新版设计模式手册

[C#]

整理制作：Terrylee

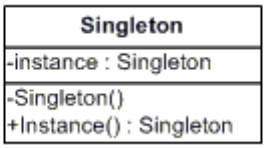
http://terrylee.cnblogs.com

目　　录

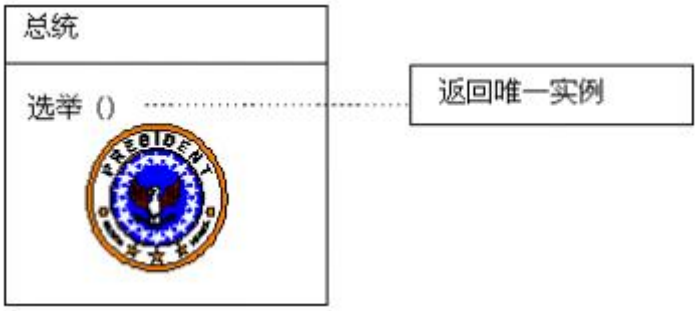
一. 创建型模式

1. 单件模式

结构图



生活例子



意图

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

适用性

- 当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。
- 当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

示意性代码

```
// 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;
    }
  }
}
```

实际应用

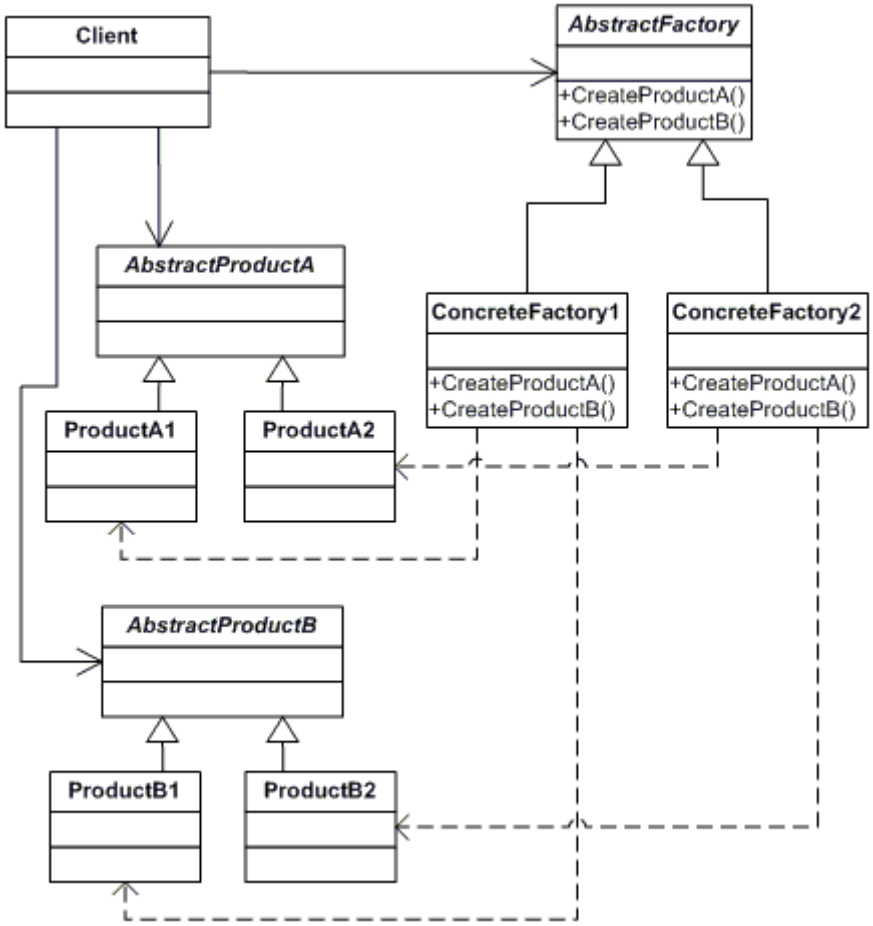
```
// Singleton pattern -- Real World example

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();
```

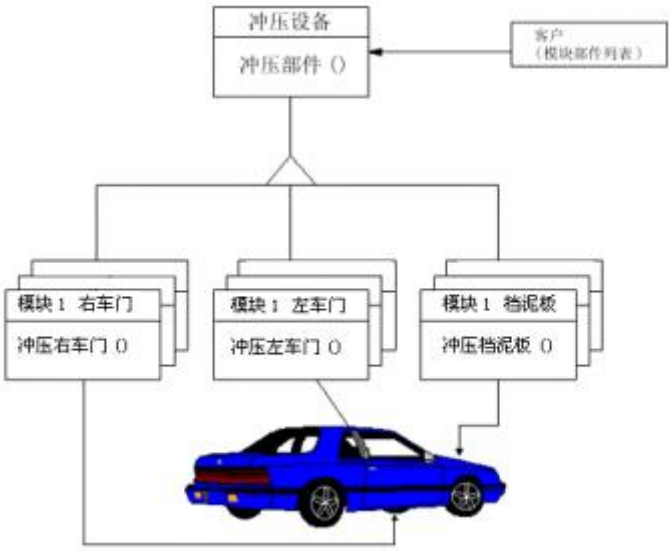
```
    // Constructor (protected)
    protected LoadBalancer()
    {
      // List of available servers
      servers.Add("ServerI");
      servers.Add("ServerII");
      servers.Add("ServerIII");
      servers.Add("ServerIV");
      servers.Add("ServerV");
    }

    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();
      }
    }
  }
}
```

2. 抽象工厂

结构图



生活例子



意图

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

适用性

- 一个系统要独立于它的产品的创建、组合和表示时。
- 一个系统要由多个产品系列中的一个来配置时。
- 当你要强调一系列相关的产品对象的设计以便进行联合使用时。
- 当你提供一个产品类库，而只想显示它们的接口而不是实现时。

示意性代码

```
// 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);
    }
  }
}
```

实际应用

```
// 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();
  }
```

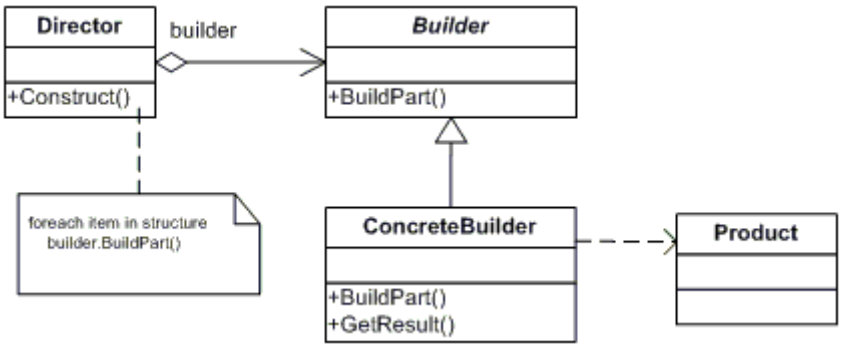
```
// "ConcreteFactory1"

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();
```

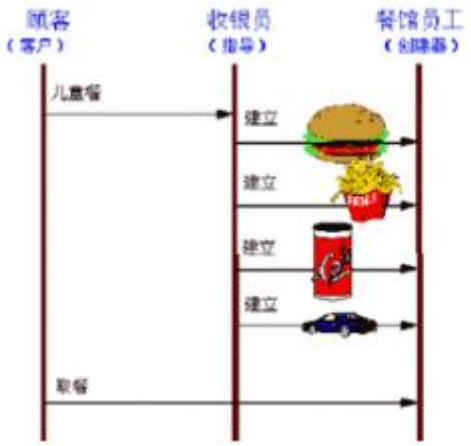
```
      herbivore = factory.CreateHerbivore();
    }

    public void RunFoodChain()
    {
      carnivore.Eat(herbivore);
    }
  }
}
```

3. 建造者模式

结构图



生活例子



意图

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

适用性

- 当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。
- 当构造过程必须允许被构造的对象有不同的表示时。

示意性代码

```
// 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);
    }
  }
}
```

实际应用

```
// 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; }
    }
```

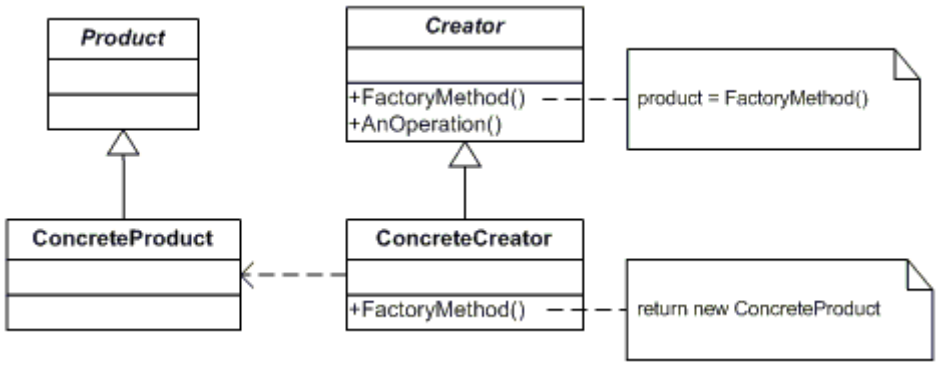
```
  public abstract void BuildFrame();
  public abstract void BuildEngine();
  public abstract void BuildWheels();
  public abstract void BuildDoors();
}

// "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
{
```

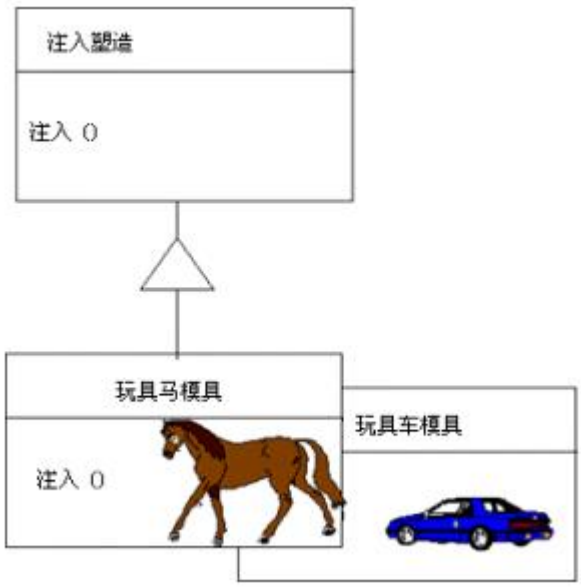
```
    private string type;
    private Hashtable parts = new Hashtable();

    // 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"]);
    }
  }
}
```

4. 工厂方法模式

结构图



生活例子



意图

定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method 使一个类的实例化延迟到其子类。

适用性

- 当一个类不知道它所必须创建的对象的类的时候。
- 当一个类希望由它的子类来指定它所创建的对象的时候。
- 当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。

示意性代码

```
// 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();
    }
  }
}
```

实际应用

```
// 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);
        }
      }
```

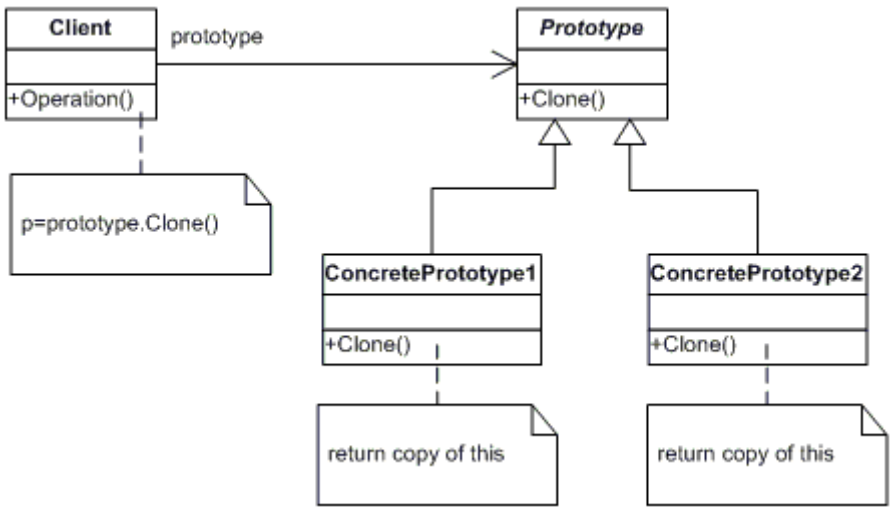
```
    // Wait for user
    Console.Read();
  }
}

// "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()
  {
```

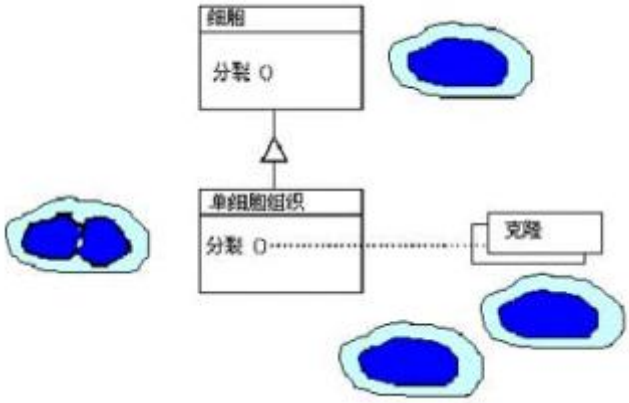
```
      Pages.Add(new SkillsPage());
      Pages.Add(new EducationPage());
      Pages.Add(new ExperiencePage());
    }
  }

  // "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());
    }
  }
}
```

5. 原型模式

结构图



生活例子



意图

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

适用性

- 当要实例化的类是在运行时刻指定时，例如，通过动态装载；或者
- 为了避免创建一个与产品类层次平行的工厂类层次时；或者
- 当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

示意性代码

```
// 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();
    }
  }
}
```

实际应用

```
// 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;
```

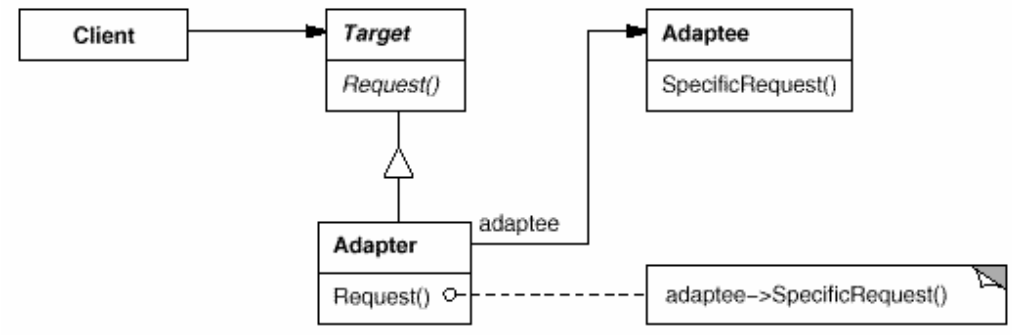
```
    // User uses selected colors
    string name = "red";
    color = colormanager[name].Clone() as Color;

    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);
```

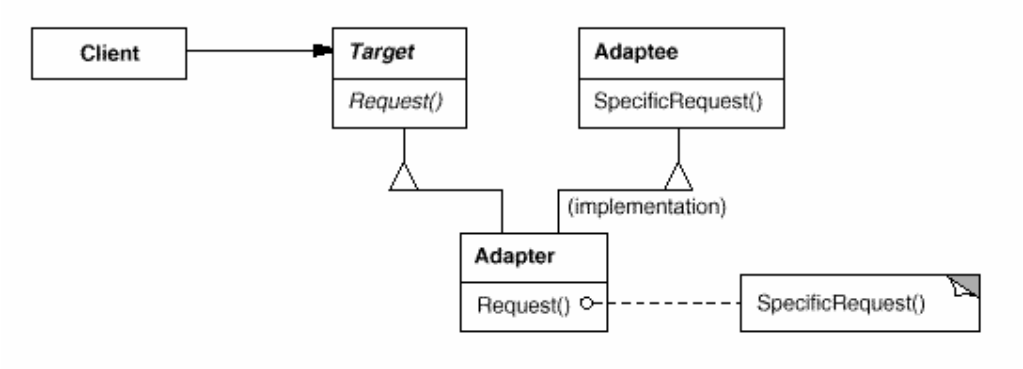
```
      return this.MemberwiseClone() as ColorPrototype;
    }
  }

  // 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);
      }
    }
  }
}
```

二．结构型模式

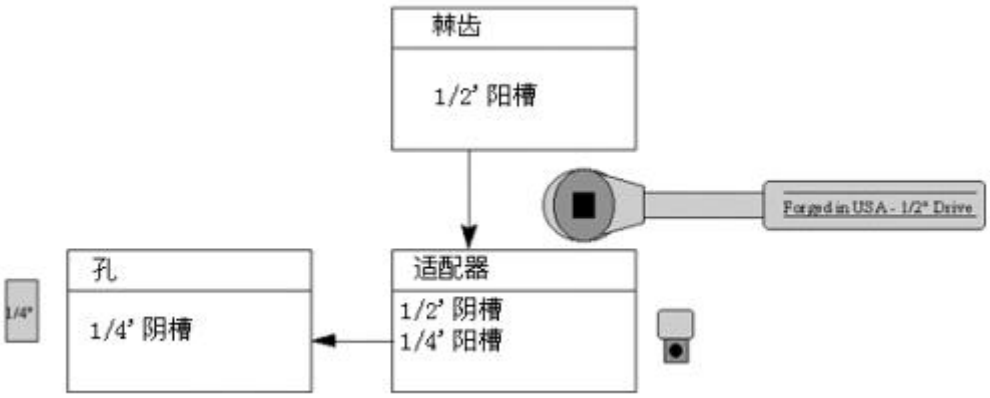
6．适配器模式

结构图





生活例子



意图

将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

适用性

- 你想使用一个已经存在的类，而它的接口不符合你的需求。
- 你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。
- （仅适用于对象 Adapter）你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。

示意性代码

```
// 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()");
```

```
    }
  }
}
```

实际应用

```
// 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;
```

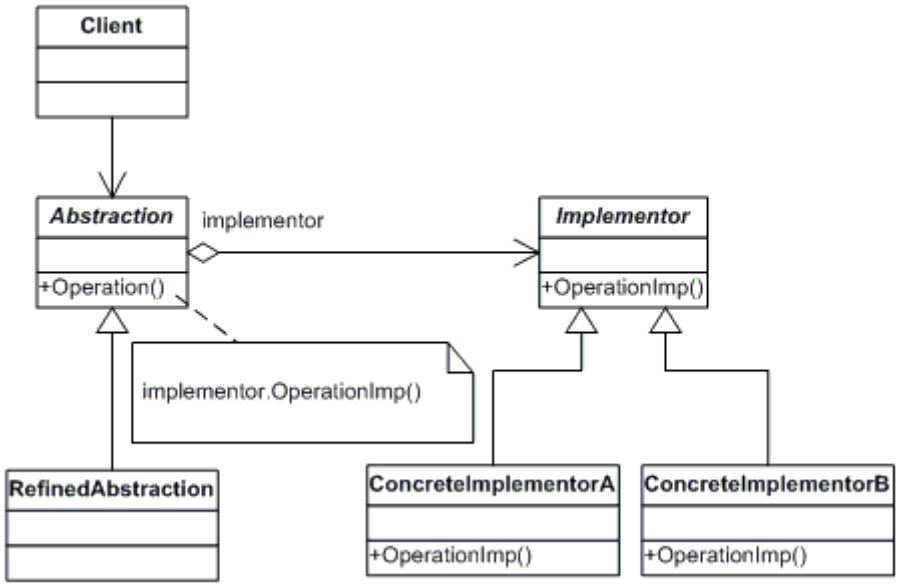
```
  // Constructor
  public Compound(string name)
  {
    this.name = name;
  }

  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)
  {
```

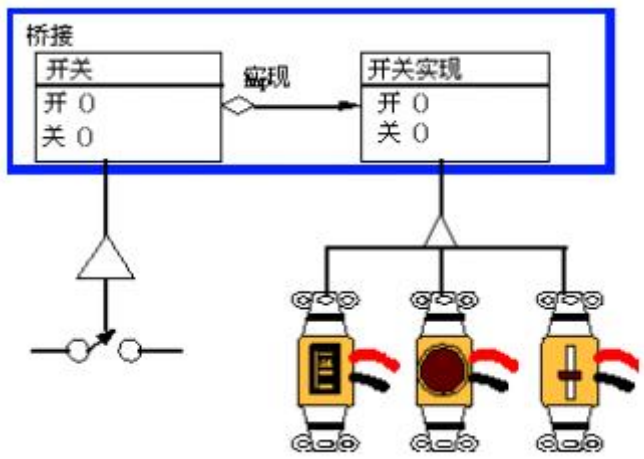
```
      double weight = 0.0;
      switch (compound.ToLower())
      {
        case "water" : weight = 18.015; break;
        case "benzene" : weight = 78.1134; break;
        case "alcohol" : weight = 46.0688; break;
      }
      return weight;
    }
  }
}
```

7. 桥接模式

结构图



生活例子



意图

将抽象部分与它的实现部分分离，使它们都可以独立地变化。

适用性

- 你不希望在抽象和它的实现部分之间有一个固定的绑定关系。例如这种情况可能是因为，在程序运行时刻实现部分应可以被选择或者切换。
- 类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。这时 Bridge 模式使你可以对不同的抽象接口和实现部分进行组合，并分别对它们进行扩充。
- 对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。
- 有许多类要生成。这样一种类层次结构说明你必须将一个对象分解成两个部分。
- 你想在多个对象间共享实现（可能使用引用计数），但同时要求客户并不知道这一点。

示意性代码

```
// 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");
    }
  }
}
```

实际应用

```
// 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();
```

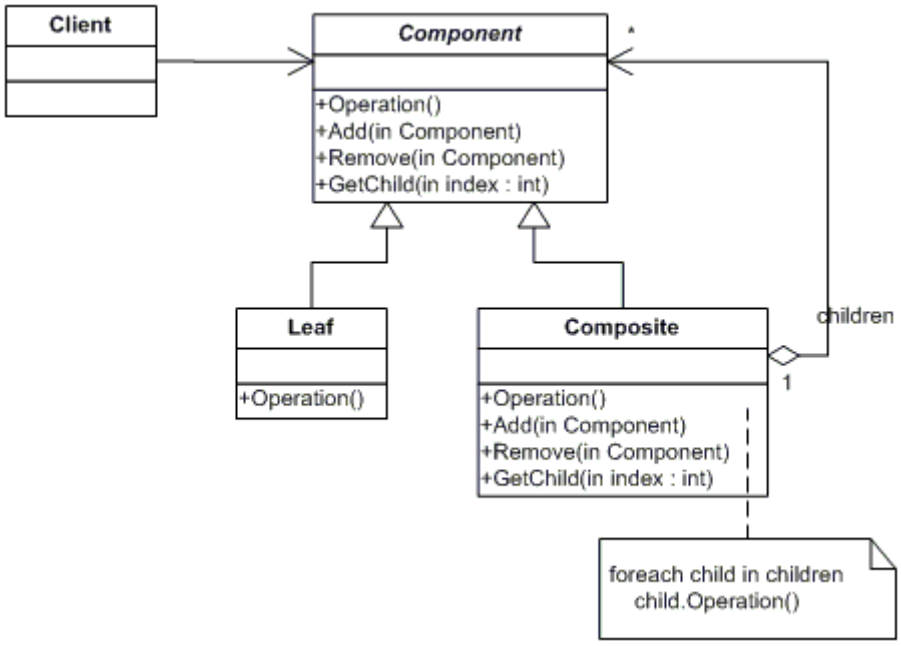
```
  }

  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]);
    }
```

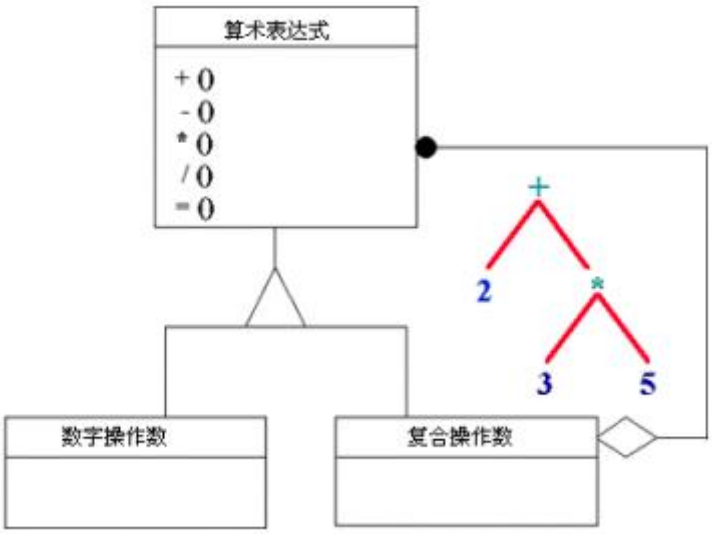
```
    public override void ShowAllRecords()
    {
      foreach (string name in customers)
      {
        Console.WriteLine(" " + name);
      }
    }
  }
}
```

8. 组合模式

结构图



生活例子



意图

将对象组合成树形结构以表示“部分-整体”的层次结构。Composite 使得用户对单个对象和组合对象的使用具有一致性。

适用性

- 你想表示对象的部分-整体层次结构。
- 你希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

示意性代码

```
// 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);
    }
  }
}
```

实际应用

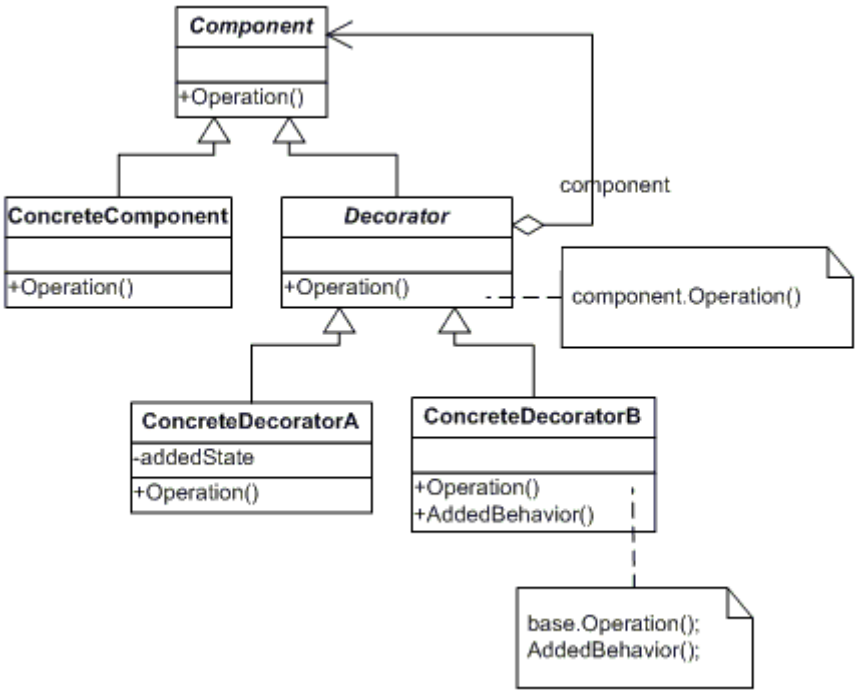
```
// Composite pattern -- Real World example
using System;
using System.Collections;

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)
  {
```

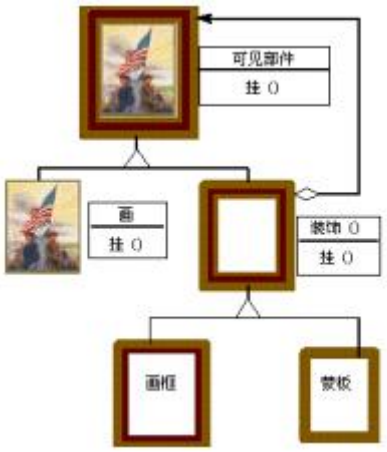
```
      Console.WriteLine(
        new String('-', indent) + " " + name);
    }
  }

  // "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);
      }
    }
  }
}
```

9. 装饰模式

结构图



生活例子



意图

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。

适用性

- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
- 处理那些可以撤消的职责。
- 当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

示意性代码

```
// 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()
    {
    }
  }
}
```

实际应用

```
// 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();
    }
```

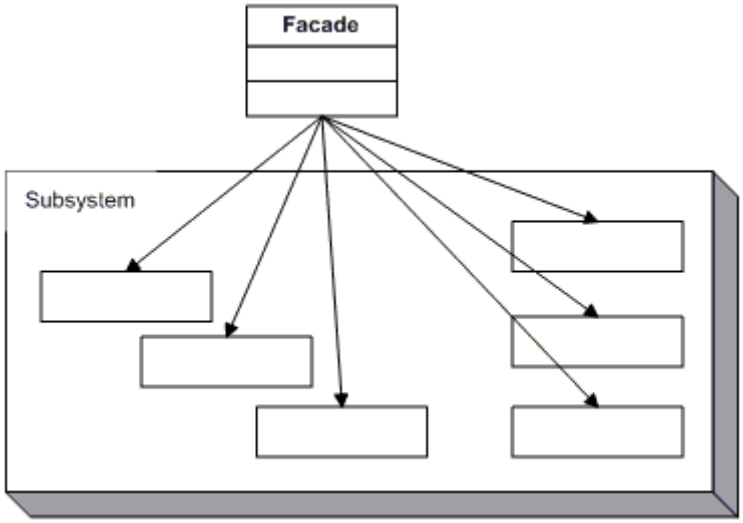
```
}

// "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();
  }
}
```

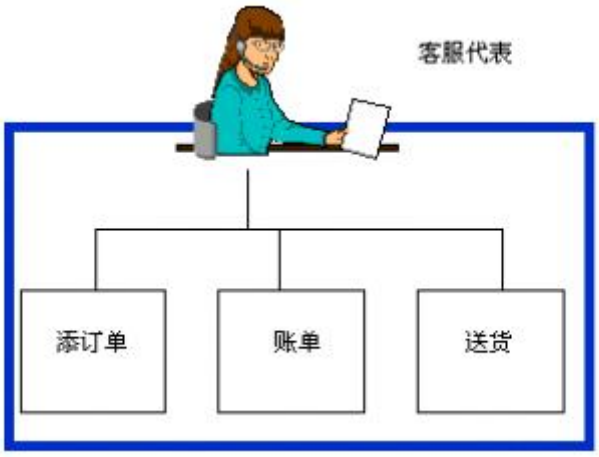
```
  // "ConcreteDecorator"

  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);
      }
    }
  }
}
```

10. 外观模式

结构图



生活例子



意图

为子系统中的一组接口提供一个一致的界面，Facade 模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

适用性

- 当你要为一个复杂子系统提供一个简单接口时。子系统往往因为不断演化而变得越来越复杂。大多数模式使用时都会产生更多更小的类。这使得子系统更具可重用性，也更容易对子系统进行定制，但这也给那些不需要定制子系统的用户带来一些使用上的困难。Facade 可以提供一个简单的缺省视图，这一视图对大多数用户来说已经足够，而那些需要更多的可定制性的用户可以越过 Facade 层。
- 客户程序与抽象类的实现部分之间存在着很大的依赖性。引入 Facade 将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。

- 当你需要构建一个层次结构的子系统时，使用 Facade 模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，你可以让它们仅通过 Facade 进行通讯，从而简化了它们之间的依赖关系。

示意性代码

```
// 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();
    }
  }
}
```

实际应用

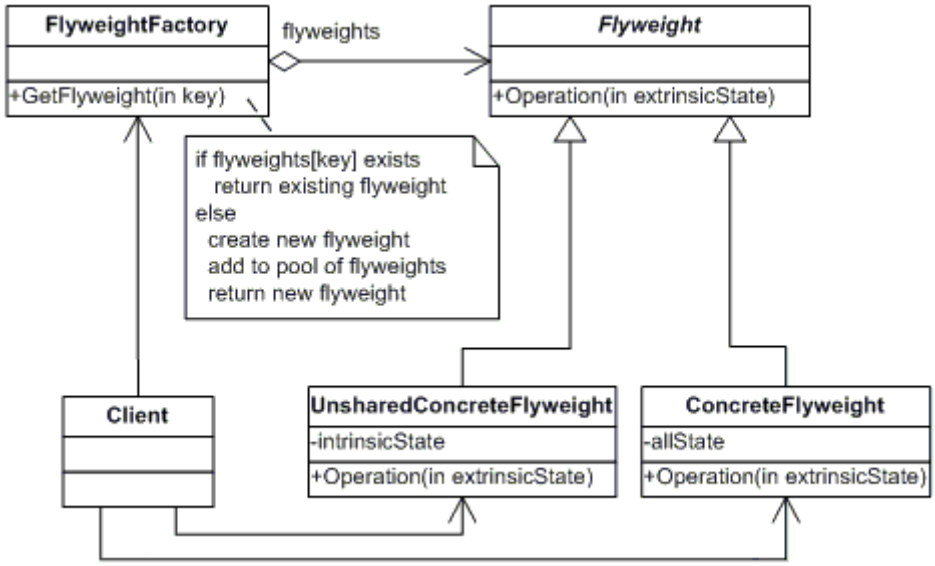
```
// Facade pattern -- Real World example
using System;

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
  {
```

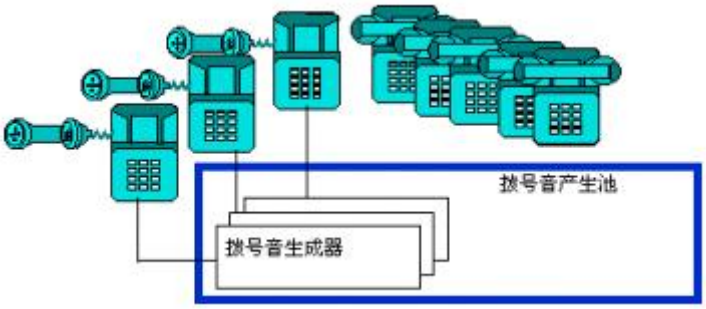
```
      get{ return name; }
    }
  }

  // "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;
    }
  }
}
```

11. 享元模式

结构图



生活例子



意图

运用共享技术有效地支持大量细粒度的对象。

适用性

- 一个应用程序使用了大量的对象。
- 完全由于使用大量的对象，造成很大的存储开销。
- 对象的大多数状态都可变为外部状态。
- 如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。
- 应用程序不依赖于对象标识。由于 Flyweight 对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。

示意性代码

```
// 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 fu = new
        UnsharedConcreteFlyweight();

      fu.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);
    }
  }
}
```

实际应用

```
 // 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)
        {
```

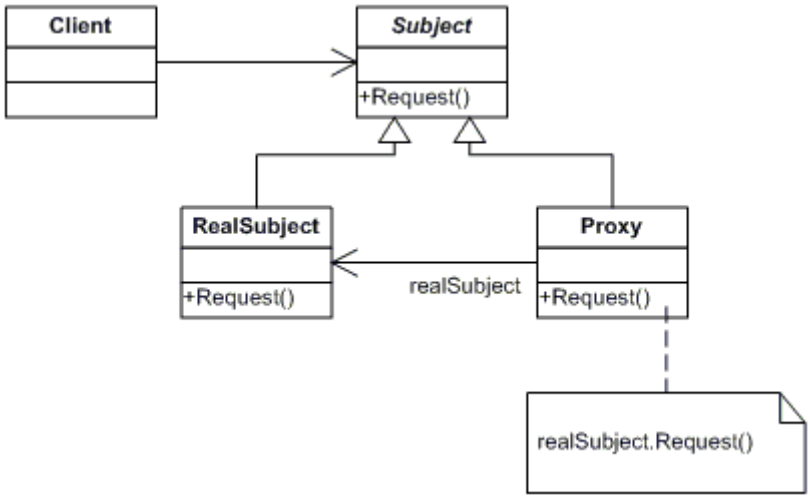
```
        case 'A': character = new CharacterA(); break;
        case 'B': character = new CharacterB(); break;
          //...
        case 'Z': character = new CharacterZ(); break;
      }
      characters.Add(key, character);
    }
    return character;
  }
}

// "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)
  {
```

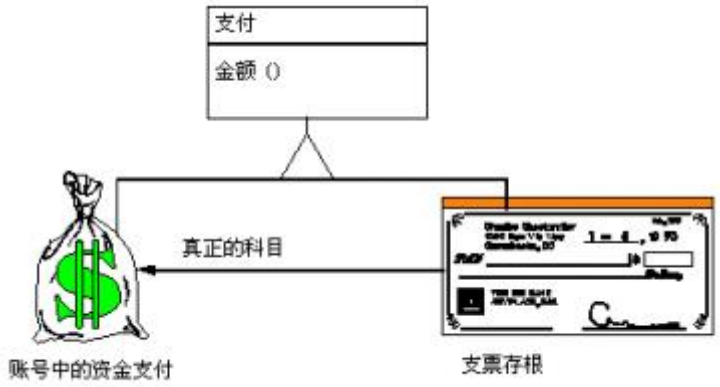
```
      this.pointSize = pointSize;
      Console.WriteLine(this.symbol +
        " (pointsize " + this.pointSize + ")");
    }
  }
}
```

12. 代理模式

结构图



生活例子



意图

为其他对象提供一种代理以控制对这个对象的访问。

适用性

- 在需要用比较通用和复杂的对象指针代替简单的指针的时候，使用 Proxy 模式。下面是一 些可以使用 Proxy 模式常见情况：

1）远程代理（Remote Proxy ）为一个对象在不同的地址空间提供局部代表。 NEXTSTEP[Add94]使用 NXProxy 类实现了这一目的。Coplien[Cop92] 称这种代理为“大使”（Ambassador）。

2)虚代理（Virtual Proxy ）根据需要创建开销很大的对象。在动机一节描述的 ImageProxy 就是这样一种代理的例子。

3）保护代理（Protection Proxy ）控制对原始对象的访问。保护代理用于对象应该有不同 的访问权限的时候。例如，在 Choices 操作系统[CIRM93]中 KemelProxies 为操作系统对象提供 了访问保护。

4)智能指引（Smart Reference ）取代了简单的指针，它在访问对象时执行一些附加操作。 它的典型用途包括：

- 对指向实际对象的引用计数，这样当该对象没有引用时，可以自动释放它(也称为 SmartPointers[Ede92]。
- 当第一次引用一个持久对象时，将它装入内存。

在访问一个实际对象前，检查是否已经锁定了它，以确保其他对象不能改变它。

示意性代码

```
// 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();
    }
  }
}
```

实际应用

```
// Proxy pattern -- Real World example
using System;

namespace DoFactory.GangOfFour.Proxy.RealWorld
{

  // Mainapp test application
```

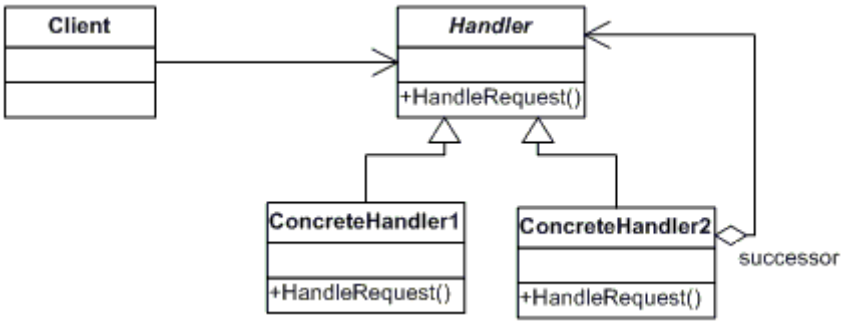
```
class MainApp
{
  static void Main()
  {
    // Create math proxy
    MathProxy p = new MathProxy();

    // 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);
    }
  }
}
```

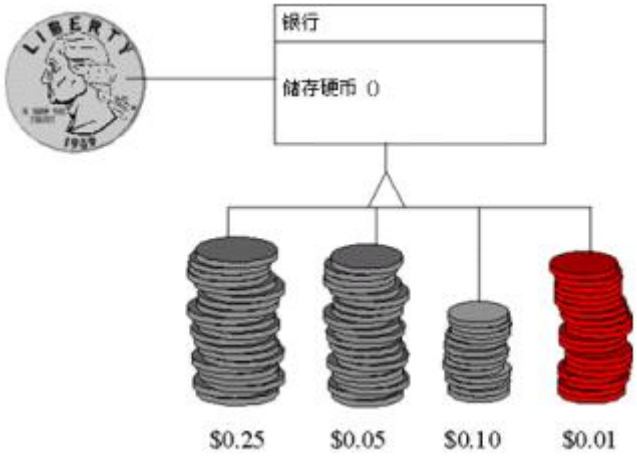
三．行为型模式

13．职责链模式

结构图



生活例子



意图

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

适用性

- 有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。
- 你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
- 可处理一个请求的对象集合应被动态指定。

示意性代码

```
// 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);
      }
    }
  }
}
```

实际应用

```
// Chain of Responsibility pattern -- Real World example
using System;

namespace DoFactory.GangOfFour.Chain.RealWorld
{

  // MainApp test application
```

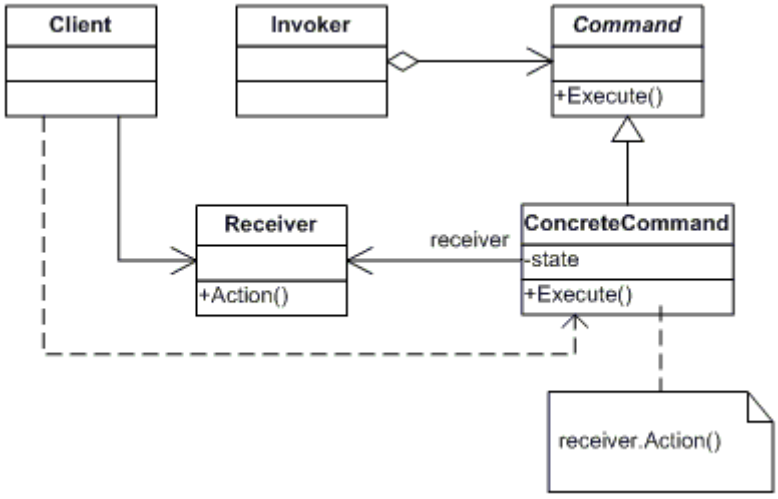
```
class MainApp
{
  static void Main()
  {
    // Setup Chain of Responsibility
    Director Larry = new Director();
    VicePresident Sam = new VicePresident();
    President Tammy = new President();
    Larry.SetSuccessor(Sam);
    Sam.SetSuccessor(Tammy);

    // 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; }
    }
  }
}
```

14. 命令模式

结构图



生活例子

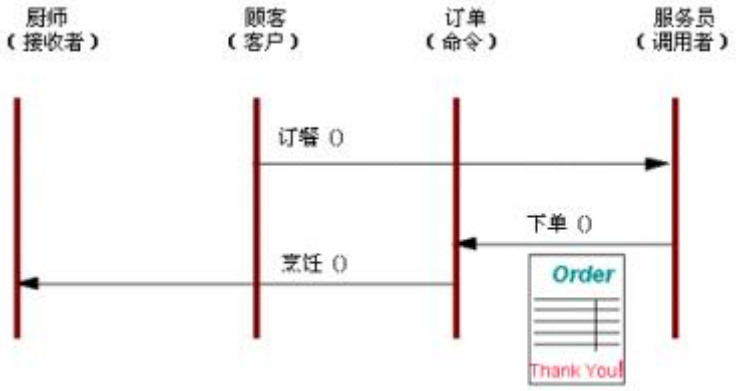


图 14：使用用餐例子的命令模式对象图

意图

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。

适用性

- 使用命令模式作为"CallBack"在面向对象系统中的替代。"CallBack"讲的便是先将一个函数登记上，然后在以后调用此函数。
- 需要在不同的时间指定请求、将请求排队。一个命令对象和原先的请求发出者可以有不同的生命期。换言之，原先的请求发出者可能已经不在了，而命令对象本身仍然是活动的。

这时命令的接收者可以是在本地，也可以在网络的另外一个地址。命令对象可以在串形化之后传送到另外一台机器上去。

- 系统需要支持命令的撤消(undo)。命令对象可以把状态存储起来，等到客户端需要撤销命令所产生的效果时，可以调用 undo()方法，把命令所产生的效果撤销掉。命令对象还可以提供 redo()方法，以供客户端在需要时，再重新实施命令效果。
- 如果一个系统要将系统中所有的数据更新到日志里，以便在系统崩溃时，可以根据日志里读回所有的数据更新命令，重新调用 Execute()方法一条一条执行这些命令，从而恢复系统在崩溃前所做的数据更新。
- 一个系统需要支持交易(Transaction)。一个交易结构封装了一组数据更新命令。使用命令模式来实现交易结构可以使系统增加新的交易类型。

示意性代码

```
// 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();
    }
  }
}
```

实际应用

```
// 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()
```

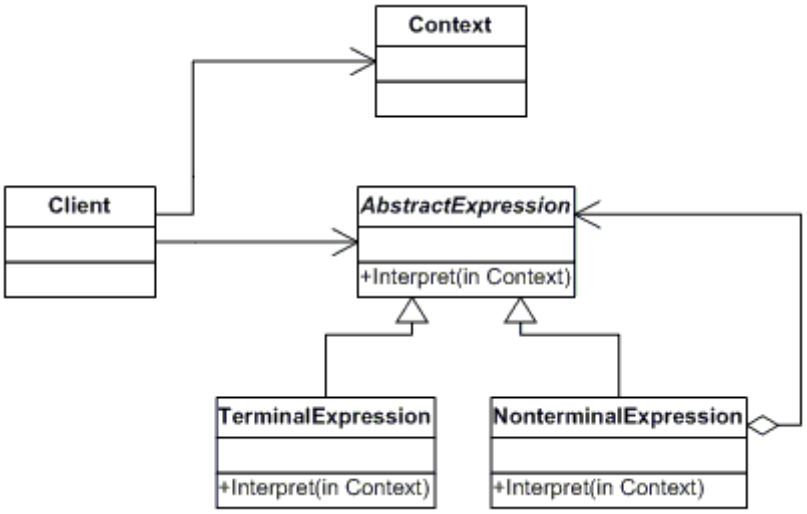
```
  {
    calculator.Operation(Undo(@operator), operand);
  }

  // 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++;
    }
  }
}
```

15. 解释器模式

结构图



生活例子

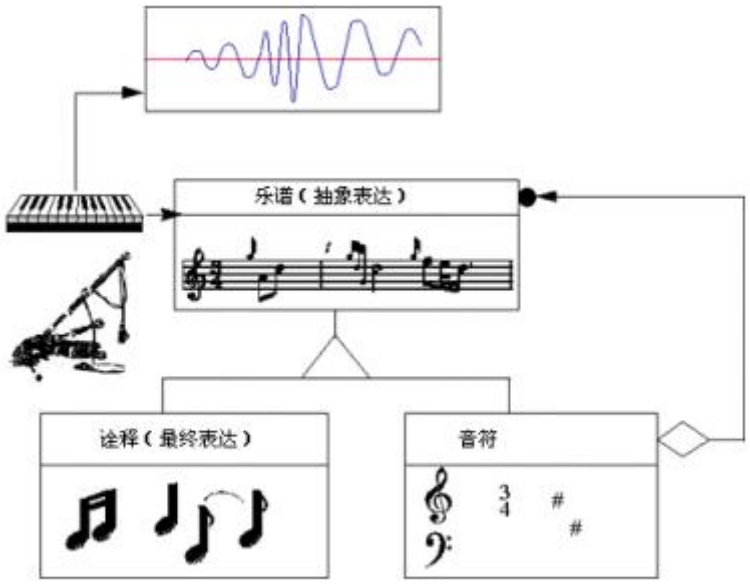


图 15：使用音乐例子的解释器模式对象图

意图

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

适用性

- 当有一个语言需要解释执行，并且你可将该语言中的句子表示为一个抽象语法树时，可使用解释器模式。而当存在以下情况时该模式效果最好：
- 该文法简单对于复杂的文法，文法的类层次变得庞大而无法管理。此时语法分析程序生成器这样的工具是更好的选择。它们无需构建抽象语法树即可解释表达式，这样可以节省空间而且还可能节省时间。
- 效率不是一个关键问题最高效的解释器通常不是通过直接解释语法分析树实现的，而是首先将它们转换成另一种形式。例如，正则表达式通常被转换成状态机。但即使在这种情况下，转换器仍可用解释器模式实现，该模式仍是有用的。

示意性代码

```
// 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()");
    }
  }
}
```

实际应用

```
// 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)
```

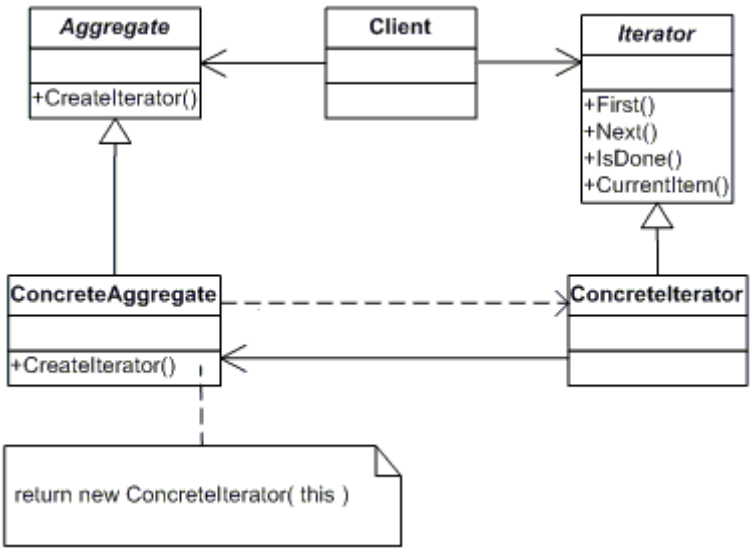
```
  {
    this.input = input;
  }

  // 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; }
  }
}
```

16. 迭代器模式

结构图



生活例子

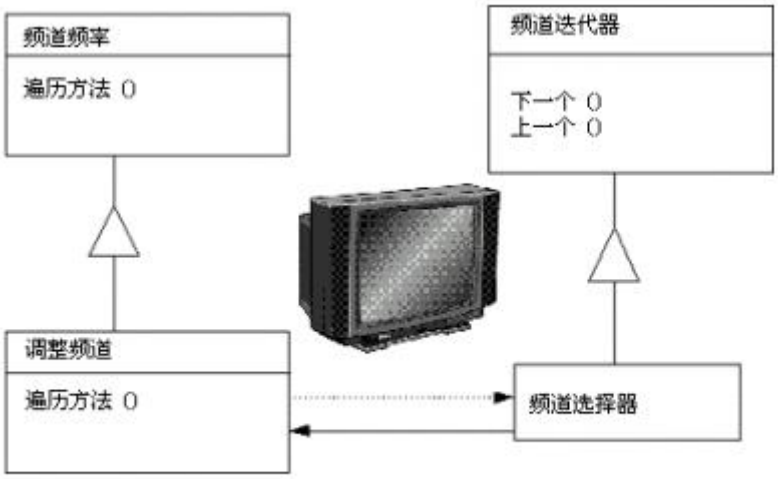


图 16：使用选频器作例子的迭代模式对象图

意图

提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。

适用性

- 访问一个聚合对象的内容而无需暴露它的内部表示。
- 支持对聚合对象的多种遍历。
- 为遍历不同的聚合结构提供一个统一的接口(即，支持多态迭代)。

示意性代码

```
// 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 ;
    }
  }
}
```

实际应用

```
// 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;
```

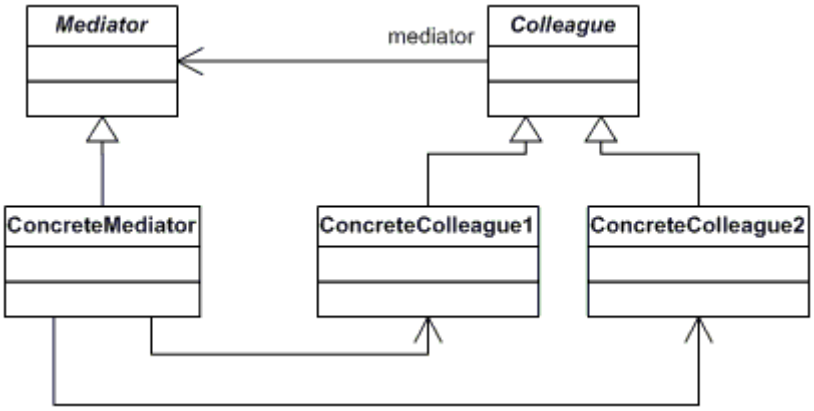
```
    Console.WriteLine("Iterating over collection:");

    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()
    {
```

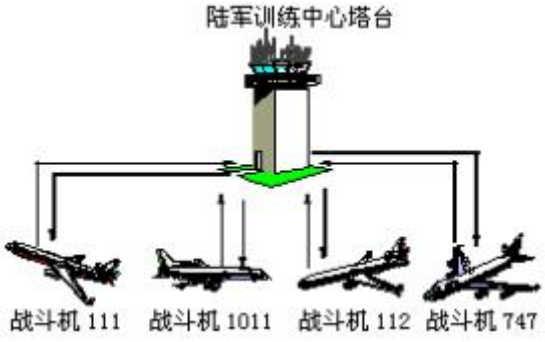
```
      current = 0;
      return collection[current] as Item;
    }

    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;
      }
    }
  }
}
```

17. 中介者模式

结构图



生活例子



意图

用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

适用性

- 一组对象以定义良好但是复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。
- 一个对象引用其他很多对象并且直接与这些对象通信,导致难以复用该对象。
- 想定制一个分布在多个类中的行为，而又不想生成太多的子类。

示意性代码

```
// 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);
    }
  }
}
```

实际应用

```
// 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();
```

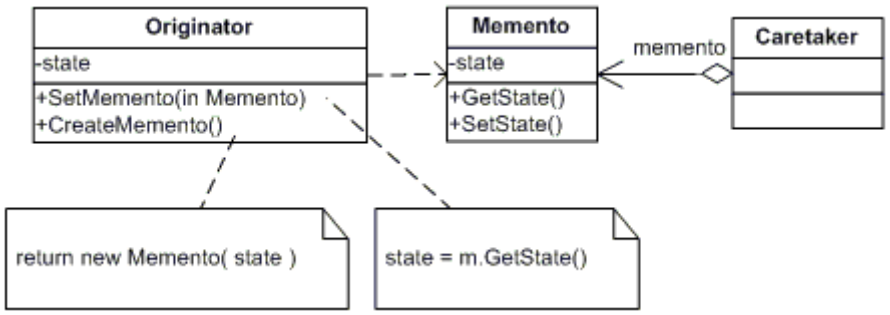
```
    // Create participants and register them
    Participant George = new Beatle("George");
    Participant Paul = new Beatle("Paul");
    Participant Ringo = new Beatle("Ringo");
    Participant John = new Beatle("John") ;
    Participant Yoko = new NonBeatle("Yoko");

    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);
    }
  }
}
```

18. 备忘录模式

结构图



生活例子

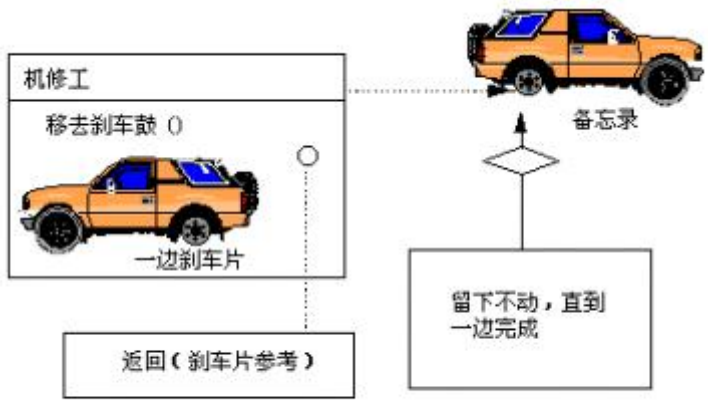


图 18：使用刹车片例子的备忘录模式对象图

意图

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

适用性

- 必须保存一个对象在某一个时刻的(部分)状态，这样以后需要时它才能恢复到先前的状态。
- 如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

示意性代码

```
// 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; }
    }
  }
}
```

实际应用

```
// 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;
```

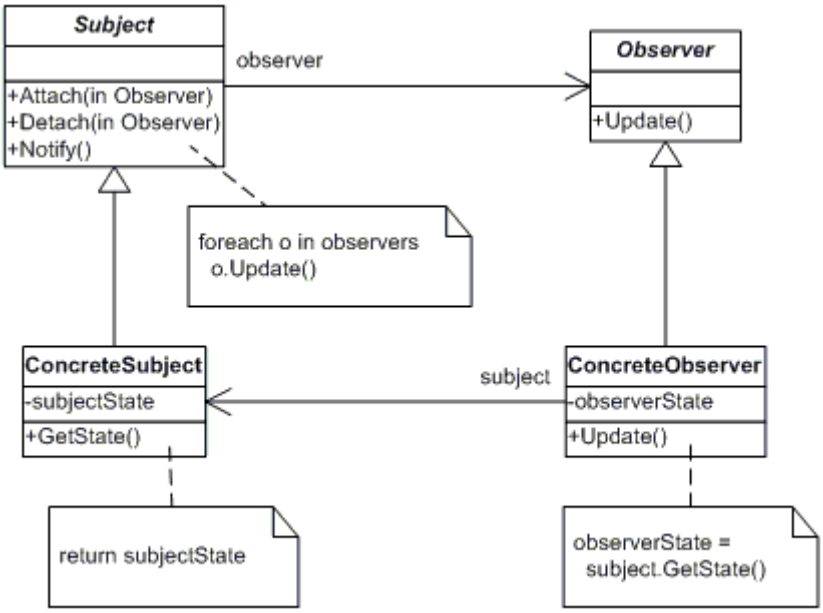
```
// Properties
public string Name
{
  get{ return name; }
  set
  {
    name = value;
    Console.WriteLine("Name: " + name);
  }
}

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; }
    }
  }
}
```

19. 观察者模式

结构图



生活例子

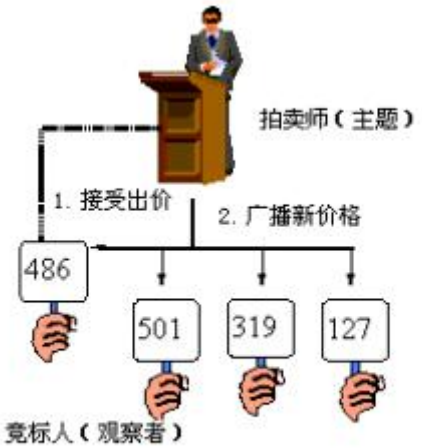


图 19：使用拍卖例子的观察者模式

意图

定义对象间的一种一对多的依赖关系,当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

适用性

- 当一个抽象模型有两个方面，其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。
- 当对一个对象的改变需要同时改变其它对象，而不知道具体有多少对象有待改变。
- 当一个对象必须通知其它对象，而它又不能假定其它对象是谁。换言之，你不希望这些对象是紧密耦合的。

示意性代码

```
// 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; }
    }
  }
}
```

实际应用

```
// Observer pattern -- Real World example
using System;
using System.Collections;

namespace DoFactory.GangOfFour.Observer.RealWorld
{

  // MainApp test application
```

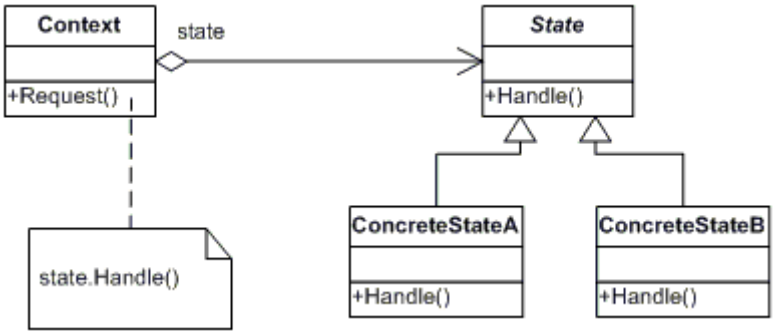
```
class MainApp
{
  static void Main()
  {
    // Create investors
    Investor s = new Investor("Sorros");
    Investor b = new Investor("Berkshire");

    // 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; }
    }
  }
}
```

20. 状态模式

结构图



生活例子

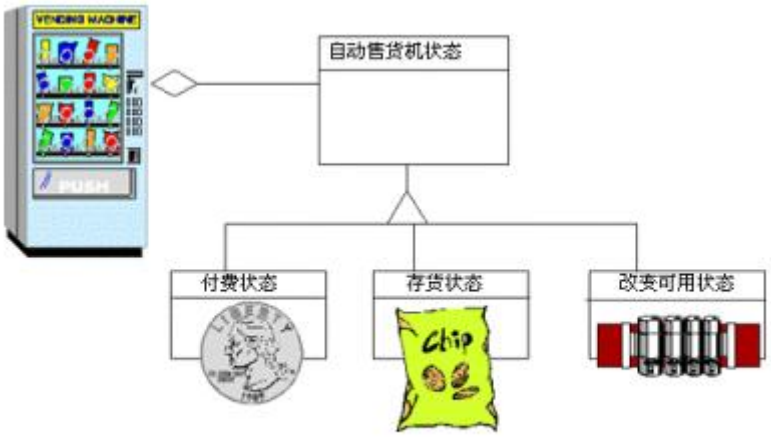


图 20：使用自动售货机例子的状态模式对象图

意图

允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

适用性

- 一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为。
- 一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。这个状态通常用一个或多个枚举常量表示。通常，有多个操作包含这一相同的条件结构。State 模式将每一个条件分支放入一个独立的类中。这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。

示意性代码

```
// 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);
    }
  }
}
```

实际应用

```
// 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();
  }
```

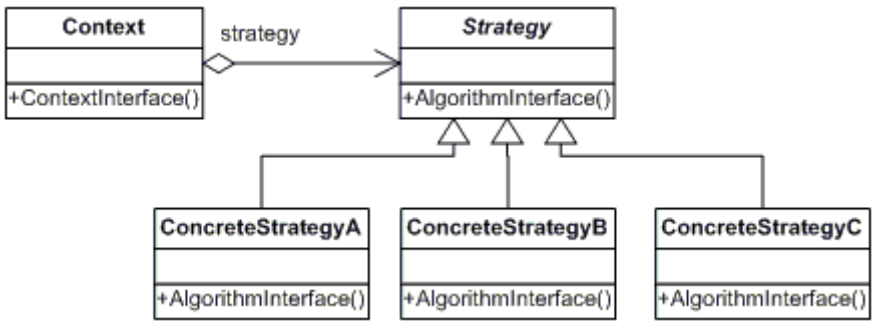
```
  private void Initialize()
  {
    // Should come from a database
    interest = 0.05;
    lowerLimit = 1000.0;
    upperLimit = 10000000.0;
  }

  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);
    }
  }
}
```

21. 策略模式

结构图



生活例子

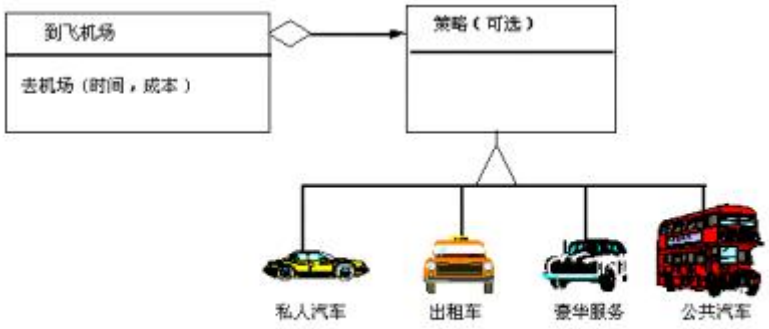


图 21：使用去机场作为例子的策略模式对象图

意图

定义一系列的算法,把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

适用性

- 许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。

- 需要使用一个算法的不同变体。例如，你可能会定义一些反映不同的空间/时间权衡的算法。当这些变体实现为一个算法的类层次时[H O 8 7]，可以使用策略模式。
- 算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。
- 一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的 Strategy 类中以代替这些条件语句。

示意性代码

```
// 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();
    }
  }
}
```

实际应用

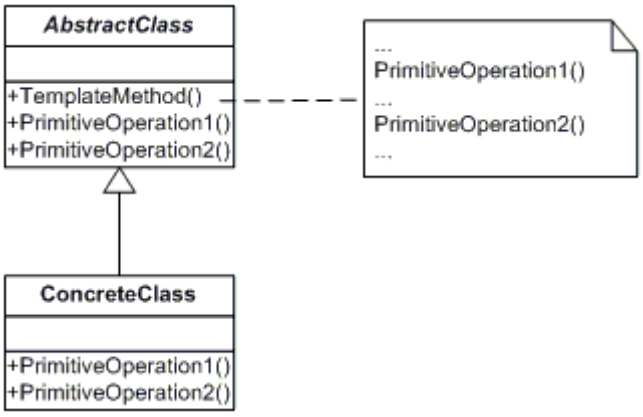
```
// Strategy pattern -- Real World example
using System;
using System.Collections;

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();
    }
  }
}
```

22. 模版方法

结构图



生活例子

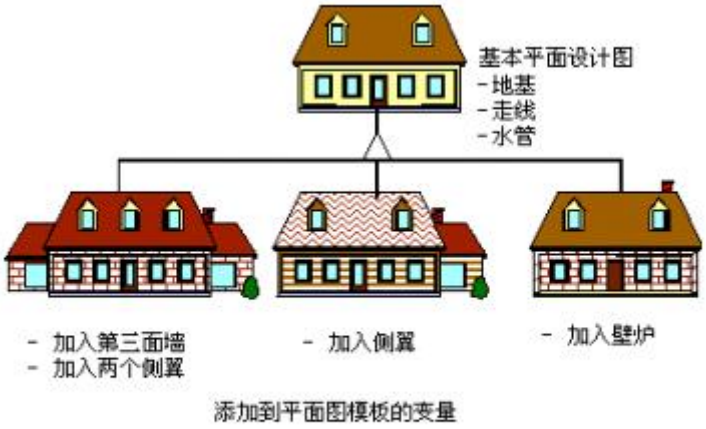


图 22：使用建筑平面图为例子的模板方法模式

意图

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

适用性

- 一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。
- 各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。这是 Opdyke 和 Johnson 所描述过的“重分解以一般化”的一个很好的例子[O J 9 3]。首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。
- 控制子类扩展。模板方法只在特定点调用“hook”操作，这样就只允许在这些点进行扩展。

示意性代码

```
// Template Method 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()");
    }
  }
}
```

实际应用

```
// Template Method 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();
```

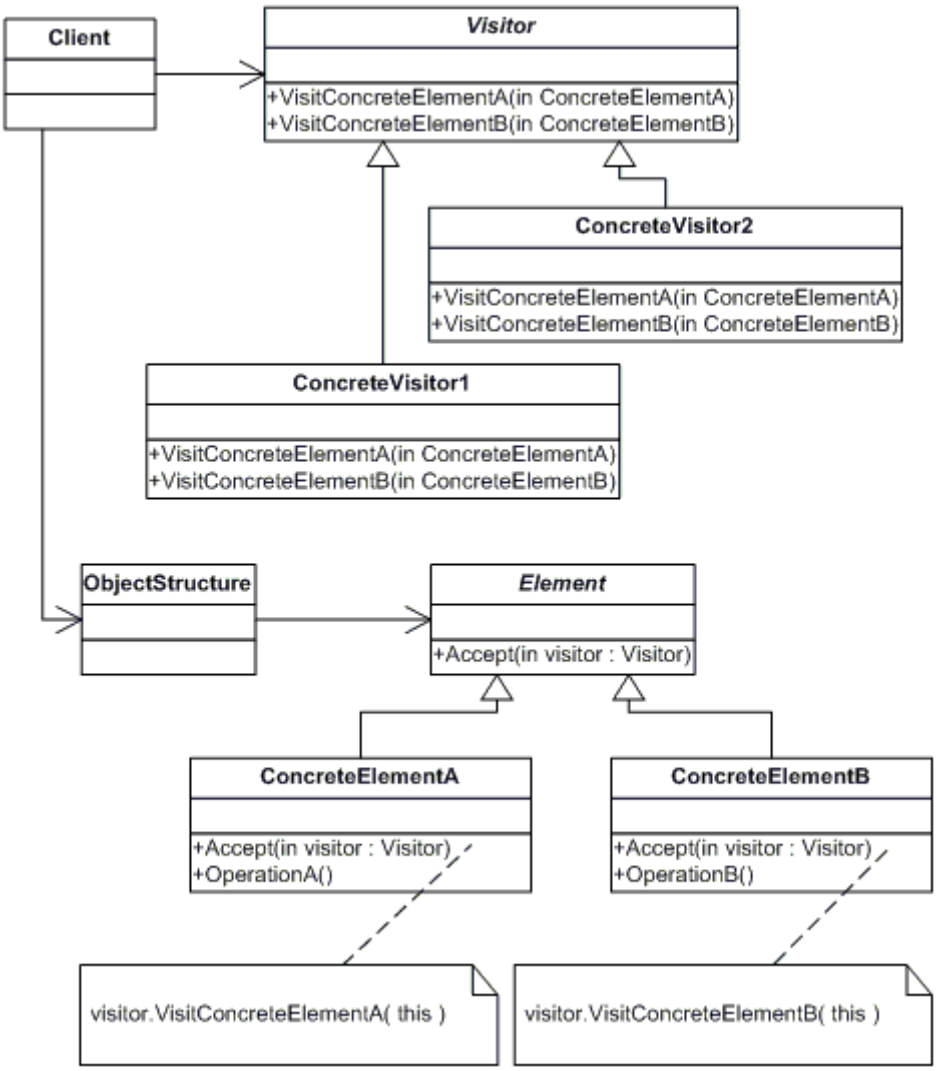
```
    dao.Run();

    // 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();
    }
  }
}
```

23. 访问者模式

结构图



生活例子

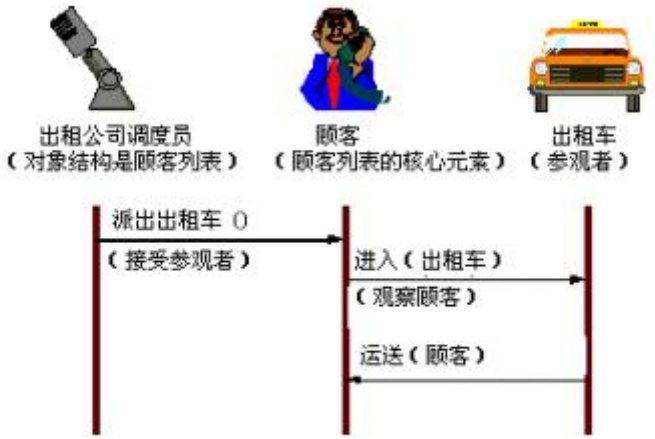


图 23：使用出租车例子的观察者模式对象图

意图

表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

适用性

- 一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作。
- 需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而你想避免让这些操作“污染”这些对象的类。Visitor 使得你可以将相关的操作集中起来定义在一个类中。当该对象结构被很多应用共享时，用 Visitor 模式让每个应用仅包含需要用到的操作。
- 定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。改变对象结构类需要重定义对所有访问者的接口，这可能需要很大的代价。如果对象结构类经常改变，那么可能还是在这些类中定义这些操作较好。

示意性代码

```
// 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);
      }
    }
  }
}
```

实际应用

```
// 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();
    }
  }
}
```

[注]

出处：博客园 http://www.cnblogs.com

整理制作：Terrylee http://terrylee.cnblogs.com

代码出处：http://www.dofactory.com/