

第1章 设计模式

设计模式 (Design Pattern) 描述了软件开发过程中若干重复出现的问题的解决方案, 这些方案不是由过程、算法等底层程序构造实体实现, 而是由软件系统中类与类之间或不同类的对象之间的共生关系组成。

设计模式可以帮助软件设计人员学习、重用前人的经验和成果。

设计模式的分类整理最早见于 Erich Gamma 在德国慕尼黑大学的博士论文。1995 年, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides 合著的《Design Patterns: Elements of Reusable Object-Oriented Software》系统地整理和描述了 23 个精选的设计模式 (GOF 模式), 为设计模式的学习、研究和推广提供了良好的范例。

第2章 GOF 设计模式

创建型

- **Abstract Factory (抽象工厂模式):** 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。
- **Builder (生成器模式):** 将一个复杂对象的构件与它的表示分离, 使得同样的构建过程可以创建不同的表述。
- **Factory Method (工厂模式):** 定义一个用于创建对象的接口, 让子类决定将哪一个类实例化。Factory Method 使一个类的实例化延迟到其子类。
- **Prototype (原型模式):** 用原型实例指定创建对象的种类, 并且通过拷贝这个原型来创建新的对象。
- **Singleton (单件模式):** 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。

结构型

- **Adapter (适配器模式):** 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。
- **Bridge (桥接模式):** 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。
- **Composite (组合模式):** 将对象组合成树形结构以表示“部分-整体”的层次结构。Composite 使得客户对单个对象和复合对象的使用具有一致性。

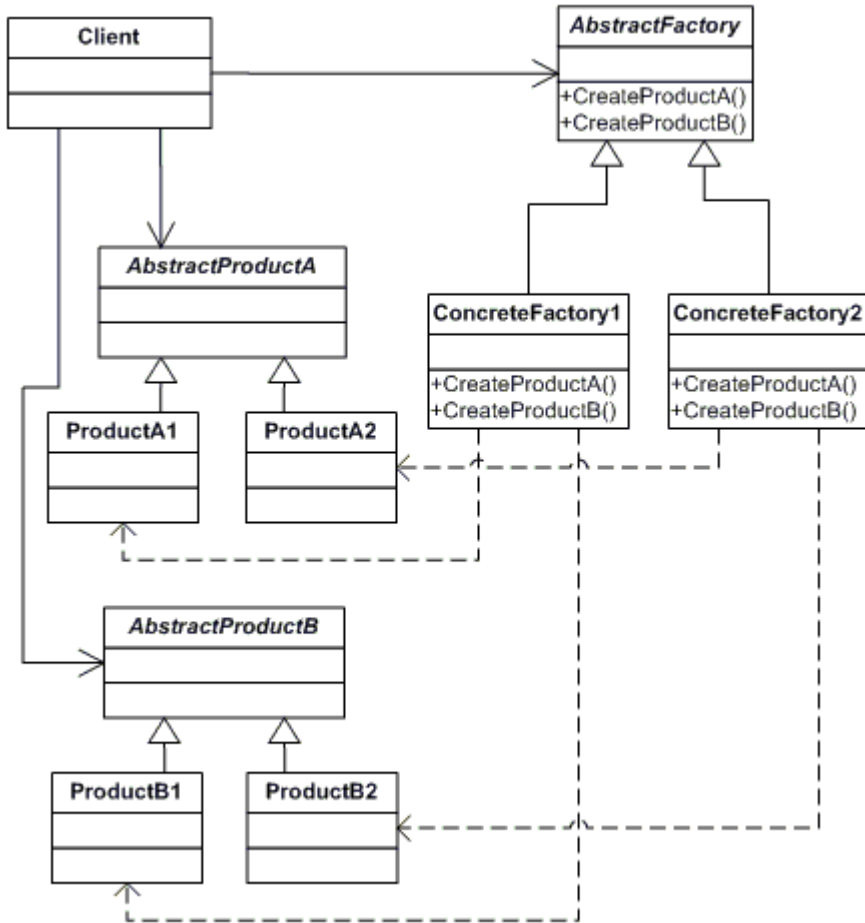
- **Decorator (装饰模式)**: 动态地给一个对象添加一些额外的职责。就扩展功能而言, Decorator 模式比生成子类方式更为灵活。
- **Facade (外观模式)**: 为子系统的一组接口提供一个一致的界面, Facade 模式定义了一个高层接口, 这个接口使得这一子系统更加容易使用。
- **Flyweight (享元模式)**: 运用共享技术有效地支持大量细粒度的对象。
- **Proxy (代理模式)**: 为其他对象提供一个代理以控制对这个对象的访问。

行为型

- **Chain of Responsibility (职责链模式)**: 为解除请求的发送者和接受者之间耦合, 而使多个对象都有机会处理这个请求。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。
- **Command (命令模式)**: 将一个请求封装为一个对象, 从而使你可以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, 以及支持可取消的操作。
- **Interpreter (解释器模式)**: 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, 该解释器使用该表示来解释语言中的句子。
- **Iterator (迭代器模式)**: 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。
- **Mediator (中介者模式)**: 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, 从而使器耦合松散, 而且可以独立地改变它们之间的交互。
- **Memento (备忘录模式)**: 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。这样以后就可以将该对象恢复到保存的状态。
- **Observer (观察者模式)**: 定义对象间的一种一对多的依赖关系, 以便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自动刷新。
- **State (状态模式)**: 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所属的类。
- **Strategy (策略模式)**: 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。本模式使得算法的变化可独立于使用它的客户。
- **TemplateMethod (模板方法模式)**: 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。
- **Visitor (访问者模式)**: 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。
- **创建型**

2.1. 创建型

2.1.1. Abstract Factory（抽象工厂模式）



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。

场景：

- 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。

重构成本：

中。

代码：

```
using System;
namespace GOF.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);
}
}
}

```

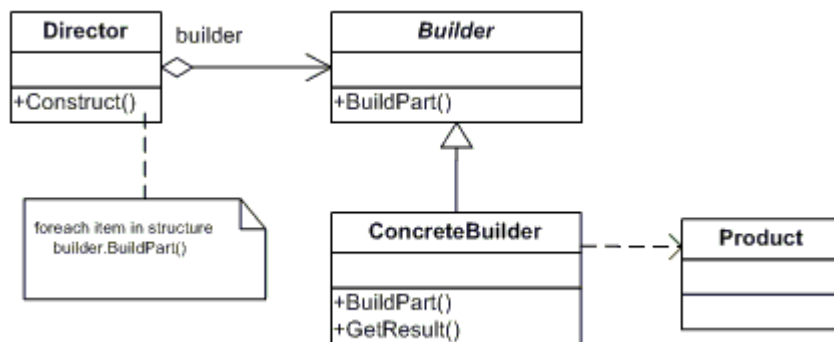
输出

```

ProductB1 interacts with ProductA1
ProductB2 interacts with ProductA2

```

2.1.2. Builder（生成器模式）



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。

场景：

- 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。

重构成本：

低。

代码：

```

using System;
using System.Collections;
namespace GOF.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);
    }
}
}
```

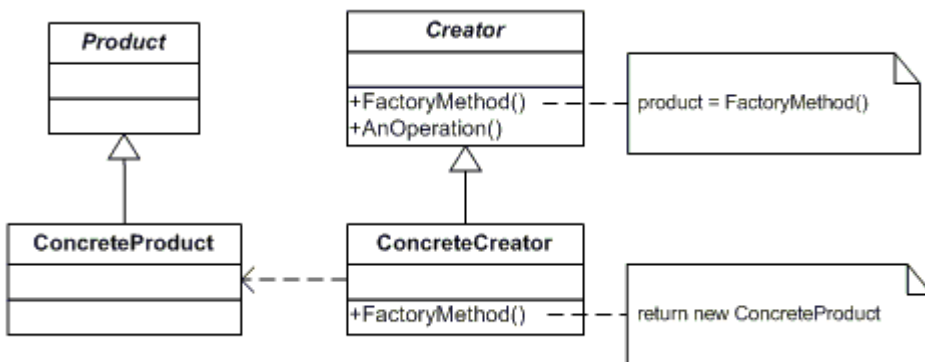
输出

```

Product Parts -----
PartA
PartB

Product Parts -----
PartX
PartY
    
```

2.1.3. Factory Method (工厂模式)



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。

场景:

- 当一个类不知道它所必须创建的对象类的类的时候。
- 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。

重构成本:

低。

代码:

```

using System;
using System.Collections;
namespace GOF.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();
    }
}

```

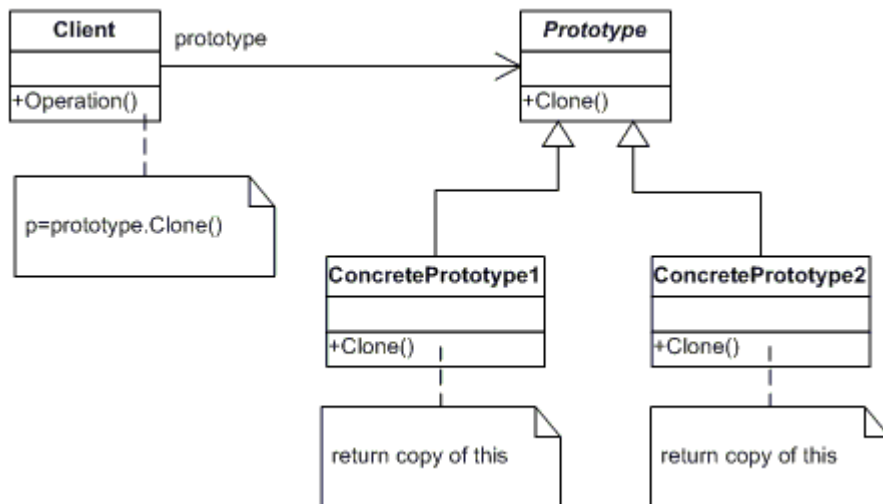
输出

```

Created ConcreteProductA
Created ConcreteProductB

```

2.1.4. Prototype（原型模式）



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。

场景:

- 当一个系统应该独立于它的产品创建、构成和表示时，要使用 Prototype 模式
- 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。

重构成本:

极低。

代码:

```
using System;
namespace GOF.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();
    }
}
}

```

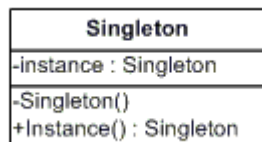
输出

```

Cloned: I
Cloned: II

```

2.1.5. Singleton（单件模式）



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。

场景：

- 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。

重构成本:

极低。

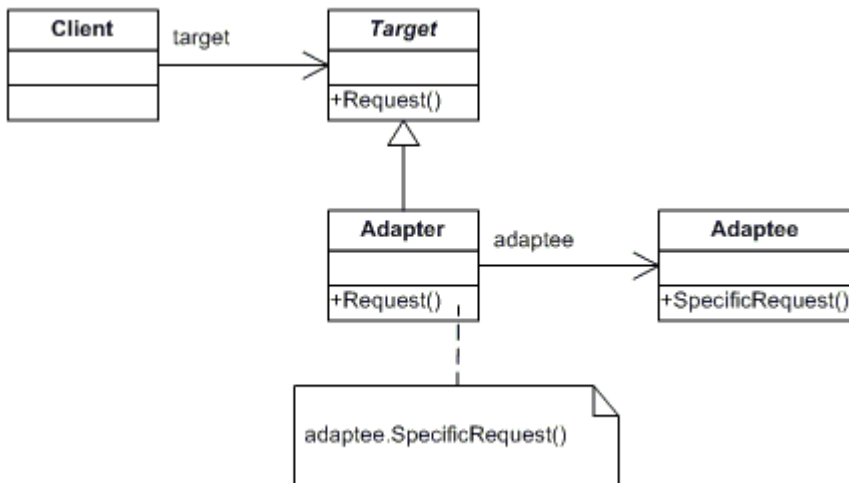
代码:

```
using System;
namespace GOF.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;
        }
    }
}
```

输出

2.2. 结构型

2.2.1. Adapter（适配器模式）



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。

场景：

- 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。

重构成本：

低。

代码：

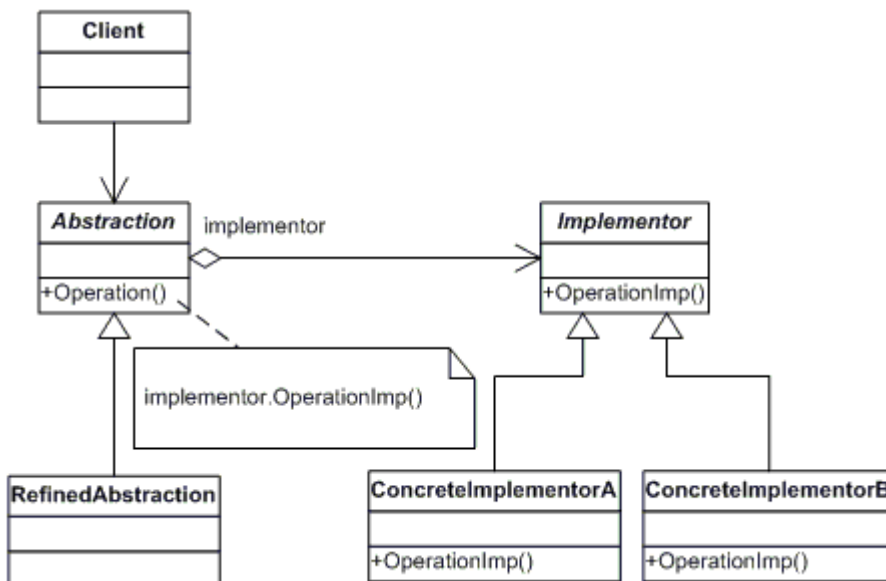
```
using System;
namespace GOF.Adapter.Structural
{
    // Mainapp test application
    class MainApp
    {
        static void Main()
        {
```

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

输出

```
Called SpecificRequest()
```

2.2.2. Bridge（桥接模式）



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。

场景：

- 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。
- （C++）你想对客户完全隐藏抽象的实现部分。在 C++ 中，类的表示在类接口中是可见的。
- 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。Rumbaugh 称这种类层次结构为“嵌套的普化”（nested generalizations）。
- 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。

重构成本：

中。

代码：

```

using System;
namespace GOF.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 implementation 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");
    }
}
}

```

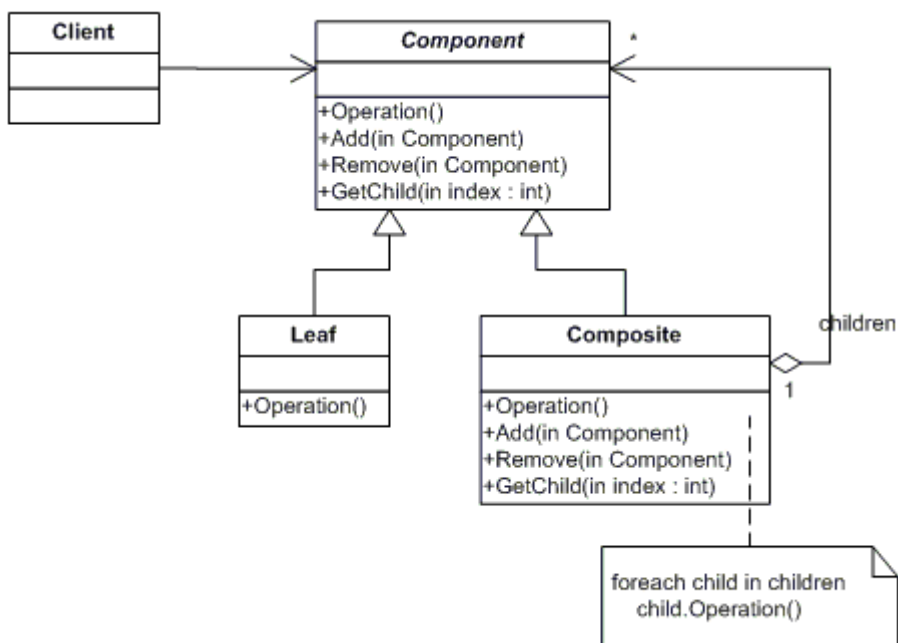
输出

```

ConcreteImplementorA Operation
ConcreteImplementorB Operation

```

2.2.3. Composite（组合模式）



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。

场景:

- 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。

重构成本:

高。

代码:

```
using System;
using System.Collections;
namespace GOF.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);
}
}
}

```

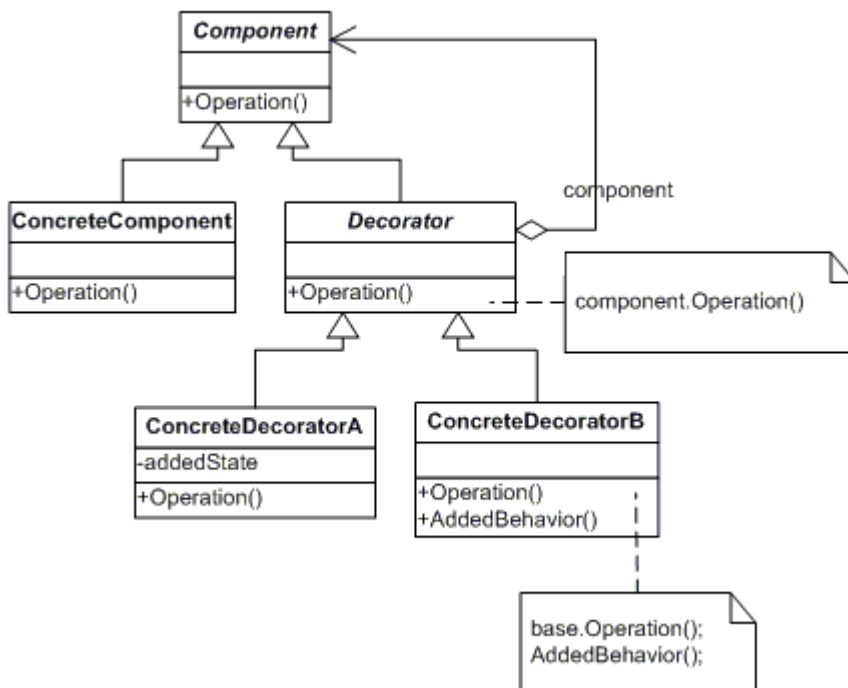
输出

```

-root
---Leaf A
---Leaf B
---Composite X
-----Leaf XA
-----Leaf XB
---Leaf C

```

2.2.4. Decorator (装饰模式)



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。

场景：

- 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。

重构成本：

低。

代码：

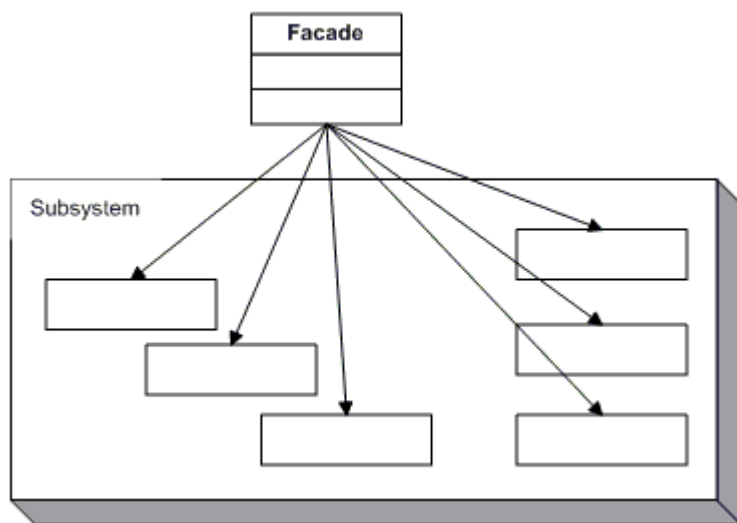
```
using System;
namespace GOF.Decorator.Structural
{
    // MainApp test application
    class MainApp
    {
        static void Main()
        {
            // Create ConcreteComponent and two Decorators
            ConcreteComponent c = new ConcreteComponent();
            ConcreteDecoratorA d1 = 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()
    {
    }
}
}
```

输出

```
ConcreteComponent.Operation()  
ConcreteDecoratorA.Operation()  
ConcreteDecoratorB.Operation()
```

2.2.5. Facade（外观模式）



意图：

为子系统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。

场景：

- 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。

重构成本：

高。

代码:

```
using System;
namespace GOF.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();
    }
}
}
```

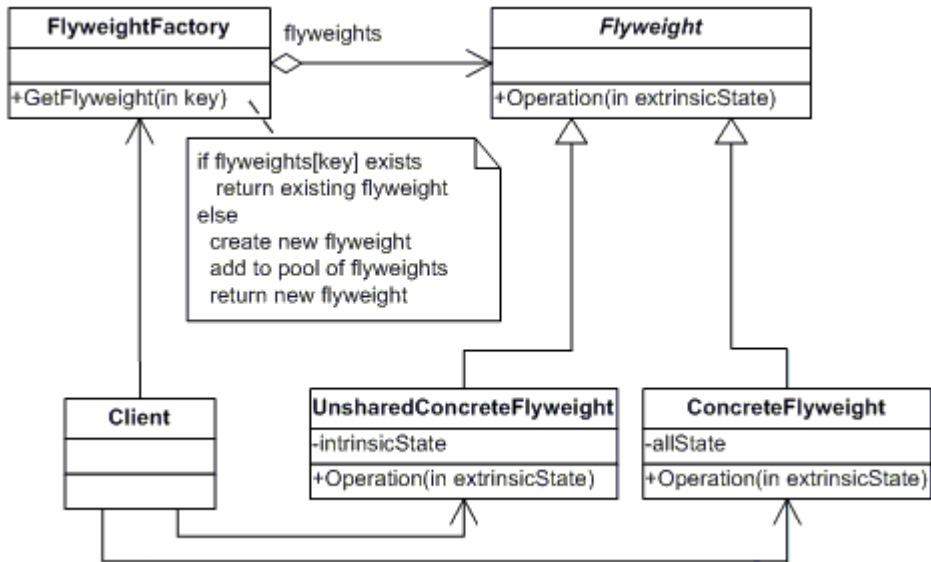
输出

```
MethodA () ----
SubSystemOne Method
SubSystemTwo Method
SubSystemFour Method

MethodB () ----
```

```
SubSystemTwo Method
SubSystemThree Method
```

2.2.6. Flyweight（享元模式）



意图：

运用共享技术有效地支持大量细粒度的对象。

场景：

- 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。

重构成本：

低。

代码：

```
using System;
using System.Collections;
namespace GOF.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);
    }
}
}
}

```

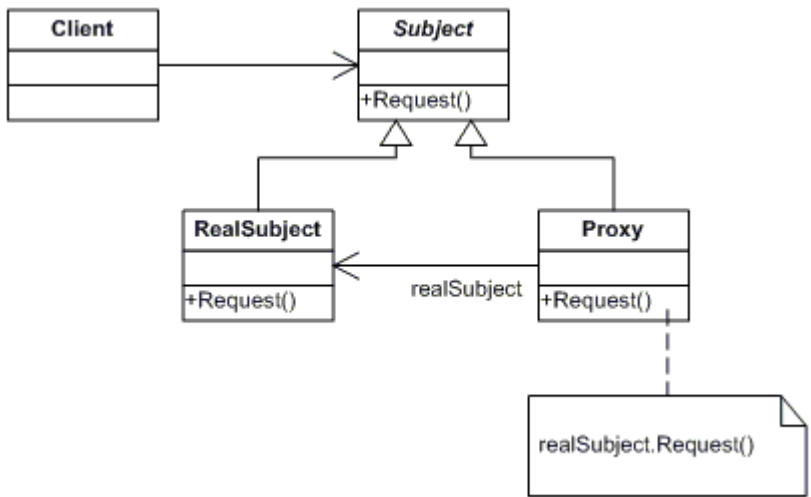
输出

```

ConcreteFlyweight: 21
ConcreteFlyweight: 20
ConcreteFlyweight: 19
UnsharedConcreteFlyweight: 18

```

2.2.7. Proxy (代理模式)



意图:

为其他对象提供一种代理以控制对这个对象的访问。

场景:

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:

- 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。
- 虚代理（Virtual Proxy）根据需要创建开销很大的对象。
- 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访问权限的时候。
- 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。
- 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。

重构成本：

低。

代码：

```
using System;
namespace GOF.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();
    }
}

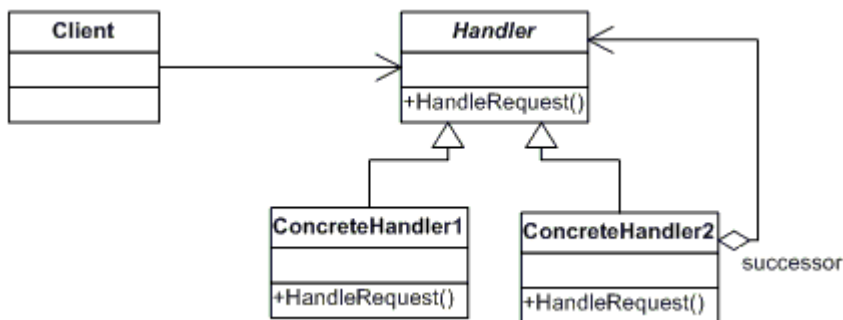
```

输出

Called RealSubject.Request()

2.3. 行为型

2.3.1. Chain of Responsibility（职责链模式）



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。

场景：

- 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。

重构成本：

中。

代码:

```
using System;
namespace GOF.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);
        }
    }
}
}
```

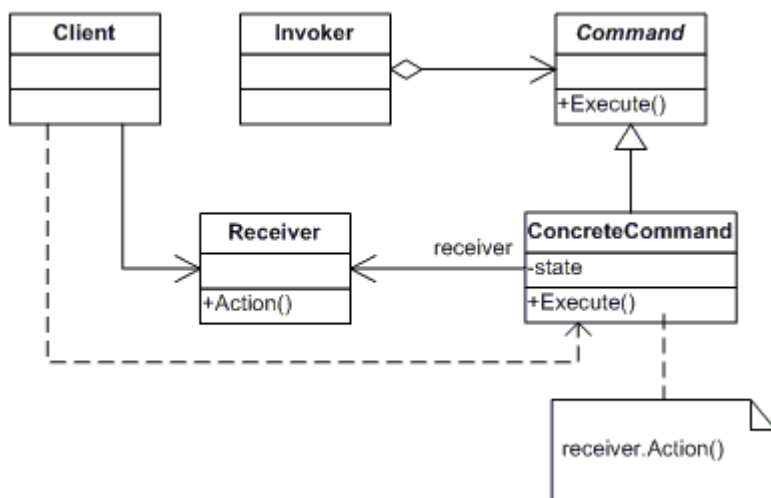
输出

```

ConcreteHandler1 handled request 2
ConcreteHandler1 handled request 5
ConcreteHandler2 handled request 14
ConcreteHandler3 handled request 22
ConcreteHandler2 handled request 18
ConcreteHandler1 handled request 3
ConcreteHandler3 handled request 27
ConcreteHandler3 handled request 20

```

2.3.2. Command（命令模式）



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，以及支持可撤销的操作。

场景：

- 抽象出待执行的动作以参数化某对象。你可用过程语言中的回调（Call back）函数表达这种参数化机制。所谓回调函数是指函数先在某处注册，而它将在稍后某个需要的时候被调用。Command 模式是回调机制的一个面向对象的替代品。
- 在不同的时刻指定、排列和执行请求。一个 Command 对象可以有一个与初始请求无关的生存期。如果一个请求的接收者可用一种与地址空间无关的方式表达，那么就可将负责该请求的命令对象传送给另一个不同的进程并在那儿实现该请求。
- 支持取消操作。Command 的 Excute 操作可在实施操作前将状态存储起来，在取消操作时这个状态用来消除该操作的影响。Command 接口必须添加一个 Unexcute 操作，该操作取消上一次 Excute 调用的效果。执行的命令被存储在一个历史列表中。可通过向后和向前遍历这一列表并分别调用 Unexcute 和 Excute 来实现重数不限的

“取消”和“重做”。

- 支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。在 **Command** 接口中添加装载操作和存储操作，可以用来保持变动的一个一致的修改日志。从崩溃中恢复的过程包括从磁盘中重新读入记录下来的命令并用 **Excute** 操作重新执行它们。
- 用构建在原语操作上的高层操作构造一个系统。这样一种结构在支持事务 (transaction) 的信息系统中很常见。一个事务封装了对数据的一组变动。模式提供了对事务进行建模的方法。**Command** 有一个公共的接口，使得你可以用同一种方式调用所有的事务。同时使用该模式也易于添加新事务以扩展系统。

重构成本：

高。

代码：

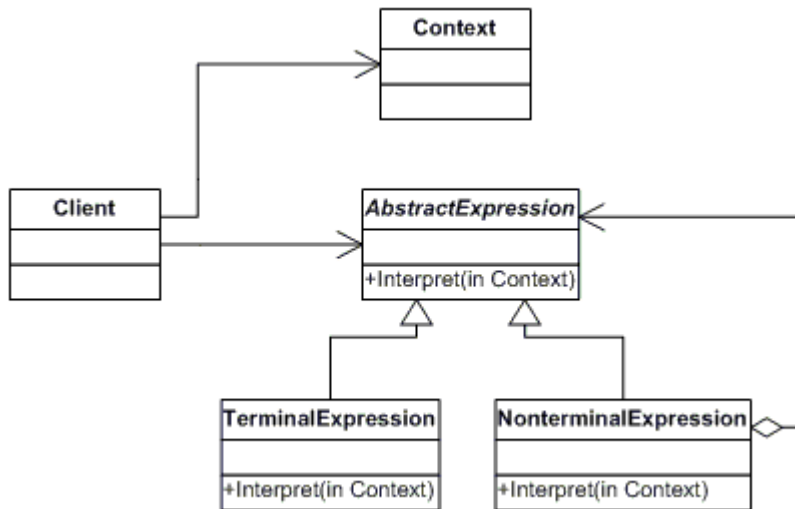
```
using System;
namespace GOF.Command.Structural
{
    // MainApp test applicatio
    class MainApp
    {
        static void Main()
        {
            // Create receiver, command, and invoker
            Receiver receiver = new Receiver();
            Command command = new ConcreteCommand(receiver);
            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();
    }
}
}
```

输出

```
Called Receiver.Action()
```

2.3.3. Interpreter (解释器模式)



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。

场景:

- 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。

重构成本:

高。

代码:

```

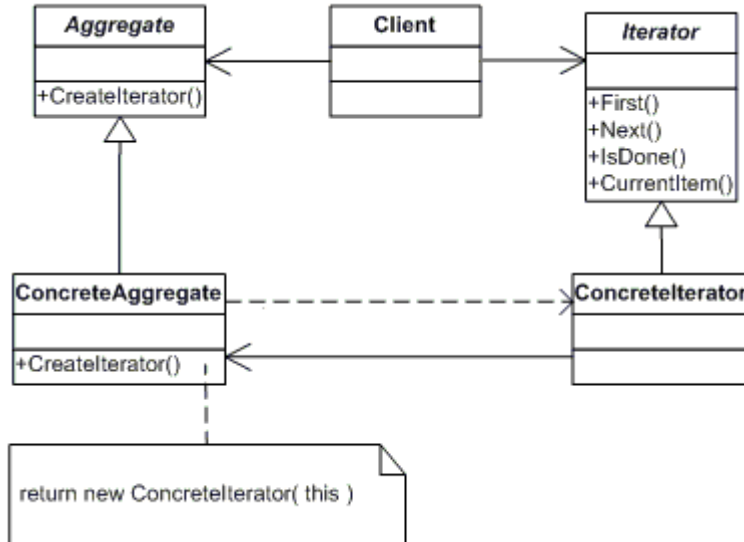
using System;
using System.Collections;
namespace GOF.Interpreter.Structural
{
    // MainApp test application
    class MainApp
    {
        static void Main()
        {
            Context context = new Context();
            // Usually a tree
            ArrayList list = new ArrayList();
            // 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()");
    }
}
}
```

输出

```
Called Terminal.Interpret()
Called Nonterminal.Interpret()
Called Terminal.Interpret()
Called Terminal.Interpret()
```

2.3.4. Iterator（迭代器模式）



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。

场景：

- 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)。

重构成本：

中。

代码：

```

using System;
using System.Collections;
namespace GOF.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();

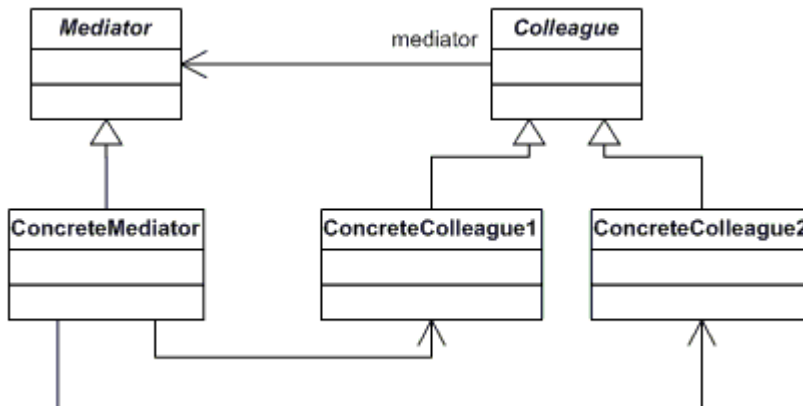
    public override Iterator CreateIterator()
    {
        return new ConcreteIterator(this);
    }
    // Property
    public int Count
    {
        get { return items.Count; }
    }
    // Indexer
    public object this[int index]
    {
        get { return items[index]; }
        set { items.Insert(index, 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;
    }
}
}
```

输出

```
Iterating over collection:
Item A
Item B
Item C
Item D
```

2.3.5. Mediator (中介者模式)



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。

场景:

- 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。

重构成本:

中。

代码:

```
using System;
using System.Collections;
namespace GOF.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
        }
    }
}
```

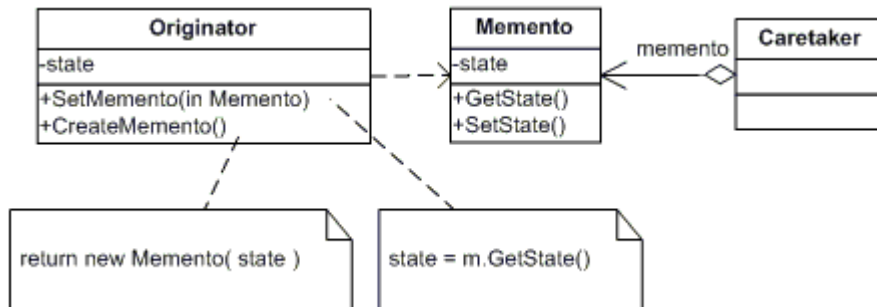
```
        Console.Read();
    }
}
// "Mediator"
abstract class Mediator
{
    public abstract void Send(string message,
        Colleague colleague);
}
// "ConcreteMediator"
class ConcreteMediator : Mediator
{
    private ConcreteColleague1 colleague1;
    private ConcreteColleague2 colleague2;
    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);
    }
}
}
```

输出

```
Colleague2 gets message: How are you?
Colleague1 gets message: Fine, thanks
```

2.3.6. Memento（备忘录模式）



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。

场景：

- 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。

重构成本：

低。

代码：

```
using System;
namespace GOF.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; }
}
}

```

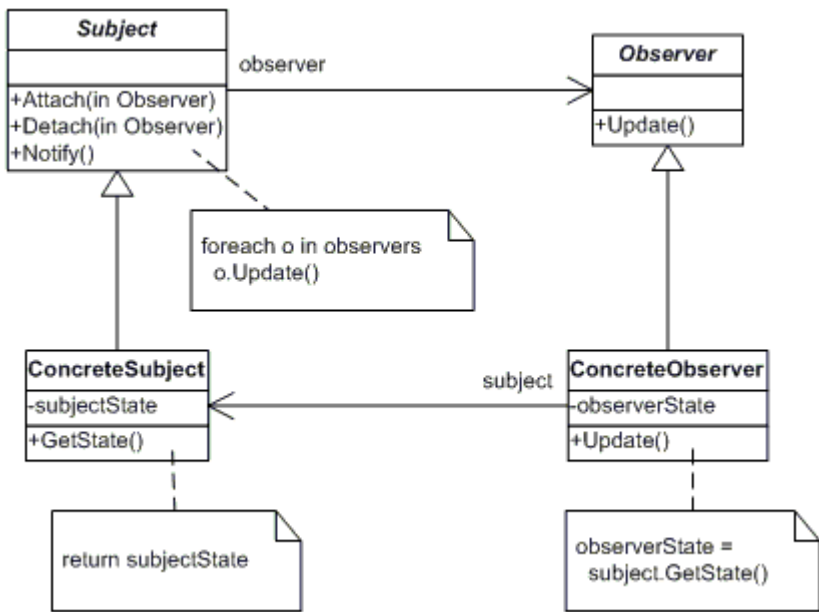
输出

```

State = On
State = Off
Restoring state:
State = On

```

2.3.7. Observer (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。

场景:

- 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。

重构成本：

高。

代码：

```
using System;
using System.Collections;
namespace GOF.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; }
    }
}
```

```

}
}

```

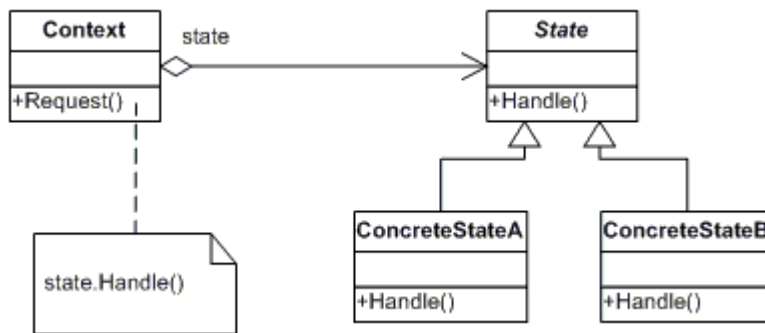
输出

```

Observer X's new state is ABC
Observer Y's new state is ABC
Observer Z's new state is ABC

```

2.3.8. State (状态模式)



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。

场景:

- 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。

重构成本:

中。

代码:

```

using System;
namespace GOF.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);
    }
}
}
```

输出

```
Deposited $500.00 ---
Balance = $500.00
Status = SilverState

Deposited $300.00 ---
Balance = $800.00
Status = SilverState

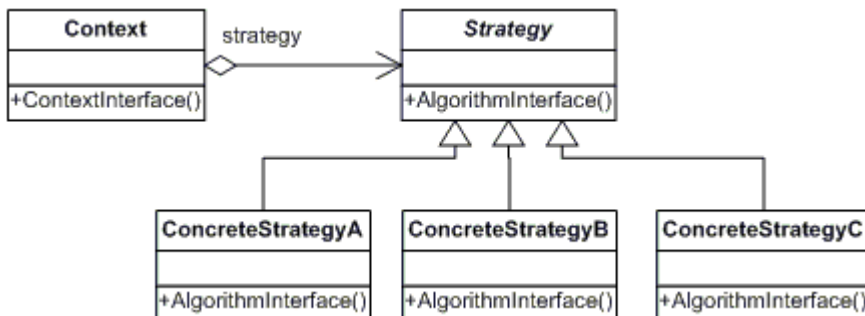
Deposited $550.00 ---
Balance = $1,350.00
Status = GoldState

Interest Paid ---
Balance = $1,417.50
Status = GoldState

Withdrew $2,000.00 ---
Balance = ($582.50)
Status = RedState

No funds available for withdrawal!
Withdrew $1,100.00 ---
Balance = ($582.50)
Status = RedState
```

2.3.9. Strategy（策略模式）



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。

场景：

- 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,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。

重构成本：

中。

代码：

```
using System;
namespace GOF.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();
    }
}
}

```

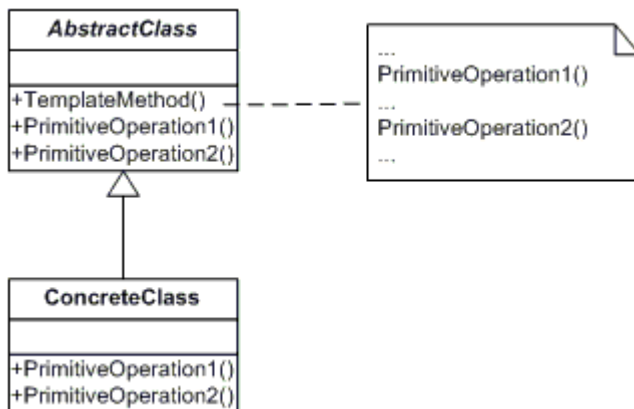
输出

```

Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyB.AlgorithmInterface()
Called ConcreteStrategyC.AlgorithmInterface()

```

2.3.10. TemplateMethod（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。TemplateMethod 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。

场景：

- 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。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。

重构成本:

低。

代码:

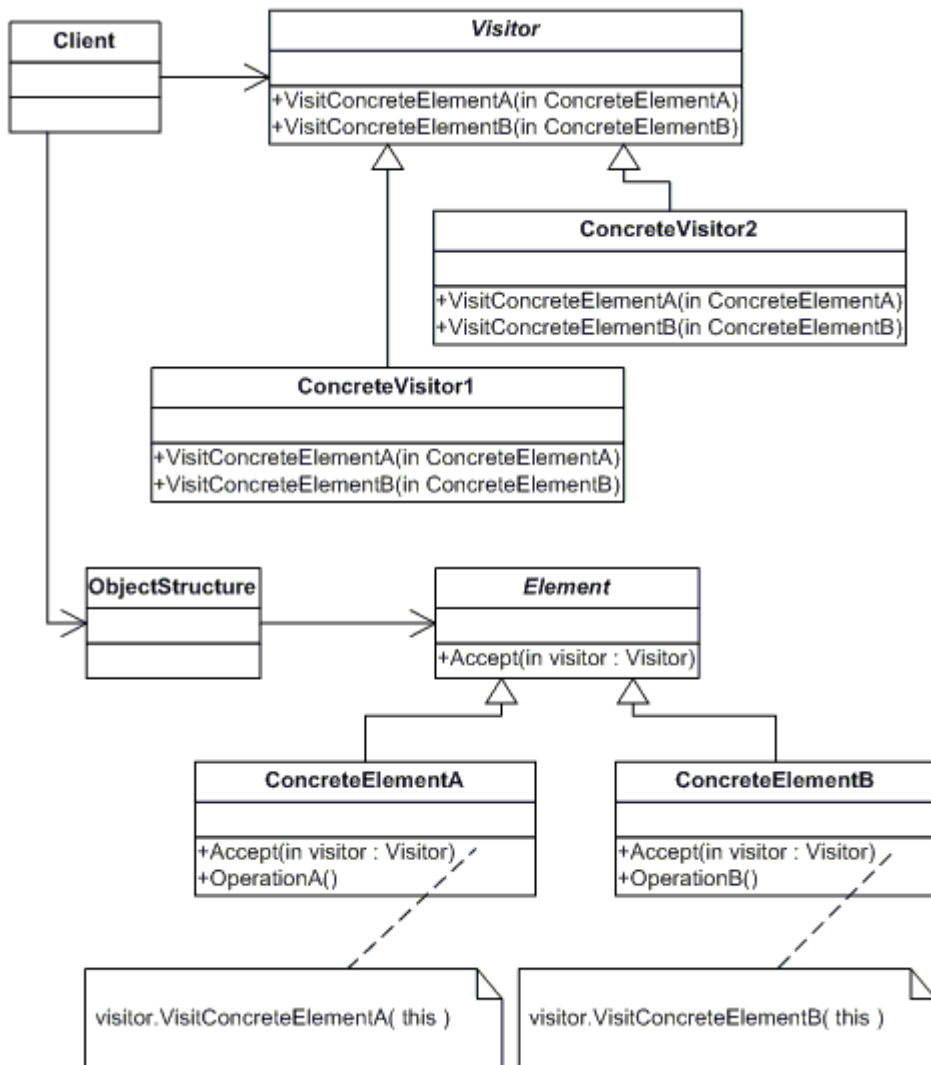
```
using System;
namespace GOF.Template.Structural
{
    // MainApp test application
    class MainApp
    {
        static void Main()
        {
            AbstractClass c;
            c = new ConcreteClassA();
            c.TemplateMethod();
            c = new ConcreteClassB();
            c.TemplateMethod();
            // 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()");  
    }  
  }  
}
```

输出

```
ConcreteClassA.PrimitiveOperation1()  
ConcreteClassA.PrimitiveOperation2()  
  
ConcreteClassB.PrimitiveOperation1()  
ConcreteClassB.PrimitiveOperation2()
```

2.3.11. Visitor（访问者模式）



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。

场景：

- 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。

重构成本：

中。

代码：

```
using System;
using System.Collections;
namespace GOF.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);
        }
    }
}
}
```

输出

```
ConcreteElementA visited by ConcreteVisitor1
ConcreteElementB visited by ConcreteVisitor1
ConcreteElementA visited by ConcreteVisitor2
ConcreteElementB visited by ConcreteVisitor2
```