {    // MainApp test application     class MainApp   {     static void Main()     {       // Non-adapted chemical compound        Compound stuff = new Compound("Unknown");       stuff.Display();              // Adapted chemical compounds        Compound water = new RichCompound("Water");       water.Display();        Compound benzene = new RichCompound("Benzene");       benzene.Display();        Compound alcohol = new RichCompound("Alcohol");       alcohol.Display();        // Wait for user        Console.Read();     }   }    // "Target"     class Compound   {     protected string name;     protected float boilingPoint;     protected float meltingPoint;     protected double molecularWeight;     protected string molecularFormula;      // Constructor      public Compound(string name)     {       this.name = name;     }      public virtual void Display()     {       Console.WriteLine("\nCompound: {0} ------ ", name);     }   }    // "Adapter"     class RichCompound : Compound   {     private ChemicalDatabank bank;      // Constructor      public RichCompound(string name) : base(name)     {     }      public override void Display()     {       // Adaptee        bank = new ChemicalDatabank();       boilingPoint = bank.GetCriticalPoint(name, "B");       meltingPoint = bank.GetCriticalPoint(name, "M");       molecularWeight = bank.GetMolecularWeight(name);       molecularFormula = bank.GetMolecularStructure(name);        base.Display();       Console.WriteLine(" Formula: {0}", molecularFormula);       Console.WriteLine(" Weight : {0}", molecularWeight);       Console.WriteLine(" Melting Pt: {0}", meltingPoint);       Console.WriteLine(" Boiling Pt: {0}", boilingPoint);     }   }    // "Adaptee"     class ChemicalDatabank   {     // The Databank 'legacy API'      public float GetCriticalPoint(string compound, string point)     {       float temperature = 0.0F;        // Melting Point        if (point == "M")       {         switch (compound.ToLower())         {           case "water" : temperature = 0.0F; break;           case "benzene" : temperature = 5.5F; break;           case "alcohol" : temperature = -114.1F; break;         }       }       // Boiling Point        else       {         switch (compound.ToLower())         {           case "water" : temperature = 100.0F; break;           case "benzene" : temperature = 80.1F; break;           case "alcohol" : temperature = 78.3F; break;         }       }       return temperature;     }      public string GetMolecularStructure(string compound)     {       string structure = "";        switch (compound.ToLower())       {         case "water" : structure = "H20"; break;         case "benzene" : structure = "C6H6"; break;         case "alcohol" : structure = "C2H6O2"; break;       }       return structure;     }      public double GetMolecularWeight(string compound)     {       double weight = 0.0;       switch (compound.ToLower())       {         case "water" : weight = 18.015; break;         case "benzene" : weight = 78.1134; break;         case "alcohol" : weight = 46.0688; break;       }       return weight;     }   } } |
| Output  Compound: Unknown ------  Compound: Water ------  Formula: H20  Weight : 18.015  Melting Pt: 0  Boiling Pt: 100  Compound: Benzene ------  Formula: C6H6  Weight : 78.1134  Melting Pt: 5.5  Boiling Pt: 80.1  Compound: Alcohol ------  Formula: C2H6O2  Weight : 46.0688  Melting Pt: -114.1  Boiling Pt: 78.3 |

**Bridge**

### definition

Decouple an abstraction from its implementation so that the two can vary independently.

### UML class diagram



### participants

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

* **Abstraction**   **(BusinessObject)**
  + defines the abstraction's interface.
  + maintains a reference to an object of type Implementor.
* **RefinedAbstraction**   **(CustomersBusinessObject)**
  + extends the interface defined by Abstraction.
* **Implementor**   **(DataObject)**
  + defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementation interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
* **ConcreteImplementor**   **(CustomersDataObject)**
  + implements the Implementor interface and defines its concrete implementation.

### sample code in C#

This structural code demonstrates the Bridge pattern which separates (decouples) the interface from its implementation. The implementation can evolve without changing clients which use the abstraction of the object.

|  |
| --- |
| // Bridge pattern -- Structural example |
| using System;  namespace DoFactory.GangOfFour.Bridge.Structural {    // MainApp test application     class MainApp   {     static void Main()     {       Abstraction ab = new RefinedAbstraction();        // Set implementation and call        ab.Implementor = new ConcreteImplementorA();       ab.Operation();        // Change implemention and call        ab.Implementor = new ConcreteImplementorB();       ab.Operation();        // Wait for user        Console.Read();     }   }    // "Abstraction"     class Abstraction   {     protected Implementor implementor;      // Property      public Implementor Implementor     {       set{ implementor = value; }     }      public virtual void Operation()     {       implementor.Operation();     }   }    // "Implementor"     abstract class Implementor   {     public abstract void Operation();   }    // "RefinedAbstraction"     class RefinedAbstraction : Abstraction   {     public override void Operation()     {       implementor.Operation();     }   }    // "ConcreteImplementorA"     class ConcreteImplementorA : Implementor   {     public override void Operation()     {       Console.WriteLine("ConcreteImplementorA Operation");     }   }    // "ConcreteImplementorB"     class ConcreteImplementorB : Implementor   {     public override void Operation()     {       Console.WriteLine("ConcreteImplementorB Operation");     }   } } |
| Output  ConcreteImplementorA Operation ConcreteImplementorB Operation |

This real-world code demonstrates the Bridge pattern in which a BusinessObject abstraction is decoupled from the implementation in DataObject. The DataObject implementations can evolve dynamically without changing any clients.

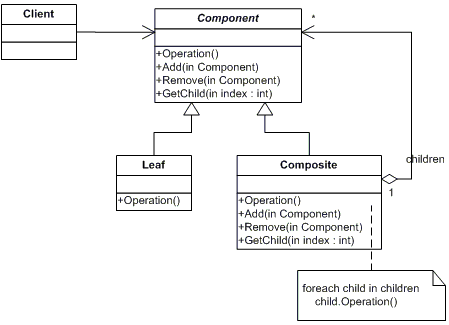
|  |
| --- |
| // Bridge pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Bridge.RealWorld {    // MainApp test application       class MainApp   {     static void Main()     {       // Create RefinedAbstraction        Customers customers =          new Customers("Chicago");        // Set ConcreteImplementor        customers.Data = new CustomersData();        // Exercise the bridge        customers.Show();       customers.Next();       customers.Show();       customers.Next();       customers.Show();       customers.New("Henry Velasquez");        customers.ShowAll();        // Wait for user        Console.Read();     }   }    // "Abstraction"     class CustomersBase   {     private DataObject dataObject;     protected string group;      public CustomersBase(string group)     {       this.group = group;     }      // Property      public DataObject Data     {       set{ dataObject = value; }       get{ return dataObject; }     }      public virtual void Next()     {       dataObject.NextRecord();     }      public virtual void Prior()     {       dataObject.PriorRecord();     }      public virtual void New(string name)     {       dataObject.NewRecord(name);     }      public virtual void Delete(string name)     {       dataObject.DeleteRecord(name);     }      public virtual void Show()     {       dataObject.ShowRecord();     }      public virtual void ShowAll()     {       Console.WriteLine("Customer Group: " + group);       dataObject.ShowAllRecords();     }   }    // "RefinedAbstraction"     class Customers : CustomersBase   {     // Constructor      public Customers(string group) : base(group)     {       }      public override void ShowAll()     {       // Add separator lines        Console.WriteLine();       Console.WriteLine ("------------------------");       base.ShowAll();       Console.WriteLine ("------------------------");     }   }    // "Implementor"     abstract class DataObject   {     public abstract void NextRecord();     public abstract void PriorRecord();     public abstract void NewRecord(string name);     public abstract void DeleteRecord(string name);     public abstract void ShowRecord();     public abstract void ShowAllRecords();   }    // "ConcreteImplementor"     class CustomersData : DataObject   {     private ArrayList customers = new ArrayList();     private int current = 0;      public CustomersData()      {       // Loaded from a database        customers.Add("Jim Jones");       customers.Add("Samual Jackson");       customers.Add("Allen Good");       customers.Add("Ann Stills");       customers.Add("Lisa Giolani");     }      public override void NextRecord()     {       if (current <= customers.Count - 1)       {         current++;       }     }      public override void PriorRecord()     {       if (current > 0)       {         current--;       }     }      public override void NewRecord(string name)     {       customers.Add(name);     }      public override void DeleteRecord(string name)     {       customers.Remove(name);     }      public override void ShowRecord()     {       Console.WriteLine(customers[current]);     }      public override void ShowAllRecords()     {       foreach (string name in customers)       {         Console.WriteLine(" " + name);       }     }   } } |
| Output  Jim Jones Samual Jackson Allen Good  ------------------------ Customer Group: Chicago Jim Jones Samual Jackson Allen Good Ann Stills Lisa Giolani Henry Velasquez ------------------------ |

**Composite**

### definition

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

### UML class diagram



### participants

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

* **Component**   **(DrawingElement)**
  + declares the interface for objects in the composition.
  + implements default behavior for the interface common to all classes, as appropriate.
  + declares an interface for accessing and managing its child components.
  + (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
* **Leaf**   **(PrimitiveElement)**
  + represents leaf objects in the composition. A leaf has no children.
  + defines behavior for primitive objects in the composition.
* **Composite**   **(CompositeElement)**
  + defines behavior for components having children.
  + stores child components.
  + implements child-related operations in the Component interface.
* **Client**  **(CompositeApp)**
  + manipulates objects in the composition through the Component interface.

### sample code in C#

This structural code demonstrates the Composite pattern which allows the creation of a tree structure in which individual nodes are accessed uniformly whether they are leaf nodes or branch (composite) nodes.

|  |
| --- |
| // Composite pattern -- Structural example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Composite.Structural {    // MainApp test application     class MainApp   {     static void Main()     {       // Create a tree structure        Composite root = new Composite("root");       root.Add(new Leaf("Leaf A"));       root.Add(new Leaf("Leaf B"));        Composite comp = new Composite("Composite X");       comp.Add(new Leaf("Leaf XA"));       comp.Add(new Leaf("Leaf XB"));        root.Add(comp);       root.Add(new Leaf("Leaf C"));        // Add and remove a leaf        Leaf leaf = new Leaf("Leaf D");       root.Add(leaf);       root.Remove(leaf);        // Recursively display tree        root.Display(1);        // Wait for user        Console.Read();     }   }    // "Component"     abstract class Component   {     protected string name;      // Constructor      public Component(string name)     {       this.name = name;     }      public abstract void Add(Component c);     public abstract void Remove(Component c);     public abstract void Display(int depth);   }    // "Composite"     class Composite : Component   {     private ArrayList children = new ArrayList();      // Constructor      public Composite(string name) : base(name)      {       }      public override void Add(Component component)     {       children.Add(component);     }      public override void Remove(Component component)     {       children.Remove(component);     }      public override void Display(int depth)     {       Console.WriteLine(new String('-', depth) + name);        // Recursively display child nodes        foreach (Component component in children)       {         component.Display(depth + 2);       }     }   }    // "Leaf"     class Leaf : Component   {     // Constructor      public Leaf(string name) : base(name)      {       }      public override void Add(Component c)     {       Console.WriteLine("Cannot add to a leaf");     }      public override void Remove(Component c)     {       Console.WriteLine("Cannot remove from a leaf");     }      public override void Display(int depth)     {       Console.WriteLine(new String('-', depth) + name);     }   } } |
| Output  -root ---Leaf A ---Leaf B ---Composite X -----Leaf XA -----Leaf XB ---Leaf C |

This real-world code demonstrates the Composite pattern used in building a graphical tree structure made up of primitive nodes (lines, circles, etc) and composite nodes (groups of drawing elements that make up more complex elements).

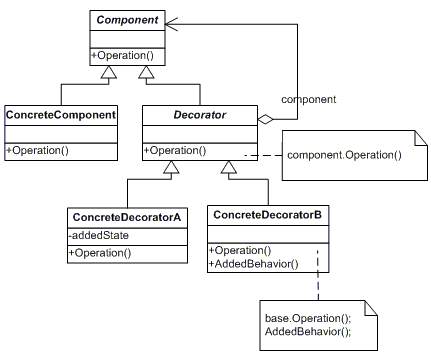
|  |
| --- |
| // Composite pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Composite.RealWorld {      // Mainapp test application     class MainApp   {     static void Main()     {       // Create a tree structure        CompositeElement root =          new CompositeElement("Picture");       root.Add(new PrimitiveElement("Red Line"));       root.Add(new PrimitiveElement("Blue Circle"));       root.Add(new PrimitiveElement("Green Box"));        CompositeElement comp =          new CompositeElement("Two Circles");       comp.Add(new PrimitiveElement("Black Circle"));       comp.Add(new PrimitiveElement("White Circle"));       root.Add(comp);        // Add and remove a PrimitiveElement        PrimitiveElement pe =          new PrimitiveElement("Yellow Line");       root.Add(pe);       root.Remove(pe);        // Recursively display nodes        root.Display(1);        // Wait for user        Console.Read();     }   }    // "Component" Treenode     abstract class DrawingElement   {     protected string name;      // Constructor      public DrawingElement(string name)     {       this.name = name;     }      public abstract void Add(DrawingElement d);     public abstract void Remove(DrawingElement d);     public abstract void Display(int indent);   }    // "Leaf"     class PrimitiveElement : DrawingElement   {     // Constructor      public PrimitiveElement(string name) : base(name)      {       }      public override void Add(DrawingElement c)     {       Console.WriteLine(         "Cannot add to a PrimitiveElement");     }      public override void Remove(DrawingElement c)     {       Console.WriteLine(         "Cannot remove from a PrimitiveElement");     }      public override void Display(int indent)     {       Console.WriteLine(         new String('-', indent) + " " + name);     }   }    // "Composite"     class CompositeElement : DrawingElement   {     private ArrayList elements = new ArrayList();        // Constructor      public CompositeElement(string name) : base(name)      {       }      public override void Add(DrawingElement d)     {       elements.Add(d);     }      public override void Remove(DrawingElement d)     {       elements.Remove(d);     }      public override void Display(int indent)     {       Console.WriteLine(new String('-', indent) +          "+ " + name);        // Display each child element on this node        foreach (DrawingElement c in elements)       {         c.Display(indent + 2);       }     }   } } |
| Output  -+ Picture --- Red Line --- Blue Circle --- Green Box ---+ Two Circles ----- Black Circle ----- White Circle |

**Decorator**

### definition

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

### UML class diagram



### participants

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

* **Component**   **(LibraryItem)**
  + defines the interface for objects that can have responsibilities added to them dynamically.
* **ConcreteComponent**   **(Book, Video)**
  + defines an object to which additional responsibilities can be attached.
* **Decorator**   **(Decorator)**
  + maintains a reference to a Component object and defines an interface that conforms to Component's interface.
* **ConcreteDecorator**   **(Borrowable)**
  + adds responsibilities to the component.

### sample code in C#

This structural code demonstrates the Decorator pattern which dynamically adds extra functionality to an existing object.

|  |
| --- |
| // Decorator pattern -- Structural example |
| using System;  namespace DoFactory.GangOfFour.Decorator.Structural {    // MainApp test application     class MainApp   {     static void Main()     {       // Create ConcreteComponent and two Decorators        ConcreteComponent c = new ConcreteComponent();       ConcreteDecoratorA d1 = new ConcreteDecoratorA();       ConcreteDecoratorB d2 = new ConcreteDecoratorB();        // Link decorators        d1.SetComponent(c);       d2.SetComponent(d1);        d2.Operation();        // Wait for user        Console.Read();     }   }    // "Component"     abstract class Component   {     public abstract void Operation();   }    // "ConcreteComponent"     class ConcreteComponent : Component   {     public override void Operation()     {       Console.WriteLine("ConcreteComponent.Operation()");     }   }    // "Decorator"     abstract class Decorator : Component   {     protected Component component;      public void SetComponent(Component component)     {       this.component = component;     }      public override void Operation()     {       if (component != null)       {         component.Operation();       }     }   }    // "ConcreteDecoratorA"     class ConcreteDecoratorA : Decorator   {     private string addedState;      public override void Operation()     {       base.Operation();       addedState = "New State";       Console.WriteLine("ConcreteDecoratorA.Operation()");     }   }    // "ConcreteDecoratorB"     class ConcreteDecoratorB : Decorator   {     public override void Operation()     {       base.Operation();       AddedBehavior();       Console.WriteLine("ConcreteDecoratorB.Operation()");     }      void AddedBehavior()     {     }   } } |
| Output  ConcreteComponent.Operation() ConcreteDecoratorA.Operation() ConcreteDecoratorB.Operation() |

This real-world code demonstrates the Decorator pattern in which 'borrowable' functionality is added to existing library items (books and videos).

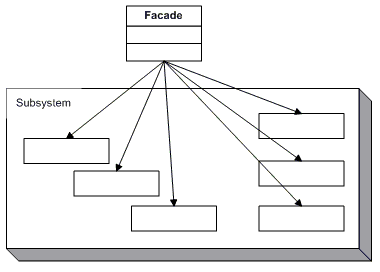
|  |
| --- |
| // Decorator pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Decorator.RealWorld {    // MainApp test application     class MainApp   {     static void Main()     {       // Create book        Book book = new Book ("Worley", "Inside ASP.NET", 10);       book.Display();        // Create video        Video video = new Video ("Spielberg", "Jaws", 23, 92);       video.Display();        // Make video borrowable, then borrow and display        Console.WriteLine("\nMaking video borrowable:");        Borrowable borrowvideo = new Borrowable(video);       borrowvideo.BorrowItem("Customer #1");       borrowvideo.BorrowItem("Customer #2");        borrowvideo.Display();        // Wait for user        Console.Read();     }   }    // "Component"     abstract class LibraryItem   {     private int numCopies;      // Property      public int NumCopies     {       get{ return numCopies; }       set{ numCopies = value; }     }      public abstract void Display();   }    // "ConcreteComponent"     class Book : LibraryItem   {     private string author;     private string title;      // Constructor      public Book(string author,string title,int numCopies)     {       this.author = author;       this.title = title;       this.NumCopies = numCopies;     }      public override void Display()     {       Console.WriteLine("\nBook ------ ");       Console.WriteLine(" Author: {0}", author);       Console.WriteLine(" Title: {0}", title);       Console.WriteLine(" # Copies: {0}", NumCopies);     }   }    // "ConcreteComponent"     class Video : LibraryItem   {     private string director;     private string title;     private int playTime;      // Constructor      public Video(string director, string title,        int numCopies, int playTime)     {       this.director = director;       this.title = title;       this.NumCopies = numCopies;       this.playTime = playTime;     }      public override void Display()     {       Console.WriteLine("\nVideo ----- ");       Console.WriteLine(" Director: {0}", director);       Console.WriteLine(" Title: {0}", title);       Console.WriteLine(" # Copies: {0}", NumCopies);       Console.WriteLine(" Playtime: {0}\n", playTime);     }   }    // "Decorator"     abstract class Decorator : LibraryItem   {     protected LibraryItem libraryItem;      // Constructor      public Decorator(LibraryItem libraryItem)     {       this.libraryItem = libraryItem;     }      public override void Display()     {       libraryItem.Display();     }   }    // "ConcreteDecorator"     class Borrowable : Decorator   {     protected ArrayList borrowers = new ArrayList();      // Constructor      public Borrowable(LibraryItem libraryItem)        : base(libraryItem)      {     }      public void BorrowItem(string name)     {       borrowers.Add(name);       libraryItem.NumCopies--;     }      public void ReturnItem(string name)     {       borrowers.Remove(name);       libraryItem.NumCopies++;     }      public override void Display()     {       base.Display();              foreach (string borrower in borrowers)       {         Console.WriteLine(" borrower: " + borrower);       }     }   } } |
| Output  Book ------ Author: Worley Title: Inside ASP.NET # Copies: 10  Video ----- Director: Spielberg Title: Jaws # Copies: 23 Playtime: 92   Making video borrowable:  Video ----- Director: Spielberg Title: Jaws # Copies: 21 Playtime: 92  borrower: Customer #1 borrower: Customer #2 |

**Façade**

### definition

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 {    // Mainapp test application     class MainApp   {     public static void Main()     {       Facade facade = new Facade();        facade.MethodA();       facade.MethodB();        // Wait for user        Console.Read();     }   }    // "Subsystem ClassA"     class SubSystemOne   {     public void MethodOne()     {       Console.WriteLine(" SubSystemOne Method");     }   }    // Subsystem ClassB"     class SubSystemTwo   {     public void MethodTwo()     {       Console.WriteLine(" SubSystemTwo Method");     }   }    // Subsystem ClassC"     class SubSystemThree   {     public void MethodThree()     {       Console.WriteLine(" SubSystemThree Method");     }   }    // Subsystem ClassD"     class SubSystemFour   {     public void MethodFour()     {       Console.WriteLine(" SubSystemFour Method");     }   }    // "Facade"     class Facade   {     SubSystemOne one;     SubSystemTwo two;     SubSystemThree three;     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 |

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.

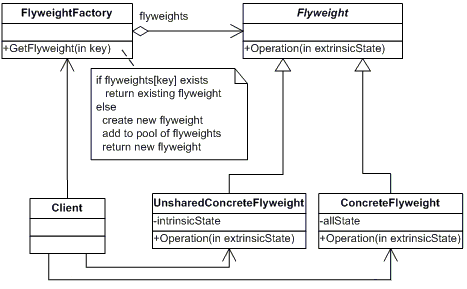
|  |
| --- |
| // Facade pattern -- Real World example |
| using System;  namespace DoFactory.GangOfFour.Facade.RealWorld {   // MainApp test application     class MainApp   {     static void Main()     {       // Facade        Mortgage mortgage = new Mortgage();        // Evaluate mortgage eligibility for customer        Customer customer = new Customer("Ann McKinsey");       bool eligable = mortgage.IsEligible(customer,125000);              Console.WriteLine("\n" + customer.Name +            " has been " + (eligable ? "Approved" : "Rejected"));        // Wait for user        Console.Read();     }   }    // "Subsystem ClassA"     class Bank   {     public bool HasSufficientSavings(Customer c, int amount)     {       Console.WriteLine("Check bank for " + c.Name);       return true;     }   }    // "Subsystem ClassB"     class Credit   {     public bool HasGoodCredit(Customer c)     {       Console.WriteLine("Check credit for " + c.Name);       return true;     }   }    // "Subsystem ClassC"     class Loan   {     public bool HasNoBadLoans(Customer c)     {       Console.WriteLine("Check loans for " + c.Name);       return true;     }   }    class Customer   {     private string name;      // Constructor      public Customer(string name)     {       this.name = name;     }      // Property      public string Name     {       get{ return name; }     }   }    // "Facade"     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;     }   } } |
| Output  Ann McKinsey applies for $125,000.00 loan  Check bank for Ann McKinsey Check loans for Ann McKinsey Check credit for Ann McKinsey  Ann McKinsey has been Approved |

**Flyweight**

### definition

Use sharing to support large numbers of fine-grained objects efficiently.

### UML class diagram



### participants

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

* **Flyweight**   **(Character)**
  + declares an interface through which flyweights can receive and act on extrinsic state.
* **ConcreteFlyweight**   **(CharacterA, CharacterB, ..., CharacterZ)**
  + implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic, that is, it must be independent of the ConcreteFlyweight object's context.
* **UnsharedConcreteFlyweight**   **( not used )**
  + not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing, but it doesn't enforce it. It is common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).
* **FlyweightFactory**   **(CharacterFactory)**
  + creates and manages flyweight objects
  + ensures that flyweight are shared properly. When a client requests a flyweight, the FlyweightFactory objects supplies an existing instance or creates one, if none exists.
* **Client**   **(FlyweightApp)**
  + maintains a reference to flyweight(s).
  + computes or stores the extrinsic state of flyweight(s).

### sample code in C#

This structural code demonstrates the Flyweight pattern in which a relatively small number of objects is shared many times by different clients.

|  |
| --- |
| // Flyweight pattern -- Structural example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Flyweight.Structural {   // MainApp test application     class MainApp   {     static void Main()     {       // Arbitrary extrinsic state        int extrinsicstate = 22;            FlyweightFactory f = new FlyweightFactory();        // Work with different flyweight instances        Flyweight fx = f.GetFlyweight("X");       fx.Operation(--extrinsicstate);        Flyweight fy = f.GetFlyweight("Y");       fy.Operation(--extrinsicstate);        Flyweight fz = f.GetFlyweight("Z");       fz.Operation(--extrinsicstate);        UnsharedConcreteFlyweight uf = new          UnsharedConcreteFlyweight();        uf.Operation(--extrinsicstate);        // Wait for user        Console.Read();     }   }    // "FlyweightFactory"     class FlyweightFactory    {     private Hashtable flyweights = new Hashtable();      // Constructor      public FlyweightFactory()     {       flyweights.Add("X", new ConcreteFlyweight());           flyweights.Add("Y", new ConcreteFlyweight());       flyweights.Add("Z", new ConcreteFlyweight());     }      public Flyweight GetFlyweight(string key)     {       return((Flyweight)flyweights[key]);      }   }    // "Flyweight"     abstract class Flyweight    {     public abstract void Operation(int extrinsicstate);   }    // "ConcreteFlyweight"     class ConcreteFlyweight : Flyweight   {     public override void Operation(int extrinsicstate)     {       Console.WriteLine("ConcreteFlyweight: " + extrinsicstate);     }   }    // "UnsharedConcreteFlyweight"     class UnsharedConcreteFlyweight : Flyweight   {     public override void Operation(int extrinsicstate)     {       Console.WriteLine("UnsharedConcreteFlyweight: " +          extrinsicstate);     }   } } |
| Output  ConcreteFlyweight: 21 ConcreteFlyweight: 20 ConcreteFlyweight: 19 UnsharedConcreteFlyweight: 18 |

This real-world code demonstrates the Flyweight pattern in which a relatively small number of Character objects is shared many times by a document that has potentially many characters.

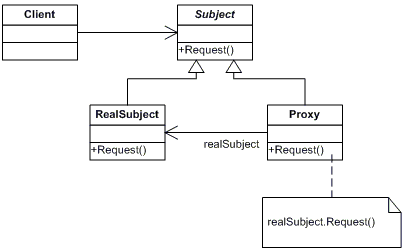
|  |
| --- |
| // Flyweight pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Flyweight.RealWorld {    // MainApp test application     class MainApp   {     static void Main()     {       // Build a document with text        string document = "AAZZBBZB";       char[] chars = document.ToCharArray();        CharacterFactory f = new CharacterFactory();        // extrinsic state        int pointSize = 10;        // For each character use a flyweight object        foreach (char c in chars)       {         pointSize++;         Character character = f.GetCharacter(c);         character.Display(pointSize);       }        // Wait for user        Console.Read();     }   }    // "FlyweightFactory"     class CharacterFactory   {     private Hashtable characters = new Hashtable();      public Character GetCharacter(char key)     {       // Uses "lazy initialization"        Character character = characters[key] as Character;       if (character == null)       {         switch (key)         {           case 'A': character = new CharacterA(); break;           case 'B': character = new CharacterB(); break;             //...            case 'Z': character = new CharacterZ(); break;         }         characters.Add(key, character);       }       return character;     }   }    // "Flyweight"     abstract class Character   {     protected char symbol;     protected int width;     protected int height;     protected int ascent;     protected int descent;     protected int pointSize;      public abstract void Display(int pointSize);   }    // "ConcreteFlyweight"     class CharacterA : Character   {     // Constructor      public CharacterA()     {       this.symbol = 'A';       this.height = 100;       this.width = 120;       this.ascent = 70;       this.descent = 0;     }      public override void Display(int pointSize)     {       this.pointSize = pointSize;       Console.WriteLine(this.symbol +          " (pointsize " + this.pointSize + ")");     }   }    // "ConcreteFlyweight"     class CharacterB : Character   {     // Constructor      public CharacterB()     {       this.symbol = 'B';       this.height = 100;       this.width = 140;       this.ascent = 72;       this.descent = 0;     }      public override void Display(int pointSize)     {       this.pointSize = pointSize;       Console.WriteLine(this.symbol +          " (pointsize " + this.pointSize + ")");     }    }    // ... C, D, E, etc.     // "ConcreteFlyweight"     class CharacterZ : Character   {     // Constructor      public CharacterZ()     {       this.symbol = 'Z';       this.height = 100;       this.width = 100;       this.ascent = 68;       this.descent = 0;     }      public override void Display(int pointSize)     {       this.pointSize = pointSize;       Console.WriteLine(this.symbol +          " (pointsize " + this.pointSize + ")");     }   } } |
| Output  A (pointsize 11) A (pointsize 12) Z (pointsize 13) Z (pointsize 14) B (pointsize 15) B (pointsize 16) Z (pointsize 17) B (pointsize 18) |

**Proxy**

### definition

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 {      // MainApp test application     class MainApp   {     static void Main()     {       // Create proxy and request a service        Proxy proxy = new Proxy();       proxy.Request();        // Wait for user        Console.Read();     }   }    // "Subject"     abstract class Subject    {     public abstract void Request();       }    // "RealSubject"     class RealSubject : Subject   {     public override void Request()     {       Console.WriteLine("Called RealSubject.Request()");     }   }    // "Proxy"     class Proxy : Subject   {     RealSubject realSubject;      public override void Request()     {       // Use 'lazy initialization'        if (realSubject == null)       {         realSubject = new RealSubject();       }        realSubject.Request();     }     } } |
| Output  Called 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 {      // Mainapp test application     class MainApp   {     static void Main()     {       // Create math proxy        MathProxy p = new MathProxy();        // Do the math        Console.WriteLine("4 + 2 = " + p.Add(4, 2));       Console.WriteLine("4 - 2 = " + p.Sub(4, 2));       Console.WriteLine("4 \* 2 = " + p.Mul(4, 2));       Console.WriteLine("4 / 2 = " + p.Div(4, 2));        // Wait for user        Console.Read();     }   }    // "Subject"     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);   }    // "RealSubject"     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;}   }    // "Proxy Object"     class MathProxy : IMath   {     Math math;      public MathProxy()     {       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);      }   } } |
| Output  4 + 2 = 6 4 - 2 = 2 4 \* 2 = 8 4 / 2 = 2 |

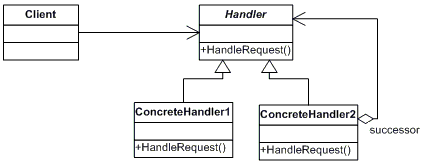
|  |
| --- |
| **Behavioral Patterns** |

**Chain of Responsibility**

### definition

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

### UML class diagram



### participants

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

* **Handler**   **(Approver)**
  + defines an interface for handling the requests
  + (optional) implements the successor link
* **ConcreteHandler**   **(Director, VicePresident, President)**
  + handles requests it is responsible for
  + can access its successor
  + if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor
* **Client**   **(ChainApp)**
  + initiates the request to a ConcreteHandler object on the chain

### sample code in C#

This structural code demonstrates the Chain of Responsibility pattern in which several linked objects (the Chain) are offered the opportunity to respond to a request or hand it off to the object next in line.

|  |
| --- |
| // Chain of Responsibility pattern -- Structural example |
| using System;  namespace DoFactory.GangOfFour.Chain.Structural {   // MainApp test application     class MainApp   {     static void Main()     {       // Setup Chain of Responsibility        Handler h1 = new ConcreteHandler1();       Handler h2 = new ConcreteHandler2();       Handler h3 = new ConcreteHandler3();       h1.SetSuccessor(h2);       h2.SetSuccessor(h3);        // Generate and process request        int[] requests = {2, 5, 14, 22, 18, 3, 27, 20};        foreach (int request in requests)       {         h1.HandleRequest(request);       }        // Wait for user        Console.Read();     }   }    // "Handler"     abstract class Handler    {     protected Handler successor;      public void SetSuccessor(Handler successor)     {       this.successor = successor;     }      public abstract void HandleRequest(int request);   }    // "ConcreteHandler1"     class ConcreteHandler1 : Handler   {     public override void HandleRequest(int request)     {       if (request >= 0 && request < 10)       {         Console.WriteLine("{0} handled request {1}",            this.GetType().Name, request);       }       else if (successor != null)       {         successor.HandleRequest(request);       }     }   }    // "ConcreteHandler2"     class ConcreteHandler2 : Handler   {     public override void HandleRequest(int request)     {       if (request >= 10 && request < 20)       {         Console.WriteLine("{0} handled request {1}",            this.GetType().Name, request);       }       else if (successor != null)       {         successor.HandleRequest(request);       }     }   }    // "ConcreteHandler3"     class ConcreteHandler3 : Handler   {     public override void HandleRequest(int request)     {       if (request >= 20 && request < 30)       {         Console.WriteLine("{0} handled request {1}",            this.GetType().Name, request);       }       else if (successor != null)       {         successor.HandleRequest(request);       }     }   } } |
| Output  ConcreteHandler1 handled request 2 ConcreteHandler1 handled request 5 ConcreteHandler2 handled request 14 ConcreteHandler3 handled request 22 ConcreteHandler2 handled request 18 ConcreteHandler1 handled request 3 ConcreteHandler3 handled request 27 ConcreteHandler3 handled request 20 |

This real-world code demonstrates the Chain of Responsibility pattern in which several linked managers and executives can respond to a purchase request or hand it off to a superior. Each position has can have its own set of rules which orders they can approve.

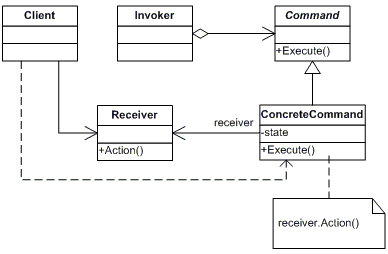
|  |
| --- |
| // Chain of Responsibility pattern -- Real World example |
| using System;  namespace DoFactory.GangOfFour.Chain.RealWorld {    // MainApp test application     class MainApp   {     static void Main()     {       // Setup Chain of Responsibility        Director Larry = new Director();       VicePresident Sam = new VicePresident();       President Tammy = new President();       Larry.SetSuccessor(Sam);       Sam.SetSuccessor(Tammy);        // Generate and process purchase requests        Purchase p = new Purchase(2034, 350.00, "Supplies");       Larry.ProcessRequest(p);        p = new Purchase(2035, 32590.10, "Project X");       Larry.ProcessRequest(p);        p = new Purchase(2036, 122100.00, "Project Y");       Larry.ProcessRequest(p);        // Wait for user        Console.Read();     }   }    // "Handler"     abstract class Approver   {     protected Approver successor;      public void SetSuccessor(Approver successor)     {       this.successor = successor;     }      public abstract void ProcessRequest(Purchase purchase);   }    // "ConcreteHandler"     class Director : Approver   {     public override void ProcessRequest(Purchase purchase)     {       if (purchase.Amount < 10000.0)       {         Console.WriteLine("{0} approved request# {1}",            this.GetType().Name, purchase.Number);       }       else if (successor != null)       {         successor.ProcessRequest(purchase);       }     }   }    // "ConcreteHandler"     class VicePresident : Approver   {     public override void ProcessRequest(Purchase purchase)     {       if (purchase.Amount < 25000.0)       {         Console.WriteLine("{0} approved request# {1}",            this.GetType().Name, purchase.Number);       }       else if (successor != null)       {         successor.ProcessRequest(purchase);       }     }   }    // "ConcreteHandler"     class President : Approver   {     public override void ProcessRequest(Purchase purchase)     {       if (purchase.Amount < 100000.0)       {         Console.WriteLine("{0} approved request# {1}",            this.GetType().Name, purchase.Number);       }       else       {         Console.WriteLine(           "Request# {0} requires an executive meeting!",            purchase.Number);       }     }   }    // Request details     class Purchase   {     private int number;     private double amount;     private string purpose;      // Constructor      public Purchase(int number, double amount, string purpose)     {       this.number = number;       this.amount = amount;       this.purpose = purpose;     }      // Properties      public double Amount     {       get{ return amount; }       set{ amount = value; }     }      public string Purpose     {       get{ return purpose; }       set{ purpose = value; }     }      public int Number     {       get{ return number; }       set{ number = value; }     }   } } |
| Output  Director Larry approved request# 2034 President Tammy approved request# 2035 Request# 2036 requires an executive meeting! |

**Command Design**

### definition

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 {      // MainApp test applicatio     class MainApp   {     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.Read();     }   }    // "Command"     abstract class Command    {     protected Receiver receiver;      // Constructor      public Command(Receiver receiver)     {       this.receiver = receiver;     }      public abstract void Execute();   }    // "ConcreteCommand"     class ConcreteCommand : Command   {     // Constructor      public ConcreteCommand(Receiver receiver) :        base(receiver)      {       }      public override void Execute()     {       receiver.Action();     }   }    // "Receiver"     class Receiver    {     public void Action()     {       Console.WriteLine("Called Receiver.Action()");     }   }    // "Invoker"     class Invoker    {     private Command command;      public void SetCommand(Command command)     {       this.command = command;     }      public void ExecuteCommand()     {       command.Execute();     }       } } |
| Output  Called Receiver.Action() |

This real-world code demonstrates the Command pattern used in a simple calculator with unlimited number of undo's and redo's. Note that in C#  the word 'operator' is a keyword. Prefixing it with '@' allows using it as an identifier.

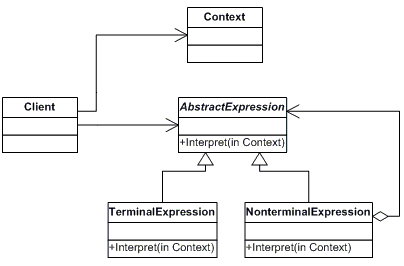
|  |
| --- |
| // Command pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Command.RealWorld {    // MainApp test application     class MainApp   {     static void Main()     {       // Create user and let her compute        User user = new User();        user.Compute('+', 100);       user.Compute('-', 50);       user.Compute('\*', 10);       user.Compute('/', 2);        // Undo 4 commands        user.Undo(4);        // Redo 3 commands        user.Redo(3);        // Wait for user        Console.Read();     }   }    // "Command"     abstract class Command   {     public abstract void Execute();     public abstract void UnExecute();   }    // "ConcreteCommand"     class CalculatorCommand : Command   {     char @operator;     int operand;     Calculator calculator;      // Constructor      public CalculatorCommand(Calculator calculator,        char @operator, int operand)     {       this.calculator = calculator;       this.@operator = @operator;       this.operand = operand;     }      public char Operator     {       set{ @operator = value; }     }      public int Operand     {       set{ operand = value; }     }      public override void Execute()     {       calculator.Operation(@operator, operand);     }      public override void UnExecute()     {       calculator.Operation(Undo(@operator), operand);     }      // Private helper function      private char Undo(char @operator)     {       char undo;       switch(@operator)       {         case '+': undo = '-'; break;         case '-': undo = '+'; break;         case '\*': undo = '/'; break;         case '/': undo = '\*'; break;         default : undo = ' '; break;       }       return undo;     }   }    // "Receiver"     class Calculator   {     private int curr = 0;      public void Operation(char @operator, int operand)     {       switch(@operator)       {         case '+': curr += operand; break;         case '-': curr -= operand; break;         case '\*': curr \*= operand; break;         case '/': curr /= operand; break;       }       Console.WriteLine(         "Current value = {0,3} (following {1} {2})",          curr, @operator, operand);     }   }    // "Invoker"     class User   {     // Initializers      private Calculator calculator = new Calculator();     private ArrayList commands = new ArrayList();      private int current = 0;      public void Redo(int levels)     {       Console.WriteLine("\n---- Redo {0} levels ", levels);       // Perform redo operations        for (int i = 0; i < levels; i++)       {         if (current < commands.Count - 1)         {           Command command = commands[current++] as Command;           command.Execute();         }       }     }      public void Undo(int levels)     {       Console.WriteLine("\n---- Undo {0} levels ", levels);       // Perform undo operations        for (int i = 0; i < levels; i++)       {         if (current > 0)         {           Command command = commands[--current] as Command;           command.UnExecute();         }       }     }      public void Compute(char @operator, int operand)     {       // Create command operation and execute it        Command command = new CalculatorCommand(         calculator, @operator, operand);       command.Execute();        // Add command to undo list        commands.Add(command);       current++;     }   } } |
| Output  Current value = 100 (following + 100) Current value =  50 (following - 50) Current value = 500 (following \* 10) Current value = 250 (following / 2)  ---- Undo 4 levels Current value = 500 (following \* 2) Current value =  50 (following / 10) Current value = 100 (following + 50) Current value =   0 (following - 100)  ---- Redo 3 levels Current value = 100 (following + 100) Current value =  50 (following - 50) Current value = 500 (following \* 10) |

**Interpreter**

### definition

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language

### UML class diagram



### participants

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

* **AbstractExpression**  **(Expression)**
  + declares an interface for executing an operation
* **TerminalExpression**  **( ThousandExpression, HundredExpression, TenExpression, OneExpression )**
  + implements an Interpret operation associated with terminal symbols in the grammar.
  + an instance is required for every terminal symbol in the sentence.
* **NonterminalExpression**  **( not used )**
  + one such class is required for every rule R ::= R1R2...Rn in the grammar
  + maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn.
  + implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.
* **Context**  **(Context)**
  + contains information that is global to the interpreter
* **Client**  **(InterpreterApp)**
  + builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes
  + invokes the Interpret operation

### sample code in C#

This structural code demonstrates the Interpreter patterns, which using a defined grammer, provides the interpreter that processes parsed statements.

|  |
| --- |
| // Interpreter pattern -- Structural example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Interpreter.Structural {      // MainApp test application     class MainApp   {     static void Main()     {       Context context = new Context();        // Usually a tree        ArrayList list = new ArrayList();         // Populate 'abstract syntax tree'        list.Add(new TerminalExpression());       list.Add(new NonterminalExpression());       list.Add(new TerminalExpression());       list.Add(new TerminalExpression());        // Interpret        foreach (AbstractExpression exp in list)       {         exp.Interpret(context);       }        // Wait for user        Console.Read();     }   }    // "Context"     class Context    {   }    // "AbstractExpression"     abstract class AbstractExpression    {     public abstract void Interpret(Context context);   }    // "TerminalExpression"     class TerminalExpression : AbstractExpression   {     public override void Interpret(Context context)       {       Console.WriteLine("Called Terminal.Interpret()");     }   }    // "NonterminalExpression"     class NonterminalExpression : AbstractExpression   {     public override void Interpret(Context context)       {       Console.WriteLine("Called Nonterminal.Interpret()");     }     } } |
| Output  Called Terminal.Interpret() Called Nonterminal.Interpret() Called Terminal.Interpret() Called Terminal.Interpret() |

This real-world code demonstrates the Interpreter pattern which is used to convert a Roman numeral to a decimal.

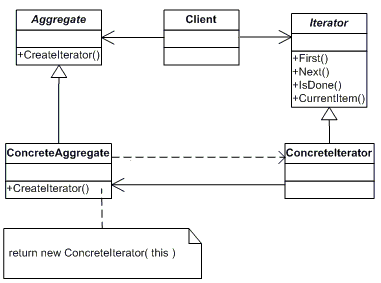
|  |
| --- |
| // Interpreter pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Interpreter.RealWorld {    // MainApp test application     class MainApp   {     static void Main()     {       string roman = "MCMXXVIII";       Context context = new Context(roman);        // Build the 'parse tree'        ArrayList tree = new ArrayList();       tree.Add(new ThousandExpression());       tree.Add(new HundredExpression());       tree.Add(new TenExpression());       tree.Add(new OneExpression());        // Interpret        foreach (Expression exp in tree)       {         exp.Interpret(context);       }        Console.WriteLine("{0} = {1}",          roman, context.Output);        // Wait for user        Console.Read();     }   }    // "Context"     class Context   {     private string input;     private int output;      // Constructor      public Context(string input)     {       this.input = input;     }      // Properties      public string Input     {       get{ return input; }       set{ input = value; }     }      public int Output     {       get{ return output; }       set{ output = value; }     }   }    // "AbstractExpression"     abstract class Expression   {     public void Interpret(Context context)     {       if (context.Input.Length == 0)          return;        if (context.Input.StartsWith(Nine()))       {         context.Output += (9 \* Multiplier());         context.Input = context.Input.Substring(2);       }       else if (context.Input.StartsWith(Four()))       {         context.Output += (4 \* Multiplier());         context.Input = context.Input.Substring(2);       }       else if (context.Input.StartsWith(Five()))       {         context.Output += (5 \* Multiplier());         context.Input = context.Input.Substring(1);       }        while (context.Input.StartsWith(One()))       {         context.Output += (1 \* Multiplier());         context.Input = context.Input.Substring(1);       }     }      public abstract string One();     public abstract string Four();     public abstract string Five();     public abstract string Nine();     public abstract int Multiplier();   }    // Thousand checks for the Roman Numeral M    // "TerminalExpression"     class ThousandExpression : Expression   {     public override string One() { return "M"; }     public override string Four(){ return " "; }     public override string Five(){ return " "; }     public override string Nine(){ return " "; }     public override int Multiplier() { return 1000; }   }    // Hundred checks C, CD, D or CM    // "TerminalExpression"     class HundredExpression : Expression   {     public override string One() { return "C"; }     public override string Four(){ return "CD"; }     public override string Five(){ return "D"; }     public override string Nine(){ return "CM"; }     public override int Multiplier() { return 100; }   }    // Ten checks for X, XL, L and XC    // "TerminalExpression"     class TenExpression : Expression   {     public override string One() { return "X"; }     public override string Four(){ return "XL"; }     public override string Five(){ return "L"; }     public override string Nine(){ return "XC"; }     public override int Multiplier() { return 10; }   }    // One checks for I, II, III, IV, V, VI, VI, VII, VIII, IX    // "TerminalExpression"     class OneExpression : Expression   {     public override string One() { return "I"; }     public override string Four(){ return "IV"; }     public override string Five(){ return "V"; }     public override string Nine(){ return "IX"; }     public override int Multiplier() { return 1; }   } } |
| Output  MCMXXVIII = 1928 |

**Iterator**

### definition

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

### UML class diagram



### participants

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

* **Iterator**  **(AbstractIterator)**
  + defines an interface for accessing and traversing elements.
* **ConcreteIterator**  **(Iterator)**
  + implements the Iterator interface.
  + keeps track of the current position in the traversal of the aggregate.
* **Aggregate**  **(AbstractCollection)**
  + defines an interface for creating an Iterator object
* **ConcreteAggregate**  **(Collection)**
  + implements the Iterator creation interface to return an instance of the proper ConcreteIterator

### sample code in C#

This structural code demonstrates the Iterator pattern which provides for a way to traverse (iterate) over a collection of items without detailing the underlying structure of the collection.

|  |
| --- |
| // Iterator pattern -- Structural example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Iterator.Structural {    // MainApp test application     class MainApp   {     static void Main()     {       ConcreteAggregate a = new ConcreteAggregate();       a[0] = "Item A";       a[1] = "Item B";       a[2] = "Item C";       a[3] = "Item D";        // Create Iterator and provide aggregate        ConcreteIterator i = new ConcreteIterator(a);        Console.WriteLine("Iterating over collection:");              object item = i.First();       while (item != null)       {         Console.WriteLine(item);         item = i.Next();       }         // Wait for user        Console.Read();     }   }    // "Aggregate"     abstract class Aggregate   {     public abstract Iterator CreateIterator();   }    // "ConcreteAggregate"     class ConcreteAggregate : Aggregate   {     private ArrayList items = new ArrayList();      public override Iterator CreateIterator()     {       return new ConcreteIterator(this);     }      // Property      public int Count     {       get{ return items.Count; }     }      // Indexer      public object this[int index]     {       get{ return items[index]; }       set{ items.Insert(index, value); }     }   }    // "Iterator"     abstract class Iterator   {     public abstract object First();     public abstract object Next();     public abstract bool IsDone();     public abstract object CurrentItem();   }    // "ConcreteIterator"     class ConcreteIterator : Iterator   {     private ConcreteAggregate aggregate;     private int current = 0;      // Constructor      public ConcreteIterator(ConcreteAggregate aggregate)     {       this.aggregate = aggregate;     }      public override object First()     {       return aggregate[0];     }      public override object Next()     {       object ret = null;       if (current < aggregate.Count - 1)       {         ret = aggregate[++current];       }              return ret;     }      public override object CurrentItem()     {       return aggregate[current];     }      public override bool IsDone()     {       return current >= aggregate.Count ? true : false ;     }   } } |
| Output  Iterating over collection: Item A Item B Item C Item D |

This real-world code demonstrates the Iterator pattern which is used to iterate over a collection of items and skip a specific number of items each iteration.

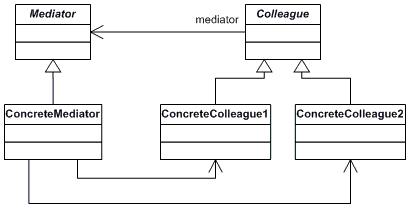
|  |
| --- |
| // Iterator pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Iterator.RealWorld {    // MainApp test application     class MainApp   {     static void Main()     {       // Build a collection        Collection collection = new Collection();       collection[0] = new Item("Item 0");       collection[1] = new Item("Item 1");       collection[2] = new Item("Item 2");       collection[3] = new Item("Item 3");       collection[4] = new Item("Item 4");       collection[5] = new Item("Item 5");       collection[6] = new Item("Item 6");       collection[7] = new Item("Item 7");       collection[8] = new Item("Item 8");        // Create iterator        Iterator iterator = new Iterator(collection);        // Skip every other item        iterator.Step = 2;        Console.WriteLine("Iterating over collection:");        for(Item item = iterator.First();          !iterator.IsDone; item = iterator.Next())       {         Console.WriteLine(item.Name);       }        // Wait for user        Console.Read();     }   }    class Item   {     string name;      // Constructor      public Item(string name)     {       this.name = name;     }      // Property      public string Name     {       get{ return name; }     }   }    // "Aggregate"     interface IAbstractCollection   {     Iterator CreateIterator();   }    // "ConcreteAggregate"     class Collection : IAbstractCollection   {     private ArrayList items = new ArrayList();      public Iterator CreateIterator()     {       return new Iterator(this);     }      // Property      public int Count     {       get{ return items.Count; }     }          // Indexer      public object this[int index]     {       get{ return items[index]; }       set{ items.Add(value); }     }   }    // "Iterator"     interface IAbstractIterator   {     Item First();     Item Next();     bool IsDone{ get; }     Item CurrentItem{ get; }   }    // "ConcreteIterator"     class Iterator : IAbstractIterator   {     private Collection collection;     private int current = 0;     private int step = 1;      // Constructor      public Iterator(Collection collection)     {       this.collection = collection;     }      public Item First()     {       current = 0;       return collection[current] as Item;     }      public Item Next()     {       current += step;       if (!IsDone)         return collection[current] as Item;       else         return null;     }      // Properties      public int Step     {       get{ return step; }       set{ step = value; }     }      public Item CurrentItem     {       get       {         return collection[current] as Item;       }     }      public bool IsDone     {       get       {         return current >= collection.Count ? true : false;       }     }   } } |
| Output  Iterating over collection: Item 0 Item 2 Item 4 Item 6 Item 8 |

**Mediator**

### definition

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

### UML class diagram



### participants

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

* **Mediator**  **(IChatroom)**
  + defines an interface for communicating with Colleague objects
* **ConcreteMediator**  **(Chatroom)**
  + implements cooperative behavior by coordinating Colleague objects
  + knows and maintains its colleagues
* **Colleague classes**  **(Participant)**
  + each Colleague class knows its Mediator object
  + each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

### sample code in C#

This structural code demonstrates the Mediator pattern facilitating loosely coupled communication between different objects and object types. The mediator is a central hub through which all interaction must take place.

|  |
| --- |
| // Mediator pattern -- Structural example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Mediator.Structural {    // Mainapp test application     class MainApp   {     static void Main()     {       ConcreteMediator m = new ConcreteMediator();        ConcreteColleague1 c1 = new ConcreteColleague1(m);       ConcreteColleague2 c2 = new ConcreteColleague2(m);        m.Colleague1 = c1;       m.Colleague2 = c2;        c1.Send("How are you?");       c2.Send("Fine, thanks");        // Wait for user        Console.Read();     }   }    // "Mediator"     abstract class Mediator   {     public abstract void Send(string message,        Colleague colleague);   }    // "ConcreteMediator"     class ConcreteMediator : Mediator   {     private ConcreteColleague1 colleague1;     private ConcreteColleague2 colleague2;      public ConcreteColleague1 Colleague1     {       set{ colleague1 = value; }     }      public ConcreteColleague2 Colleague2     {       set{ colleague2 = value; }     }      public override void Send(string message,        Colleague colleague)     {       if (colleague == colleague1)       {         colleague2.Notify(message);       }       else       {         colleague1.Notify(message);       }     }   }    // "Colleague"     abstract class Colleague   {     protected Mediator mediator;      // Constructor      public Colleague(Mediator mediator)     {       this.mediator = mediator;     }   }    // "ConcreteColleague1"     class ConcreteColleague1 : Colleague   {     // Constructor      public ConcreteColleague1(Mediator mediator)        : base(mediator)      {      }      public void Send(string message)     {       mediator.Send(message, this);     }      public void Notify(string message)     {       Console.WriteLine("Colleague1 gets message: "          + message);     }   }    // "ConcreteColleague2"     class ConcreteColleague2 : Colleague   {     // Constructor      public ConcreteColleague2(Mediator mediator)        : base(mediator)      {      }      public void Send(string message)     {       mediator.Send(message, this);     }      public void Notify(string message)     {       Console.WriteLine("Colleague2 gets message: "          + message);     }   } } |
| Output  Colleague2 gets message: How are you? Colleague1 gets message: Fine, thanks |

This real-world code demonstrates the Mediator pattern facilitating loosely coupled communication between different Participants registering with a Chatroom. The Chatroom is the central hub through which all communication takes place. At this point only one-to-one communication is implemented in the Chatroom, but would be trivial to change to one-to-many.

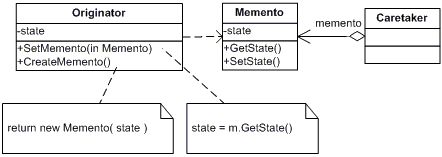
|  |
| --- |
| // Mediator pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Mediator.RealWorld {      // MainApp test application     class MainApp   {     static void Main()     {       // Create chatroom        Chatroom chatroom = new Chatroom();        // Create participants and register them        Participant George = new Beatle("George");       Participant Paul = new Beatle("Paul");       Participant Ringo = new Beatle("Ringo");       Participant John = new Beatle("John") ;       Participant Yoko = new NonBeatle("Yoko");        chatroom.Register(George);       chatroom.Register(Paul);       chatroom.Register(Ringo);       chatroom.Register(John);       chatroom.Register(Yoko);        // Chatting participants        Yoko.Send ("John", "Hi John!");       Paul.Send ("Ringo", "All you need is love");       Ringo.Send("George", "My sweet Lord");       Paul.Send ("John", "Can't buy me love");       John.Send ("Yoko", "My sweet love") ;        // Wait for user        Console.Read();     }   }    // "Mediator"     abstract class AbstractChatroom   {     public abstract void Register(Participant participant);     public abstract void Send(       string from, string to, string message);   }    // "ConcreteMediator"     class Chatroom : AbstractChatroom   {     private Hashtable participants = new Hashtable();      public override void Register(Participant participant)     {       if (participants[participant.Name] == null)       {         participants[participant.Name] = participant;       }        participant.Chatroom = this;     }      public override void Send(       string from, string to, string message)     {       Participant pto = (Participant)participants[to];       if (pto != null)       {         pto.Receive(from, message);       }     }   }    // "AbstractColleague"     class Participant   {     private Chatroom chatroom;     private string name;      // Constructor      public Participant(string name)     {       this.name = name;     }      // Properties      public string Name     {       get{ return name; }     }      public Chatroom Chatroom     {       set{ chatroom = value; }       get{ return chatroom; }     }      public void Send(string to, string message)     {       chatroom.Send(name, to, message);     }      public virtual void Receive(       string from, string message)     {       Console.WriteLine("{0} to {1}: '{2}'",         from, Name, message);     }   }    //" ConcreteColleague1"     class Beatle : Participant   {     // Constructor      public Beatle(string name) : base(name)      {      }      public override void Receive(string from, string message)     {       Console.Write("To a Beatle: ");       base.Receive(from, message);     }   }    //" ConcreteColleague2"     class NonBeatle : Participant   {     // Constructor      public NonBeatle(string name) : base(name)      {      }      public override void Receive(string from, string message)     {       Console.Write("To a non-Beatle: ");       base.Receive(from, message);     }   } } |
| Output  To a Beatle: Yoko to John: 'Hi John!' To a Beatle: Paul to Ringo: 'All you need is love' To a Beatle: Ringo to George: 'My sweet Lord' To a Beatle: Paul to John: 'Can't buy me love' To a non-Beatle: John to Yoko: 'My sweet love' |

**Memento**

### definition

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

### UML class diagram



### participants

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

* **Memento**  **(Memento)**
  + stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  + protect against access by objects of other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento -- it can only pass the memento to the other objects. Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produces the memento would be permitted to access the memento's internal state.
* **Originator**  **(SalesProspect)**
  + creates a memento containing a snapshot of its current internal state.
  + uses the memento to restore its internal state
* **Caretaker**  **(Caretaker)**
  + is responsible for the memento's safekeeping
  + never operates on or examines the contents of a memento.

### sample code in C#

This structural code demonstrates the Memento pattern which temporary saves and restores another object's internal state.

|  |
| --- |
| // Memento pattern -- Structural example |
| using System;  namespace DoFactory.GangOfFour.Memento.Structural {    // MainApp test application     class MainApp   {     static void Main()     {       Originator o = new Originator();       o.State = "On";        // Store internal state        Caretaker c = new Caretaker();       c.Memento = o.CreateMemento();        // Continue changing originator        o.State = "Off";        // Restore saved state        o.SetMemento(c.Memento);        // Wait for user        Console.Read();     }   }    // "Originator"     class Originator   {     private string state;      // Property      public string State     {       get{ return state; }       set       {          state = value;          Console.WriteLine("State = " + state);       }     }      public Memento CreateMemento()     {       return (new Memento(state));     }      public void SetMemento(Memento memento)     {       Console.WriteLine("Restoring state:");       State = memento.State;     }   }    // "Memento"     class Memento   {     private string state;      // Constructor      public Memento(string state)     {       this.state = state;     }      // Property      public string State     {       get{ return state; }     }   }    // "Caretaker"     class Caretaker   {     private Memento memento;      // Property      public Memento Memento     {       set{ memento = value; }       get{ return memento; }     }   } } |
| Output  State = On State = Off Restoring state: State = On |

This real-world code demonstrates the Memento pattern which temporarily saves and then restores the SalesProspect's internal state.

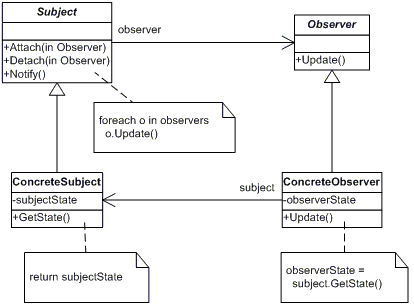
|  |
| --- |
| // Memento pattern -- Real World example |
| using System;  namespace DoFactory.GangOfFour.Memento.RealWorld {    // MainApp test application     class MainApp   {     static void Main()     {       SalesProspect s = new SalesProspect();       s.Name = "Noel van Halen";       s.Phone = "(412) 256-0990";       s.Budget = 25000.0;        // Store internal state        ProspectMemory m = new ProspectMemory();       m.Memento = s.SaveMemento();        // Continue changing originator        s.Name = "Leo Welch";       s.Phone = "(310) 209-7111";       s.Budget = 1000000.0;        // Restore saved state        s.RestoreMemento(m.Memento);        // Wait for user        Console.Read();     }   }    // "Originator"     class SalesProspect   {     private string name;     private string phone;     private double budget;      // Properties      public string Name     {       get{ return name; }       set       {          name = value;          Console.WriteLine("Name: " + name);       }     }      public string Phone     {       get{ return phone; }       set       {          phone = value;          Console.WriteLine("Phone: " + phone);       }     }      public double Budget     {       get{ return budget; }       set       {          budget = value;          Console.WriteLine("Budget: " + budget);       }     }      public Memento SaveMemento()     {       Console.WriteLine("\nSaving state --\n");       return new Memento(name, phone, budget);     }      public void RestoreMemento(Memento memento)     {       Console.WriteLine("\nRestoring state --\n");       this.Name = memento.Name;       this.Phone = memento.Phone;       this.Budget = memento.Budget;     }   }    // "Memento"     class Memento   {     private string name;     private string phone;     private double budget;      // Constructor      public Memento(string name, string phone, double budget)     {       this.name = name;       this.phone = phone;       this.budget = budget;     }      // Properties      public string Name     {       get{ return name; }       set{ name = value; }     }      public string Phone     {       get{ return phone; }       set{ phone = value; }     }      public double Budget     {       get{ return budget; }       set{ budget = value; }     }   }    // "Caretaker"     class ProspectMemory   {     private Memento memento;      // Property      public Memento Memento     {       set{ memento = value; }       get{ return memento; }     }   } } |
| Output  Name:   Noel van Halen Phone:  (412) 256-0990 Budget: 25000  Saving state --  Name:   Leo Welch Phone:  (310) 209-7111 Budget: 1000000  Restoring state --  Name:   Noel van Halen Phone:  (412) 256-0990 Budget: 25000 |

**Observer**

### definition

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

### UML class diagram



### participants

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

* **Subject**  **(Stock)**
  + knows its observers. Any number of Observer objects may observe a subject
  + provides an interface for attaching and detaching Observer objects.
* **ConcreteSubject**  **(IBM)**
  + stores state of interest to ConcreteObserver
  + sends a notification to its observers when its state changes
* **Observer**  **(IInvestor)**
  + defines an updating interface for objects that should be notified of changes in a subject.
* **ConcreteObserver**  **(Investor)**
  + maintains a reference to a ConcreteSubject object
  + stores state that should stay consistent with the subject's
  + implements the Observer updating interface to keep its state consistent with the subject's

### sample code in C#

This structural code demonstrates the Observer pattern in which registered objects are notified of and updated with a state change.

|  |
| --- |
| // Observer pattern -- Structural example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Observer.Structural {    // MainApp test application     class MainApp   {     static void Main()     {       // Configure Observer pattern        ConcreteSubject s = new ConcreteSubject();        s.Attach(new ConcreteObserver(s,"X"));       s.Attach(new ConcreteObserver(s,"Y"));       s.Attach(new ConcreteObserver(s,"Z"));        // Change subject and notify observers        s.SubjectState = "ABC";       s.Notify();        // Wait for user        Console.Read();     }   }    // "Subject"     abstract class Subject   {     private ArrayList observers = new ArrayList();      public void Attach(Observer observer)     {       observers.Add(observer);     }      public void Detach(Observer observer)     {       observers.Remove(observer);     }      public void Notify()     {       foreach (Observer o in observers)       {         o.Update();       }     }   }    // "ConcreteSubject"     class ConcreteSubject : Subject   {     private string subjectState;      // Property      public string SubjectState     {       get{ return subjectState; }       set{ subjectState = value; }     }   }    // "Observer"     abstract class Observer   {     public abstract void Update();   }    // "ConcreteObserver"     class ConcreteObserver : Observer   {     private string name;     private string observerState;     private ConcreteSubject subject;      // Constructor      public ConcreteObserver(       ConcreteSubject subject, string name)     {       this.subject = subject;       this.name = name;     }      public override void Update()     {       observerState = subject.SubjectState;       Console.WriteLine("Observer {0}'s new state is {1}",         name, observerState);     }      // Property      public ConcreteSubject Subject     {       get { return subject; }       set { subject = value; }     }   } } |
| Output  Observer X's new state is ABC Observer Y's new state is ABC Observer Z's new state is ABC |

This real-world code demonstrates the Observer pattern in which registered investors are notified every time a stock changes value.

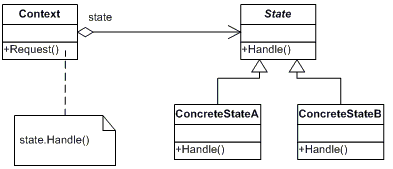
|  |
| --- |
| // Observer pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Observer.RealWorld {      // MainApp test application     class MainApp   {     static void Main()     {       // Create investors        Investor s = new Investor("Sorros");       Investor b = new Investor("Berkshire");        // Create IBM stock and attach investors        IBM ibm = new IBM("IBM", 120.00);       ibm.Attach(s);       ibm.Attach(b);        // Change price, which notifies investors        ibm.Price = 120.10;       ibm.Price = 121.00;       ibm.Price = 120.50;       ibm.Price = 120.75;        // Wait for user        Console.Read();     }   }    // "Subject"     abstract class Stock   {     protected string symbol;     protected double price;     private ArrayList investors = new ArrayList();      // Constructor      public Stock(string symbol, double price)     {       this.symbol = symbol;       this.price = price;     }      public void Attach(Investor investor)     {       investors.Add(investor);     }      public void Detach(Investor investor)     {       investors.Remove(investor);     }      public void Notify()     {       foreach (Investor investor in investors)       {         investor.Update(this);       }       Console.WriteLine("");     }      // Properties      public double Price     {       get{ return price; }       set       {         price = value;         Notify();        }     }      public string Symbol     {       get{ return symbol; }       set{ symbol = value; }     }   }    // "ConcreteSubject"     class IBM : Stock   {     // Constructor      public IBM(string symbol, double price)       : base(symbol, price)     {     }   }    // "Observer"     interface IInvestor   {     void Update(Stock stock);   }    // "ConcreteObserver"     class Investor : IInvestor   {     private string name;     private Stock stock;      // Constructor      public Investor(string name)     {       this.name = name;     }      public void Update(Stock stock)     {       Console.WriteLine("Notified {0} of {1}'s " +         "change to {2:C}", name, stock.Symbol, stock.Price);     }      // Property      public Stock Stock     {       get{ return stock; }       set{ stock = value; }     }   } } |
| Output  Notified Sorros of IBM's change to $120.10 Notified Berkshire of IBM's change to $120.10  Notified Sorros of IBM's change to $121.00 Notified Berkshire of IBM's change to $121.00  Notified Sorros of IBM's change to $120.50 Notified Berkshire of IBM's change to $120.50  Notified Sorros of IBM's change to $120.75 Notified Berkshire of IBM's change to $120.75 |

**State**

### definition

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

### UML class diagram



### participants

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

* **Context**  **(Account)**
  + defines the interface of interest to clients
  + maintains an instance of a ConcreteState subclass that defines the current state.
* **State**  **(State)**
  + defines an interface for encapsulating the behavior associated with a particular state of the Context.
* **Concrete State**  **(RedState, SilverState, GoldState)**
  + each subclass implements a behavior associated with a state of Context

### sample code in C#

This structural code demonstrates the State pattern which allows an object to behave differently depending on its internal state. The difference in behavior is delegated to objects that represent this state.

|  |
| --- |
| // State pattern -- Structural example |
| using System;  namespace DoFactory.GangOfFour.State.Structural {      // MainApp test application     class MainApp   {     static void Main()     {       // Setup context in a state        Context c = new Context(new ConcreteStateA());        // Issue requests, which toggles state        c.Request();       c.Request();       c.Request();       c.Request();        // Wait for user        Console.Read();     }   }    // "State"     abstract class State   {     public abstract void Handle(Context context);   }    // "ConcreteStateA"     class ConcreteStateA : State   {     public override void Handle(Context context)     {       context.State = new ConcreteStateB();     }   }    // "ConcreteStateB"     class ConcreteStateB : State   {     public override void Handle(Context context)     {       context.State = new ConcreteStateA();     }   }    // "Context"     class Context   {     private State state;      // Constructor      public Context(State state)     {       this.State = state;     }      // Property      public State State     {       get{ return state; }       set       {          state = value;          Console.WriteLine("State: " +            state.GetType().Name);       }     }      public void Request()     {       state.Handle(this);     }   } } |
| Output  State: ConcreteStateA State: ConcreteStateB State: ConcreteStateA State: ConcreteStateB State: ConcreteStateA |

This real-world code demonstrates the State pattern which allows an Account to behave differently depending on its balance. The difference in behavior is delegated to State objects called RedState, SilverState and GoldState. These states represent overdrawn accounts, starter accounts, and accounts in good standing.

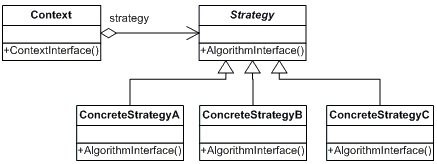
|  |
| --- |
| // State pattern -- Real World example |
| using System;  namespace DoFactory.GangOfFour.State.RealWorld {      // MainApp test application     class MainApp   {     static void Main()     {       // Open a new account        Account account = new Account("Jim Johnson");        // Apply financial transactions        account.Deposit(500.0);       account.Deposit(300.0);       account.Deposit(550.0);       account.PayInterest();       account.Withdraw(2000.00);       account.Withdraw(1100.00);        // Wait for user        Console.Read();     }   }    // "State"     abstract class State   {     protected Account account;     protected double balance;      protected double interest;     protected double lowerLimit;     protected double upperLimit;      // Properties      public Account Account     {       get{ return account; }       set{ account = value; }     }      public double Balance     {       get{ return balance; }       set{ balance = value; }     }      public abstract void Deposit(double amount);     public abstract void Withdraw(double amount);     public abstract void PayInterest();   }    // "ConcreteState"     // Account is overdrawn     class RedState : State   {     double serviceFee;      // Constructor      public RedState(State state)     {       this.balance = state.Balance;       this.account = state.Account;       Initialize();     }      private void Initialize()     {       // Should come from a datasource        interest = 0.0;       lowerLimit = -100.0;       upperLimit = 0.0;       serviceFee = 15.00;     }      public override void Deposit(double amount)     {       balance += amount;       StateChangeCheck();     }      public override void Withdraw(double amount)     {       amount = amount - serviceFee;       Console.WriteLine("No funds available for withdrawal!");     }      public override void PayInterest()     {       // No interest is paid      }      private void StateChangeCheck()     {       if (balance > upperLimit)       {         account.State = new SilverState(this);       }     }   }    // "ConcreteState"     // Silver is non-interest bearing state     class SilverState : State   {     // Overloaded constructors       public SilverState(State state) :        this( state.Balance, state.Account)     {       }      public SilverState(double balance, Account account)     {       this.balance = balance;       this.account = account;       Initialize();     }      private void Initialize()     {       // Should come from a datasource        interest = 0.0;       lowerLimit = 0.0;       upperLimit = 1000.0;     }      public override void Deposit(double amount)     {       balance += amount;       StateChangeCheck();     }      public override void Withdraw(double amount)     {       balance -= amount;       StateChangeCheck();     }      public override void PayInterest()     {       balance += interest \* balance;       StateChangeCheck();     }      private void StateChangeCheck()     {       if (balance < lowerLimit)       {         account.State = new RedState(this);       }       else if (balance > upperLimit)       {         account.State = new GoldState(this);       }     }   }    // "ConcreteState"     // Interest bearing state     class GoldState : State   {     // Overloaded constructors      public GoldState(State state)        : this(state.Balance,state.Account)     {       }      public GoldState(double balance, Account account)     {       this.balance = balance;       this.account = account;       Initialize();     }      private void Initialize()     {       // Should come from a database        interest = 0.05;       lowerLimit = 1000.0;       upperLimit = 10000000.0;     }      public override void Deposit(double amount)     {       balance += amount;       StateChangeCheck();     }      public override void Withdraw(double amount)     {       balance -= amount;       StateChangeCheck();     }      public override void PayInterest()     {       balance += interest \* balance;       StateChangeCheck();     }      private void StateChangeCheck()     {       if (balance < 0.0)       {         account.State = new RedState(this);       }       else if (balance < lowerLimit)       {         account.State = new SilverState(this);       }     }   }    // "Context"     class Account   {     private State state;     private string owner;      // Constructor      public Account(string owner)     {       // New accounts are 'Silver' by default        this.owner = owner;       state = new SilverState(0.0, this);     }      // Properties      public double Balance     {       get{ return state.Balance; }     }      public State State     {       get{ return state; }       set{ state = value; }     }      public void Deposit(double amount)     {       state.Deposit(amount);       Console.WriteLine("Deposited {0:C} --- ", amount);       Console.WriteLine(" Balance = {0:C}", this.Balance);       Console.WriteLine(" Status = {0}\n" ,          this.State.GetType().Name);       Console.WriteLine("");     }      public void Withdraw(double amount)     {       state.Withdraw(amount);       Console.WriteLine("Withdrew {0:C} --- ", amount);       Console.WriteLine(" Balance = {0:C}", this.Balance);       Console.WriteLine(" Status = {0}\n" ,          this.State.GetType().Name);     }      public void PayInterest()     {       state.PayInterest();       Console.WriteLine("Interest Paid --- ");       Console.WriteLine(" Balance = {0:C}", this.Balance);       Console.WriteLine(" Status = {0}\n" ,          this.State.GetType().Name);     }   } } |
| Output  Deposited $500.00 ---  Balance = $500.00  Status = SilverState   Deposited $300.00 ---  Balance = $800.00  Status = SilverState   Deposited $550.00 ---  Balance = $1,350.00  Status = GoldState   Interest Paid ---  Balance = $1,417.50  Status = GoldState  Withdrew $2,000.00 ---  Balance = ($582.50)  Status = RedState  No funds available for withdrawal! Withdrew $1,100.00 ---  Balance = ($582.50)  Status = RedState |

**Strategy**

### definition

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

### UML class diagram



### participants

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

* **Strategy**  **(SortStrategy)**
  + declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
* **ConcreteStrategy**  **(QuickSort, ShellSort, MergeSort)**
  + implements the algorithm using the Strategy interface
* **Context**  **(SortedList)**
  + is configured with a ConcreteStrategy object
  + maintains a reference to a Strategy object
  + may define an interface that lets Strategy access its data.

### sample code in C#

This structural code demonstrates the Strategy pattern which encapsulates functionality in the form of an object. This allows clients to dynamically change algorithmic strategies.

|  |
| --- |
| // Strategy pattern -- Structural example |
| using System;  namespace DoFactory.GangOfFour.Strategy.Structural {      // MainApp test application     class MainApp   {     static void Main()     {       Context context;        // Three contexts following different strategies        context = new Context(new ConcreteStrategyA());       context.ContextInterface();        context = new Context(new ConcreteStrategyB());       context.ContextInterface();        context = new Context(new ConcreteStrategyC());       context.ContextInterface();        // Wait for user        Console.Read();     }   }    // "Strategy"     abstract class Strategy   {     public abstract void AlgorithmInterface();   }    // "ConcreteStrategyA"     class ConcreteStrategyA : Strategy   {     public override void AlgorithmInterface()     {       Console.WriteLine(         "Called ConcreteStrategyA.AlgorithmInterface()");     }   }    // "ConcreteStrategyB"     class ConcreteStrategyB : Strategy   {     public override void AlgorithmInterface()     {       Console.WriteLine(         "Called ConcreteStrategyB.AlgorithmInterface()");     }   }    // "ConcreteStrategyC"     class ConcreteStrategyC : Strategy   {     public override void AlgorithmInterface()     {       Console.WriteLine(         "Called ConcreteStrategyC.AlgorithmInterface()");     }   }    // "Context"     class Context   {     Strategy strategy;      // Constructor      public Context(Strategy strategy)     {       this.strategy = strategy;     }      public void ContextInterface()     {       strategy.AlgorithmInterface();     }   } } |
| Output  Called ConcreteStrategyA.AlgorithmInterface() Called ConcreteStrategyB.AlgorithmInterface() Called ConcreteStrategyC.AlgorithmInterface() |

This real-world code demonstrates the Strategy pattern which encapsulates sorting algorithms in the form of sorting objects. This allows clients to dynamically change sorting strategies including Quicksort, Shellsort, and Mergesort.

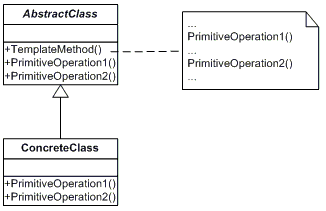
|  |
| --- |
| // Strategy pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Strategy.RealWorld {      // MainApp test application     class MainApp   {     static void Main()     {       // Two contexts following different strategies        SortedList studentRecords = new SortedList();        studentRecords.Add("Samual");       studentRecords.Add("Jimmy");       studentRecords.Add("Sandra");       studentRecords.Add("Vivek");       studentRecords.Add("Anna");        studentRecords.SetSortStrategy(new QuickSort());       studentRecords.Sort();        studentRecords.SetSortStrategy(new ShellSort());       studentRecords.Sort();        studentRecords.SetSortStrategy(new MergeSort());       studentRecords.Sort();        // Wait for user        Console.Read();     }   }    // "Strategy"     abstract class SortStrategy   {     public abstract void Sort(ArrayList list);   }    // "ConcreteStrategy"     class QuickSort : SortStrategy   {     public override void Sort(ArrayList list)     {       list.Sort(); // Default is Quicksort        Console.WriteLine("QuickSorted list ");     }   }    // "ConcreteStrategy"     class ShellSort : SortStrategy   {     public override void Sort(ArrayList list)     {        //list.ShellSort(); not-implemented        Console.WriteLine("ShellSorted list ");     }   }    // "ConcreteStrategy"     class MergeSort : SortStrategy   {     public override void Sort(ArrayList list)     {       //list.MergeSort(); not-implemented        Console.WriteLine("MergeSorted list ");     }   }    // "Context"     class SortedList   {     private ArrayList list = new ArrayList();     private SortStrategy sortstrategy;      public void SetSortStrategy(SortStrategy sortstrategy)     {       this.sortstrategy = sortstrategy;     }      public void Add(string name)     {       list.Add(name);     }      public void Sort()     {       sortstrategy.Sort(list);        // Display results        foreach (string name in list)       {         Console.WriteLine(" " + name);       }       Console.WriteLine();     }   } } |
| Output  QuickSorted list  Anna  Jimmy  Samual  Sandra  Vivek  ShellSorted list  Anna  Jimmy  Samual  Sandra  Vivek  MergeSorted list  Anna  Jimmy  Samual  Sandra  Vivek |

**Template**

### definition

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

### UML class diagram



### participants

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

* **AbstractClass**  **(DataObject)**
  + defines abstract *primitive operations* that concrete subclasses define to implement steps of an algorithm
  + implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
* **ConcreteClass**  **(CustomerDataObject)**
  + implements the primitive operations ot carry out subclass-specific steps of the algorithm

### sample code in C#

This structural code demonstrates the Template method which provides a skeleton calling sequence of methods. One or more steps can be deferred to subclasses which implement these steps without changing the overall calling sequence.

|  |
| --- |
| // Template pattern -- Structural example |
| using System;  namespace DoFactory.GangOfFour.Template.Structural {      // MainApp test application     class MainApp   {     static void Main()     {       AbstractClass c;              c = new ConcreteClassA();       c.TemplateMethod();        c = new ConcreteClassB();       c.TemplateMethod();        // Wait for user        Console.Read();     }   }    // "AbstractClass"     abstract class AbstractClass   {     public abstract void PrimitiveOperation1();     public abstract void PrimitiveOperation2();      // The "Template method"      public void TemplateMethod()     {       PrimitiveOperation1();       PrimitiveOperation2();       Console.WriteLine("");     }   }    // "ConcreteClass"     class ConcreteClassA : AbstractClass   {     public override void PrimitiveOperation1()     {       Console.WriteLine("ConcreteClassA.PrimitiveOperation1()");     }     public override void PrimitiveOperation2()     {       Console.WriteLine("ConcreteClassA.PrimitiveOperation2()");     }   }    class ConcreteClassB : AbstractClass   {     public override void PrimitiveOperation1()     {       Console.WriteLine("ConcreteClassB.PrimitiveOperation1()");     }     public override void PrimitiveOperation2()     {       Console.WriteLine("ConcreteClassB.PrimitiveOperation2()");     }   } } |
| Output  ConcreteClassA.PrimitiveOperation1() ConcreteClassA.PrimitiveOperation2() |

This real-world code demonstrates a Template method named Run() which provides a skeleton calling sequence of methods. Implementation of these steps are deferred to the CustomerDataObject subclass which implements the Connect, Select, Process, and Disconnect methods.

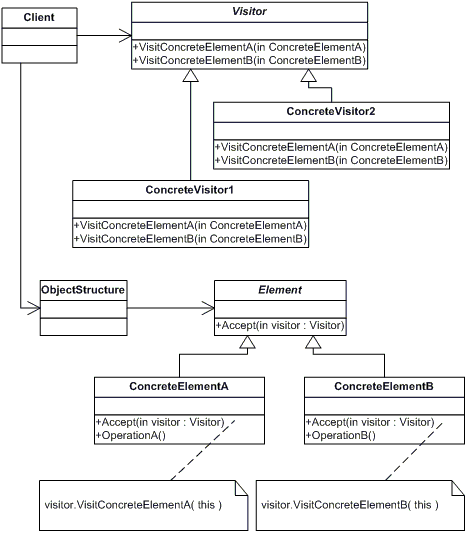
|  |
| --- |
| // Template pattern -- Real World example |
| using System; using System.Data; using System.Data.OleDb;  namespace DoFactory.GangOfFour.Template.RealWorld {      // MainApp test application     class MainApp   {     static void Main()     {       DataAccessObject dao;              dao = new Categories();       dao.Run();        dao = new Products();       dao.Run();        // Wait for user        Console.Read();     }   }    // "AbstractClass"     abstract class DataAccessObject   {     protected string connectionString;      protected DataSet dataSet;      public virtual void Connect()     {       // Make sure mdb is on c:\        connectionString =          "provider=Microsoft.JET.OLEDB.4.0; " +         "data source=c:\\nwind.mdb";     }      public abstract void Select();     public abstract void Process();      public virtual void Disconnect()     {       connectionString = "";     }      // The "Template Method"       public void Run()     {       Connect();       Select();       Process();       Disconnect();     }   }    // "ConcreteClass"     class Categories : DataAccessObject   {     public override void Select()     {       string sql = "select CategoryName from Categories";       OleDbDataAdapter dataAdapter = new OleDbDataAdapter(         sql, connectionString);        dataSet = new DataSet();       dataAdapter.Fill(dataSet, "Categories");     }      public override void Process()     {       Console.WriteLine("Categories ---- ");              DataTable dataTable = dataSet.Tables["Categories"];       foreach (DataRow row in dataTable.Rows)       {         Console.WriteLine(row["CategoryName"]);       }       Console.WriteLine();     }   }    class Products : DataAccessObject   {     public override void Select()     {       string sql = "select ProductName from Products";       OleDbDataAdapter dataAdapter = new OleDbDataAdapter(         sql, connectionString);        dataSet = new DataSet();       dataAdapter.Fill(dataSet, "Products");     }      public override void Process()     {       Console.WriteLine("Products ---- ");       DataTable dataTable = dataSet.Tables["Products"];       foreach (DataRow row in dataTable.Rows)       {         Console.WriteLine(row["ProductName"]);       }       Console.WriteLine();     }   } } |
| Output  Categories ---- Beverages Condiments Confections Dairy Products Grains/Cereals Meat/Poultry Produce Seafood  Products ---- Chai Chang Aniseed Syrup Chef Anton's Cajun Seasoning Chef Anton's Gumbo Mix Grandma's Boysenberry Spread Uncle Bob's Organic Dried Pears Northwoods Cranberry Sauce Mishi Kobe Niku |

**Visitor**

### definition

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

### UML class diagram



### participants

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

* **Visitor**  **(Visitor)**
  + declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the elements directly through its particular interface
* **ConcreteVisitor**  **(IncomeVisitor, VacationVisitor)**
  + implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class or object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
* **Element**  **(Element)**
  + defines an Accept operation that takes a visitor as an argument.
* **ConcreteElement**  **(Employee)**
  + implements an Accept operation that takes a visitor as an argument
* **ObjectStructure**  **(Employees)**
  + can enumerate its elements
  + may provide a high-level interface to allow the visitor to visit its elements
  + may either be a Composite (pattern) or a collection such as a list or a set

### sample code in C#

This structural code demonstrates the Visitor pattern in which an object traverses an object structure and performs the same operation on each node in this structure. Different visitor objects define different operations.

|  |
| --- |
| // Visitor pattern -- Structural example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Visitor.Structural {      // MainApp test application     class MainApp   {     static void Main()     {       // Setup structure        ObjectStructure o = new ObjectStructure();       o.Attach(new ConcreteElementA());       o.Attach(new ConcreteElementB());        // Create visitor objects        ConcreteVisitor1 v1 = new ConcreteVisitor1();       ConcreteVisitor2 v2 = new ConcreteVisitor2();        // Structure accepting visitors        o.Accept(v1);       o.Accept(v2);        // Wait for user        Console.Read();     }   }    // "Visitor"     abstract class Visitor   {     public abstract void VisitConcreteElementA(       ConcreteElementA concreteElementA);     public abstract void VisitConcreteElementB(       ConcreteElementB concreteElementB);   }    // "ConcreteVisitor1"     class ConcreteVisitor1 : Visitor   {     public override void VisitConcreteElementA(       ConcreteElementA concreteElementA)     {       Console.WriteLine("{0} visited by {1}",         concreteElementA.GetType().Name, this.GetType().Name);     }      public override void VisitConcreteElementB(       ConcreteElementB concreteElementB)     {       Console.WriteLine("{0} visited by {1}",         concreteElementB.GetType().Name, this.GetType().Name);     }   }    // "ConcreteVisitor2"     class ConcreteVisitor2 : Visitor   {     public override void VisitConcreteElementA(       ConcreteElementA concreteElementA)     {       Console.WriteLine("{0} visited by {1}",         concreteElementA.GetType().Name, this.GetType().Name);     }      public override void VisitConcreteElementB(       ConcreteElementB concreteElementB)     {       Console.WriteLine("{0} visited by {1}",         concreteElementB.GetType().Name, this.GetType().Name);     }   }    // "Element"     abstract class Element   {     public abstract void Accept(Visitor visitor);   }    // "ConcreteElementA"     class ConcreteElementA : Element   {     public override void Accept(Visitor visitor)     {       visitor.VisitConcreteElementA(this);     }      public void OperationA()     {     }   }    // "ConcreteElementB"     class ConcreteElementB : Element   {     public override void Accept(Visitor visitor)     {       visitor.VisitConcreteElementB(this);     }      public void OperationB()     {     }   }    // "ObjectStructure"     class ObjectStructure   {     private ArrayList elements = new ArrayList();      public void Attach(Element element)     {       elements.Add(element);     }      public void Detach(Element element)     {       elements.Remove(element);     }      public void Accept(Visitor visitor)     {       foreach (Element e in elements)       {         e.Accept(visitor);       }     }   } } |
| Output  ConcreteElementA visited by ConcreteVisitor1 ConcreteElementB visited by ConcreteVisitor1 ConcreteElementA visited by ConcreteVisitor2 ConcreteElementB visited by ConcreteVisitor2 |

This real-world code demonstrates the Visitor pattern in which two objects traverse a list of Employees and performs the same operation on each Employee. The two visitor objects define different operations -- one adjusts vacation days and the other income.

|  |
| --- |
| // Visitor pattern -- Real World example |
| using System; using System.Collections;  namespace DoFactory.GangOfFour.Visitor.RealWorld {      // MainApp startup application     class MainApp   {     static void Main()     {       // Setup employee collection        Employees e = new Employees();       e.Attach(new Clerk());       e.Attach(new Director());       e.Attach(new President());        // Employees are 'visited'        e.Accept(new IncomeVisitor());       e.Accept(new VacationVisitor());        // Wait for user        Console.Read();     }   }    // "Visitor"     interface IVisitor   {     void Visit(Element element);   }    // "ConcreteVisitor1"     class IncomeVisitor : IVisitor   {     public void Visit(Element element)     {       Employee employee = element as Employee;        // Provide 10% pay raise        employee.Income \*= 1.10;       Console.WriteLine("{0} {1}'s new income: {2:C}",          employee.GetType().Name, employee.Name,          employee.Income);     }   }    // "ConcreteVisitor2"     class VacationVisitor : IVisitor   {     public void Visit(Element element)     {       Employee employee = element as Employee;              // Provide 3 extra vacation days        Console.WriteLine("{0} {1}'s new vacation days: {2}",          employee.GetType().Name, employee.Name,          employee.VacationDays);     }   }    class Clerk : Employee   {     // Constructor      public Clerk() : base("Hank", 25000.0, 14)     {      }   }    class Director : Employee   {     // Constructor      public Director() : base("Elly", 35000.0, 16)     {       }   }    class President : Employee   {     // Constructor      public President() : base("Dick", 45000.0, 21)     {       }   }    // "Element"     abstract class Element   {     public abstract void Accept(IVisitor visitor);   }    // "ConcreteElement"     class Employee : Element   {     string name;     double income;     int vacationDays;      // Constructor      public Employee(string name, double income,        int vacationDays)     {       this.name = name;       this.income = income;       this.vacationDays = vacationDays;     }      // Properties      public string Name     {       get{ return name; }       set{ name = value; }     }      public double Income     {       get{ return income; }       set{ income = value; }     }      public int VacationDays     {       get{ return vacationDays; }       set{ vacationDays = value; }     }      public override void Accept(IVisitor visitor)     {       visitor.Visit(this);     }   }    // "ObjectStructure"     class Employees   {     private ArrayList employees = new ArrayList();      public void Attach(Employee employee)     {       employees.Add(employee);     }      public void Detach(Employee employee)     {       employees.Remove(employee);     }      public void Accept(IVisitor visitor)     {       foreach (Employee e in employees)       {         e.Accept(visitor);       }       Console.WriteLine();     }   } } |
| Output  Clerk Hank's new income: $27,500.00 Director Elly's new income: $38,500.00 President Dick's new income: $49,500.00  Clerk Hank's new vacation days: 14 Director Elly's new vacation days: 16 President Dick's new vacation days: 21 |