GoF 设计模式

创建型

- Abstract Factory（抽象工厂模式）
- Builder（生成器模式）
- Factory Method（工厂模式）
- Prototype（原型模式）
- Singleton（单件模式）

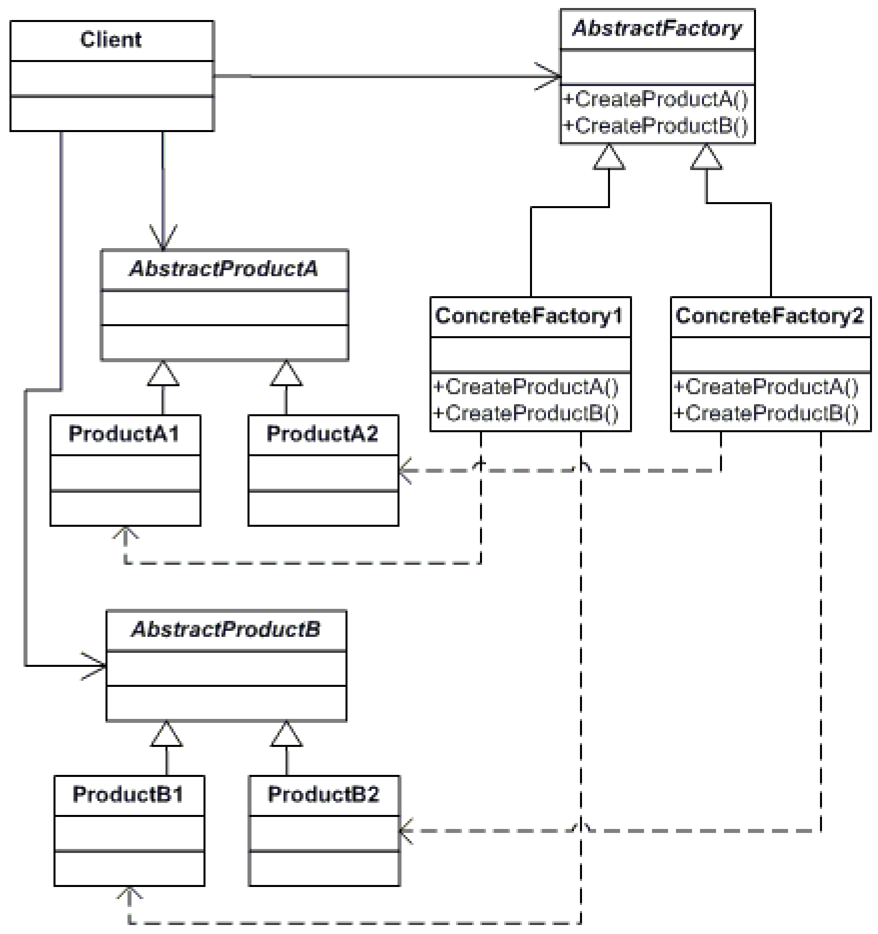
结构型

- Adapter（适配器模式）
- Bridge（桥接模式）
- Composite（组合模式）
- Decorator（装饰模式
- Facade（外观模式）
- Flyweight（享元模式）
- Proxy（代理模式）

行为型

- Chain of Responsibility（职责链模式）
- Command（命令模式）
- Interpreter（解释器模式）
- Iteartor（迭代器模式）
- Mediator（中介者模式）
- Memento（备忘录模式）
- Observer（观察者模式）
- State（状态模式）
- Strategy（策略模式）
- TemplateMethod（模板方法模式）
- Visitor（访问者模式）

Abstract Factory



```
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
```

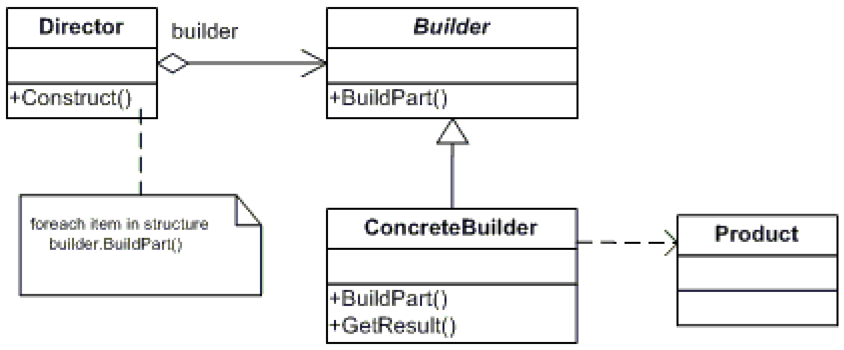
```
                Console.Read();
            }
        }
        // "AbstractFactory"
        abstract class AbstractFactory
        {
            public abstract AbstractProductA CreateProductA();
            public abstract AbstractProductB CreateProductB();
        }
        // "ConcreteFactory1"
        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);
        }
    }
}
```

输出

```
ProductB1 interacts with ProductA1
ProductB2 interacts with ProductA2
```

Builder



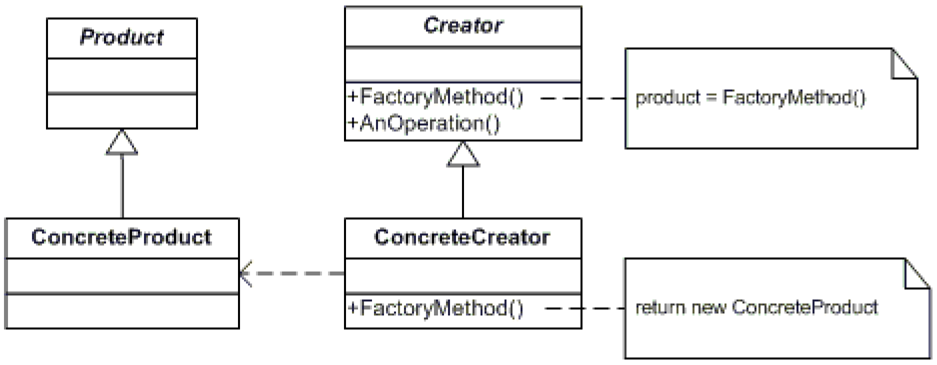
```
using System;
using System.Collections;
namespace DoFactory.GangOfFour.Builder.Structural
{
    // MainApp test application
    public class MainApp
    {
        public static void Main()
        {
            // Create director and builders
            Director director = new Director();
            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);
		}
	}
}
```

输出

```
Product Parts -------
PartA
PartB

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

Factory Method



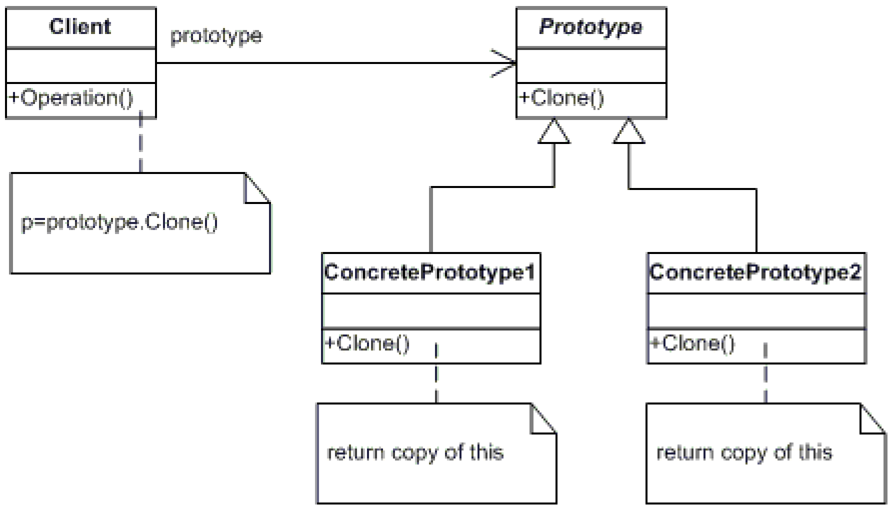
```
using System;
using System.Collections;
namespace DoFactory.GangOfFour.Factory.Structural
{
	// MainApp test application
	class MainApp
	{
		static void Main()
		{
			// An array of creators
			Creator[] creators = new Creator[2];
			creators[0] = new ConcreteCreatorA();
			creators[1] = new ConcreteCreatorB();
			// 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();
                }
        }
}
```

输出

```
Created ConcreteProductA
Created ConcreteProductB
```

Prototype



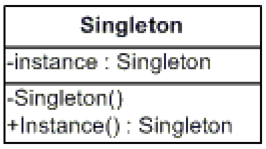
```
using System;
namespace DoFactory.GangOfFour.Prototype.Structural
{
        // MainApp test application
        class MainApp
        {
                static void Main()
                {
                        // Create two instances and clone each
                        ConcretePrototype1 p1 = 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();
            }
        }
    }
```

输出

```
Cloned: I
Cloned: II
```

Singleton

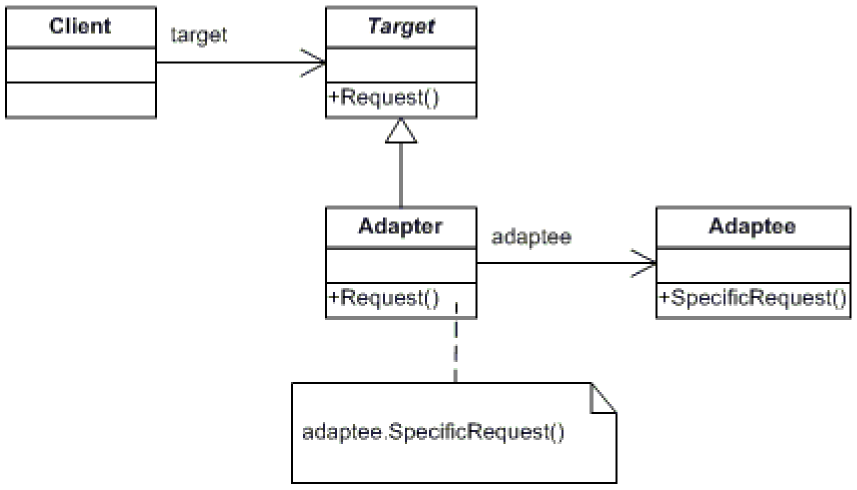


```
using System;
namespace DoFactory.GangOfFour.Singleton.Structural
{
    // MainApp test application
    class MainApp
    {
        static void Main()
        {
            // Constructor is protected -- cannot use new
            Singleton s1 = Singleton.Instance();
            Singleton s2 = Singleton.Instance();
            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;
        }
    }
}
```

输出

```
Objects are the same instance
```

Adapter



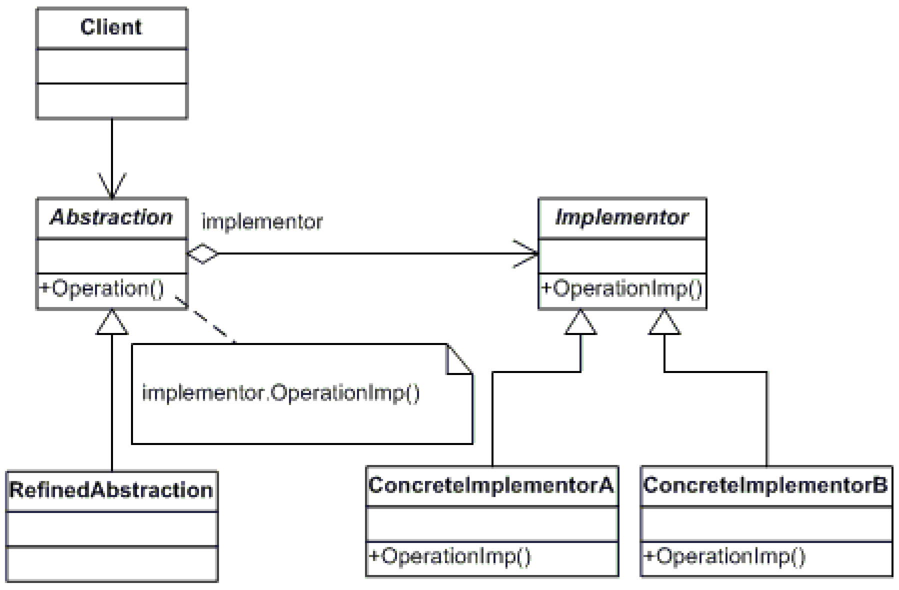
```
using System;
namespace DoFactory.GangOfFour.Adapter.Structural
{
    // Mainapp test application
    class MainApp
    {
        static void Main()
        {
            // Create adapter and place a request
            Target target = new Adapter();
            target.Request();
            // 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()");
            }
        }
    }
```

输出

```
Called SpecificRequest()
```

Bridge



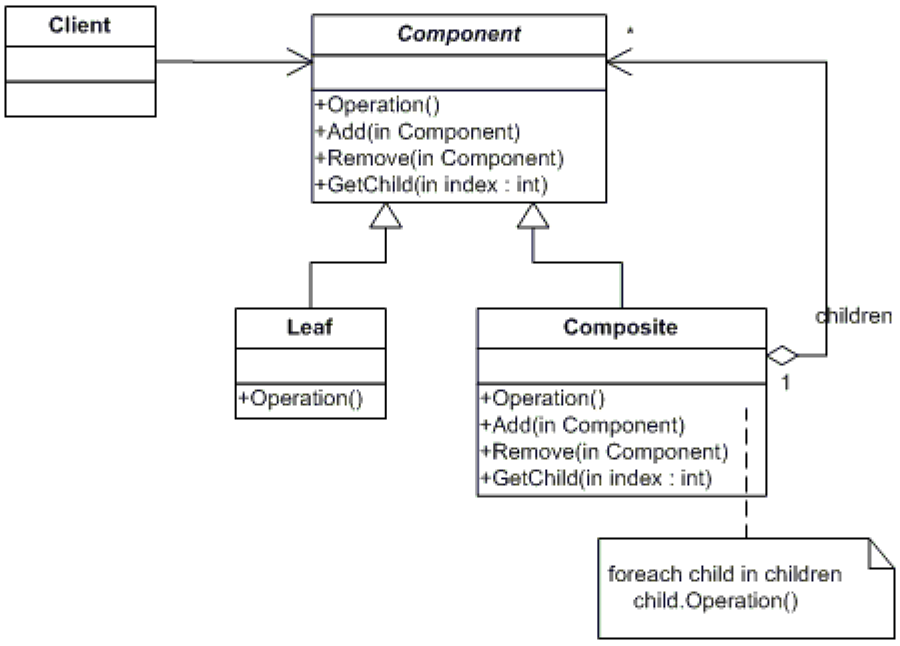
```
using System;
namespace DoFactory.GangOfFour.Bridge.Structural
{
        // MainApp test application
        class MainApp
        {
                static void Main()
                {
                        Abstraction ab = new RefinedAbstraction();
                        // Set implementation 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");
            }
        }
    }
```

输出

```
ConcreteImplementorA Operation
ConcreteImplementorB Operation
```

Composite



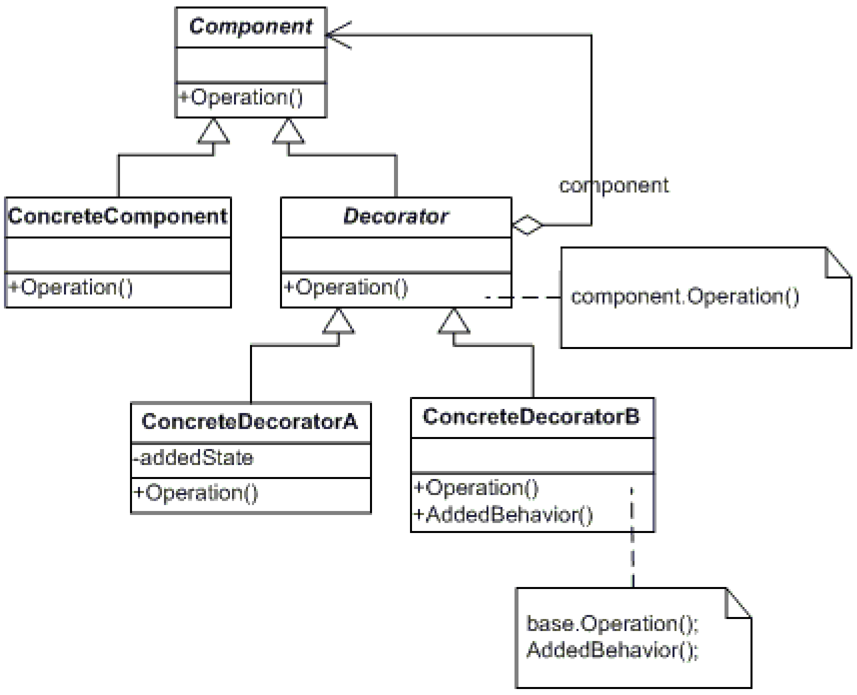
```
using System;
using System.Collections;
namespace DoFactory.GangOfFour.Composite.Structural
{
        // MainApp test application
        class MainApp
        {
                static void Main()
                {
                        // Create a tree structure
                        Composite 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);
                }
        }
}
```

输出

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

Decorator



```
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
```

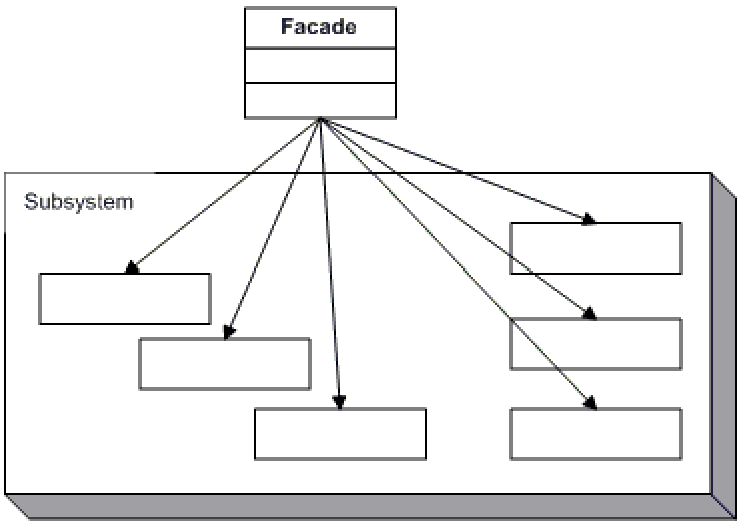
```
{
        public abstract void Operation();
}
// "ConcreteComponent"
class ConcreteComponent : Component
{
        public override void Operation()
        {
                Console.WriteLine("ConcreteComponent.Operation()");
        }
}
// "Decorator"
abstract class Decorator : Component
{
        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()
                {
                }
        }
}
```

输出

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

Facade

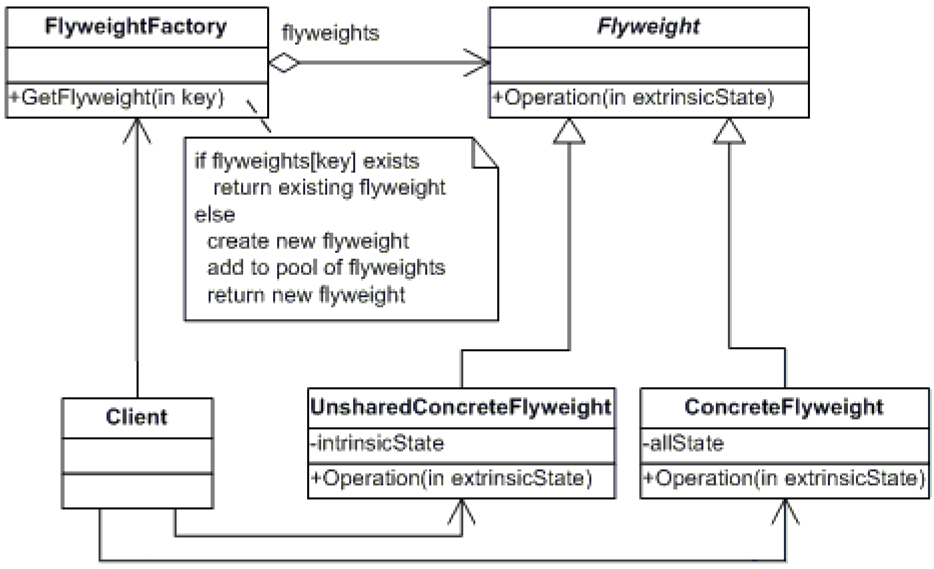


```
using System;
namespace DoFactory.GangOfFour.Facade.Structural
{
        // Mainapp test application
        class MainApp
        {
                public static void Main()
                {
                        Facade facade = new Facade();
                        facade.MethodA();
                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();
            }
        }
    }
```

输出

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

MethodB() ----
SubSystemTwo Method
SubSystemThree Method
```

Flyweight


FlyweightFactory
+GetFlyweight(in key)
flyweights
Flyweight
+Operation(in extrinsicState)
if flyweights[key] exists
return existing flyweight
else
create new flyweight
add to pool of flyweights
return new flyweight
Client
UnsharedConcreteFlyweight
-intrinsicState
+Operation(in extrinsicState)
ConcreteFlyweight
-allState
+Operation(in extrinsicState)

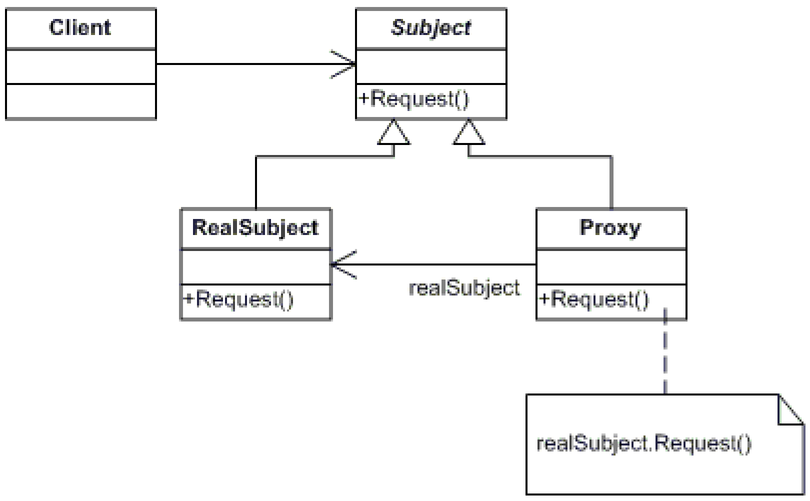
```
using System;
using System.Collections;
namespace DoFactory.GangOfFour.Flyweight.Structural
{
        // MainApp test application
        class MainApp
        {
                static void Main()
                {
                        // Arbitrary extrinsic state
                        int extrinsicstate = 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);
                }
        }
}
```

输出

```
ConcreteFlyweight: 21
ConcreteFlyweight: 20
```

```
ConcreteFlyweight: 19
UnsharedConcreteFlyweight: 18
```

Proxy

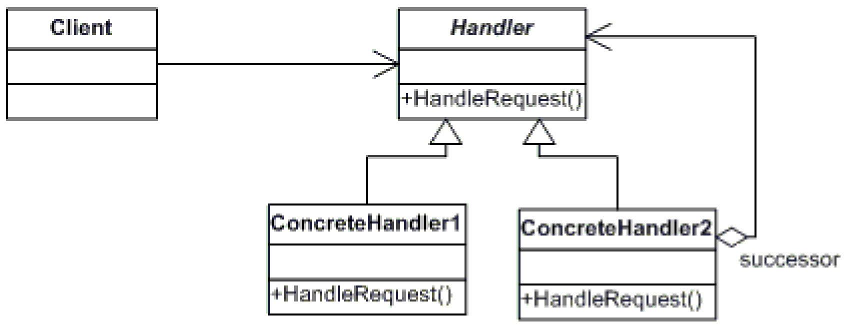


```
using System;
namespace DoFactory.GangOfFour.Proxy.Structural
{
    // MainApp test application
    class MainApp
    {
        static void Main()
        {
            // Create proxy and request a service
            Proxy proxy = new Proxy();
            proxy.Request();
            // Wait for 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();
        }
    }
}
```

输出

```
Called RealSubject.Request()
```

Chain of Responsibility

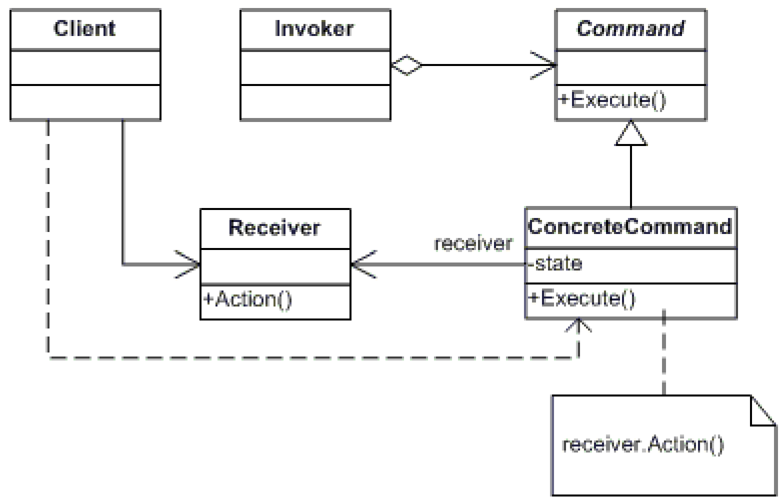


```
using System;
namespace DoFactory.GangOfFour.Chain.Structural
{
    // MainApp test application
    class MainApp
    {
        static void Main()
        {
            // Setup Chain of Responsibility
            Handler h1 = new ConcreteHandler1();
            Handler h2 = new ConcreteHandler2();
            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);
            }
        }
    }
}
```

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
```

Command

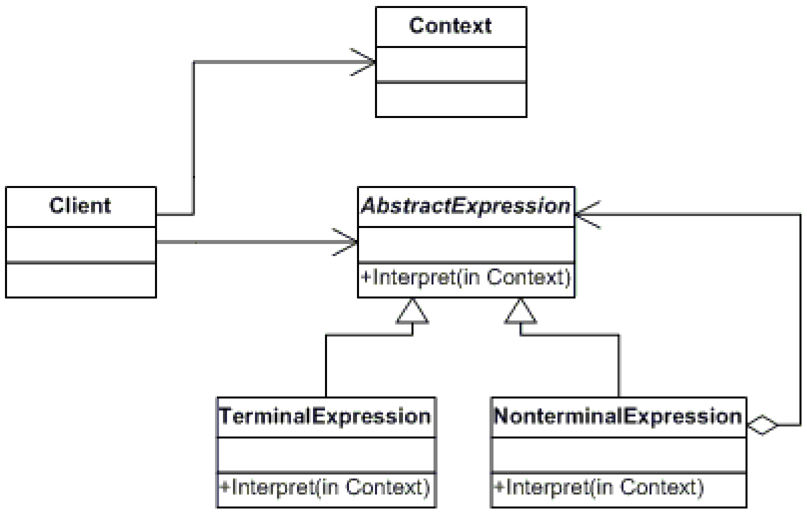


```
using System;
namespace DoFactory.GangOfFour.Command.Structural
{
        // MainApp test applicatio
        class MainApp
        {
                static void Main()
                {
                        // Create receiver, command, and invoker
                        Receiver receiver = new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

输出

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

Interpreter

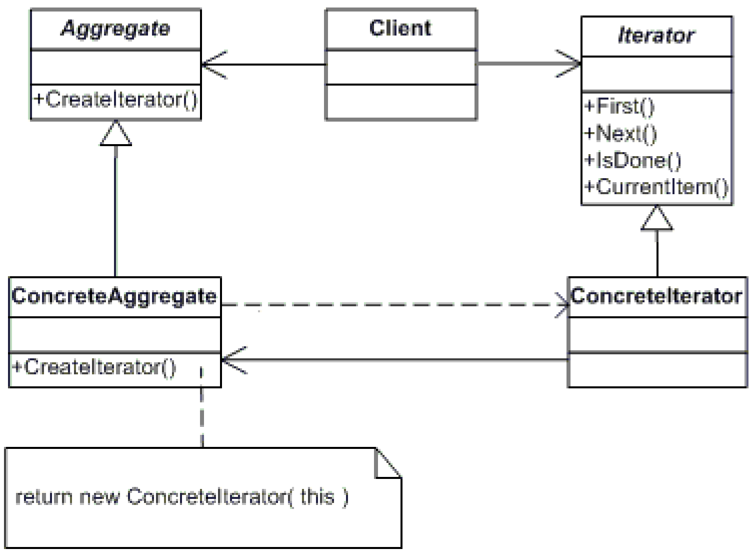


```
using System;
using System.Collections;
namespace DoFactory.GangOfFour.Interpreter.Structural
{
        // MainApp test application
        class MainApp
        {
                static void Main()
                {
                        Context context = new Context();
                        // 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()");
                }
        }
}
```

输出

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

Iterator



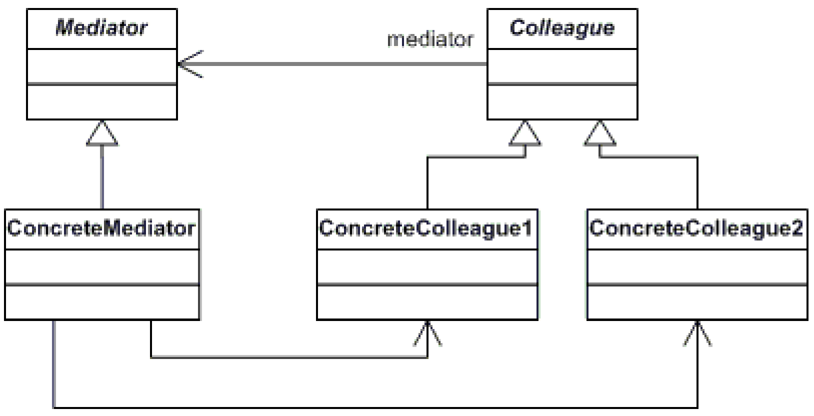
```
using System;
using System.Collections;
namespace DoFactory.GangOfFour.Iterator.Structural
{
        // MainApp test application
        class MainApp
        {
                static void Main()
                {
                        ConcreteAggregate a = new ConcreteAggregate();
                        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;
        }
    }
}
```

输出

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

Mediator

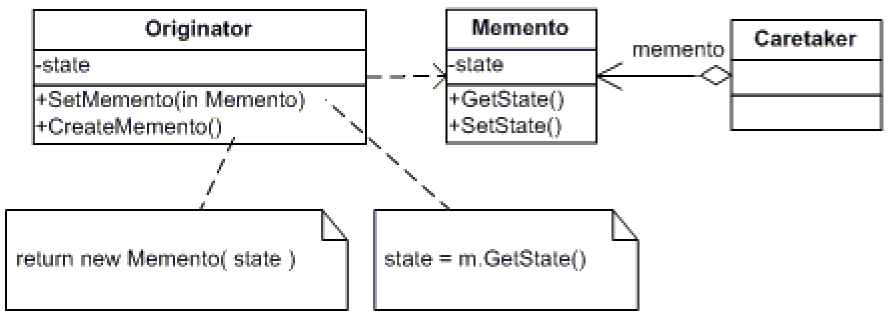


```
using System;
using System.Collections;
namespace DoFactory.GangOfFour.Mediator.Structural
{
	// Mainapp test application
	class MainApp
	{
		static void Main()
		{
			ConcreteMediator m = new ConcreteMediator();
			ConcreteColleague1 c1 = new ConcreteColleague1(m);
			ConcreteColleague2 c2 = new 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);
        }
    }
}
```

输出

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

Memento

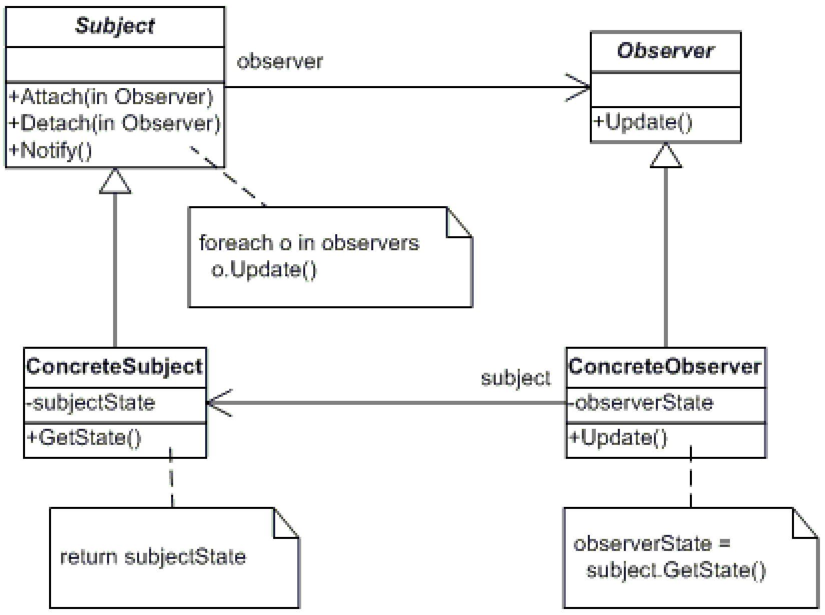


```
using System;
namespace DoFactory.GangOfFour.Memento.Structural
{
        // MainApp test application
        class MainApp
        {
                static void Main()
                {
                        Originator o = new Originator();
                        o.State = "On";
                        // 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; }
                }
        }
}
```

输出

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

Observer

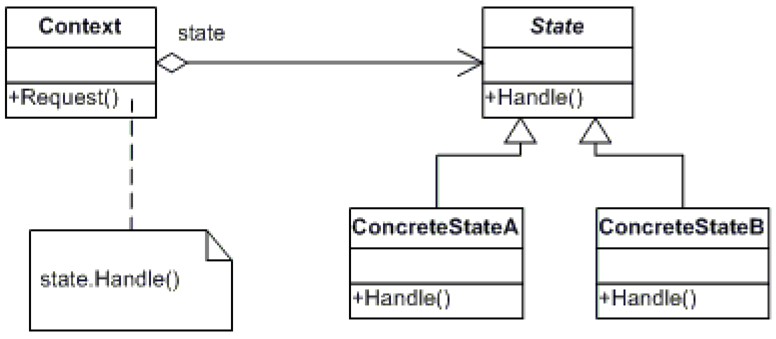


```
using System;
using System.Collections;
namespace DoFactory.GangOfFour.Observer.Structural
{
    // MainApp test application
    class MainApp
    {
        static void Main()
        {
            // Configure Observer pattern
            ConcreteSubject s = new ConcreteSubject();
            s.Attach(new 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; }
        }
    }
}
```

输出

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

State



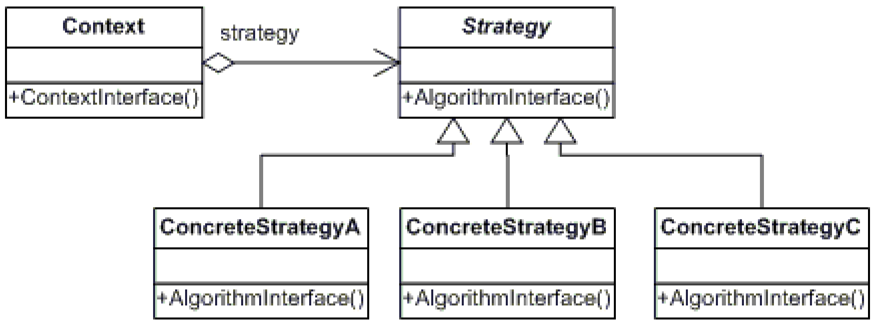
```
using System;
namespace DoFactory.GangOfFour.State.RealWorld
{
    // MainApp test application
    class MainApp
    {
        static void Main()
        {
            // Open a new account
            Account account = new Account("Jim Johnson");
            // Apply financial transactions
            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);
            }
        }
}
```

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
```

Strategy

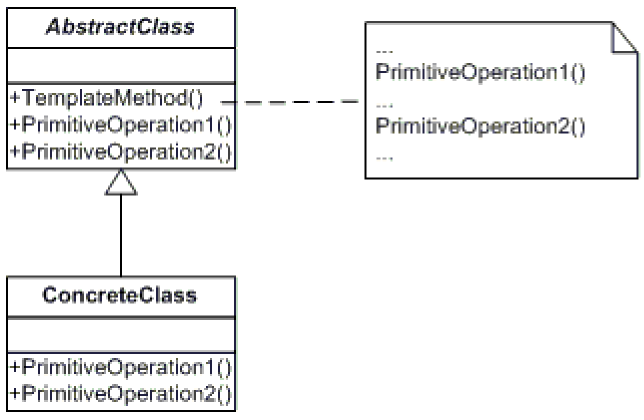


```
using System;
namespace DoFactory.GangOfFour.Strategy.Structural
{
        // MainApp test application
        class MainApp
        {
                static void Main()
                {
                        Context context;
                        // Three contexts following different strategies
                        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();
        }
    }
}
```

输出

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

Template Method

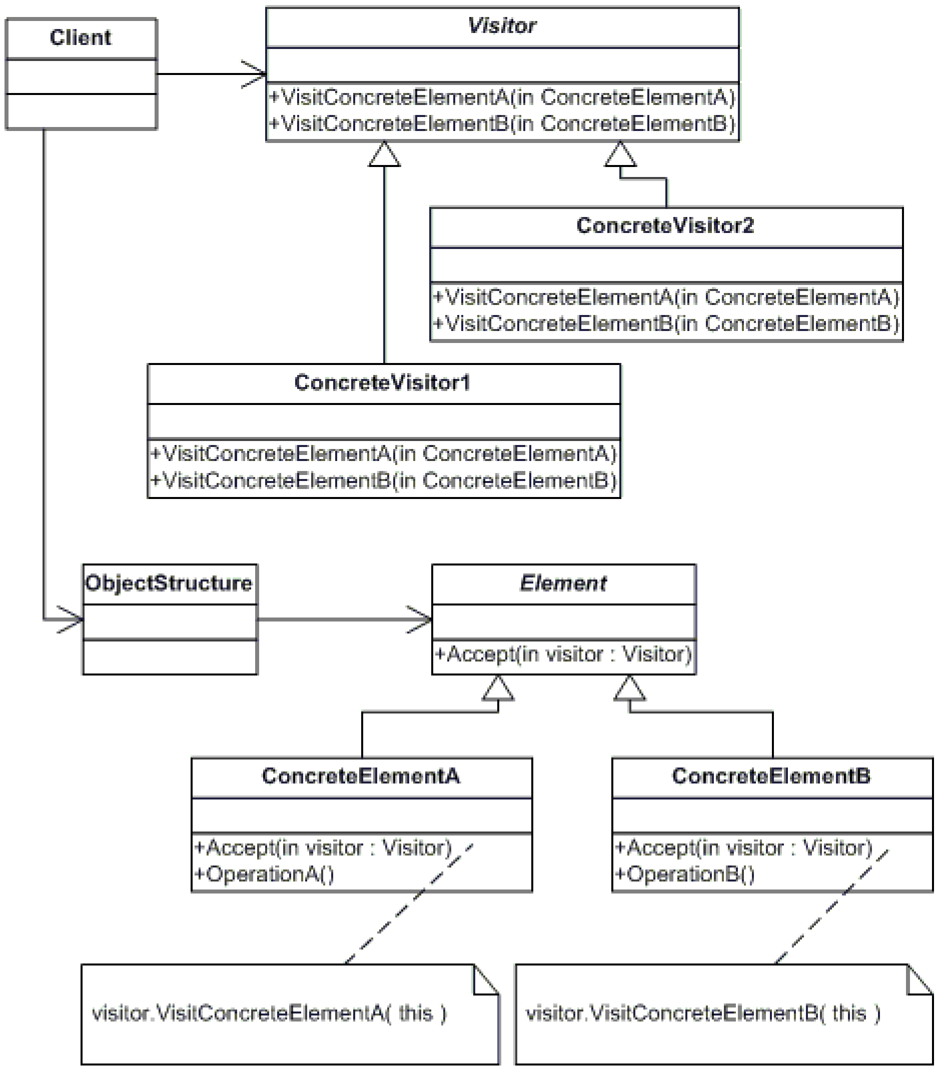


```
using System;
namespace DoFactory.GangOfFour.Template.Structural
{
    // MainApp test application
    class MainApp
    {
        static void Main()
        {
            AbstractClass c;
            c = new ConcreteClassA();
            c.TemplateMethod();
            c = new ConcreteClassB();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

输出

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

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

Visitor


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this)
visitor.VisitConcreteElementB(this)

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

输出

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