设计模式详细说明及代码

制作者：ITtray

设计模式类型

创建型、结构型、行为型

创建型

抽象工厂模式、生成器模式、工厂模式、单件模式、原型模式。

结构型

适配器模式、桥接模式、组合模式、装饰模式、外观模式（门面模式）、
享元模式、代理模式。

行为型

职责链模式、命令模式、解释器模式、迭代器模式、中介者模式、备忘录模式、
观察者模式、状态模式、策略模式、模板方法模式、访问者模式。

设计模式索引

各种设计模式名称、意图、适用环境、代码、调用、输出结果

1、抽象工厂模式

```
/** 名称：抽象工厂模式
 * 意图：提供一个创建一系列相关的或相互依赖对象的接口，而无需指定它们具体的类。
 * 适用环境：
 * 1、一个系统要独立它的产品的创建、组合和表示时
 * 2、一个系统要由多个产品系列中的一个来配置时。
 * 3、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。
 * 4、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。
 **/
abstract class AbstraactFactory
```

```
{
    public abstract AbstractProductA CreateProductA();
    public abstract AbstractProductB CreateProductB();
}
class ConcreteFactory1 : AbstraactFactory
{
    public override AbstractProductA CreateProductA()
    {
        return new ProudctA1();
    }
    public override AbstractProductB CreateProductB()
    {
        return new ProductB1();
    }
}
class ConcreteFactory2 : AbstraactFactory
{
    public override AbstractProductA CreateProductA()
    {
        return new ProductA2();
    }
    public override AbstractProductB CreateProductB()
    {
        return new ProductB2();
    }
}
abstract class AbstractProductA
{
}
abstract class AbstractProductB
{
    public abstract void Interact(AbstractProductA a);
}
class ProudctA1 : AbstractProductA
{
}
class ProductB1 : AbstractProductB
{
    public override void Interact(AbstractProductA a)
    {
        Console.WriteLine(this.GetType().Name +
            " interacts with " + a.GetType().Name);
    }
}
```

```
class ProductA2 : AbstractProductA
{
}
class ProductB2 : AbstractProductB
{
    public override void Interact(AbstractProductA a)
    {
        Console.WriteLine(this.GetType().Name +
            " interacts with " + a.GetType().Name);
    }
}
class Client
{
    private AbstractProductA _abstractProductA;
    private AbstractProductB _abstractProductB;
    public Client(AbstraactFactory factory)
    {
        _abstractProductB = factory.CreateProductB();
        _abstractProductA = factory.CreateProductA();
    }
    public void Run()
    {
        _abstractProductB.Interact(_abstractProductA);
    }
}
```

调用：

```
AbstraactFactory factory1 = new ConcreteFactory1();
Client client1 = new Client(factory1);
client1.Run();
AbstraactFactory factory2 = new ConcreteFactory2();
Client client2 = new Client(factory2);
client2.Run();
```

输出：

```
ProductB1 interacts with ProudctA1
ProductB2 interacts with ProductA2
```

2、生成器模式

```
/** 名称：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。
 * 适用环境：
 * 1、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。
 * 2、当构造过程必须允许被构造的对象有不同的表示时。
 **/
```

```
class Director
{
    public void Construct(Buidler builder)
    {
        builder.BuildPartA();
        builder.BuildPartB();
    }
}
abstract class Buidler
{
    public abstract void BuildPartA();
    public abstract void BuildPartB();
    public abstract Product GetResult();
}
class ConcreateBuilder1 : Buidler
{
    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;
    }
}
class ConcreateBuilder2 : Buidler
{
    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;
        }
    }
    class Product
    {
        private List<string> _parts = new List<string>();
        public void Add(string part)
        {
            _parts.Add(part);
        }
        public void Show()
        {
            Console.WriteLine("\nProduct parts 产品类型（汽车、摩托车）");
            foreach (string part in _parts)
            {
                Console.WriteLine(part);
            }
        }
    }
```

调用：

```
            Director director = new Director();
            Buidler b1 = new ConcreateBuilder1();
            Buidler b2 = new ConcreateBuilder2();
            director.Construct(b1);
            Product p1 = b1.GetResult();
            p1.Show();

            director.Construct(b2);
            Product p2 = b2.GetResult();
            p2.Show();
```

输出：

```
Product parts 产品类型（汽车、摩托车）
PartA
PartB

Product parts 产品类型（汽车、摩托车）
PartX
PartY
```

3、工厂模式

```
/** 名称：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。
* 适用环境：
* 1、当一个类不知道它所必须创建的对象的类的时候
* 2、当一个类希望由它的子类来指定它所创建的对象的时候。
* 3、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。
**/
abstract class ProductFM
{
}
class ConcreteProductA : ProductFM
{
}
class ConcreteProductB : ProductFM
{
}
abstract class CreatorFM
{
    public abstract ProductFM FactoryMethod();
}
class ConcreteCreatorA : CreatorFM
{
    public override ProductFM FactoryMethod()
    {
        return new ConcreteProductA();
    }
}
class ConcreteCreatorB : CreatorFM
{
    public override ProductFM FactoryMethod()
    {
        return new ConcreteProductB();
    }
}
```

调用：

```
        CreatorFM[] creators = new CreatorFM[2];
        creators[0] = new ConcreteCreatorA();
        creators[1] = new ConcreteCreatorB();
```

```
            foreach (CreatorFM creator in creators)
            {
                ProductFM product = creator.FactoryMethod();
                Console.WriteLine("Created {0}",
                    product.GetType().Name);
            }
```

输出：

Created ConcreteProductA

Created ConcreteProductB

4、原型模式

```
    /** 名称：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。
    * 适用环境：
    * 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；
    * 以及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。
    **/
    abstract class Prototype
    {
        private string _id;

        public Prototype(string id)
        {
            this._id = id;
        }
        public string Id
        {
            get { return _id; }
        }
        public abstract Prototype Clone();
    }
    class ConcretePrototype1 : Prototype
    {
        public ConcretePrototype1(string id)
            : base(id)
        {
        }
        public override Prototype Clone()
        {
```

```
            return (Prototype)this.MemberwiseClone();
        }
    }
    class ConcretePrototype2 : Prototype
    {
        public ConcretePrototype2(string id)
            : base(id)
        {
        }
        public override Prototype Clone()
        {
            return (Prototype)this.MemberwiseClone();
        }
    }
```

调用：

```
            ConcretePrototype1 cp1 = new ConcretePrototype1("I");
            ConcretePrototype1 c1 = (ConcretePrototype1)cp1.Clone();
            Console.WriteLine("Cloned:{0}", c1.Id);

            ConcretePrototype2 cp2 = new ConcretePrototype2("II");
            ConcretePrototype2 c2 = (ConcretePrototype2)cp2.Clone();
            Console.WriteLine("Cloned:{0}", c2.Id);
```

输出：

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

5、单件模式

```
/** 名称：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。
  * 适用环境：
  * 1、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。
  * 2、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。
  **/
  class Singleton
  {
      private static Singleton _instance;
      protected Singleton()
      {
      }
      public static Singleton Instance()
      {
```

```
            // Uses lazy initialization.
            // Note: this is not thread safe
            if (_instance == null)
            {
                _instance = new Singleton();
            }
            return _instance;
        }
    }
```

调用：

```
            Singleton s1 = Singleton.Instance();
            Singleton s2 = Singleton.Instance();

            if (s1 == s2)
            {
                Console.WriteLine("Objects are same instance");
            }
```

输出：

Objects are same instance

6、适配器模式

```
/*
 * 名称：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。
 * 适用环境：
 * 1、你想使用一个已经存在的类，而它的接口不符合你的需求。
 * 2、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。
 * 3、（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。
 * */
class Target
{
    public virtual void Request()
    {
        Console.WriteLine("Called Target Request()");
    }
}
class Adapter : Target
{
```

```
        private Adaptee _adaptee = new Adaptee();
        public override void Request()
        {
            _adaptee.SpecificRequest();
        }
    }
    class Adaptee
    {
        public void SpecificRequest()
        {
            Console.WriteLine("Called SpecificRequest()");
        }
    }
```

调用：

```
            Target target = new Adapter();
            target.Request();
```

输出：

Called SpecificRequest()

7、桥接模式

```
/** 名称：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。
 * 适用环境：
 * 1、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。
 * 2、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。
 * 3、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。
 * 4、（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以见
 * 的。
 * 5、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”（netsted generalizations)。
 * 6、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。
 **/
class Abstraction
{
    protected Implementor implementor;

    public Implementor Implementor
    {
        set {implementor=value;}
    }
```

```
        public virtual void Operation()
        {
            implementor.Operation();
        }
}
abstract class Implementor
{
        public abstract void Operation();
}
class RefinedAbstraction : Abstraction
{
        public override void Operation()
        {
            implementor.Operation();
        }
}
class ConcreteImplementorA : Implementor
{
        public override void Operation()
        {
            Console.WriteLine("ConcreteImplementorA Operation");
        }
}
class ConcreteImplementorB : Implementor
{
        public override void Operation()
        {
            Console.WriteLine("ConcreteImplementorB Operation");
        }
}
```

调用：

```
            //set
            Abstraction ab = new RefinedAbstraction();
            ab.Implementor = new ConcreteImplementorA();
            ab.Operation();
            //change
            ab.Implementor = new ConcreteImplementorB();
            ab.Operation();
```

输出：

ConcreteImplementorA Operation

ConcreteImplementorB Operation

8、组合模式

```
/* * 名称：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。
 * 适用环境：
 * 1、你想表示对象的部分-整体层次结构。
 * 2、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。
 * */
abstract class Component
{
    protected string name;

    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);
}
class Composite : Component
{
    private List<Component> _children = new List<Component>();
    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);
        foreach (Component component in _children)
        {
            component.Display(depth + 2);

        }
```

```
        }
    }
    class Leaf : Component
    {
        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);
        }
    }
```

调用：

```
            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);
            root.Display(1);
```

输出：

```
-root
---Leaf A
---Leaf B
---Composite X
```

-----Leaf XA
-----Leaf XB
---Leaf C

9、装饰模式

```
/** 名称：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。
 * 适用环境：
 *1、在不影响其他对象的情况下，以动态、透明的试给单个对象添加职责。
 *2、处理那些可以撤消的职责。
 *3、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。别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。
 **/
abstract class DComponent
{
    public abstract void Operation();
}
class ConcreteComponent : DComponent
{
    public override void Operation()
    {
        Console.WriteLine("ConcreteComponent.Operation()");
    }
}
abstract class Decorator : DComponent
{
    protected DComponent component;
    public void SetComponent(DComponent component)
    {
        this.component = component;
    }
    public override void Operation()
    {
        if (component != null)
        {
            component.Operation();
        }
    }
}
class ConcreteDecoratorA : Decorator
```

```
{
    public override void Operation()
    {
        base.Operation();
        Console.WriteLine("ConcreteDecoratorA.Operation()");
    }
}
class ConcreteDecoratorB : Decorator
{
    public override void Operation()
    {
        base.Operation();
        AddedBehavior();
        Console.WriteLine("ConcreteDecoratorB.Operation()");
    }
    void AddedBehavior()
    {
    }
}
```

调用：

```
        ConcreteComponent c = new ConcreteComponent();
        ConcreteDecoratorA d1 = new ConcreteDecoratorA();
        ConcreteDecoratorB d2 = new ConcreteDecoratorB();
        d1.SetComponent(c);
        d2.SetComponent(d1);
        d2.Operation();
```

输出：

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

10、外观模式、门面模式

```
/** 名称：外观模式、门面模式
 * 意图：为子系统中的一组接口提供一个一致的界面，Facade模式定义一个高层接口，这个接
 * 口使得这一子系更加容易使用。
 * 适用环境：
 * 1、当你要为一个复杂子系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。Facde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。
```

```
 * 2、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。
 * 3、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。
 * */
class SubSystemOne
{
    public void MethodOne()
    {
        Console.WriteLine(" SubSystemOne Method");
    }
}
class SubSystemTwo
{
    public void MethodTwo()
    {
        Console.WriteLine(" SubSystemTwo Method");
    }
}
class SubSystemThree
{
    public void MethodThree()
    {
        Console.WriteLine(" SubSystemThree Method");
    }
}
class SubSystemFour
{
    public void MethodFour()
    {
        Console.WriteLine(" SubSystemFour Method");
    }
}
class Facade
{
    private SubSystemOne _one;
    private SubSystemTwo _two;
    private SubSystemThree _three;
    private 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();
        }
}
```

调用：

```
            Facade facade = new Facade();

            facade.MethodA();
            facade.MethodB();
```

输出：

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

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

11、享元模式

```
/** 名称：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
  * 适用环境：
  * 1、一个应用程序使用了大量的对象。
  * 2、完全由于使用大量的对象，造成很大的存储开销。
  * 3、对象的大多数状态都可变为外部状态。
  * 4、如果删除对象的外部状，那么可以用相对较少的共享对象取代很多组对象。
```

```
 * 5、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。
 * */
class FlyweightFactory
{
    private Hashtable flyweights = new Hashtable();
    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];
    }
}
abstract class Flyweight
{
    public abstract void Operation(int extrinsicstate);
}
class ConcreteFlyweight:Flyweight
{
    public override void Operation(int extrinsicstate)
    {
        Console.WriteLine("ConcreteFlyweight: " + extrinsicstate);
    }
}
class UnsharedConcreteFlyweight:Flyweight
{
    public override void Operation(int extrinsicstate)
    {
        Console.WriteLine("UnsharedConcreteFlyweight:" +
            extrinsicstate);
    }
}
```

调用：

```
        int extrinsicstate = 22;
        FlyweightFactory factory = new FlyweightFactory();
        Flyweight fx = factory.GetFlyweight("X");
        fx.Operation(--extrinsicstate);

        Flyweight fy = factory.GetFlyweight("Y");
        fy.Operation(--extrinsicstate);
```

```
            Flyweight fz = factory.GetFlyweight("Z");
            fz.Operation(--extrinsicstate);
```

输出：

ConcreteFlyweight: 21

ConcreteFlyweight: 20

ConcreteFlyweight: 19

12、代理模式

```
/** 名称：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。
  * 适用环境：
  * 在需要用比较通用和复杂的对象指针代替简单的指针的时候，使用Proxy模式。下面是一些可
  * 以使用Proxy模式常见情况：
  * 1、远程代理为一个对象在不同的地址空间提供局部代表。
  * 2、虎代理根据需要创建开销很大的对象。
  * 3、保护代理控制对原始对象的访问。保护代理用于对象应该有不同的访问权限的时候。
  * 4、智能指引取代了简单的指针，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。
  **/
abstract class Subject
{
    public abstract void Request();
}
class RealSubject : Subject
{
    public override void Request()
    {
        Console.WriteLine("Called RealSubject.Request()");
    }
}
class Proxy : Subject
{
    private RealSubject _realSubject;
    public override void Request()
    {
        if (_realSubject == null)
        {
```

```
            _realSubject = new RealSubject();
        }
        _realSubject.Request();
    }
}
```

调用：

```
        Proxy proxy = new Proxy();
        proxy.Request();
```

输出：

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

13、职责链模式

```
/** 名称：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。
 * 适用环境：
 *1、有多个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。
 *2、你想在不明确指定的接收者的情况下，向多个对象中的一个提交一个请求，。
 *3、可处理一个请求的对象集合应被动态指定。
 **/
abstract class Handler
{
    protected Handler successor;
    public void SetSuccessor(Handler successor)
    {
        this.successor = successor;
    }
    public abstract void HandleRequest(int request);
}
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);
        }
```

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

        }
    }
```

调用：

```
            Handler h1 = new ConcreteHandler1();
            Handler h2 = new ConcreteHandler2();
            Handler h3 = new ConcreteHandler3();
            h1.SetSuccessor(h2);
            h2.SetSuccessor(h3);

            int[] requests = { 2, 5, 14, 22, 18, 3, 27, 20 };
            foreach (int request in requests)
            {
                h1.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
```

14、命令模式

```
/** 名称：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，以及支持可撤消的操作。
 * 适用环境：
 *1、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。
 *2、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
 * 的命令对象送给另一个不同的进程并在那儿实现该请求。
 *3、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。
 *4、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的信息系统中
 * 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
 * 事务以护展系统。
 **/
abstract class Command
{
    public abstract void Execute();
    public abstract void UnExecute();
}
class CalculatorCommand : Command
{
    private char _operator;
    private int _operand;
    private Calculator _calculator;
```

```
        public CalculatorCommand(Calculator calculator,
            char @operator, int operand)
        {
            this._calculator = calculator;
            this._operator = @operator;
            this._operand = operand;
        }

        public char Operator
        {
            set { _operator = value; }
        }

        public int Operand
        {
            set { _operand = value; }
        }

        public override void Execute()
        {
            _calculator.Operation(_operator, _operand);
        }

        public override void UnExecute()
        {
            _calculator.Operation(Undo(_operator), _operand);
        }
        public char Undo(char @operator)
        {
            switch (@operator)
            {
                case '+': return '-';
                case '-': return '+';
                case '*': return '/';
                case '/': return '*';
                default:throw new
                    ArgumentException("@operator");
            }
        }
}
class Calculator
{
        private int _curr = 0;
        public void Operation(char @operator, int operand)
```

```
        {
            switch (@operator)
            {
                case '+': _curr += operand; break;
                case '-': _curr -= operand; break;
                case '*': _curr *= operand; break;
                case '/': _curr /= operand; break;
            }
            Console.WriteLine(
                "Current value={0,3} (following {1} {2})",
                _curr, @operator, operand);
        }
    }
    class User
    {
        private Calculator _calculator = new Calculator();
        private List<Command> _commands = new List<Command>();
        private int _current = 0;

        public void Redo(int levels)
        {
            Console.WriteLine("\n---- Redo {0} levels ", levels);
            for (int i = 0; i < levels; i++)
            {
                if (_current < _commands.Count - 1)
                {
                    Command command = _commands[_current++];
                    command.Execute();
                }
            }
        }
        public void Undo(int levels)
        {
            Console.WriteLine("\n---- Undo {0} levels ", levels);
            for (int i = 0; i < levels; i++)
            {
                if (_current > 0)
                {
                    Command command = _commands[--_current] as Command;
                    command.UnExecute();
                }
            }
        }
        public void Compute(char @operator, int operand)
```

```
        {
            Command command = new CalculatorCommand(
                _calculator, @operator, operand);
            command.Execute();

            _commands.Add(command);
            _current++;
        }
    }
```

调用：

```
            User user = new User();
            user.Compute('+', 100);
            user.Compute('-', 50);
            user.Compute('*', 10);
            user.Compute('/', 20);

            user.Undo(4);
            user.Redo(3);
```

输出：

```
Current value=100 (following + 100)
Current value= 50 (following - 50)
Current value=500 (following * 10)
Current value= 25 (following / 20)

---- Undo 4 levels
Current value=500 (following * 20)
Current value= 50 (following / 10)
Current value=100 (following + 50)
Current value=    0 (following - 100)

---- Redo 3 levels
Current value=100 (following + 100)
Current value= 50 (following - 50)
Current value=500 (following * 10)
```

15、解释器模式

```
/** 名称：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。
  * 适用环境：
  * 1、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对时，可使
  * 用解释器模式。而当存在以下情况该模式效果最好：
```

```
 * 该文法法简单对于复杂文法，文法的类层变得宠大庞大而无法管理。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，而是首选
 * 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。
 * */
class Context
{
}
abstract class AbstractExpression
{
    public abstract void Interpret(Context context);
}
class TerminalExpression:AbstractExpression
{
    public override void Interpret(Context context)
    {
        Console.WriteLine("Called Terminal.Interpret()");
    }
}
class NoterminalExpression : AbstractExpression
{
    public override void Interpret(Context context)
    {
        Console.WriteLine("Called Noterminal.Interpret()");
    }
}
```

调用：

```
        Context context = new Context();
        ArrayList list = new ArrayList();

        list.Add(new TerminalExpression());
        list.Add(new NoterminalExpression());
        list.Add(new TerminalExpression());
        list.Add(new TerminalExpression());
        foreach (AbstractExpression exp in list)
        {
            exp.Interpret(context);
        }
```

输出：

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

Called Noterminal.Interpret()
Called Terminal.Interpret()
Called Terminal.Interpret()

16、迭代器模式

```
/*
 * 名称：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。
 * 适用环境：
 * 1、访问一个取合对象的内容而无需暴露它的内部表示。
 * 2、支持对聚合对象的多种遍历。
 * 3、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）。
 * */
abstract class Aggregate
{
    public abstract Iterator CreateIterator();
}
class ConcreteAggregate : Aggregate
{
    private ArrayList _items = new ArrayList();
    public override Iterator CreateIterator()
    {
        return new ConcreteIterator(this);
    }
    public int Count
    {
        get { return _items.Count; }
    }
    public object this[int index]
    {
        get { return _items[index]; }
        set { _items.Insert(index, value); }
    }
}
abstract class Iterator
{
    public abstract object First();
    public abstract object Next();
    public abstract bool IsDone();
    public abstract object CurrentItem();
}
class ConcreteIterator : Iterator
{
```

```
        private ConcreteAggregate _aggregate;
        private int _current = 0;

        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 _current >= _aggregate.Count;
        }
        public override bool IsDone()
        {
            return _current >= _aggregate.Count;
        }
}
```

调用：

```
            ConcreteAggregate a = new ConcreteAggregate();
            a[0] = "Item A";
            a[1] = "Item B";
            a[2] = "Item C";
            a[3] = "Item D";

            ConcreteIterator i = new ConcreteIterator(a);
            Console.WriteLine("Iterating over collection:");

            object item = i.First();
            while (item != null)
            {
                Console.WriteLine(item);
                item = i.Next();
```

```
        }
```

输出：

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

17、中介者模式

```
/** 名称：中介者模式
 * 意图：用一个中介对象封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从
 * 而使其耦合松散，
 * 而且可以独立地改变它们之间的交互。
 * 适用环境：
 *1、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。
 *2、一个对象引用其他很多对象并且直接与这些对象通信，导致难惟复用该对象。
 *3、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。
 **/
abstract class Mediator
{
    public abstract void Send(string messages,
        Colleague colleague);

}
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);
            }
        }
}
abstract class Colleague
{
        protected Mediator mediator;
        public Colleague(Mediator mediator)
        {
            this.mediator = mediator;
        }
}
class ConcreteColleague1 : Colleague
{
        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);
        }
}
class ConcreteColleague2 : Colleague
{
        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);
        }
    }
    调用：
            ConcreteMediator m = new ConcreteMediator();
            ConcreteColleague1 cc1 = new ConcreteColleague1(m);
            ConcreteColleague2 cc2 = new ConcreteColleague2(m);

            m.Colleague1 = cc1;
            m.Colleague2 = cc2;
            cc1.Send("How are You?");
            cc2.Send("Fine, thanks");

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

18、备忘录模式

```
/** 名称：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。
  * 适用环境：
  * 1、必须保存一个对象的在某一个时记刻的（部分）状态，这样以后需要时它才能恢复到先前
  * 的状态。
  * 2、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存坏对象的
  * 封装性。
  **/
 class Originator
 {
     private string _state;
     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;
        }
    }
    class Memento
    {
        private string _state;
        public Memento(string state)
        {
            this._state = state;
        }
        public string State
        {
            get { return _state; }
        }
    }
    class Caretaker
    {
        private Memento _memento;
        public Memento Memento
        {
            set { _memento = value; }
            get { return _memento; }
        }
    }
```

调用：

```
            Originator o = new Originator();
            o.State = "On";
            Caretaker ct = new Caretaker();
            ct.Memento = o.CreateMemento();

            o.State = "Off";

            o.SetMemento(ct.Memento);
```

输出：

State = On

State = Off

Restoring state...
State = On

19、观察者模式

```
/** 名称：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
 * 的对象的都得到通知并被自动更新。
 * 适用环境：
 * 1、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。
 * 2、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。
 * 3、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。
 **/
abstract class SubjectO
{
    private List<Observer> _observers = new List<Observer>();
    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();
        }
    }
}
class ConcreteSubject : SubjectO
{
    private string _subjectState;
    public string SubjectState
    {
        get { return _subjectState; }
        set { _subjectState = value; }
    }
}
```

```
abstract class Observer
{
        public abstract void Update();
}
class ConcreteObserver : Observer
{
        private string _name;
        private string _observerState;
        private ConcreteSubject _subject;
        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);
        }
        public ConcreteSubject Subject
        {
                get { return _subject; }
                set { _subject = value; }
        }
}
```

调用：

```
                ConcreteSubject s = new ConcreteSubject();
                s.Attach(new ConcreteObserver(s, "X"));
                s.Attach(new ConcreteObserver(s, "Y"));
                s.Attach(new ConcreteObserver(s, "Z"));

                s.SubjectState = "ABC";
                s.Notify();
```

输出：

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

20、状态模式

```
/** 名称：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。
```

```
 * 适用环境：
 * 1、一个对象的行为取于它的状态，并且它必须在运行时刻根据状态改变它的行为。
 * 2、一个操作中含有庞大的多分支条件语句，且这些分支依该对象的状。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。
 * */
abstract class State
{
    public abstract void Handle(ContextS context);
}
class ConcreteStateA : State
{
    public override void Handle(ContextS context)
    {
        context.State = new ConcreteStateB();
    }
}
class ConcreteStateB : State
{
    public override void Handle(ContextS context)
    {
        context.State = new ConcreteStateA();
    }
}
class ContextS
{
    private State _state;
    public ContextS(State state)
    {
        this.State = state;
    }
    public State State
    {
        get { return _state; }
        set
        {
            _state = value;
            Console.WriteLine("State: " +
                _state.GetType().Name);
        }
    }
    public void Request()
```

```
        {
            _state.Handle(this);
        }
    }
```

调用：

```
            ContextS cs = new ContextS(new ConcreteStateA());
            cs.Request();
            cs.Request();
            cs.Request();
            cs.Request();
```

输出：

```
State: ConcreteStateA
State: ConcreteStateB
State: ConcreteStateA
State: ConcreteStateB
State: ConcreteStateA
```

21、策略模式

```
/** 名称：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。
 * 适用环境：
 *1、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。
 *2、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。
 *3、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。
 *4、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。
 **/
abstract class Strategy
{
    public abstract void AlgorithmInterFace();
}
class ConcreteStrategyA : Strategy
{
    public override void AlgorithmInterFace()
    {
        Console.WriteLine(
            "Called ConcreteStrategyA.AlgorithmInterface()");
    }
```

```
}
class ConcreteStrategyB : Strategy
{
    public override void AlgorithmInterFace()
    {
        Console.WriteLine(
            "Called ConcreteStrategyB.AlgorithmInterface()");
    }
}
class ConcreteStrategyC:Strategy
{
    public override void AlgorithmInterFace()
    {
        Console.WriteLine(
            "Called ConcreteStrategyC.AlgorithmInterface()");
    }
}
class ContextCS
{
    private Strategy _strategy;
    public ContextCS(Strategy strategy)
    {
        this._strategy = strategy;
    }
    public void ContextInterface()
    {
        _strategy.AlgorithmInterFace();
    }
}
```

调用：

```
        ContextCS contextcs;

        contextcs = new ContextCS(new ConcreteStrategyA());
        contextcs.ContextInterface();

        contextcs = new ContextCS(new ConcreteStrategyB());
        contextcs.ContextInterface();

        contextcs = new ContextCS(new ConcreteStrategyC());
        contextcs.ContextInterface();
```

输出：

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

Called ConcreteStrategyC.AlgorithmInterface()

22、模板方法模式

```
/** 名称：模板方法模式
 * 意图：定义一个操作中的算法的骨架，而将一些步聚延迟到子类中。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。
 * 适用环境：
 * 1、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。
 * 2、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。首先识别现在代码中的不同之处，
 * 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。
 * 3、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。
 **/
abstract class AbstractClass
{
    public abstract void PrimitiveOperation1();
    public abstract void PrimitiveOperation2();

    public void TemplateMehod()
    {
        PrimitiveOperation1();
        PrimitiveOperation2();
        Console.WriteLine("");
    }
}
class ConcreteClassA : AbstractClass
{
    public override void PrimitiveOperation1()
    {
        Console.WriteLine("ConcreteClassA.PrimitiveOpration1()");
    }
    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()");
        }
    }
```

调用：

```
            AbstractClass aA = new ConcreteClassA();
            aA.TemplateMehod();

            AbstractClass aB = new ConcreteClassB();
            aB.TemplateMehod();
```

输出：

```
ConcreteClassA.PrimitiveOpration1()
ConcreteClassA.PrimitiveOperation2()

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

23、访问者模式

```
/** 名称：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。
 * 适用环境：
 * 1、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。
 * 2、需要对一个对象结构中的对象进行很多不同的并且不想的操作，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。
 * 3、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。
 **/
abstract class Visitor
{
    public abstract void VisitConcreteElementA(
        ConcreteElementA concreteElementA);
    public abstract void VisitConcreteElementB(
        ConcreteElementB concreteElementB);
}
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);
        }
}
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);
        }
}
abstract class Element
{
        public abstract void Accept(Visitor visitor);
}
class ConcreteElementA : Element
{
        public override void Accept(Visitor visitor)
        {
            visitor.VisitConcreteElementA(this);
        }
        public void OperationA()
        {
        }
}
class ConcreteElementB : Element
{
        public override void Accept(Visitor visitor)
        {
            visitor.VisitConcreteElementB(this);
        }
```

```
        public void OperationB()
        {
        }
}
class ObjectStructure
{
        private List<Element> _elements = new List<Element>();
        public void Attach(Element element)
        {
                _elements.Add(element);
        }
        public void Detach(Element element)
        {
                _elements.Add(element);
        }
        public void Accept(Visitor vistor)
        {
                foreach (Element element in _elements)
                {
                        element.Accept(vistor);
                }
        }
}
```

调用：

```
            ObjectStructure ov = new ObjectStructure();

                ov.Attach(new ConcreteElementA());
                ov.Attach(new ConcreteElementB());

                ConcreteVisitor1 v1 = new ConcreteVisitor1();
                ConcreteVisitor2 v2 = new ConcreteVisitor2();

                ov.Accept(v1);
                ov.Accept(v2);
```

输出：

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

制作者：ITtray