常用的设计模式包括：

Abstract Factory 模式（抽象工厂）
Adapter 模式（适配器）
Composite 模式（组合）
Decorator 模式（装饰）
Factory 模式（工厂方法）
Observer 模式（观察者）
Strategy 模式（策略）
Template 模式（模板方法）

C#设计模式(23 种设计模式)

创建型：

1. 单件模式(Singleton Pattern)
2. 抽象工厂（Abstract Factory）
3. 建造者模式(Builder)
4. 工厂方法模式（Factory Method)
5. 原型模式(Prototype)

结构型：

6. 适配器模式（Adapter Pattern)
7. 桥接模式（Bridge Pattern)
8. 装饰模式(Decorator Pattern)
9. 组合模式(Composite Pattern)
10. 外观模式（Facade Pattern)
11. 享元模式(Flyweight Pattern)
12. 代理模式(Proxy Pattern)
13. 模板方法(Template Method)
14. 命令模式(Command Pattern)
15. 迭代器模式(Iterator Pattern)

行为型：

16. 观察者模式(Observer Pattern）
17. 解释器模式(Interpreter Pattern)
18. 中介者模式(Mediator Pattern)
19. 职责链模式(Chain of Responsibility Pattern)
20. 备忘录模式(Memento Pattern)
21. 策略模式(Strategy Pattern)
22. 访问者模式(Visitor Pattern)
23. 状态模式(State Pattern)

@Author kwming

抽象工厂（Abstract Factory）

常规的对象创建方法：

```
//创建一个 Road 对象
Road road =new Road();
```

new 的问题：

实现依赖，不能应对“具体实例化类型”的变化。

解决思路：

封装变化点-----哪里变化，封装哪里

潜台词： 如果没有变化，当然不需要额外的封装！

工厂模式的缘起

变化点在“对象创建”，因此就封装“对象创建”

面向接口编程----依赖接口，而非依赖实现

最简单的解决方法：

```
1 class RoadFactory{
2 public static Road CreateRoad()
3 {
4   return new Road();
5 }
6 }
7 //创建一个 Road 对象
8 Road road=roadFactory.CreateRoad();
```

创建一系列相互依赖对象的创建工作：

假设一个游戏开场景：

我们需要构造"道路"、"房屋"、"地道"，"从林"...等等对象

工厂方法如下：

```
1    class RoadFactory
2    {
3        public static Road CreateRoad()
4        {
5            return new Road();
6        }
```

```
7       public static Building CreateBuilding()
8       {
9          return new Building();
10      }
11      public static Tunnel CreateTunnel()
12      {
13         return new Tunnel();
14      }
15      public static Jungle CreateJungle()
16      {
17         return new Jungle();
18      }
19   }
```

调用方式如下：

```
1       Road road =  RoadFactory.CreateRoad();
3       Building building = RoadFactory.CreateBuilding();
4       Tunnel tunnel = RoadFactory.CreateTunnel();
5       Jungle jungle = RoadFactory.CreateJungle();
```

如上可见**简单工厂的问题：**

不能应对"不同系列对象"的变化。比如有不同风格的场景---对应不同风格的道路，房屋、地道....

如何解决：

使用面向对象的技术来"封装"变化点。

动机(Motivate)：

在软件系统中，经常面临着"一系统相互依赖的对象"的创建工作：同时，由于需求的变化，往往存在更多系列对象的创建工作。

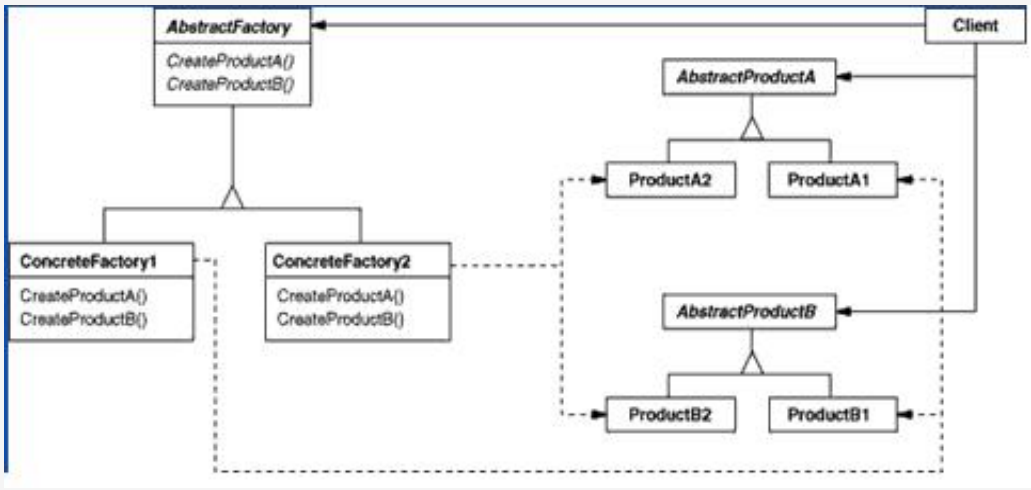
如何应对这种变化？如何绕过常规的对象创建方法（new),提供一种"封装机制"来避免客户程序和这种"多系列具体对象创建工作"的紧耦合？

意图(Intent):

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

----《设计模式》GOF

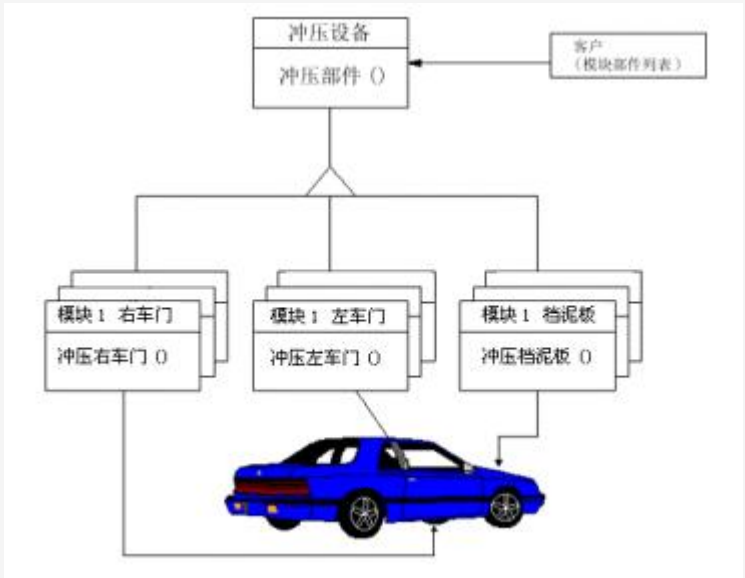
结构图（Struct）：



适用性：

1. 一个系统要独立于它的产品的创建、组合和表示时。
2. 一个系统要由多个产品系统中的一个来配置时。
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不是实现时。

生活例子：



结构图代码实现：

```
1  abstract class AbstractFactory
2    {
```

```
3       public abstract AbstractProductA CreateProductA();
4       public abstract AbstractProductB CreateProductB();
5     }
```

```
1   abstract class AbstractProductA
2     {
3       public abstract void Interact(AbstractProductB b);
4     }
```

```
1   abstract class AbstractProductB
2     {
3       public abstract void Interact(AbstractProductA a);
4     }
```

```
1    class Client
2     {
3        private AbstractProductA AbstractProductA;
4        private AbstractProductB AbstractProductB;
5        public Client(AbstractFactory factory)
6        {
7           AbstractProductA = factory.CreateProductA();
8           AbstractProductB = factory.CreateProductB();
9        }
10        public void Run()
11        {
12           AbstractProductB.Interact(AbstractProductA);
13           AbstractProductA.Interact(AbstractProductB);
14        }
15     }
1   class ConcreteFactory1:AbstractFactory
2     {
3        public override AbstractProductA CreateProductA()
4        {
```

```
5           return new ProductA1();
6       }
7       public override AbstractProductB CreateProductB()
8       {
9           return new ProductB1();
10       }
11   }
1    class ConcreteFactory2:AbstractFactory
2    {
3       public override AbstractProductA CreateProductA()
4       {
5           return new ProdcutA2();
6       }
7       public override AbstractProductB CreateProductB()
8       {
9           return new ProductB2();
10       }
11   }
1  class ProductA1:AbstractProductA
2    {
3       public override void Interact(AbstractProductB b)
4       {
5           Console.WriteLine(this.GetType().Name + "interact with" + b.GetType(
).Name);
6       }
7    }
1   class ProductB1:AbstractProductB
2    {
3       public override void Interact(AbstractProductA a)
4       {
5           Console.WriteLine(this.GetType().Name + "interact with" + a.GetType(
).Name);
6       }
7    }
```

```
1  class ProdcutA2:AbstractProductA
2    {
3       public override void Interact(AbstractProductB b)
4       {
5          Console.WriteLine(this.GetType().Name + "interact with" + b.GetType(
).Name);
6       }
7    }
1  class ProductB2:AbstractProductB
2    {
3       public override void Interact(AbstractProductA a)
4       {
5          Console.WriteLine(this.GetType().Name + "interact with" + a.GetType(
).Name);
6       }
7    }
 1   public static void Main()
 2       {
 3        // Abstractfactory1
 4          AbstractFactory factory1 = new ConcreteFactory1();
 5          Client c1 = new Client(factory1);
 6          c1.Run();
 7       // Abstractfactory2
 8          AbstractFactory factory2 = new ConcreteFactory2();
 9          Client c2 = new Client(factory2);
10           c2.Run();
11       }
```

Abstract Factory 注意的几点：

如果不存在”多系列对象创建“的需求变化，则没必要应用 Abstract Factory 模式，静态工厂方法足矣。

"系列对象"指的是这些对象之间有相互依赖、或作用的关系。例如游戏开发场景中的"道路"与"房屋"依赖，“道路”与“地道”的依赖。

Abstract Factory 模式主要在于应对"新系列"的需求变动。其**缺点**在于难以

应对”新对象“的需求变动。

Abstract Factory 模式经常和 **Factory Method** 模式共同组合来应对“**对象创建**”的需求变化。

单件模式(Singleton Pattern)

创建型模式---单件模式(Singleton Pattern)

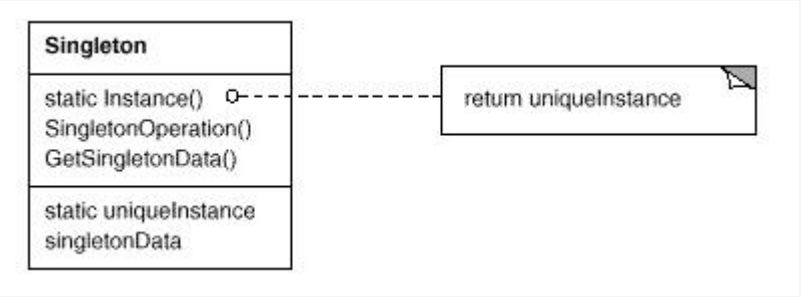
动机（Motivation）:

在软件系统中，经常有这样一些特殊的类，必须保证它们在系统中只存在一个实例，才能确保它们的逻辑正确性、以及良好的效率。

如何绕过常规的构造器，提供一种机制来保证一个类只创建一个实例？

这应该是类设计者的责任，而不是类使用者的责任。

结构图：

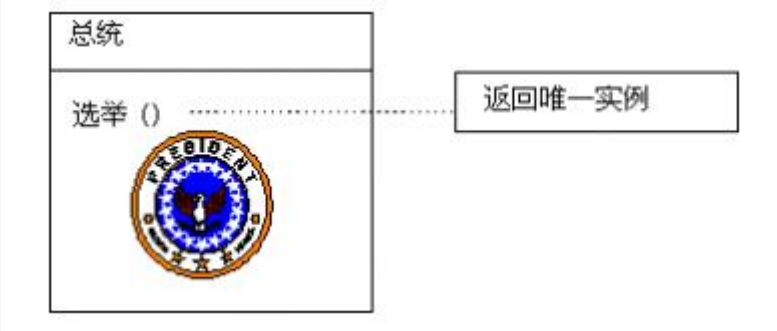


意图：

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。

------<<设计模式>>GOF

生活的例子：



适用性：

（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。

代码实现：

（1）单线程 Singleton 实现

```
class SingleThread_Singleton
{
    private static SingleThread_Singleton instance = null;
    private  SingleThread_Singleton(){}
    public static SingleThread_Singleton Instance
    {
       get
       {
          if (instance == null)
          {
             instance = new SingleThread_Singleton();
          }
          return instance;
       }
    }
}
```

以上代码在单线程情况下不会出现任何问题。但是在多线程的情况下却不是安全的。

如两个线程同时运行到 if (instance == null)判断是否被实例化，一个线程判断为 True 后，在进行创建

instance = new SingleThread_Singleton();之前，另一个线程也判断

(instance == null)，结果也为 True.

这样就就违背了 Singleton 模式的原则（保证一个类仅有一个实例）。

怎样在多线程情况下实现 Singleton?

（2）多线程 Singleton 实现：

```
1   class MultiThread_Singleton
2    {
3      private  static volatile MultiThread_Singleton instance = null;
4      private static object lockHelper = new object();
5      private MultiThread_Singleton() { }
6      public static MultiThread_Singleton Instance
7      {
8         get
9         {
```

```
10            if (instance == null)
11            {
12               lock (lockHelper)
13               {
14                  if (instance == null)
15                  {
16                     instance = new MultiThread_Singleton();
17                  }
18               }
19            }
20            return instance;
21         }
22      }
23
```

此程序对多线程是安全的，使用了一个辅助对象 lockHelper，保证只有一个线程创建实例（如果 instance 为空，保证只有一个线程 instance = new MultiThread_Singleton();创建唯一的一个实例）。（Double Check)

请注意一个关键字 volatile，如果去掉这个关键字，还是有可能发生线程不是安全的。

volatile 保证严格意义的多线程编译器在代码编译时对指令不进行微调。

(3)静态 Singleton 实现

```
3        class Static_Singleton
4        {
5           public
static readonly Static_Singleton instance = new Static_Singleton();
6           private Static_Singleton() { }
7        }
```

以上代码展开等同于

```
1     class Static_Singleton
2        {
3           public static readonly Static_Singleton instance;
4           static Static_Singleton()
```

```
5           {
6               instance = new Static_Singleton();
7           }
8           private Static_Singleton() { }
9       }
```

由此可以看出，完全符合 Singleton 的原则。

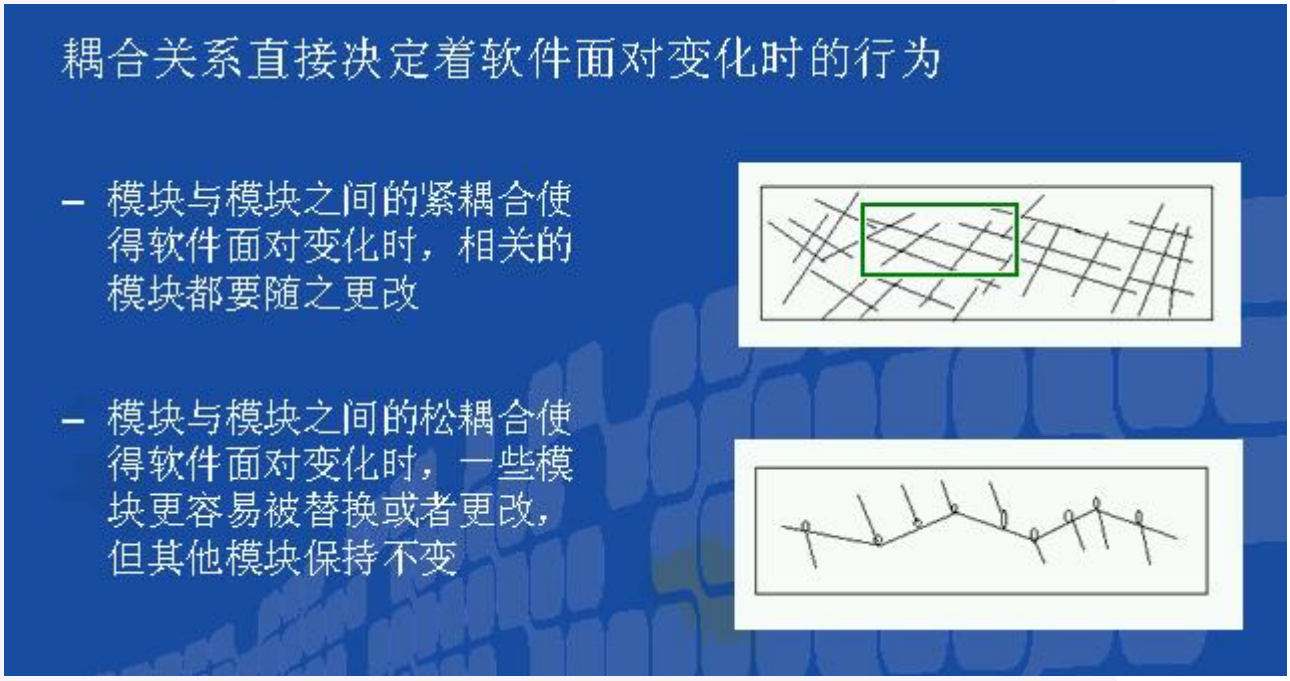
优点： 简洁，易懂

缺点： 不可以实现带参数实例的创建。

（注：以上代码及信息借鉴于李建忠老师的 MSDN 和 TerryLee 的文章。）

工厂方法模式（Factory Method)

耦合关系：



动机(Motivation):

在软件系统中，由于需求的变化，"这个对象的具体实现"经常面临着剧烈的变化，但它却有比较稳定的接口。

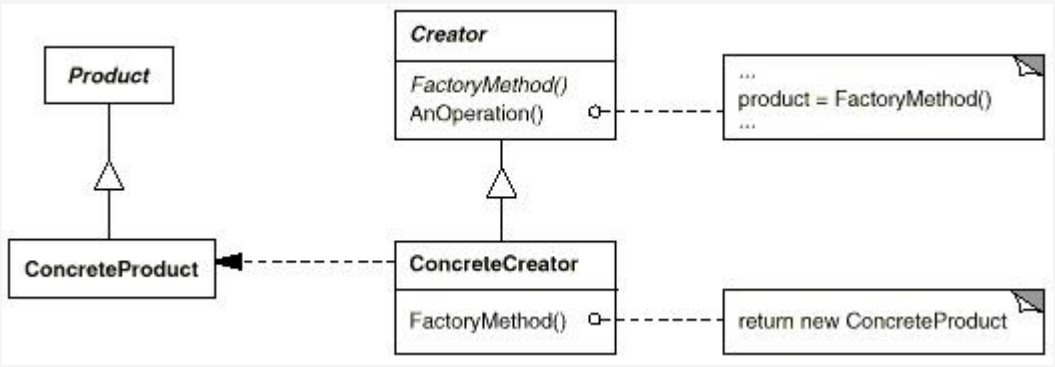
如何应对这种变化呢？提供一种封装机制来隔离出"这个易变对象"的变化，从而保持系统中"其它依赖的对象"不随需求的变化而变化。

意图(Intent):

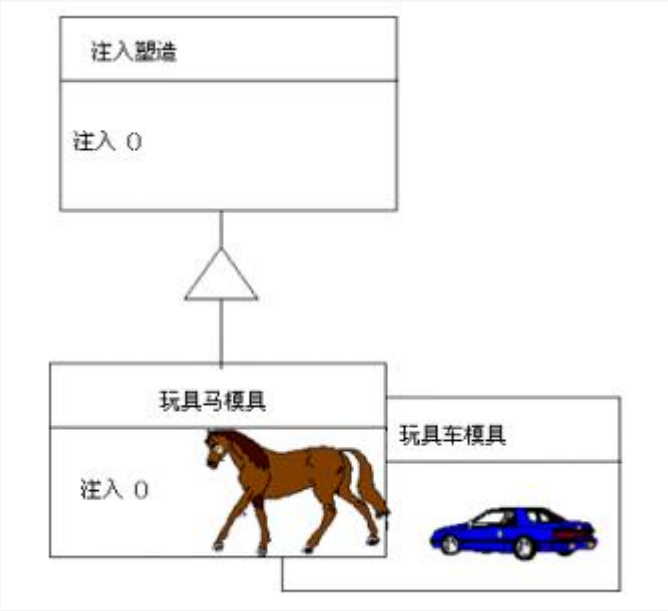
定义一个用户创建对象的接口，让子类决定实例哪一个类。Factory Method 使一个类的实例化延迟到子类。

----------《设计模式》GOF

结构图(Struct):



生活实例：



适用性：

1.当一个类不知道它所必须创建的对象类的时候。

2.当一个类希望由它子类来指定它所创建对象的时候。

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。

实例代码：

CarFactory 类：

```
1    public abstract class CarFactory
2     {
3        public abstract Car CarCreate();
4     }
```

Car 类：

```
1   public abstract class Car
2     {
```

```
3       public abstract void StartUp();
4       public abstract void Run();
5       public abstract void Stop();
6
7     }
```

HongQiCarFactory 类：

```
1   public class HongQiCarFactory:CarFactory
2     {
3       public override Car CarCreate()
4       {
5           return new HongQiCar();
6       }
7     }
```

BMWCarFactory 类：

```
1   public class BMWCarFactory:CarFactory
2     {
3       public override Car CarCreate()
4       {
5           return new BMWCar();
6       }
7     }
```

HongQiCar 类：

```
 1   public  class HongQiCar:Car
 2     {
 3       public override void StartUp()
 4       {
 5           Console.WriteLine("Test HongQiCar start-up speed!");
 6       }
 7       public override void Run()
 8       {
 9           Console.WriteLine("The HongQiCar run is very quickly!");
10       }
11       public override void Stop()
12       {
```

```
13          Console.WriteLine("The slow stop time is 3 second ");
14      }
15    }
```

BMWCar 类：

```
1   public  class BMWCar:Car
2    {
3      public override void StartUp()
4      {
5         Console.WriteLine("The BMWCar start-up speed is very quickly");
6      }
7      public override void Run()
8      {
9         Console.WriteLine("The BMWCar run is quitely fast and safe!!!");
10      }
11      public override void Stop()
12      {
13         Console.WriteLine("The slow stop time is 2 second");
14      }
15    }
```

app.config

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <configuration>
3   <appSettings>
4     <add key="No1" value="HongQiCarFactory"/>
5     <add key="No2" value="BMWCarFactory"/>
6   </appSettings>
7 </configuration>
```

Program 类:

```
1  class Program
2    {
3       static void Main(string[] args)
4       {
5          Console.WriteLine("Please Enter Factory Method No:");
6          Console.WriteLine("*****************************");
```

```
7           Console.WriteLine("no        Factory Method");
8           Console.WriteLine("1         HongQiCarFactory");
9           Console.WriteLine("2         BMWCarFactory");
10          Console.WriteLine("*****************************");
11          int no=Int32.Parse(Console.ReadLine().ToString());
12          string factoryType=ConfigurationManager.AppSettings["No"+no];
13          //CarFactory factory = new HongQiCarFactory();
14          CarFactory factory = (CarFactory)Assembly.Load("FactoryMehtod").Cr
eateInstance("FactoryMehtod." + factoryType); ;
15          Car car=factory.CarCreate();
16          car.StartUp();
17          car.Run();
18          car.Stop();
19
20      }
21  }
```

Factory Method 模式的几个要点：

Factory Method 模式主要用于隔离类对象的使用者和具体类型之间的耦合关系。面对一个经常变化的具体类型，紧耦合关系会导致软件的脆弱。

Factory Method 模式通过面向对象的手法，将所要创建的具体对象工作延迟到子类，从而实现一种扩展（而非更改）的策略，较好地解决了这种紧耦合关系。

Factory Mehtod 模式解决"单个对象"的需求变化，**AbstractFactory 模式**解决"系列对象"的需求变化，**Builder 模式**解决"对象部分"的需求变化。

建造者模式(Builder)

Builder 模式的缘起：

假设创建游戏中的一个房屋 House 设施，该房屋的构建由几部分组成，且各个部分富于变化。如果使用最直观的设计方法，每一个房屋部分的变化，都将导致房屋构建的重新修正.....

动机（Motivation）:

在软件系统中，有时候面临一个"复杂对象"的创建工作，其通常由各个部分的子对象用一定算法构成;由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合到一起的算法却相对稳定。

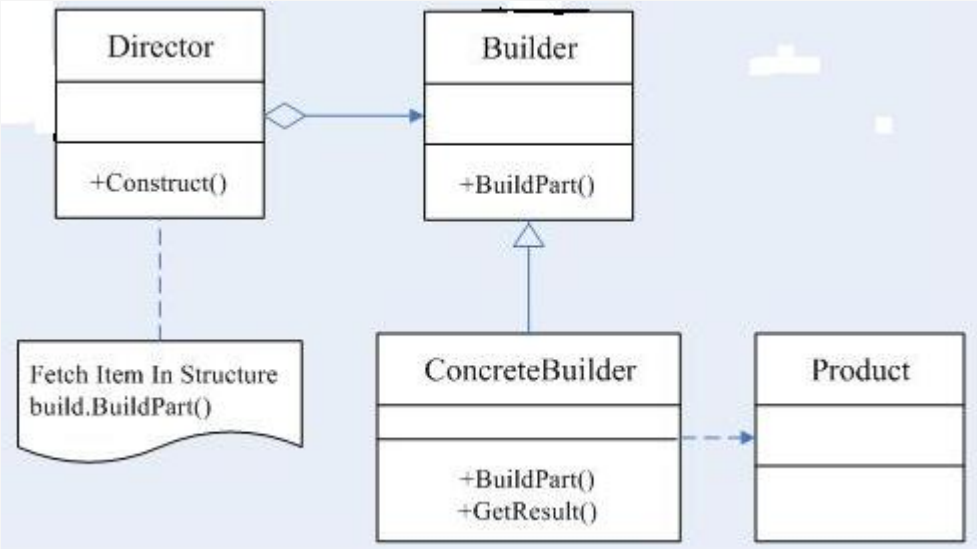
如何应对种变化呢？如何提供一种"封装机制"来隔离出"复杂对象的各个部分"的变化，从而保持系统中的"稳定构建算法"不随需求的改变而改变？

意图(Intent)：

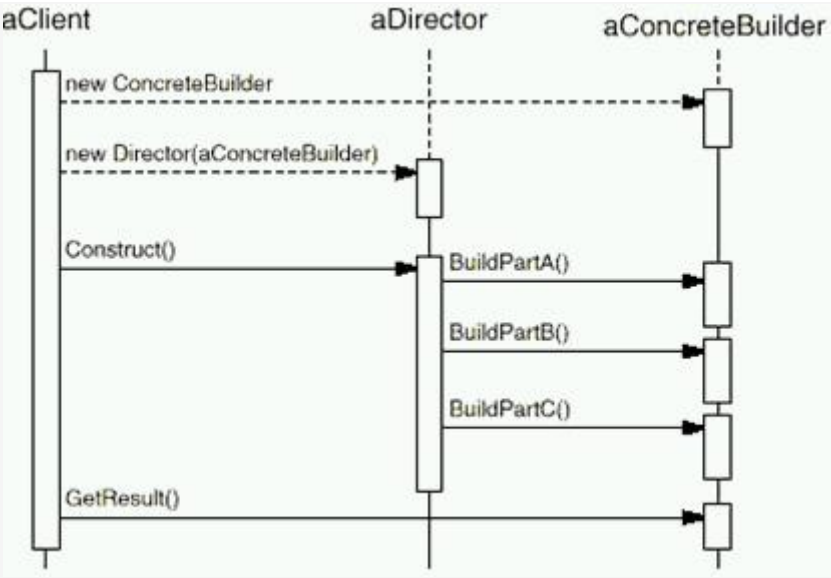
将一个复杂对象的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

-------《设计模式》GOF

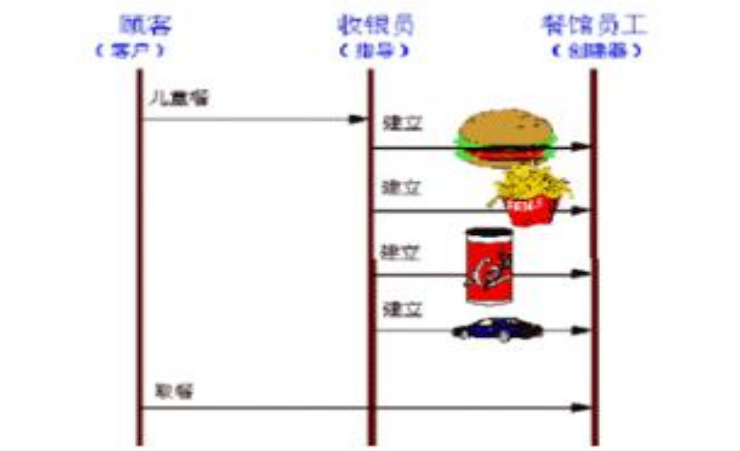
结构图(Struct):



协作(Collaborations):



生活中的例子：



适用性：

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。

实例代码：

Builder 类：

```
1 public abstract class Builder
2    {
3        public abstract void BuildDoor();
4        public abstract void BuildWall();
5        public abstract void BuildWindows();
6        public abstract void BuildFloor();
7        public abstract void BuildHouseCeiling();
8
```

```
9        public abstract House GetHouse();
10    }
```

Director 类：这一部分是 组合到一起的算法（相对稳定）。

```
1 public class Director
2    {
3     public void Construct(Builder builder)
4     {
5        builder.BuildWall();
6        builder.BuildHouseCeiling();
7        builder.BuildDoor();
8        builder.BuildWindows();
9        builder.BuildFloor();
10     }
11    }
```

ChineseBuilder 类

```
1   public class ChineseBuilder:Builder
2    {
3      private House ChineseHouse = new House();
4       public override void BuildDoor()
5       {
6          Console.WriteLine("this Door 's style of Chinese");
7       }
8       public override void BuildWall()
9       {
10         Console.WriteLine("this Wall 's style of Chinese");
11      }
12      public override void BuildWindows()
13      {
14         Console.WriteLine("this Windows 's style of Chinese");
15      }
16      public override void BuildFloor()
17      {
18         Console.WriteLine("this Floor 's style of Chinese");
19      }
```

```
20        public override void BuildHouseCeiling()
21        {
22            Console.WriteLine("this Ceiling 's style of Chinese");
23        }
24        public override House GetHouse()
25        {
26            return ChineseHouse;
27        }
28    }
```

RomanBuilder 类：

```
1    class RomanBuilder:Builder
2    {
3        private House RomanHouse = new House();
4        public override void BuildDoor()
5        {
6            Console.WriteLine("this Door 's style of Roman");
7        }
8        public override void BuildWall()
9        {
10           Console.WriteLine("this Wall 's style of Roman");
11       }
12       public override void BuildWindows()
13       {
14           Console.WriteLine("this Windows 's style of Roman");
15       }
16       public override void BuildFloor()
17       {
18           Console.WriteLine("this Floor 's style of Roman");
19       }
20       public override void BuildHouseCeiling()
21       {
22           Console.WriteLine("this Ceiling 's style of Roman");
23       }
24       public override House GetHouse()
```

```
25        {
26            return RomanHouse;
27        }
28    }
```

ChineseBuilder 和 RomanBuilder 这两个是：这个复杂对象的两个部分经常面临着剧烈的变化。

```
1   public class Client
2     {
3        public static void Main(string[] args)
4        {
5            Director director = new Director();
6
7            Builder instance;
8
9            Console.WriteLine("Please Enter House No:");
10
11           string No = Console.ReadLine();
12
13           string houseType = ConfigurationSettings.AppSettings["No" + No];
14
15           instance = (Builder)Assembly.Load("House").CreateInstance("House."
 + houseType);
16
17           director.Construct(instance);
18
19           House house= instance.GetHouse();
20           house.Show();
21
22           Console.ReadLine();
23       }
24    }
```

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="No1" value="RomanBuilder"></add>
    <add key="No2" value="ChineseBuilder"></add>
  </appSettings>
</configuration>
```

Builder 模式的几个要点:

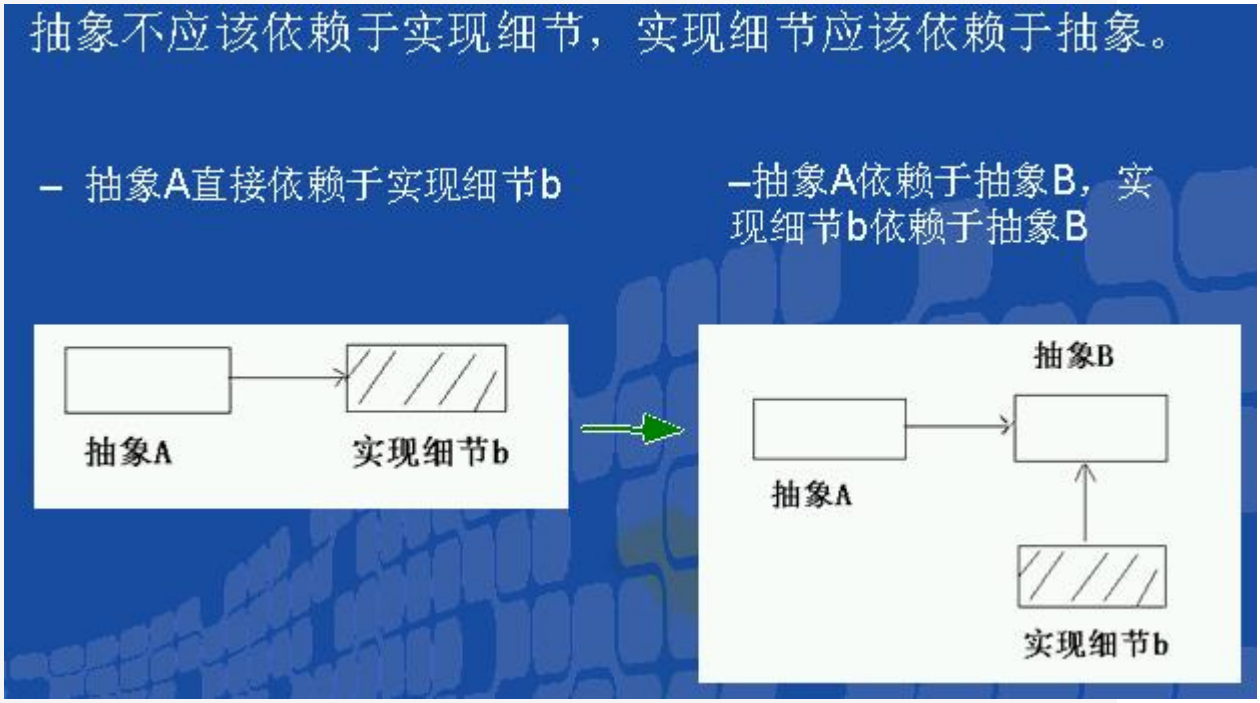
Builder 模式 主要用于“分步骤构建一个复杂的对象”。在这其中“分步骤”是一个稳定的乘法，而复杂对象的各个部分则经常变化。

Builder 模式主要在于应对“复杂对象各个部分”的频繁需求变动。其**缺点在于**难以应对“分步骤构建算法”的需求变动。

Abstract Factory 模式解决“系列对象”的需求变化，**Builder 模式**解决“对象部分”的需求变化。Builder 械通常和 Composite 模式组合使用。

原型模式(Prototype)

依赖关系倒置：



动机(Motivate)：

在软件系统中，经常面临着“某些结构复杂的对象”的创建工作;由于需求的变化，这些对象经常面临着

剧烈的变化,但是它们却拥有比较稳定一致的接口。

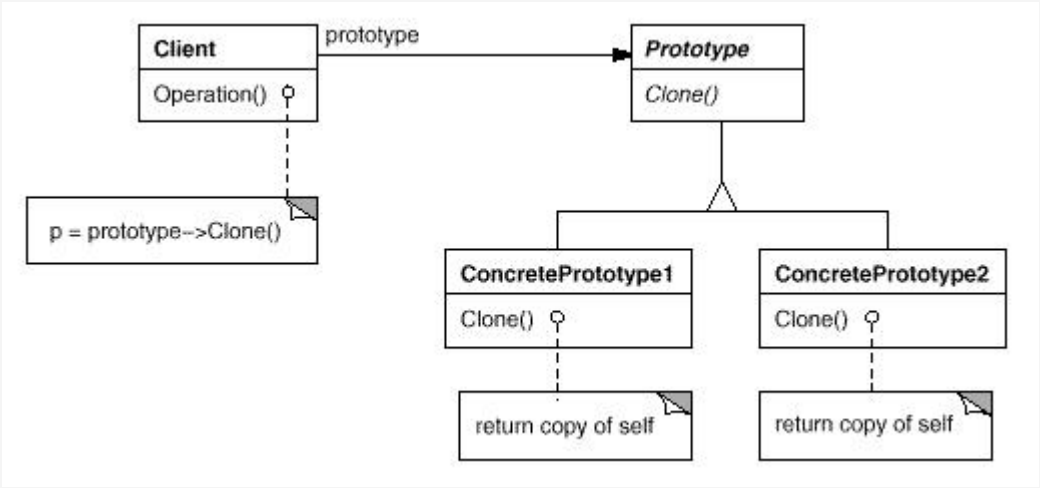
如何应对这种变化？如何向“客户程序（使用这些对象的程序）"隔离出“这些易变对象”，从而使得“依赖这些易变对象的客户程序”不随着需求改变而改变？

意图(Intent)：

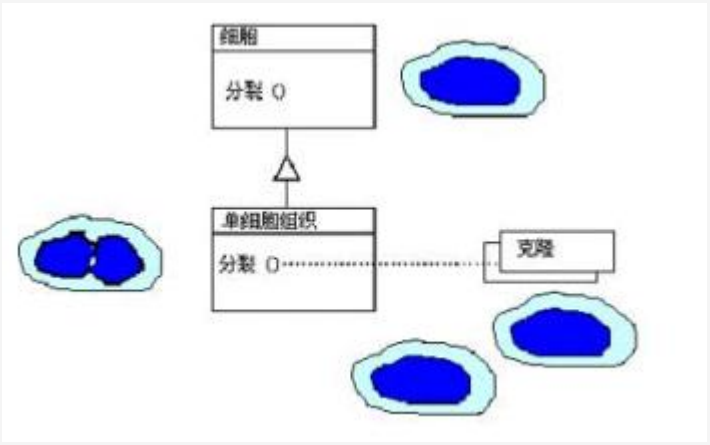
用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

------《设计模式》GOF

结构图(Struct)：



生活例子：



适用性：

1．当一个系统应该独立于它的产品创建，构成和表示时；

2．当要实例化的类是在运行时刻指定时，例如，通过动态装载；

3．为了避免创建一个与产品类层次平行的工厂类层次时；

4．当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

示意性代码例子：

```
1 public  abstract class NormalActor
2    {
3       public abstract NormalActor clone();
4    }
```

```
1   public class NormalActorA:NormalActor
2    {
3      public override NormalActor clone()
4      {
5         Console.WriteLine("NormalActorA is call");
6         return (NormalActor)this.MemberwiseClone();
7
8      }
9    }
```

```
1  public class NormalActorB :NormalActor
2    {
3      public override NormalActor clone()
4      {
5         Console.WriteLine("NormalActorB  was called");
6         return (NormalActor)this.MemberwiseClone();
7
8      }
9    }
```

```
public class GameSystem
 {
    public void Run(NormalActor normalActor)
    {
       NormalActor normalActor1 = normalActor.clone();
       NormalActor normalActor2 = normalActor.clone();
```

```
        NormalActor normalActor3 = normalActor.clone();
        NormalActor normalActor4 = normalActor.clone();
        NormalActor normalActor5 = normalActor.clone();

      }
    }

    class
Program

    {
       static void Main(string[] args)
       {
          GameSystem gameSystem = new GameSystem();

         gameSystem.Run(new NormalActorA());
       }
    }
```

如果又需要创建新的对象（flyActor)，只需创建此抽象类，然后具体类进行克隆。

```
  public abstract class FlyActor
    {
       public abstract FlyActor clone();
    }

  public   class FlyActorB:FlyActor
    {
     /// <summary>
     /// 浅拷贝，如果用深拷贝，可使用序列化
     /// </summary>
     /// <returns></returns>
     public override FlyActor clone()
     {
```

```
        return (FlyActor)this.MemberwiseClone();
     }
   }
```

此时，调用的 Main()函数只需如下：

```
  class
Program


   {
      static void Main(string[] args)
      {
         GameSystem gameSystem = new GameSystem();


        gameSystem.Run(new NormalActorA(), new FlyActorB());
      }
   }
```

Prototype 的几个要点：

Prototype 模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这
些“易变类”拥有“稳定的接口”。

Prototype 模式对于“如何创建易变类的实体对象“采用“原型克隆”的方法来做，它使得我们可以
非常灵活地动态创建“拥有某些稳定接口中”的新对象----所需工作仅仅是注册的地方不断地 Clone.

Prototype 模式中的 Clone 方法可以利用.net 中的 object 类的 memberwiseClone()方法或者序列化来实现深拷贝。

有关创建型模式的讨论：

Singleton 模式解决的是实体对象个数的问题。除了 Singleton 之外，其他创建型模式解决的是都是 new 所带来的耦合关系。

Factory Method ,Abstract Factory,Builder 都需要一个额外的工厂类来负责实例化“易变对象”，而 Prototype 则是通过原型（一个特殊的工厂类）来克隆“易变对象”。

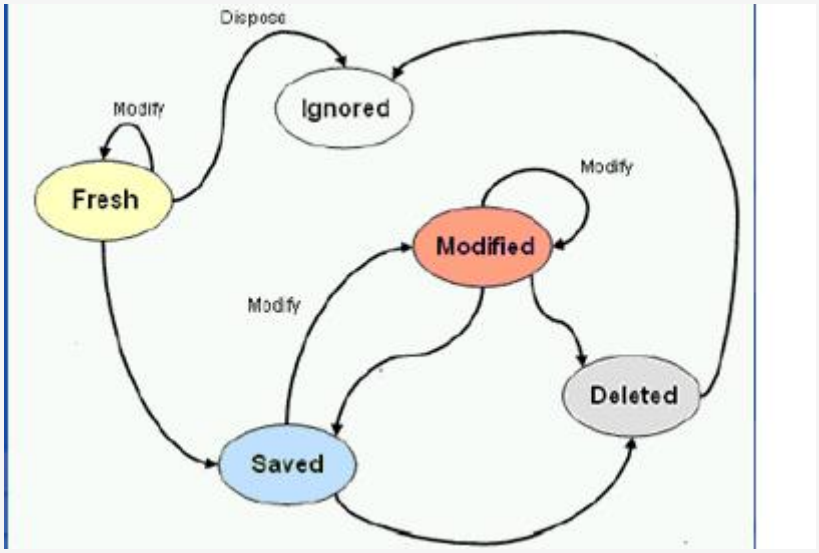
如果遇到“易变类”，起初的设计通常从 Factory Mehtod 开始，当遇到更多

的复杂变化时，再考虑重重构为其他三种工厂模式（Abstract Factory,Builder,Prototype).

备忘录模式(Memento Pattern)

对象状态的回溯：

对象状态的变化无端，如何回溯/恢复对象在某个点的状态？



动机：

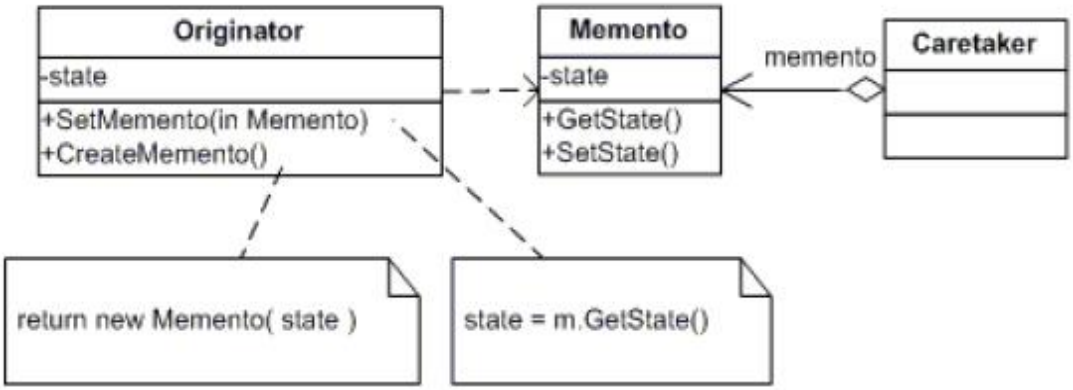
在软件构建过程中，某些对象的状态在转换过程中，可能由于某种需要，要求程序能够回溯到对象之前处于某个点时的状态。如果使用一些公有接口来让其他对象得到对象的状态，便会暴露对象的细节实现。

如何实现对象状态的良好保存与恢复？但同时又不会因此而破坏对象本身的封装性。

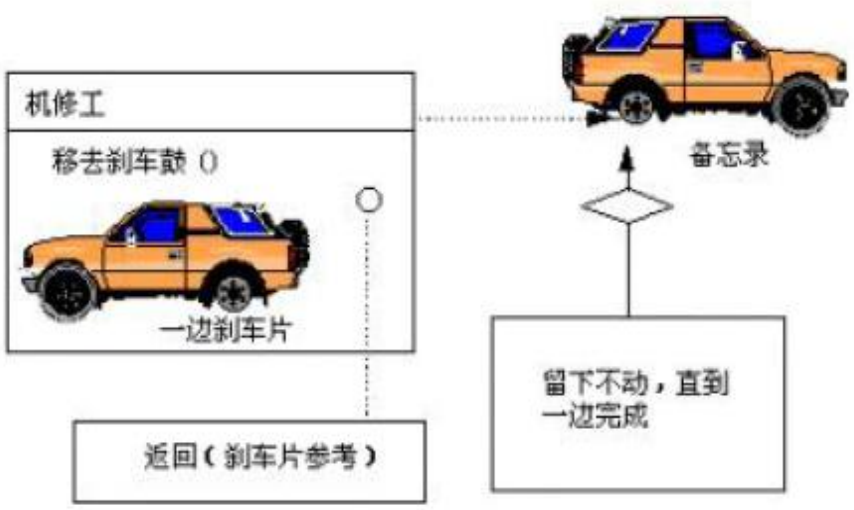
意图：

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。这样以后可以将该对象恢复到原先保存的状态。

结构图



生活例子



适用性：

1.必须保存一个对象某一个时刻的(部分)状态，这样以后需要时它才能恢复到先前的状态。

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破坏对象的封装性。

代码实现：

```
1    class Memento
2    {
3        private string name;
4        private string phone;
5        private double budget;
6
```

```
7       //Constructor
8       public Memento(string name, string phone, double budget)
9       {
10          this.name = name;
11          this.phone = phone;
12          this.budget = budget;
13      }
14      //Properties
15      public string Name
16      {
17          get { return name; }
18          set { name = value; }
19      }
20      public string Phone
21      {
22          get { return phone; }
23          set { phone = value; }
24      }
25      public double Budget
26      {
27          get { return budget; }
28          set { budget = value; }
29      }
30  }
```

```
1   class ProspectMemory
2   {
3       private Memento memento;
4
5       //Property
6       public Memento Memento
7       {
8           set { memento = value; }
9           get { return memento; }
```

```
    }
  }
```

```
//Originator
 class SalesProspect
 {
    private string name;
    private string phone;
    private double budget;

    //Properties
    public string Name
     {
        get { return name; }
        set { name = value; Console.WriteLine("Name:" + name); }
     }
     public string Phone
     {
        get { return phone; }
        set { phone = value; Console.WriteLine("Phone:" + phone); }
     }
     public double Budget
     {
        get { return Budget; }
        set { budget = value; Console.WriteLine("Budget:" + budget); }
     }
     public Memento SaveMemento()
     {
        Console.WriteLine("\nSaving state --\n");
        return new Memento(name, phone, budget);
     }
     public void RestoreMemento(Memento memento)
     {
        Console.WriteLine("\nRestoring state --\n");
```

```
32          this.Name = memento.Name;
33          this.Phone = memento.Phone;
34          this.Budget = memento.Budget;
35      }
36  }
```

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5           SalesProspect s = new SalesProspect();
6           s.Name = "xiaoming";
7           s.Phone = "(010)65236523";
8           s.Budget = 28000.0;
9
10          //Store internal state
11          ProspectMemory m = new ProspectMemory();
12          m.Memento = s.SaveMemento();
13
14          //Continue changing originator
15          s.Name = "deke";
16          s.Phone = "(029)85423657";
17          s.Budget = 80000.0;
18
19          //Restore saved state
20          s.RestoreMemento(m.Memento);
21
22          //Wait for user
23          Console.Read();
24      }
25  }
```

Memento 需要注意的几个要点：

1.备忘录(Memento)存储原发器(Originator)对象的内部状态，在需要时恢复原发器状态。

Memento 模式适用于“由原发器管理，却又必须存储在原发器之外的信息”。

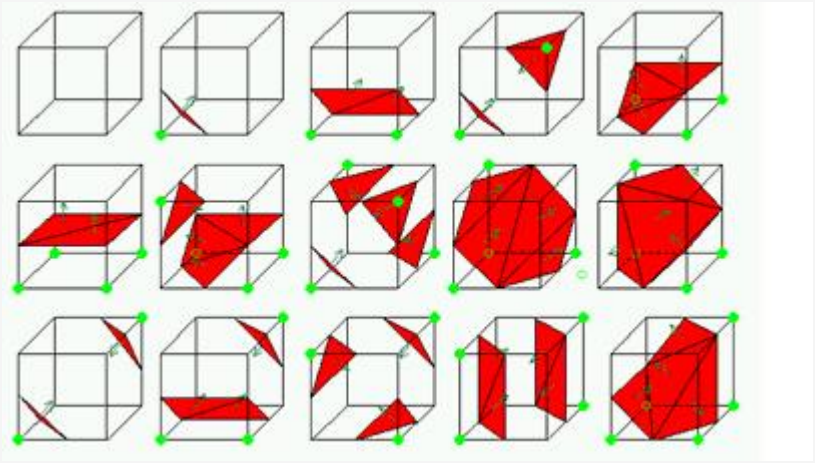
2.在实现 Memento 模式中，要防止原发器以外的对象访问备忘录对象。备忘录对象有两个接口，一个为原发器的宽接口;一个为其他对象使用的窄接口。

3.在实现 Memento 模式时，要考虑拷贝对象状态的效率问题，如果对象开销比较大，可以采用某种增量式改变来改进 Memento 模式。

策略模式(Strategy Pattern)

算法与对象的耦合：

对象可能经常需要使用多种不同的算法，但是如果变化频繁，会将类型变得脆弱...



动机：

在软件构建过程中，某些对象使用的算法可能多种多样，经常改变，如果将这些算法都编码对象中，将会使对象变得异常复杂;而且有时候支持不使用的算法也是一个性能负担。

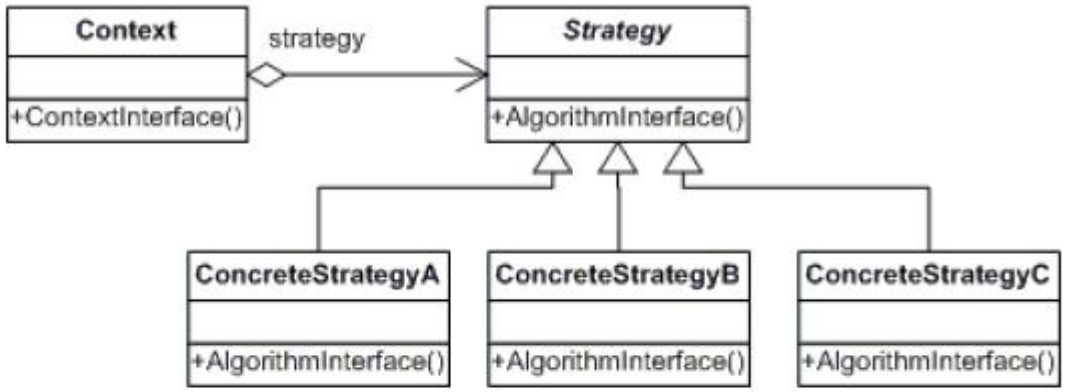
如何在运行时根据需要透明地更改对象的算法？将算法与对象本身解耦，从而避免上述问题？

意图：

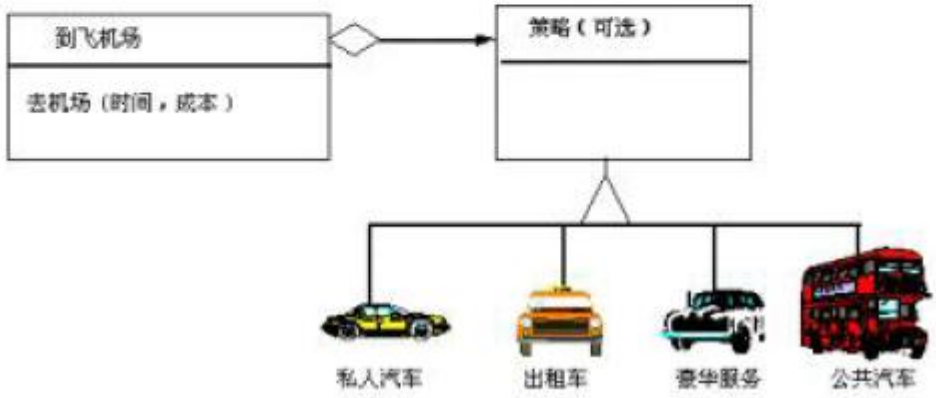
定义一系统的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

--------《设计模式》GOF

结构图



生活例子



适用性：

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[H087]，可以使用策略模式。

3.算法使用客户不应该知道数据。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。

代码实现：

```
1  enum SortType
2    {
3        QuickSort,
```

```
4       ShellSort,
5       MergeSort,
6   }
```

```
1   class Sort
2   {
3       public void SortList(SortType s)
4       {
5           if (s == SortType.QuickSort)
6           {
7               ProcessA();
8           }
9           else if (s == SortType.ShellSort)
10          {
11              ProcessB();
12          }
13          else if (s == SortType.MergeSort)
14          {
15              ProcessC();
16          }
17          Console.WriteLine();
18      }
19
20      protected void ProcessA()
21      {
22          Console.WriteLine("QuickSort List");
23      }
24      protected void ProcessB()
25      {
26          Console.WriteLine("ShellSort List");
27      }
28      protected void ProcessC()
29      {
30          Console.WriteLine("MergeSort List");
```

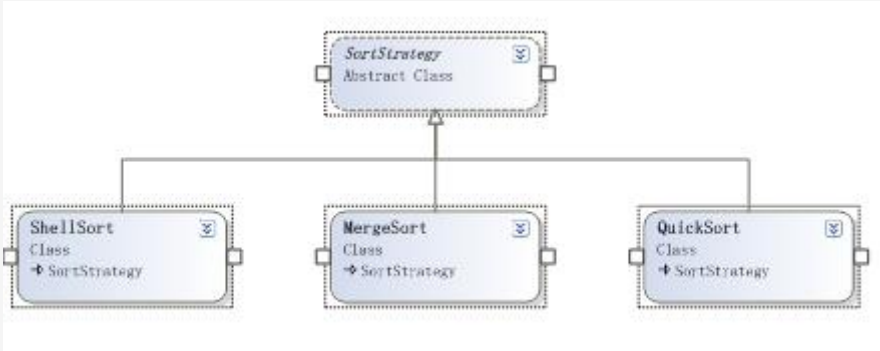
```
31        }
32    }
```

客户端调用：

```
1    class Test
2    {
3        public static void Main()
4        {
5            Sort sort = new Sort();
6            sort.SortList(SortType.QuickSort);
7            sort.SortList(SortType.ShellSort);
8            sort.SortList(SortType.MergeSort);
9        }
10   }
```

由此可见，由于客户端新增调用方式的选择，就会修改 SortType 及 Sort 里的判断语句。在类 Sort 中会增加 if 语句的判断，用敏捷软件开发的语言说，你应该闻到了代码的臭味道了，也就是设计模式中说的存在了变化的地方。

重构以上代码，增加一层中间层来处理变化。类结构如下：




```
1    //Stategy  表达抽象算法
2    abstract  class SortStrategy
3    {
4      public abstract void Sort(ArrayList list);
5    }
```

```
1    //ConcreteStrategy
2     class ShellSort :SortStrategy
3     {
```

```
4       public override void Sort(System.Collections.ArrayList list)
5       {
6               list.Sort(); //no-implement
7               Console.WriteLine("ShellSorted List");
8
9       }
10    }
```

```
1   //ConcreteStrategy
2    class MergeSort :SortStrategy
3    {
4       public override void Sort(System.Collections.ArrayList list)
5       {
6          list.Sort(); //no-implement
7          Console.WriteLine("MergeSort List ");
8       }
9    }
1    //ConcreteStrategy
2    class QuickSort :SortStrategy
3    {
4       public override void Sort(System.Collections.ArrayList list)
5       {
6          list.Sort(); //Default is Quicksort
7          Console.WriteLine("QuickSorted List");
8       }
9    }
```

```
1    //Context
2    class SortdList
3    {
4       private ArrayList list = new ArrayList();
5       private SortStrategy sortstrategy;  //对象组合
6       public void SetSortStrategy(SortStrategy sortstrategy)
7       {
```

```
        this.sortstrategy = sortstrategy;
    }
    public void Add(string name)
    {
        list.Add(name);
    }
    public void Sort()
    {
        sortstrategy.Sort(list);
        //Display results
        foreach (string name in list)
        {
            Console.WriteLine(" " + name);
        }
        Console.WriteLine();
    }
}
```

客户端代码如下：

```
class Program
{
    static void Main(string[] args)
    {
        //Two contexts following different strategies
        SortdList studentRecords = new SortdList();

        studentRecords.Add("Satu");
        studentRecords.Add("Jim");
        studentRecords.Add("Palo");
        studentRecords.Add("Terry");
        studentRecords.Add("Annaro");

        studentRecords.SetSortStrategy(new QuickSort());
        studentRecords.Sort();

```

```
17          studentRecords.SetSortStrategy(new ShellSort());
18          studentRecords.Sort();
19
20          studentRecords.SetSortStrategy(new MergeSort());
21          studentRecords.Sort();
22
23          Console.Read();
24      }
25  }
```

由此可见，更好地满足开放封闭原则。

Strategy 模式的几个要点：

1.Strategy 及其子类为组件提供了一系列可重用的算法，从而可以使得类型在运行时方便地根据需要在各个算法之间进行切换。所谓封装算法，支持算法的变化。

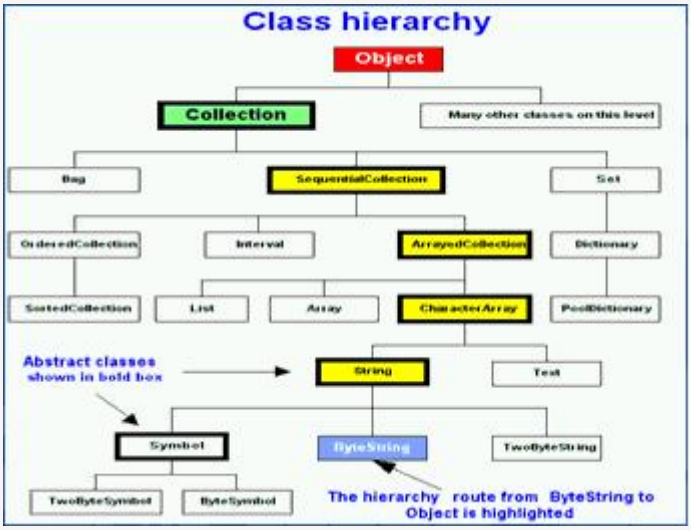
2.Strategy 模式提供了用条件判断语句以外的另一种选择，消除条件判断语句，就是在解耦合。含有许多条件判断语句的代码通常都需要 Strategy 模式。

3.与 State 类似，如果 Strategy 对象没有实例变量，那么各个上下文可以共享同一个 Strategy 对象，从而节省对象开销

访问者模式(Visitor Pattern)

类层次结构的变化：

类层次结构中可能经常由于引入新的操作，从而将类型变得脆弱...



动机：

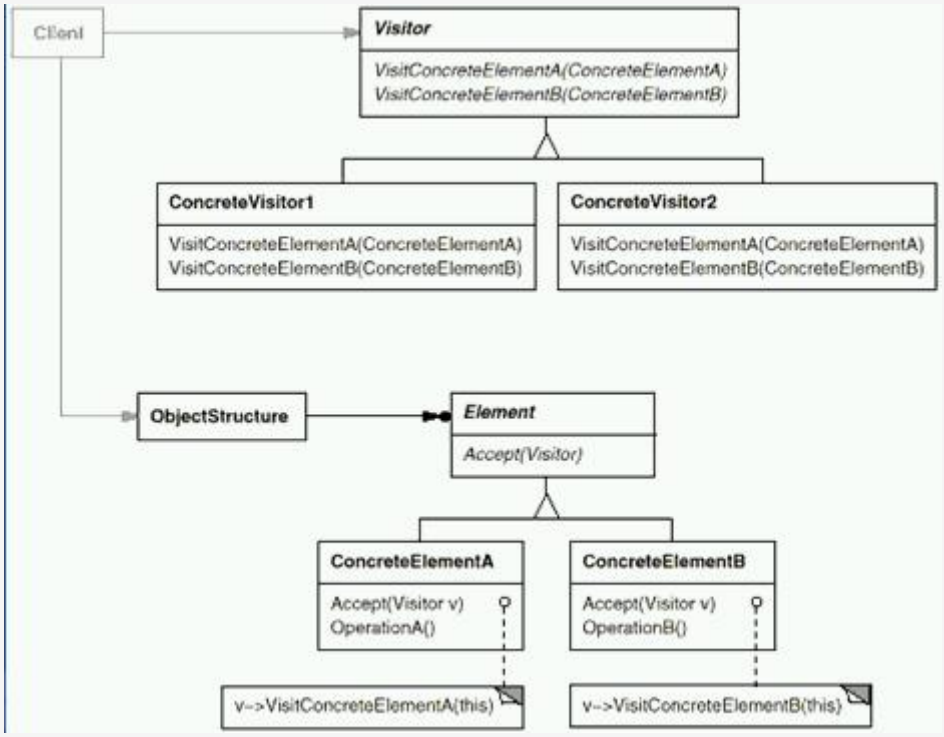
在软件构建过程中，由于需求的改变，某些类层次结构中常常需要增加新的行为(方法),如果直接在基类中做这样的更改，将会给子类带来很繁重的变更负担，甚至破坏原有设计。

如何在不更改类层次结构的前提下，在运行时根据需要透明地为类层次结构上的各个类动态添加新的操作，从而避免上述问题？

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引起元素的新操作。

结构：



适用性：

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相关的操作，而你想避免让这些操作"污染"这些对象的类。Visitor 使得你可以将相关的操作集中起来定义在一个类中。当该对象结构被很多应用共享时，用 Visitor 模式让每个应用仅包含需要用到的操作。

3.定义对象结构的类很少改变，但经常需要在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。

代码实现：

```
1 // MainApp startup application
2
3  class MainApp
4  {
5    static void Main()
6    {
7      // Setup employee collection
8      Employees e = new Employees();
9      e.Attach(new Clerk());
```

```
10      e.Attach(new Director());
11      e.Attach(new President());
12
13      // Employees are 'visited'
14      e.Accept(new IncomeVisitor());
15      e.Accept(new VacationVisitor());
16
17      // Wait for user
18      Console.Read();
19    }
20  }
21
22  // "Visitor"
23
24  interface IVisitor
25  {
26    void Visit(Element element);
27  }
28
29  // "ConcreteVisitor1"
30
31  class IncomeVisitor : IVisitor
32  {
33    public void Visit(Element element)
34    {
35      Employee employee = element as Employee;
36
37      // Provide 10% pay raise
38      employee.Income *= 1.10;
39      Console.WriteLine("{0} {1}'s new income: {2:C}",
40        employee.GetType().Name, employee.Name,
41        employee.Income);
42    }
43  }
```

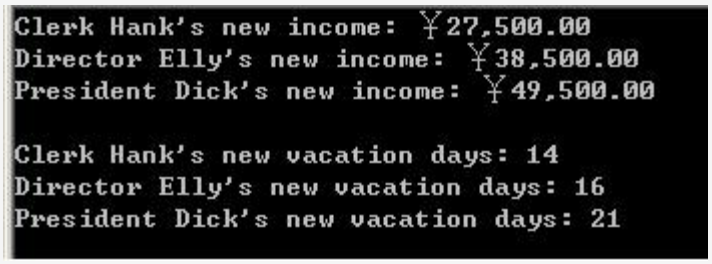
```
44
45  // "ConcreteVisitor2"
46
47  class VacationVisitor : IVisitor
48  {
49    public void Visit(Element element)
50    {
51      Employee employee = element as Employee;
52
53      // Provide 3 extra vacation days
54      Console.WriteLine("{0} {1}'s new vacation days: {2}",
55        employee.GetType().Name, employee.Name,
56        employee.VacationDays);
57    }
58  }
59
60  class Clerk : Employee
61  {
62    // Constructor
63    public Clerk() : base("Hank", 25000.0, 14)
64    {
65    }
66  }
67
68  class Director : Employee
69  {
70    // Constructor
71    public Director() : base("Elly", 35000.0, 16)
72    {
73    }
74  }
75
76  class President : Employee
77  {
```

```
78    // Constructor
79    public President() : base("Dick", 45000.0, 21)
80    {
81    }
82  }
83
84  // "Element"
85
86  abstract class Element
87  {
88    public abstract void Accept(IVisitor visitor);
89  }
90
91  // "ConcreteElement"
92
93  class Employee : Element
94  {
95    string name;
96    double income;
97    int vacationDays;
98
99    // Constructor
100   public Employee(string name, double income,
101     int vacationDays)
102   {
103     this.name = name;
104     this.income = income;
105     this.vacationDays = vacationDays;
106   }
107
108   // Properties
109   public string Name
110   {
111     get{ return name; }
```

```
112     set{ name = value; }
113   }
114
115   public double Income
116   {
117     get{ return income; }
118     set{ income = value; }
119   }
120
121   public int VacationDays
122   {
123     get{ return vacationDays; }
124     set{ vacationDays = value; }
125   }
126
127   public override void Accept(IVisitor visitor)
128   {
129     visitor.Visit(this);
130   }
131 }
132
133 // "ObjectStructure"
134
135 class Employees
136 {
137   private ArrayList employees = new ArrayList();
138
139   public void Attach(Employee employee)
140   {
141     employees.Add(employee);
142   }
143
144   public void Detach(Employee employee)
145   {
```

```
146      employees.Remove(employee);
147    }
148
149    public void Accept(IVisitor visitor)
150    {
151      foreach (Employee e in employees)
152      {
153        e.Accept(visitor);
154      }
155      Console.WriteLine();
156    }
157  }
```

运行结果：



Visoitr 模式的几个要点：

1.Visitor 模式通过所谓双重分发(double dispatch)来实现在不更改 Element 类层次结构的前提下，在运行时透明地为类层次结构上的各个类动态添加新的操作。

2.所谓双重分发却 Visotor 模式中间包括了两个多态分发（注意其中的多态机制);第一个为 accept 方法的多态辨析;第二个为 visitor 方法的多态辨析。

3.Visotor 模式的最大缺点在于扩展类层次结构(增添新的 Element 子类），会导致 Visitor 类的改变。因此 Visiotr 模式适用"Element"类层次结构稳定，而其中的操作却经常面临频繁改动".

观察者模式(Observer Pattern）

动机(Motivate):

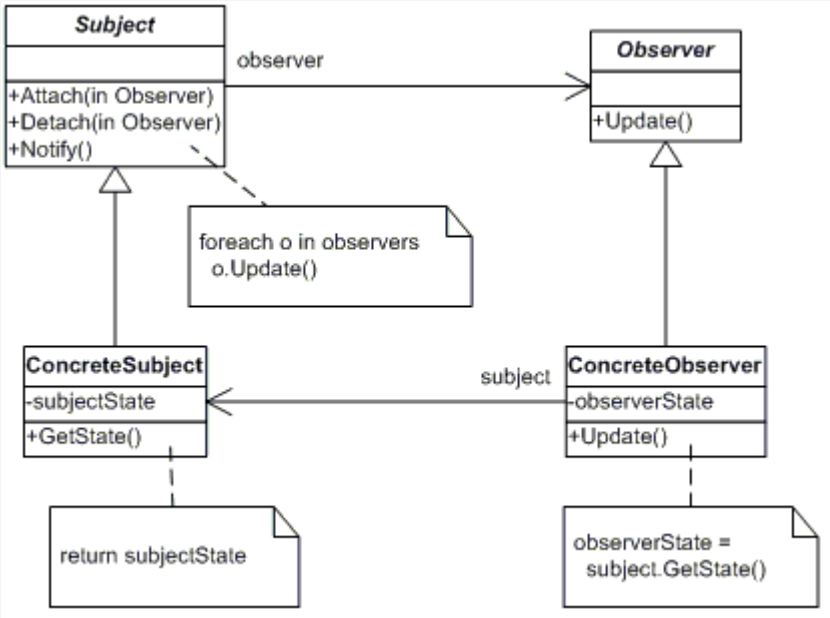
在软件构建过程中，我们需要为某些对象建立一种“通知依赖关系” ---------一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知。如果这样的依赖关系过于紧密，将使软件不能很好地抵御变化。使用面向对象技术，可以将这种依赖关系弱化，并形成一种稳定的依赖关系。从而实现软件体系结构的松耦合。

意图(Intent):

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

--------《设计模式》GOF

结构图(Struct):



适用性：

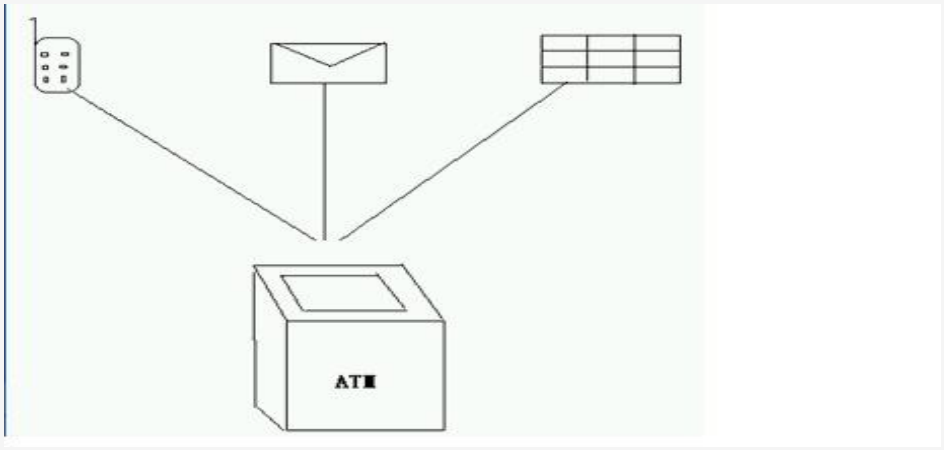
1．当一个抽象模型有两个方面，其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

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。

生活中的例子：

观察者定义了对象间一对多的关系，当一个对象的状态变化时，所有依赖它的对象都得到通知并且自动地更新。在 ATM 取款，当取款成功后，以手机、邮件等方式进行通知。



代码实现：

```
1   public class BankAccount
2   {
3       Emailer emailer;  //强信赖关系
4       Mobile  phoneNumber;    //强信赖关系
5
6       private double _money;
7
8       public Emailer Emailer
9       {
10          get { return emailer; }
11          set { this.emailer = value; }
12      }
13      public Mobile PhoneNumber
14      {
15          get { return phoneNumber; }
16          set { this.phoneNumber = value; }
17      }
18      public double Money
19      {
```

```
20          get { return _money; }
21          set { this._money = value; }
22      }
23
24      public void WithDraw()
25      {
26          emailer.SendEmail(this);
27          phoneNumber.SendNotification(this);
28      }
29
30  }
```

```
1   public class Emailer
2   {
3     private string _emailer;
4     public Emailer(string emailer)
5     {
6        this._emailer = emailer;
7     }
8      public void SendEmail(BankAccount ba)
9      {
10         //..
11         Console.WriteLine("Notified : Emailer is {0}, You withdraw  {1:C} ", _
emailer, ba.Money);
12     }
13  }
```

```
1   public class Mobile
2   {
3      private long _phoneNumber;
4      public Mobile(long phoneNumber)
5      {
6         this._phoneNumber = phoneNumber;
7      }
```

```
8       public void SendNotification(BankAccount ba)
9       {
10          Console.WriteLine("Notified :Phone number is {0} You withdraw  {1:C
} ", _phoneNumber, ba.Money);
11       }
12    }
```

此时简单的客户端调用如下：

```
1 class Test
2 {
3    static void Main(string[] args)
4    {
5       BankAccount ba = new BankAccount();
6       Emailer emailer = new Emailer("abcdwxc@163.com");
7       Mobile mobile = new Mobile(13901234567);
8       ba.Emailer = emailer;
9       ba.PhoneNumber = mobile;
10       ba.Money = 2000;
11       ba.WithDraw();
12    }
13 }
```

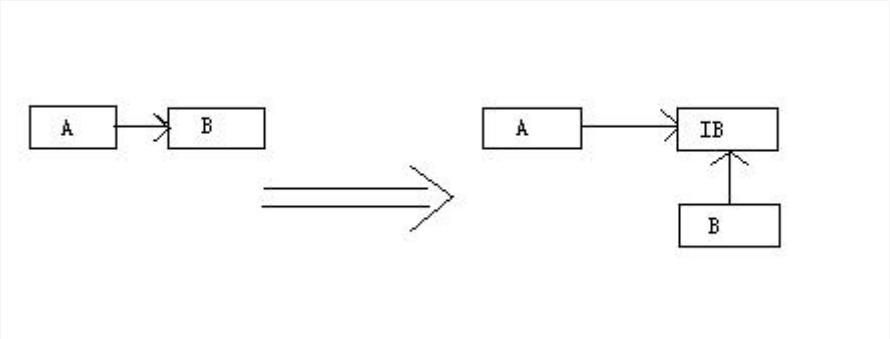
运行结果如下：

由此可见程序可以正常运行，但请注意 BandAccount 和 Emailer 及 Mobile 之间形成了一种双向的依赖关系，即 BankAccount 调用了 Emailer 及 Mobile 的方法，而 Emailer 及 Mobile 调用了 BnadAccount 类的属性。如果有其中一个类变化，有可能会引起另一个的变化。如果又需添加一种新的通知方式，就得在 BankAccount 的 WithDraw()方法中增加对该中通知方式的调用。

显然这样的设计极大的违背了“开放-封闭”原则，这不是我们所想要的，仅仅是新增加了一种通知对象，就需要对原有的 BankAccount 类进行修改，这样的设计是很糟糕的。对此做进一步

的抽象，既然出现了多个通知对象，我们就为这些对象之间抽象出一个接口，用它来取消 BankAccount 和具体的通知对象之间依赖。

由此我们由左图转换到右图。



实例代码如下：

```
1  public interface IObserverAccount
2      {
3          void Update(BankAccount ba);
4      }
```

```
1  public class BankAccount
2      {
3          IObserverAccount emailer;       //依赖于接口
4          IObserverAccount phoneNumber;    //依赖于接口
5
6          private double _money;
7
8          public IObserverAccount Emailer
9          {
10             get { return emailer; }
11             set { this.emailer = value; }
12         }
13         public IObserverAccount PhoneNumber
14         {
15             get { return phoneNumber; }
16             set { this.phoneNumber = value; }
17         }
18         public double Money
```

```
19          {
20              get { return _money; }
21              set { this._money = value; }
22          }
23
24          public void WithDraw()
25          {
26              emailer.Update(this);
27              phoneNumber.Update(this);
28          }
29
30      }
```

```
1     public class Emailer : IObserverAccount
2       {
3          private string _emailer;
4          public Emailer(string emailer)
5          {
6              this._emailer = emailer;
7          }
8          public void Update(BankAccount ba)
9          {
10             //..
11             Console.WriteLine("Notified : Emailer is {0}, You withdraw  {1:C} "
, _emailer, ba.Money);
12         }
13      }
```

```
1     public class Mobile : IObserverAccount
2       {
3          private long _phoneNumber;
4          public Mobile(long phoneNumber)
5          {
6              this._phoneNumber = phoneNumber;
```

```
7          }
8          public void Update(BankAccount ba)
9          {
10             Console.WriteLine("Notified :Phone number is {0} You withdraw  {
1:C} ", _phoneNumber, ba.Money);
11         }
12      }
```

客户端与上方相同，其运行结果也相同。但 BankAccount 增加和删除通知对象时，还需对其进行修改。对此我们再做如下重构，在 BankAccount 中维护一个 IObserver 列表，同时提供相应的维护方法。

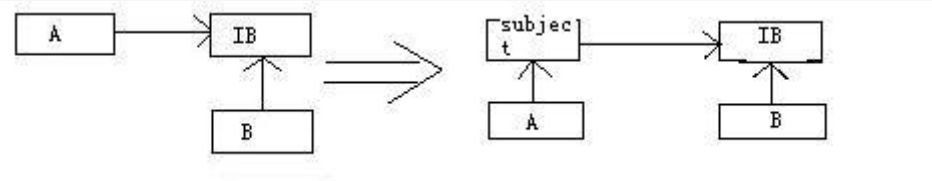
```
1   public class BankAccount
2   {
3      private List<IObserverAccount> Observers = new List<IObserverAccount
>();
4
5
6      private double _money;
7
8      public double Money
9      {
10        get { return _money; }
11        set { this._money = value; }
12     }
13
14     public void WithDraw()
15     {
16        foreach (IObserverAccount ob in Observers)
17        {
18           ob.Update(this);
19
20        }
21     }
22     public void AddObserver(IObserverAccount observer)
23     {
```

```
24          Observers.Add(observer);
25       }
26       public void RemoverObserver(IObserverAccount observer)
27       {
28          Observers.Remove(observer);
29       }
30
31    }
```

此时客户端代码如下：

```
1  class Test
2    {
3       static void Main(string[] args)
4       {
5          BankAccount ba = new BankAccount();
6          IObserverAccount emailer = new Emailer("abcdwxc@163.com");
7          IObserverAccount mobile = new Mobile(13901234567);
8
9          ba.Money = 2000;
10         ba.AddObserver(emailer);
11         ba.AddObserver(mobile);
12
13         ba.WithDraw();
14      }
15   }
```

走到这一步，已经有了 Observer 模式的影子了，BankAccount 类不再依赖于具体的 Emailer 或 Mobile，而是依赖于抽象的 IObserverAccount。存在着的一个问题是 Emailer 或 Mobile 仍然依赖于具体的 BankAccount，解决这样的问题很简单，只需要再对 BankAccount 类做一次抽象。如下图：




```
1  public abstract class Subject
2    {
```

```
3       private List<IObserverAccount> Observers = new List<IObserverAccount
>();
4
5       private double _money;
6       public Subject(double money)
7       {
8          this._money = money;
9       }
10
11       public double Money
12       {
13          get { return _money; }
14       }
15
16       public void WithDraw()
17       {
18          foreach (IObserverAccount ob in Observers)
19          {
20             ob.Update(this);
21
22          }
23       }
24       public void AddObserver(IObserverAccount observer)
25       {
26          Observers.Add(observer);
27       }
28       public void RemoverObserver(IObserverAccount observer)
29       {
30          Observers.Remove(observer);
31       }
32
33    }
```

```
public interface IObserverAccount
{
    void Update(Subject subject);
}
```

```
public class BankAccount : Subject
{
    public BankAccount(double money)
        : base(money)
    { }

}
```

```
 public class Emailer : IObserverAccount
 {
     private string _emalier;
     public Emailer(string emailer )
     {
         this._emalier = emailer;
     }
     public void Update(Subject subject)
     {
         Console.WriteLine("Notified : Emailer is {0}, You withdraw  {1:C} ", _
emalier, subject.Money);
     }
 }
```

```
public class Mobile : IObserverAccount
 {
     private long _phoneNumber;
     public Mobile(long phoneNumber)
     {
         this._phoneNumber = phoneNumber;
     }
```

```
8        public void Update(Subject subject)
9        {
10           Console.WriteLine("Notified :Phone number is {0} You withdraw  {1:C
} ", _phoneNumber, subject.Money);
11       }
12    }
```

此时客户端实现如下：

```
1   class Test
2    {
3       static void Main(string[] args)
4       {
5          Subject subject = new BankAccount(2000);
6          subject.AddObserver(new Emailer("abcdwxc@163.com"));
7          subject.AddObserver(new Mobile(13901234567));
8
9          subject.WithDraw();
10      }
11   }
```

推模式与拉模式

对于发布-订阅模型，大家都很容易能想到推模式与拉模式，用 SQL Server 做过数据库复制的朋友对这一点很清楚。在 Observer 模式中同样区分推模式和拉模式，我先简单的解释一下两者的区别：推模式是当有消息时，把消息信息以参数的形式传递（推）给所有观察者，而拉模式是当有消息时，通知消息的方法本身并不带任何的参数，是由观察者自己到主体对象那儿取回（拉）消息。知道了这一点，大家可能很容易发现上面我所举的例子其实是一种推模式的 Observer 模式。我们先看看这种模式带来了什么好处：当有消息时，所有的观察者都会直接得到全部的消息，并进行相应的处理程序，与主体对象没什么关系，两者之间的关系是一种松散耦合。但是它也有缺陷，第一是所有的观察者得到的消息是一样的，也许有些信息对某个观察者来说根本就用不上，也就是观察者不能“按需所取”；第二，当通知消息的参数有变化时，所有的观察者对象都要变化。鉴于以上问题，拉模式就应运而生了，它是由观察者自己主动去取消息，需要什么信息，就可以取什么，不会像推模式那样得到所有的消息参数。

拉模式实现如下：

```
1 public abstract class Subject
2    {
```

```
3       private List<IObserverAccount> Observers = new List<IObserverAccount
>();
4
5
6       private double _money;
7
8       public double Money
9       {
10          get { return _money; }
11      }
12      public Subject(double money)
13      {
14          this._money = money;
15      }
16      public void WithDraw()
17      {
18          foreach (IObserverAccount ob in Observers)
19          {
20              ob.Update();
21
22          }
23      }
24      public void AddObserver(IObserverAccount observer)
25      {
26          Observers.Add(observer);
27      }
28      public void RemoverObserver(IObserverAccount observer)
29      {
30          Observers.Remove(observer);
31      }
32
33  }
```

```
1   public interface IObserverAccount
2    {
3       void Update();
4    }
```

```
1    public class BankAccount :Subject
2    {
3       public BankAccount(double money)
4          : base(money)
5       { }
6
7    }
```

```
1    public class Emailer : IObserverAccount
2    {
3       private string _emalier;
4       private Subject _subject;
5       public Emailer(string emailer,Subject subject)
6       {
7          this._emalier = emailer;
8          this._subject = subject;
9       }
10       public void Update()
11       {
12          //..
13          Console.WriteLine("Notified : Emailer is {0}, You withdraw  {1:C} ", _
emalier,_subject.Money);
14       }
15    }
```

```
1    public class Mobile : IObserverAccount
2    {
3       private long _phoneNumber;
4       private Subject _subject;
```

```
5       public Mobile(long phoneNumber,Subject subject)
6       {
7          this._phoneNumber = phoneNumber;
8          this._subject = subject;
9       }
10       public void Update()
11       {
12          Console.WriteLine("Notified :Phone number is {0} You withdraw  {1:C
} ", _phoneNumber,_subject.Money);
13       }
14    }
```

此时客户端调用如下：

```
1   class Test
2    {
3       static void Main(string[] args)
4       {
5         Subject subject= new BankAccount(2000);
6          subject.AddObserver(new Emailer("abcdwxc@163.com",subject));
7          subject.AddObserver(new Mobile(13901234567,subject));
8
9          subject.WithDraw();
10       }
11    }
```

.NET 中 Observer 实现：

用事件和委托来实现 Observer 模式我认为更加的简单和优雅，也是一种更好的解决方案。

```
1  public  class Subject
2    {
3       public event NotifyEventHandler NotifyEvent;
4
5       private double _money;
6       public Subject(double money)
7       {
8          this._money = money;
```

```
9       }
10
11      public double Money
12      {
13          get { return _money; }
14      }
15
16      public void WithDraw()
17      {
18          OnNotifyChange();
19      }
20      public void OnNotifyChange()
21      {
22          if (NotifyEvent != null)
23          {
24              NotifyEvent(this);
25          }
26
27      }
28
29  }
```

```
1   public class Emailer
2   {
3       private string _emalier;
4       public Emailer(string emailer)
5       {
6           this._emalier = emailer;
7       }
8       public void Update(object obj)
9       {
10          if (obj is Subject)
11          {
12              Subject subject = (Subject)obj;
```

```
13
14              Console.WriteLine("Notified : Emailer is {0}, You withdraw  {1:C} "
, _emalier, subject.Money);
15          }
16      }
17 }
public delegate void NotifyEventHandler(object sender);
```

客户端调用如下：

```
1   class Test
2       {
3          static void Main(string[] args)
4          {
5             Subject subject = new Subject(2000);
6             Emailer emailer = new Emailer("abcdwxc@163.com");
7             subject.NotifyEvent += new NotifyEventHandler(emailer.Update);
8
9
10             subject.WithDraw();
11          }
12      }
```

Observer 实现要点：

1. 使用面向对象的抽象，Observer 模式使得我们可以独立地改变目标与观察者，从而使二者之间的依赖关系达到松耦合。

2. 目标发送通知时，无需指定观察者，通知（可以携带通知信息作为参数）会自动传播。观察者自己决定是否需要订阅通知。目标对象对此一无所知。

3. 在 C#中的 Event。委托充当了抽象的 Observer 接口，而提供事件的对象充当了目标对象，委托是比抽象 Observer 接口更为松耦合的设计。

解释器模式(Interpreter Pattern)

动机(Motivate):

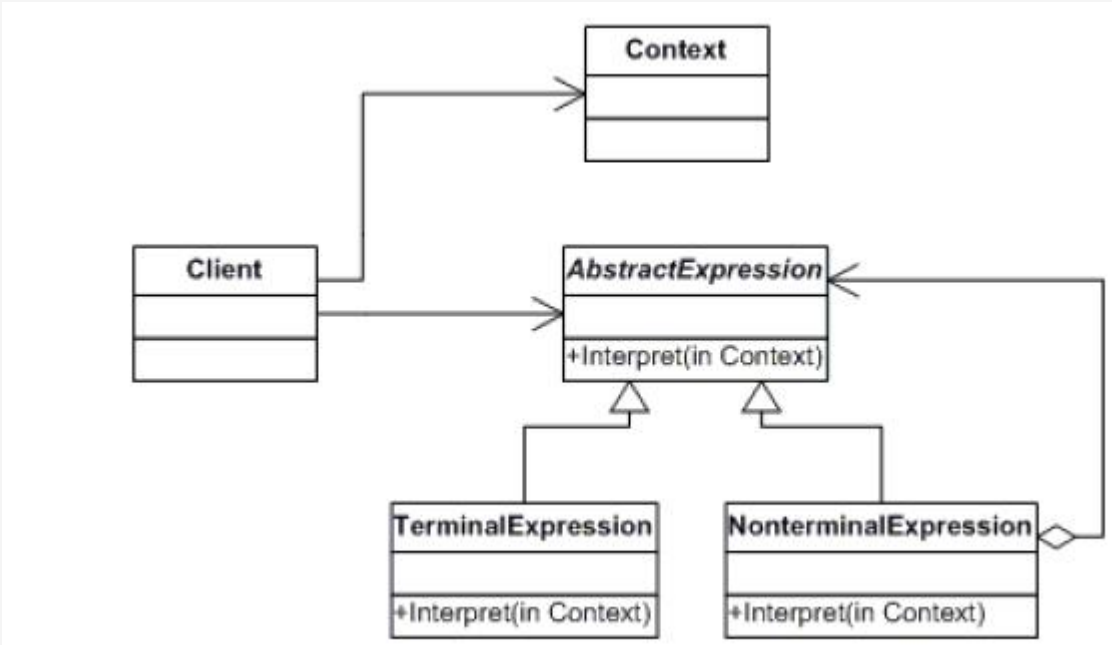
在软件构建过程中，如果某一特定领域的问题比较复杂，类似的模式不断重复出现，如果使用普通的编程方式来实现将面临非常频繁的变化。

在这种情况下，将特定领域的问题表达为某种文法规则下的句子，然后构建一个解释器来解释这样的句子，从而达到解决问题的目的。

意图(Intent):

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。

结构图(Struct):



生活中的例子：

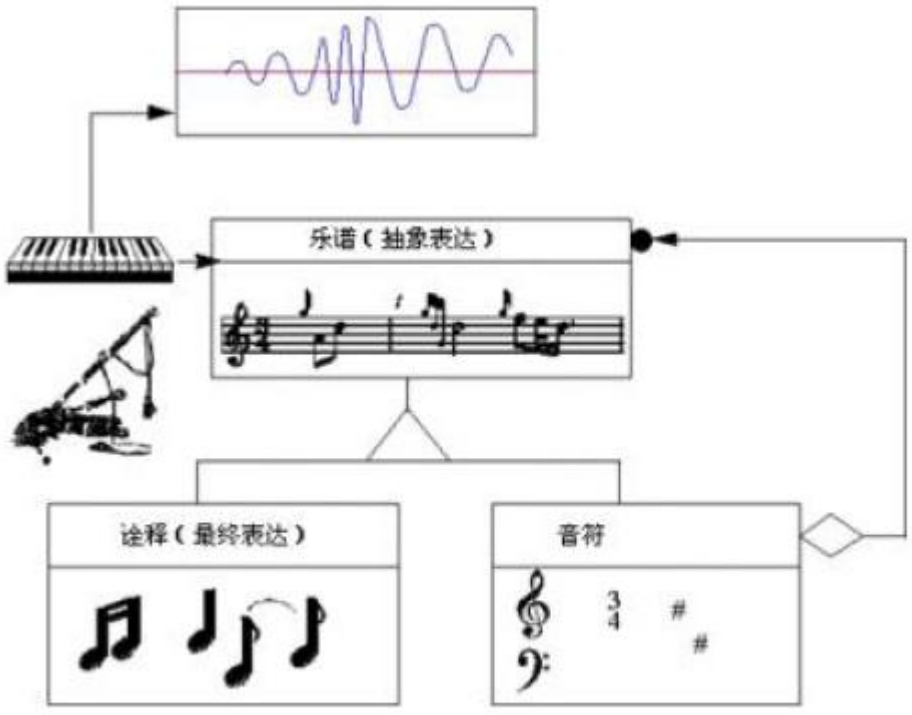


图 15：使用音乐例子的解释器模式对象图

适用性：

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树时，可使用解释器模式。

而当存在以下情况时该模式效果最好：

2.该文法简单对于复杂的文法，文法的类层次变得庞大而无法管理。此时语法分析程序生成器这样的工具是更好的选择。它们无需构建抽象语法树即可解释表达工，这样可以节省空间而且还可能节省时间。

3.效率不是一个关键问题最高效的解释器通常不是通过直接解释语法分析树实现的，而是首先将它们转换成另一种

形式。例如：正则表达式通常被转换成状态机。但即使在这种情况下，转换器仍可用解释器模式实现，该模式仍

是有用的。

代码实现：

客户端代码如下：

```
1    class Program
2    {
3        static void Main(string[] args)
4        {
```

```
5          string roman = "五千四百三十二"; //5432
6          Context context = new Context(roman);
7
8          //Build the 'parse tree'
9          ArrayList tree = new ArrayList();
10          tree.Add(new OneExpression());
11          tree.Add(new TenExpression());
12          tree.Add(new HundredExpression());
13          tree.Add(new ThousandExpression());
14
15          //Interpret
16          foreach (Expression exp in tree)
17          {
18             exp.Interpret(context);
19          }
20          Console.WriteLine("{0} = {1}", roman, context.Data);
21          //Wait for user
22          Console.Read();
23       }
24    }
```

创建一个抽象类 Expression，来描述共同的操作。

```
1    public abstract class Expression
2    {
3       protected Dictionary<string, int> table = new Dictionary<string, int>(9);
4       public Expression()
5       {
6          table.Add("一", 1);
7          table.Add("二", 2);
8          table.Add("三", 3);
9          table.Add("四", 4);
10          table.Add("五", 5);
11          table.Add("六", 6);
12          table.Add("七", 7);
13          table.Add("八", 8);
```

```
14          table.Add("九", 9);
15      }
16      public virtual void Interpret(Context context)
17      {
18       if(context.Statement.Length==0)
19          {
20           return;
21          }
22      foreach(string key in table.Keys)
23      {
24       int value=table[key];
25       if(context.Statement.EndsWith(key + GetPostifix()))
26          {
27           context.Data +=value*Multiplier();
28           context.Statement = context.Statement.Substring(0,context.Statem
ent.Length- this.GetLength());
29          }
30
31          if(context.Statement.EndsWith("零"))
32          {
33             context.Statement = context.Statement.Substring(0, context.State
ment.Length - 1);
34          }
35          if (context.Statement.Length == 0)
36          {
37             return;
38          }
39       }
40      }
41
42      public abstract string GetPostifix();
43      public abstract int Multiplier();
44      public virtual int GetLength()
45      {
```

```
46          return this.GetPostifix().Length + 1;
47       }
48    }
```

然后创建一个公共类 Context,定义一些全局信息。

```
1   public class Context
2    {
3       private string statement;
4       private int data;
5
6       //Constructor
7       public Context(string statement)
8       {
9          this.statement = statement;
10      }
11      //Properties
12      public string Statement
13      {
14         get { return statement; }
15         set { statement = value; }
16      }
17      public int  Data
18      {
19         get { return data; }
20         set { data  = value; }
21      }
22   }
```

```
1   public class OneExpression：Expression
2   {
3      public override string GetPostifix()
4      {
5         return "";
6      }
7      public override int Multiplier() { return 1; }
```

```
8       public override int GetLength()
9       {
10          return 1;
11      }
12  }
13  public class TenExpression：Expression
14  {
15      public override string GetPostifix()
16      {
17          return "十";
18      }
19      public override int Multiplier() { return 10; }
20      public override int GetLength()
21      {
22          return 2;
23      }
24  }
25  public class HundredExpression：Expression
26  {
27      public override string GetPostifix()
28      {
29          return "百";
30      }
31      public override int Multiplier() { return 100; }
32      public override int GetLength()
33      {
34          return 2;
35      }
36  }
37  public class ThousandExpression：Expression
38  {
39      public override string GetPostifix()
40      {
41          return "千";
```

```
42        }
43        public override int Multiplier() { return 1000; }
44        public override int GetLength()
45        {
46            return 2;
47        }
48    }
```

Interpreter 实现要点：

Interpreter 模式的应用场合是 interpreter 模式应用中的难点，只有满足"业务规则频繁变化，且类似的模式不断重复出现，并且容易抽象为语法规则的问题"才适合使用 Interpreter 模式。

使用 Interpreter 模式来表示文法规则，从而可以使用面向对象技巧来方便地“扩展”文法。

Interpreter 模式比较适合简单的文法表示，对于复杂的文法表示，Interpreter 模式会产生比较大的类层次结构，需要求助于语法分析生成器这样的标准工具。

职责链模式(Chain of Responsibility Pattern)

动机(Motivate)：

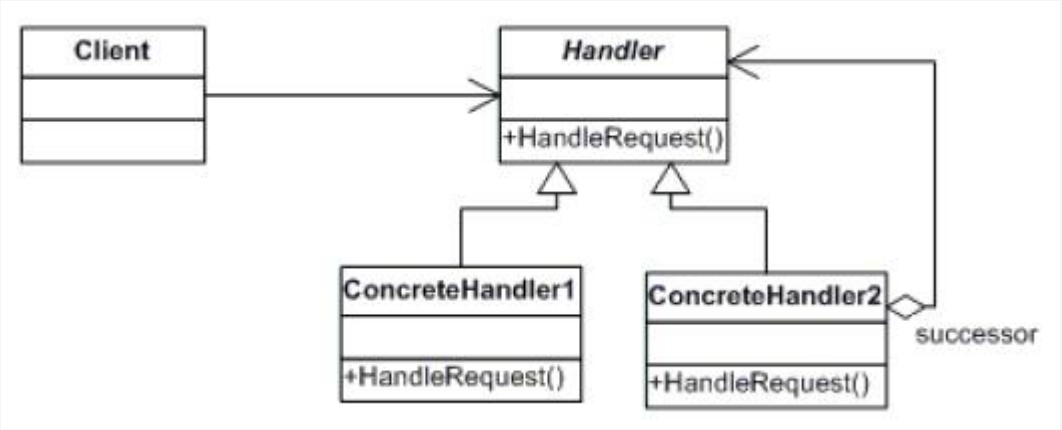
在软件构建过程中，一个请求可能被多个对象处理，但是每个请求在运行时只能有一个接受者，如果显示指定，将必不可少地带来请求发送者与接受者的紧耦合。

如何使请求的发送者不需要指定具体的接受者？让请求的接受者自己在运行时决定来处理请求，从而使两者解耦。

意图(Intent)：

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。

结构图(Struct)：



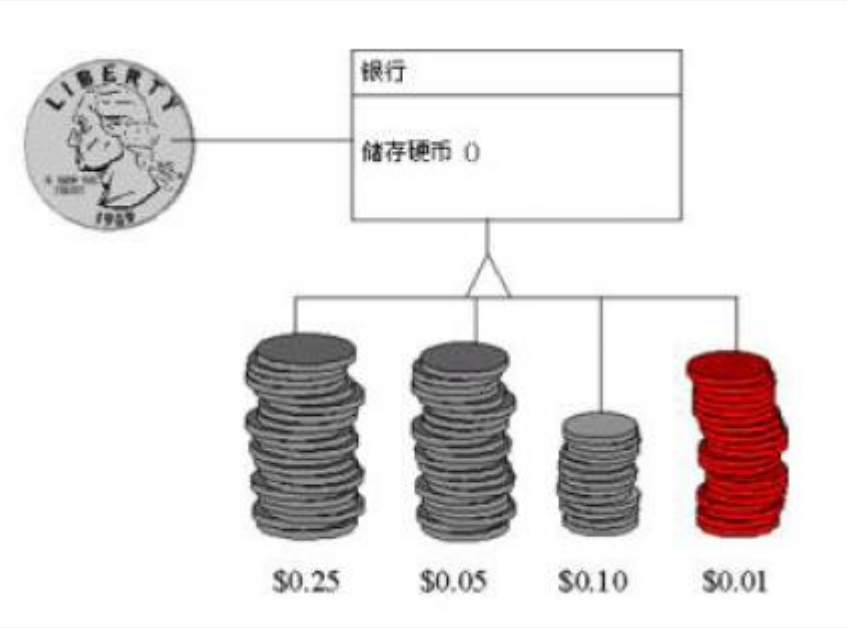
适用性：

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。

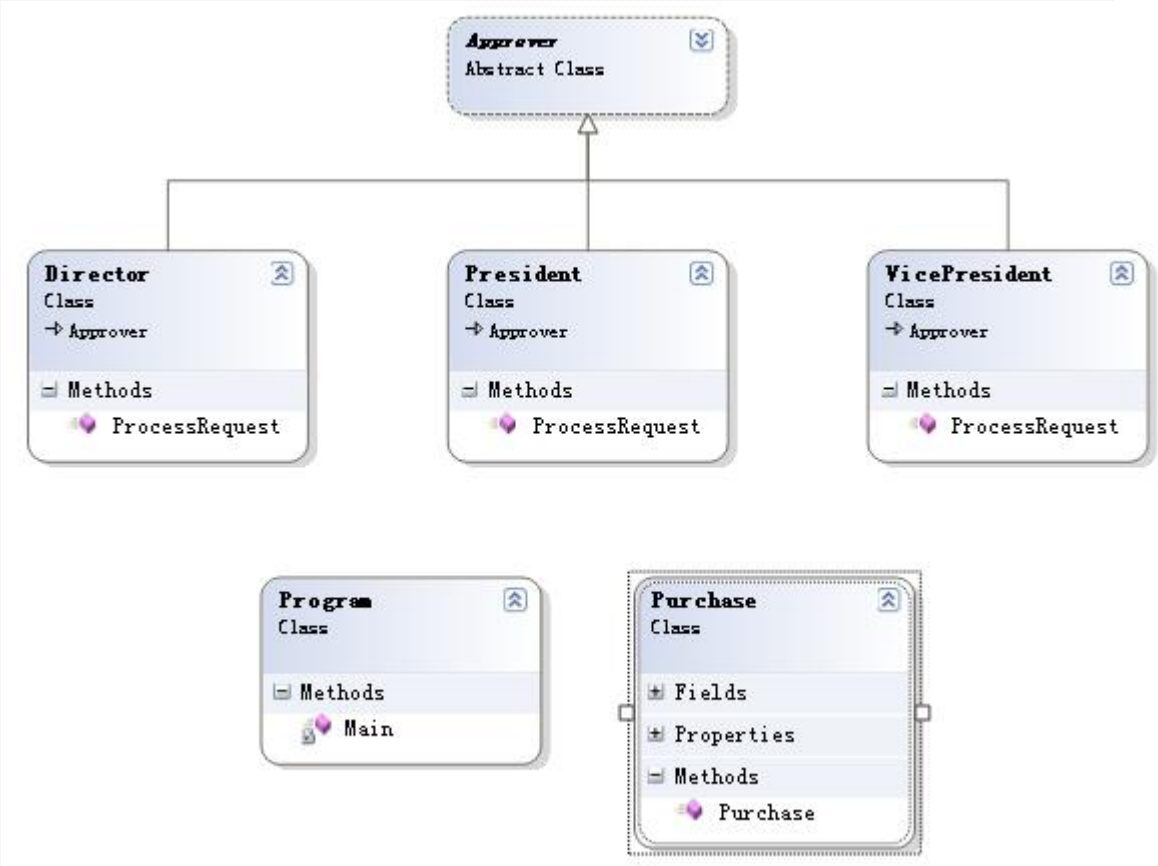
2.你想在不明确接收者的情况下，向多个对象中的一个提交一个请求。

3.可处理一个请求的对象集合应被动态指定。

生活中的例子：



代码实现：




```
1    //Handler
2   abstract class Approver
3    {
```

```
4       protected Approver successor;
5       public void SetSuccessor(Approver successor)
6       {
7          this.successor = successor;
8       }
9       public abstract void ProcessRequest(Purchase purchase);
10
11    }
12
13
14
```

```
1     //ConcreteHandler
2     class Director :Approver
3     {
4        public override void ProcessRequest(Purchase purchase)
5        {
6           if (purchase.Amount < 10000.0)
7           {
8              Console.WriteLine("{0} approved request# {1}", this.GetType().Na
me, purchase.Number);
9
10          }
11          else if(successor !=null)
12          {
13             successor.ProcessRequest(purchase);
14          }
15       }
16    }
```

```
1
2
3
4     class VicePresident :Approver
```

```
5    {
6        public override void ProcessRequest(Purchase purchase)
7        {
8            if (purchase.Amount < 25000.0)
9            {
10               Console.WriteLine("{0} approved request# {1}", this.GetType().N
ame, purchase.Number);
11
12           }
13           else if (successor != null)
14           {
15               successor.ProcessRequest(purchase);
16           }
17       }
18   }
```

```
1
2    class President :Approver
3    {
4        public override void ProcessRequest(Purchase purchase)
5        {
6            if (purchase.Amount < 100000.0)
7            {
8                Console.WriteLine("{0} approved request# {1}", this.GetType().Na
me, purchase.Number);
9
10           }
11           else
12           {
13               Console.WriteLine("Request! {0} requires an executive meeting!",
purchase.Number);
14           }
15       }
16   }
```

```
1
2
3   //Request details
4   class Purchase
5   {
6       private int number;
7       private double amount;
8       private string purpose;
9
10      //Constructor
11      public Purchase(int number, double amount, string purpose)
12      {
13          this.number = number;
14          this.amount = amount;
15          this.purpose = purpose;
16      }
17      //Properties
18      public double Amount
19      {
20          get { return amount; }
21          set { amount = value; }
22      }
23      public string Purpose
24      {
25          get { return purpose; }
26          set { purpose = value; }
27      }
28      public int Number
29      {
30          get { return number; }
31          set { number = value; }
32      }
33  }
```

客户端调用如下：

```
1
2   class Program
3   {
4      static void Main(string[] args)
5      {
6         //Setup Chain of Responsibility
7         Director Larry = new Director();
8         VicePresident Sam = new VicePresident();
9         President Tammy = new President();
10         Larry.SetSuccessor(Sam);
11         Sam.SetSuccessor(Tammy);
12
13         //Generate and process purchase requests
14         Purchase p = new Purchase(1034, 350.00, "Supplies");
15         Larry.ProcessRequest(p);
16
17         p = new Purchase(2035, 32590.10, "Project X");
18         Larry.ProcessRequest(p);
19
20         p = new Purchase(2036, 122100.00, "Project Y");
21         Larry.ProcessRequest(p);
22
23         //Wait for user
24         Console.Read();
25      }
26   }
```

运行结果如下：

Chain of Responsibility 实现要点：

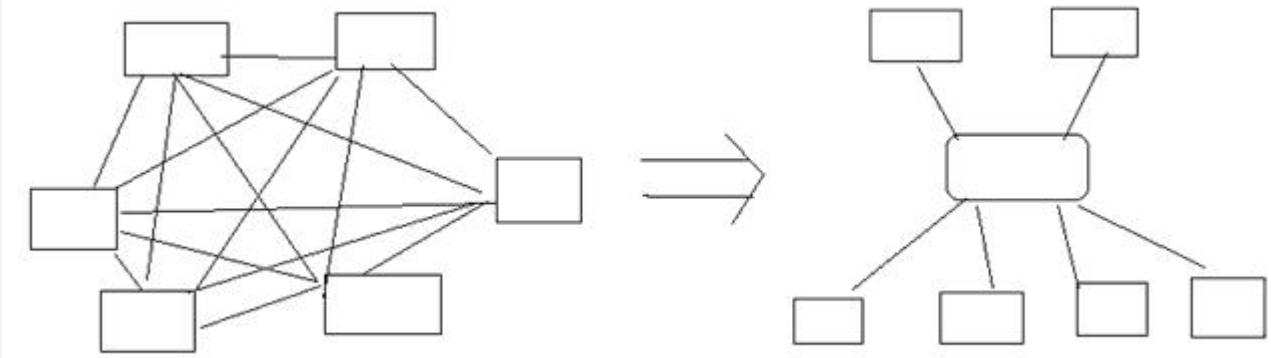
1.Chain of Responsibility 模式的应用场合在于“一个请求可能有多个接受者，但是最后真正的接受者只胡一个”，只有这时候请求发送者与接受者的耦合才胡可能出现“变化脆弱”的症状，职责链的目的就是将二者解耦，从而更好地应对变化。

2.应用了 Chain of Responsibility 模式后，对象的职责分派将更具灵活性。我们可以在运行时动态添加/修改请求的处理职责。

3.如果请求传递到职责链的末尾仍得不到处理，应该有一个合理的缺省机制。这也是每一个接受对象的责任，而不是发出请求的对象的责任。

中介者模式(Mediator Pattern)

依赖关系的转化:



动机(Motivate):

在软件构建过程中，经常会出现多个对象互相关联交互的情况，对象之间常常会维持一种复杂的引用关系，如果遇到一些需求的更改，这种直接的引用关系将面临不断的变化。

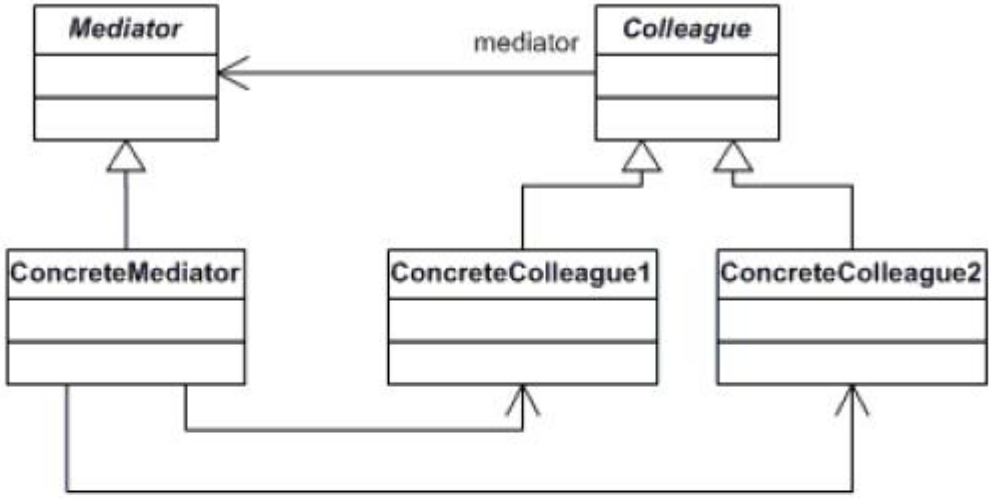
在这种情况下，我们可使用一个“中介对象”来管理对象间的关联关系，避免相互交互的对象之间的紧耦合引用关系，从而更好地抵御变化。

意图(Intent)：

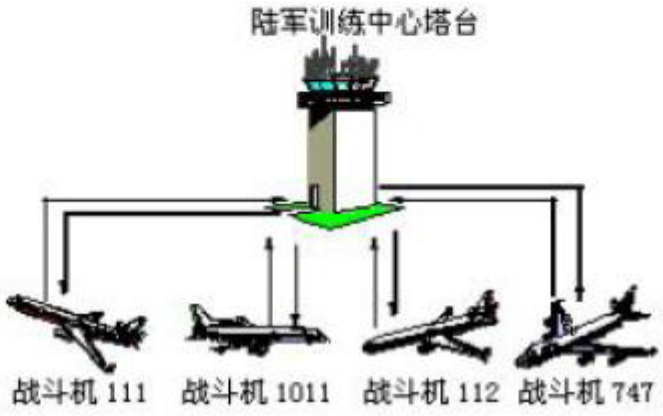
用一个中介对象来封装一系列对象交互。中介者使各对象不需要相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交

互。 ------《设计模式》GOF

结构图(Struct):



生活例子



适用性：

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，导致难以复用该对象。

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。

代码实现：

```
1    //Mediator
2   abstract class AbstractChatroom
3    {
4      public abstract void Register(Participant participant);
5      public abstract void Send(string from, string to, string message);
6    }
```

```
1   //ConcreteMediator
2   class Chatroom :AbstractChatroom
3   {
4      private Hashtable participants = new Hashtable();
5      public override void Register(Participant participant)
6      {
7         if (participants[participant.Name] == null)
8         {
9            participants[participant.Name]=participant;
10        }
11        participant.Chatroom = this;
12     }
13     public override void Send(string from, string to, string message)
14     {
15        Participant pto = (Participant)participants[to];
16        if (pto != null)
17        {
18           pto.Receive(from, message);
19        }
20     }
21  }
```

```
1   //AbstractColleague
2   class Participant
3   {
4      private Chatroom chatroom;
5      private string name;
6
7      //Constructor
8      public Participant(string name)
9      {
10        this.name = name;
11     }
```

```
12      //Properties
13      public string Name
14      {
15          get { return name; }
16      }
17      public Chatroom Chatroom
18      {
19          set { chatroom = value; }
20          get { return chatroom; }
21
22      }
23      public void Send(string to, string message)
24      {
25          chatroom.Send(name, to, message);
26      }
27      public virtual void Receive(string from, string message)
28      {
29          Console.WriteLine("{0} to {1}:'{2}'", from, name, message);
30      }
31  }
```

```
1   //ConcreteColleaguel
2   class Beatle :Participant
3   {
4    //Constructor
5       public Beatle(string name)
6           : base(name)
7       { }
8       public override void Receive(string from, string message)
9       {
10          Console.Write("To a Beatle: ");
11          base.Receive(from, message);
12      }
13  }
```

```
1   //ConcreteColleague2
2   class NonBeatle :Participant
3   {
4      //Constructor
5      public NonBeatle(string name)
6         : base(name)
7      { }
8      public override void Receive(string from, string message)
9      {
10        Console.Write("To a non-Beatle:");
11        base.Receive(from, message);
12     }
13  }
```

客户端调用如下：

```
1 static void Main(string[] args)
2      {
3         //create chatroom
4         Chatroom chatroom = new Chatroom();
5         //Create participants and register them
6         Participant George = new Beatle("George");
7         Participant Paul = new Beatle("Paul");
8         Participant Ringo = new Beatle("Ringo");
9         Participant John = new Beatle("John");
10        Participant Yoko = new Beatle("Yoko");
11        chatroom.Register(George);
12        chatroom.Register(Paul);
13        chatroom.Register(Ringo);
14        chatroom.Register(John);
15        chatroom.Register(Yoko);
16
17        //chatting participants
18        Yoko.Send("John", "Hi John");
19        Paul.Send("Ringo", "All you need is love");
```

```
20          Ringo.Send("George", "My sweet Lord");
21          Paul.Send("John", "Can't buy me love");
22          John.Send("Yoko", "My sweet love");
23      }
```

运行结果如下:

Mediator 实现要点：

1.将多个对象间复杂的关联关系解耦，Mediator 模式将多个对象间的控制逻辑进行集中管理，变“多个对象互相关系”为多“个对象和一个中介者关联”，简化了系统的维护，抵御了可能的变化。

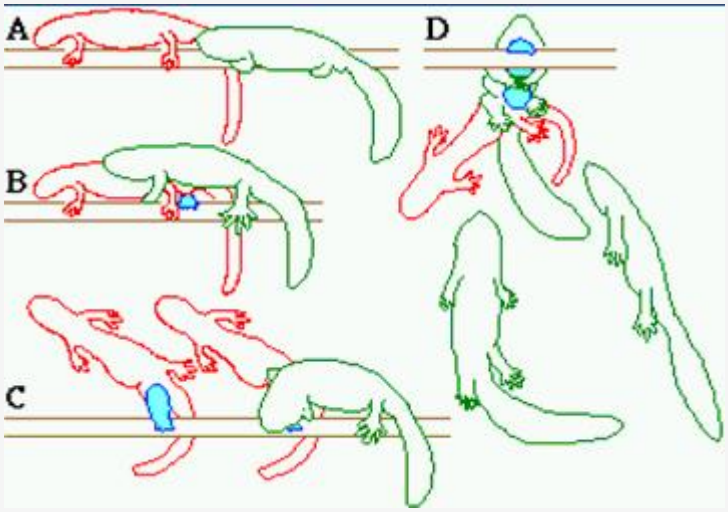
2.随着控制逻辑的复杂化，Mediator 具体对象的实现可能相当复杂。 这时候可以对 Mediator 对象进行分解处理。

3.Facade 模式是解耦系统外到系统内(单向)的对象关系关系;Mediator 模式是解耦系统内各个对象之间(双向)的关联关系

状态模式(State Pattern)

对象状态影响对象行为：

对象拥有不同的状态，往往会行使不同的行为...



动机：

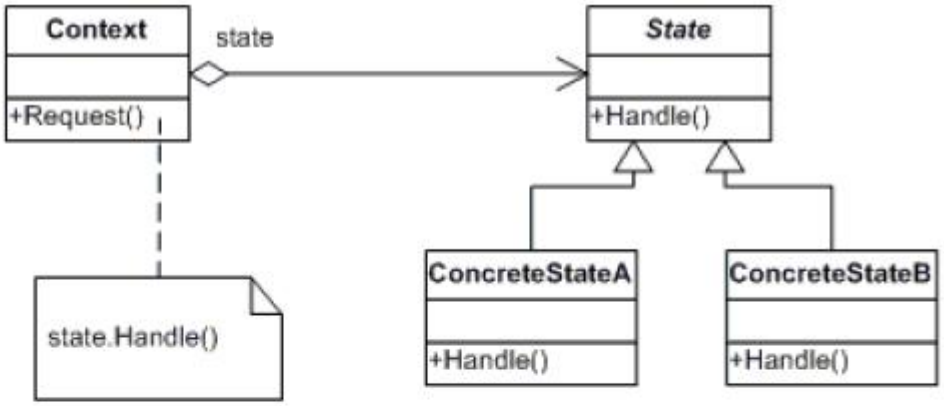
在软件构建过程中，某些对象的状态如果改变以及其行为也会随之而发生变化，比如文档处于只读状态，其支持的行为和读写状态支持的行为就可能完全不同。

如何在运行时根据对象的状态来透明更改对象的行为？而不会为对象操作和状态转化之间引入紧耦合？

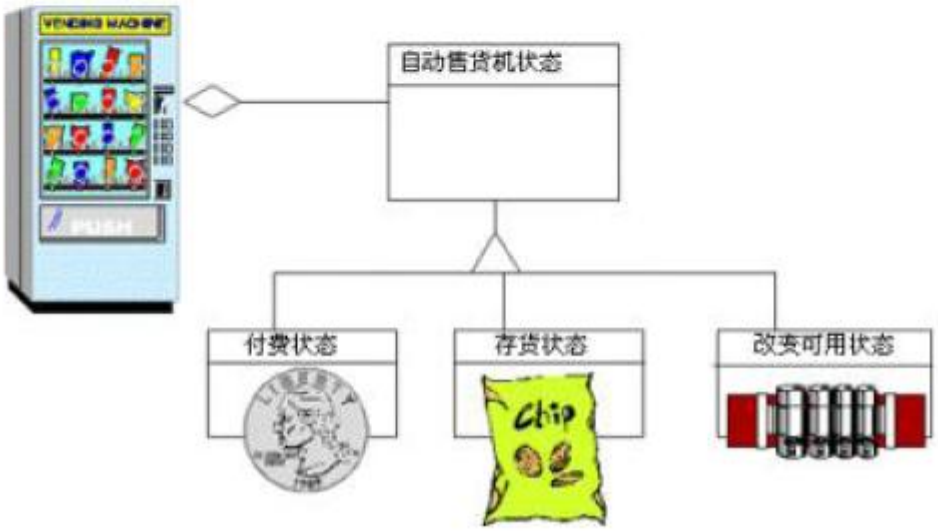
意图：

允许一个对象在其内部状态改变时改变它的行为。从而使对象看起来似乎修改了其行为。 ------《设计模式》GOF

结构图：



生活例子



适用性：

1.一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为。

2.一个操作中含有庞大的多分支的等条件语句，且这些分支依赖于该对象的状态。这个状态通常用一个或多个枚举常量表示。通常，有多个操作包含这一相同的条件结构。State 模式将每一个分支放入一个独立的类中。这使得你可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。

代码实现：

```
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);
  }
}
```

结果：

```
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
```

State 模式的几个要点：

1.State 模式将所有一个特定状态相关的行为都放入一个 State 的子类对象中，在对象状态切换时，切换相应的对象;但同时维持 State 的接口，这样实现了具体操作与状态转换之间的解耦。

2.为不同的状态引入不同的对象使得状态转换变得更加明确，而且可以保证不会出现状态不一致的情况，因为转换是原子性的----即要么彻底转换过来，要么不转换。

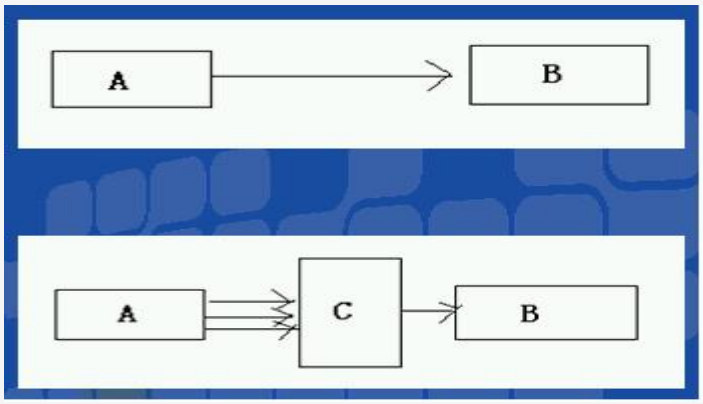
3.如果 State 对象没有实例变量，那么各个上下文可以共享 同一个 State 对象，从而节省对象开销。

代理模式(Proxy Pattern)

直接与间接：

人们对复杂的软件系统常有一种处理手法，即增加一层间接层，从而对系统获得一种更为灵活、

满足特定需求的解决方案。



动机(Motivate)：

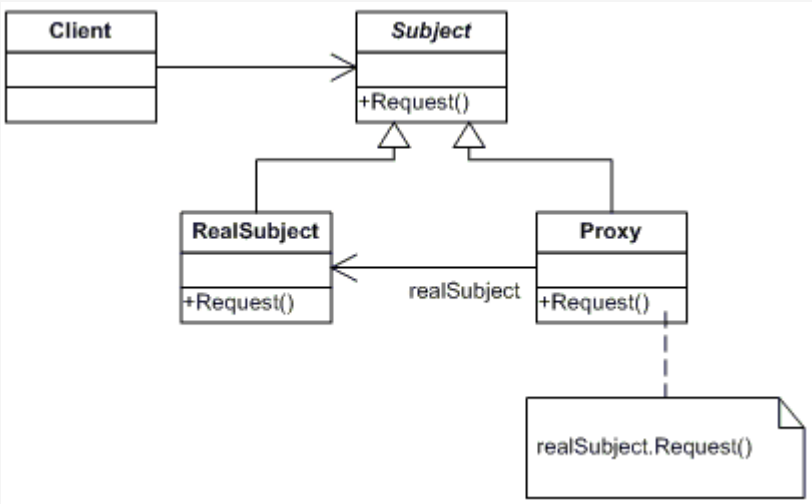
在面向对象系统中，有些对象由于某种原因（比如对象创建的开销很大，或者某些操作需要安全控制，或者需要进程外的访问等），直接访问会给使用者、或者系统结构带来很多麻烦。

如何在不失去透明操作对象的同时来管理/控制这些对象特有的复杂性？增加一层间接层是软件开发中常见的解决方式。

意图(Intent)：

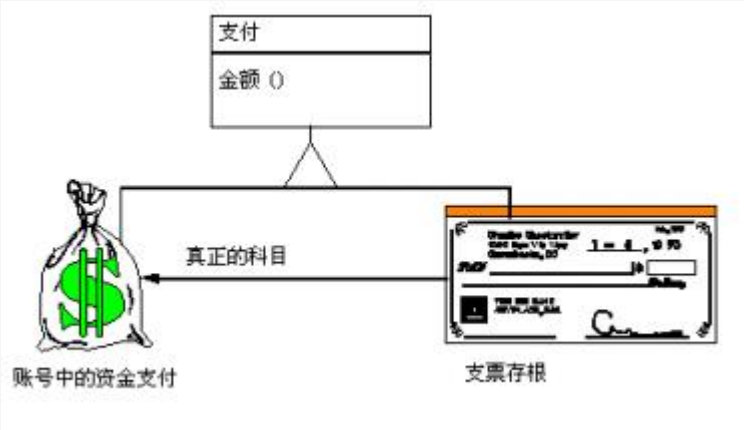
为其他对象提供一种代理以控制对这个对象的访问。 --------《设计模式》GOF

结构图(Struct)：



生活中的例子：

代理模式提供一个中介以控制对这个对象的访问。一张支票或银行存单是账户中资金的代理。支票在市场交易中用来代替现金，并提供对签发人账号上资金的控制。



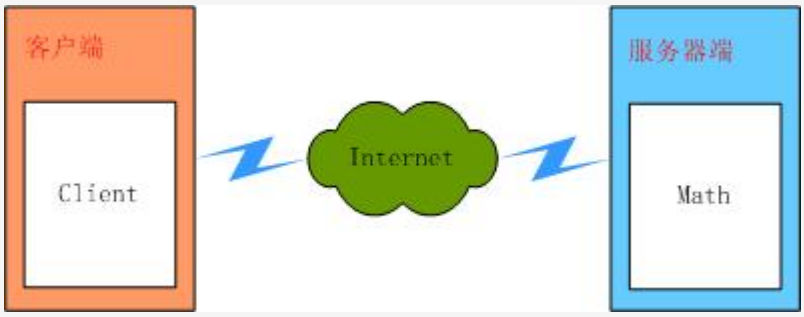
代码实例：

在软件系统中，我们无时不在跨越障碍，当我们访问网络上一台计算机的资源时，我们正在跨越网络障碍，当我们去访问服务器上数据库时，我们又在跨越数据库访问障碍，同时还有网络障碍。跨越这些障碍有时候是非常复杂的，如果我们更多的去关注处理这些障碍问题，可能就会忽视了本来应该关注的业务逻辑问题，Proxy 模式有助于我们去解决这些问题。我们以一个简单的数学计算程序为例，这个程序只负责进行简单的加减乘除运算：

```
1 public class Math
2 {
3    public double Add(double x,double y)
4    {
5       return x + y;
6    }
```

```
7
8     public double Sub(double x,double y)
9     {
10        return x - y;
11    }
12
13    public double Mul(double x,double y)
14    {
15        return x * y;
16    }
17
18    public double Dev(double x,double y)
19    {
20        return x / y;
21    }
22 }
```

如果说这个计算程序部署在我们本地计算机上，使用就非常之简单了，我们也就不用去考虑 Proxy 模式了。但现在问题是这个 Math 类并没有部署在我们本地，而是部署在一台服务器上，也就是说 Math 类根本和我们的客户程序不在同一个地址空间之内，我们现在要面对的是跨越 Internet 这样一个网络障碍：

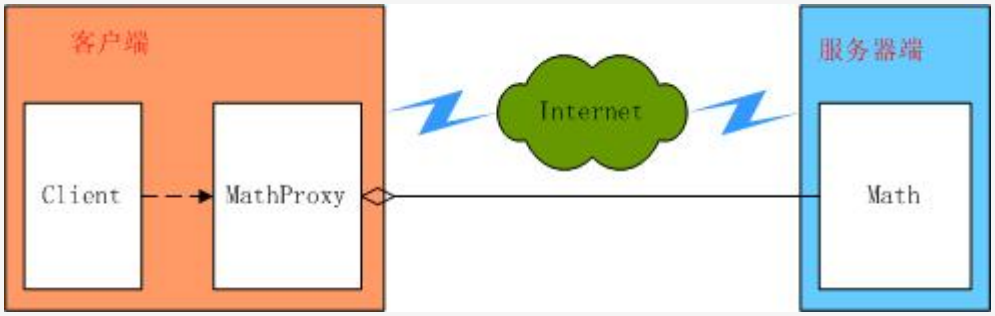


这时候调用 Math 类的方法就没有下面那么简单了，因为我们更多的还要去考虑网络的问题，对接收到的结果解包等一系列操作。

```
1 public class App
2 {
3     public static void Main()
4     {
5         Math math = new Math();
```

```
6
7       // 对接收到的结果数据进行解包
8
9       double addresult = math.Add(2,3);
10
11       double subresult = math.Sub(6,4);
12
13       double mulresult = math.Mul(2,3);
14
15       double devresult = math.Dev(2,3);
16    }
17 }
```

为了解决由于网络等障碍引起复杂性，就引出了 Proxy 模式，我们使用一个本地的代理来替 Math 类打点一切，即为我们的系统引入了一层间接层，示意图如下：

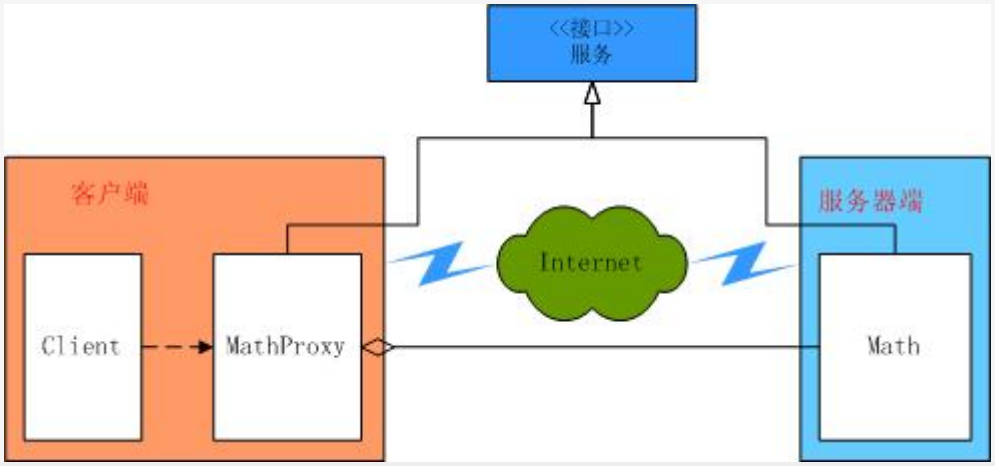


我们在 MathProxy 中对实现 Math 数据类的访问，让 MathProxy 来代替网络上的 Math 类，这样我们看到 MathProxy 就好像是本地 Math 类，它与客户程序处在了同一地址空间内：

```
1 public class MathProxy
2 {
3    private Math math = new Math();
4
5    // 以下的方法中，可能不仅仅是简单的调用 Math 类的方法
6
7    public double Add(double x,double y)
8    {
9       return math.Add(x,y);
```

```
10     }
11
12     public double Sub(double x,double y)
13     {
14         return math.Sub(x,y);
15     }
16
17     public double Mul(double x,double y)
18     {
19         return math.Mul(x,y);
20     }
21
22     public double Dev(double x,double y)
23     {
24         return math.Dev(x,y);
25     }
26 }
```

现在可以说我们已经实现了对 Math 类的代理，存在的一个问题是我们在 MathProxy 类中调用了原实现类 Math 的方法，但是 Math 并不一定实现了所有的方法，为了强迫 Math 类实现所有的方法，另一方面，为了我们更加透明的去操作对象，我们在 Math 类和 MathProxy 类的基础上加上一层抽象，即它们都实现与 IMath 接口，示意图如下：




```
1 public interface IMath
2 {
3     double Add(double x,double y);
```

```
4
5    double Sub(double x,double y);
6
7    double Mul(double x,double y);
8
9    double Dev(double x,double y);
10 }
11
12 Math 类和 MathProxy 类分别实现 IMath 接口：
13
14 public class MathProxy：IMath
15 {
16    //
17 }
18
19 public class Math：IMath
20 {
21    //
22 }
```

此时我们在客户程序中就可以像使用 Math 类一样来使用 MathProxy 类了：

```
1 public class App
2 {
3    public static void Main()
4    {
5       MathProxy proxy = new MathProxy();
6
7       double addresult = proxy.Add(2,3);
8
9       double subresult = proxy.Sub(6,4);
10
11       double mulresult = proxy.Mul(2,3);
12
13       double devresult = proxy.Dev(2,3);
```

```
14     }
15 }
```

到这儿整个使用 Proxy 模式的过程就完成了，回顾前面我们的解决方案，无非是在客户程序和 Math 类之间加了一个间接层，这也是我们比较常见的解决问题的手段之一。另外，对于程序中的接口 Imath，并不是必须的，大多数情况下，我们为了保持对对象操作的透明性，并强制实现类实现代理类所要调用的所有的方法，我们会让它们实现与同一个接口。但是我们说代理类它其实只是在一定程度上代表了原来的实现类，所以它们有时候也可以不实现于同一个接口。

代理模式实现要点：

1．远程（Remote）代理：为一个位于不同的　　地址空间的对象提供一个局域代表对象。这个不同的地址空间可以是在本机器中，也可是在另一台机器中。远程代理又叫做大使（Ambassador）。好处是系统可以将网络的细节隐藏起来，使得客户端不必考虑网络的存在。客户完全可以认为被代理的对象是局域的而不是远程的，而代理对象承担了大部份的网络通讯工作。由于客户可能没有意识到会启动一个耗费时间的远程调用，因此客户没有必要的思想准备。

2．虚拟（Virtual）代理：根据需要创建一个资源消耗较大的对象，使得此对象只在需要时才会被真正创建。使用虚拟代理模式的好处就是代理对象可以在必要的时候才将被代理的对象加载；代理可以对加载的过程加以必要的优化。当一个模块的加载十分耗费资源的情况下，虚拟代理的好处就非常明显。

3．Copy-on-Write 代理：虚拟代理的一种。把复制（克隆）拖延到只有在客户端需要时，才真正采取行动。

4．保护（Protect or Access）代理：控制对一个对象的访问，如果需要，可以给不同的用户提供不同级别的使用权限。保护代理的好处是它可以在运行时间对用户的有关权限进行检查，然后在核实后决定将调用传递给被代理的对象。

5．Cache 代理：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果。

6．防火墙（Firewall）代理：保护目标，不让恶意用户接近。

7．同步化（Synchronization）代理：使几个用户能够同时使用一个对象而没有冲突。

8．智能引用（Smart Reference）代理：当一个对象被引用时，提供一些额外的操作，比如将对此对象调用的次数记录下来等

迭代器模式(Iterator Pattern)

动机(Motivate):

在软件构建过程中，集合对象内部结构常常变化各异。但对于这些集合对象，我们希望在不暴露其内部结构的同时，可以让外部客户代码透明地访问其中包含的元素;同时这种“透明遍历”也为“ 同一种算法在多种集合对象上进行操作”提供了可能。

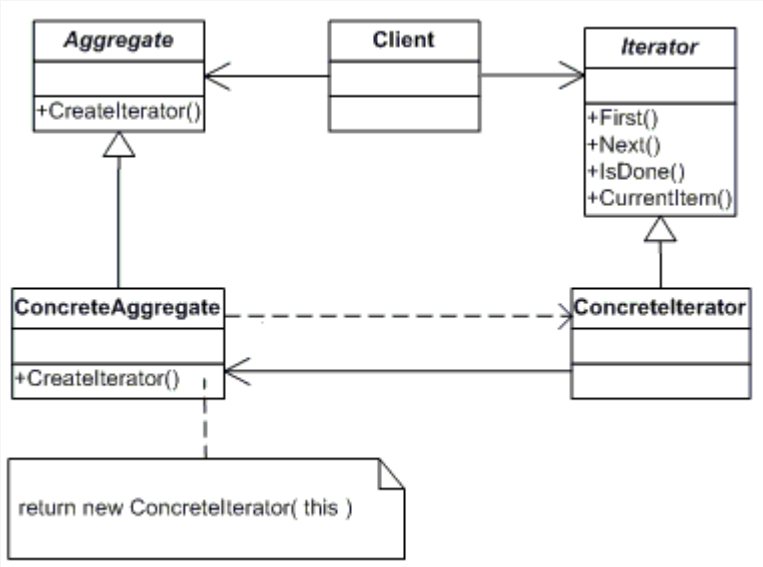
使用面向对象技术将这种遍历机制抽象为“迭代器对象”为“应对变化中的集合对象”提供了一种优雅的方法。

意图(Intent):

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。

--------《设计模式》GOF

结构图（Struct):



适用性：

1. 访问一个聚合对象的内容而无需暴露它的内部表示。

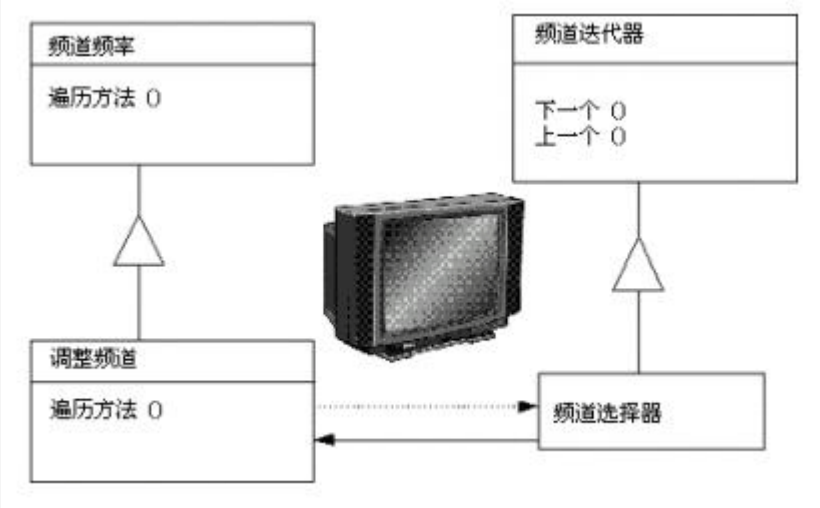
2. 支持对聚合对象的多种遍历。

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)。

生活中的例子：

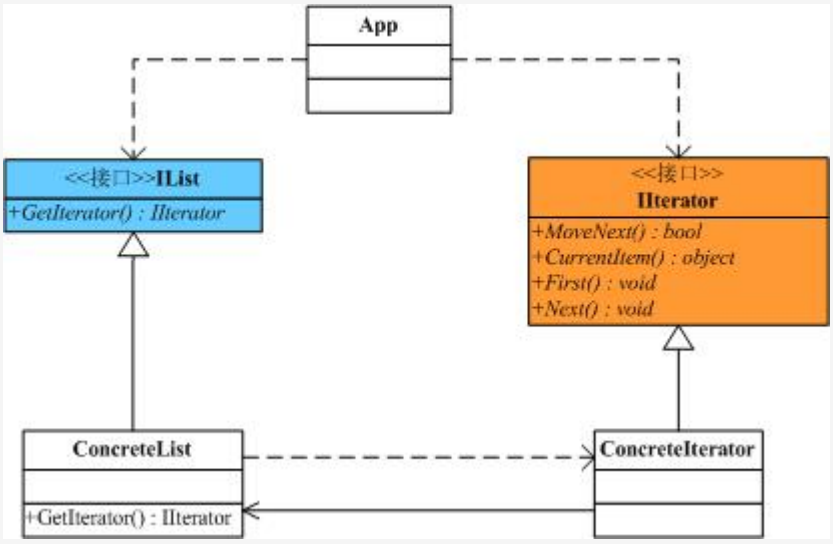
迭代器提供一种方法顺序访问一个集合对象中各个元素，而又不需要暴露该对象的内部表示。在早期的电视机中，一个拨盘用来改变频道。当改变频道时，需要手工转动拨盘移过每一个频道，而不论这个频道是否有信号。现在的电视机，使用[后一个]和[前一个]按钮。当按下[后一个]按钮时，将切换到下一个预置的频道。想象一下在陌生的城市中的旅店中看电视。当改变

频道时，重要的不是几频道，而是节目内容。如果对一个频道的节目不感兴趣，那么可以换下一个频道，而不需要知道它是几频道。



代码实现：

在面向对象的软件设计中，我们经常会遇到一类集合对象，这类集合对象的内部结构可能有着各种各样的实现，但是归结起来，无非有两点是需要我们去关心的：一是集合内部的数据存储结构，二是遍历集合内部的数据。面向对象设计原则中有一条是类的单一职责原则，所以我们要尽可能的去分解这些职责，用不同的类去承担不同的职责。Iterator 模式就是分离了集合对象的遍历行为，抽象出一个迭代器类来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明的访问集合内部的数据。下面看一个简单的示意性例子，类结构图如下：



首先有一个抽象的聚集，所谓的聚集就是就是数据的集合，可以循环去访问它。它只有一个方法 GetIterator()让子类去实现，用来获得一个迭代器对象。

```
/// <summary>

/// 抽象聚集

/// </summary>

public interface IList

{
    IIterator GetIterator();
}
```

抽象的迭代器，它是用来访问聚集的类，封装了一些方法，用来把聚集中的数据按顺序读取出来。通常会有 MoveNext()、CurrentItem()、Fisrt()、Next()等几个方法让子类去实现。

```
/// <summary>

/// 抽象迭代器

/// </summary>

public interface IIterator
{
   bool MoveNext();

    Object CurrentItem();

    void First();

    void Next();
}
```

具体的聚集，它实现了抽象聚集中的唯一的方法，同时在里面保存了一组数据，这里我们加上 Length 属性和 GetElement()方法是为了便于访问聚集中的数据。

```
/// <summary>

```

```
3 /// 具体聚集
4
5 /// </summary>
6
7 public class ConcreteList ： IList
8 {
9    int[] list;
10
11    public ConcreteList()
12
13    {
14       list = new int[] { 1,2,3,4,5};
15    }
16
17    public IIterator GetIterator()
18
19    {
20       return new ConcreteIterator(this);
21    }
22
23    public int Length
24
25    {
26       get { return list.Length; }
27    }
28
29    public int GetElement(int index)
30
31    {
32       return list[index];
33    }
34 }
```

具体迭代器，实现了抽象迭代器中的四个方法，在它的构造函数中需要接受一个具体聚集类型的参数，在这里面我们可以根据实际的情况去编写不同的迭代方式。

```
1 /**//// <summary>
2
3 /// 具体迭代器
4
5 /// </summary>
6
7 public class ConcreteIterator : IIterator
8
9 {
10    private ConcreteList list;
11
12    private int index;
13
14    public ConcreteIterator(ConcreteList list)
15
16    {
17       this.list = list;
18
19       index = 0;
20    }
21
22    public bool MoveNext()
23
24    {
25       if (index < list.Length)
26
27          return true;
28
29       else
30
31          return false;
32    }
33
34    public Object CurrentItem()
```

```
35
36    {
37        return list.GetElement(index) ;
38    }
39
40    public void First()
41
42    {
43        index = 0;
44    }
45
46    public void Next()
47
48    {
49        if (index < list.Length)
50
51        {
52            index++;
53        }
54    }
55 }
```

简单的客户端程序调用：

```
1 /**//// <summary>
2
3 /// 客户端程序
4
5 /// </summary>
6
7 class Program
8
9 {
10    static void Main(string[] args)
11
12    {
```

```
13        IIterator iterator;
14
15        IList list = new ConcreteList();
16
17        iterator = list.GetIterator();
18
19        while (iterator.MoveNext())
20
21        {
22            int i = (int)iterator.CurrentItem();
23            Console.WriteLine(i.ToString());
24
25            iterator.Next();
26        }
27
28        Console.Read();
29
30    }
31
32 }
```

.NET 中 Iterator 中的应用：

在.NET 下实现 Iterator 模式，对于聚集接口和迭代器接口已经存在了，其中 IEnumerator 扮演的就是迭代器的角色，它的实现如下：

```
1 public interface IEumerator
2
3 {
4    object Current
5    {
6       get;
7    }
8
9    bool MoveNext();
10
11    void Reset();
```

```
12
13 }
```

属性 Current 返回当前集合中的元素，Reset()方法恢复初始化指向的位置，MoveNext()方法返回值 true 表示迭代器成功前进到集合中的下一个元素，返回值 false 表示已经位于集合的末尾。能够提供元素遍历的集合对象，在.Net 中都实现了 IEnumerator 接口。

IEnumerable 则扮演的就是抽象聚集的角色，只有一个 GetEnumerator()方法，如果集合对象需要具备跌代遍历的功能，就必须实现该接口。

```
1 public interface IEnumerable
2
3 {
4     IEumerator GetEnumerator();
5 }
```

Iterator 实现要点：

1. 迭代抽象：访问一个聚合对象的内容而无需暴露它的内部表示。

2. 迭代多态：为遍历不同的集合结构提供一个统一的接口，从而支持同样的算法在不同的集合结构上进行操作。

3. 迭代器的健壮性考虑：遍历的同时更改迭代器所在的集合结构，会导致问题。

命令模式(Command Pattern)

耦合与变化：

耦合是软件不能抵御变化灾难的根本性原因。不仅实体对象与实体对象之间存在耦合关系，实体对象与行为操作之间也存在耦合关
系。

动机(Motivate)：

在软件系统中，“行为请求者”与“行为实现者”通常呈现一种“紧耦合”。但在某些场合，比如要对行为进行“记录、撤销/重做、事务”等处理，这种无法抵御变化的紧耦合是不合适的。

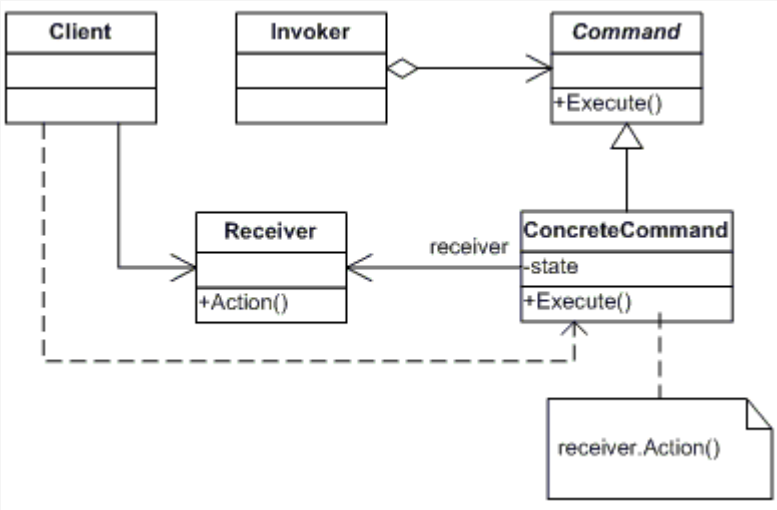
在这种情况下，如何将“行为请求者”与“行为实现者”解耦？将一组行为抽象为对象，可以实现二者之间的松耦合。

意图(Intent)：

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。

--------《设计模式》GOF

结构图(Struct)：



适用性：

1. 使用命令模式作为"CallBack"在面向对象系统中的替代。"CallBack"讲的便是先将一个函数登记上，然后在以后调用此函数。

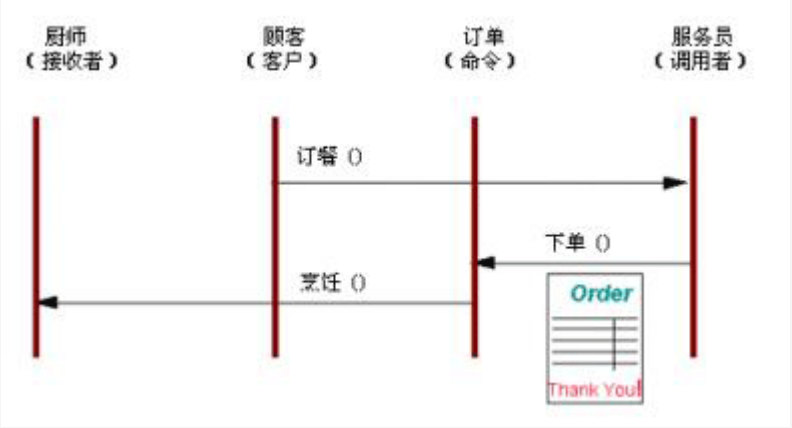
2．需要在不同的时间指定请求、将请求排队。一个命令对象和原先的请求发出者可以有不同的生命期。换言之，原先的请求发出者可能已经不在了，而命令对象本身仍然是活动的。这时命令的接收者可以是在本地，也可以在网络的另外一个地址。命令对象可以在串形化之后传送到另外一台机器上去。

3．系统需要支持命令的撤消(undo)。命令对象可以把状态存储起来，等到客户端需要撤销命令所产生的效果时，可以调用 undo()方法，把命令所产生的效果撤销掉。命令对象还可以提供 redo()方法，以供客户端在需要时，再重新实施命令效果。

4．如果一个系统要将系统中所有的数据更新到日志里，以便在系统崩溃时，可以根据日志里读回所有的数据更新命令，重新调用 Execute()方法一条一条执行这些命令，从而恢复系统在崩溃前所做的数据更新。

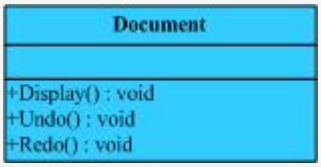
生活中的例子：

Command 模式将一个请求封装为一个对象，从而使你可以使用不同的请求对客户进行参数化。用餐时的账单是 Command 模式的一个例子。服务员接受顾客的点单，把它记在账单上封装。这个点单被排队等待烹饪。注意这里的"账单"是不依赖于菜单的，它可以被不同的顾客使用，因此它可以添入不同的点单项目。



代码实现：

在众多的设计模式中，Command 模式是很简单也很优雅的一种设计模式。Command 模式它封装的是命令，把命令发出者的责任和命令执行者的责任分开。我们知道，一个类是一组操作和相应的一些变量的集合，现在有这样一个类 Document，如下：




```
1 /// <summary>
2
3 /// 文档类
4
5 /// </summary>
6
7 public class Document
8
9 {
10     /**//// <summary>
11
12     /// 显示操作
13
14     /// </summary>
15
16     public void Display()
17
18     {
19         Console.WriteLine("Display");
20     }
21
22     /**//// <summary>
23
24     /// 撤销操作
25
26     /// </summary>
27
28     public void Undo()
29
30     {
```

```
31        Console.WriteLine("Undo");
32    }
33
34    /**/// <summary>
35
36    /// 恢复操作
37
38    /// </summary>
39
40    public void Redo()
41
42    {
43        Console.WriteLine("Redo");
44    }
45 }
```

通常客户端实现代码如下：

```
1 class Program
2
3 {
4    static void Main(string[] args)
5
6    {
7        Document doc = new Document();
8
9        doc.Display();
10
11       doc.Undo();
12
13       doc.Redo();
14   }
15 }
```

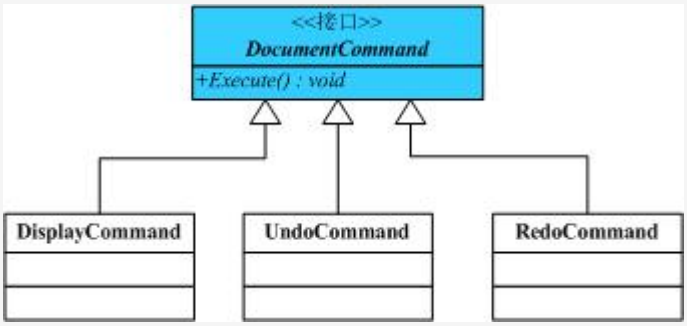
这样的使用本来是没有任何问题的，但是我们看到在这个特定的应用中，出现了 Undo/Redo

的操作，这时如果行为的请求者和行为的实现者之间还是呈现这样一种紧耦合，就不太合适了。可以看到，客户程序是依赖于具体 Document 的命令（方法）的，引入 Command 模式，需要对 Document 中的三个命令进行抽象，这是 Command 模式最有意思的地方，因为在我们看来 Display()，Undo()，Redo()这三个方法都应该是 Document 所具有的，如果单独抽象出来成一个命令对象，那就是把函数层面的功能提到了类的层面，有点功能分解的味道，我觉得这正是 Command 模式解决这类问题的优雅之处，先对命令对象进行抽象：


```
/// <summary>

/// 抽象命令

/// </summary>

public abstract class DocumentCommand

{
    Document _document;

    public DocumentCommand(Document doc)

    {
        this._document = doc;
    }

    /**/// <summary>

    /// 执行

    /// </summary>

    public abstract void Execute();
```

```
25
26 }
```

其他的具体命令类都继承于该抽象类，如下：



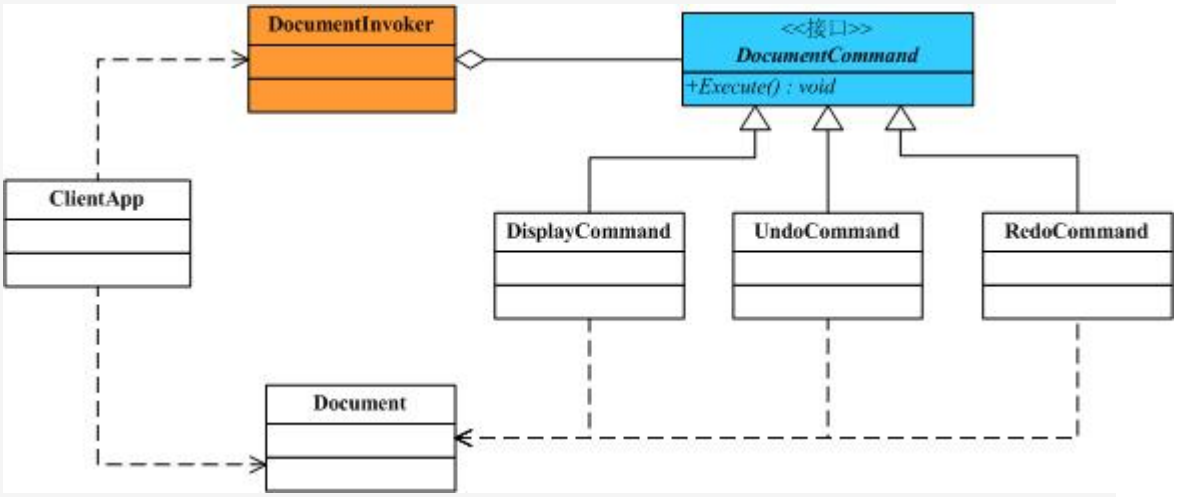
示意性代码如下：

```
1 /// <summary>
2
3 /// 显示命令
4
5 /// </summary>
6
7 public class DisplayCommand：DocumentCommand
8
9 {
10    public DisplayCommand(Document doc)
11
12       : base(doc)
13    {
14
15    }
16
17    public override void Execute()
18
19    {
20       _document.Display();
21    }
22 }
23
24
```

```
25 /**/// <summary>
26
27 /// 撤销命令
28
29 /// </summary>
30
31 public class UndoCommand：DocumentCommand
32
33 {
34    public UndoCommand(Document doc)
35
36       : base(doc)
37    {
38
39    }
40
41    public override void Execute()
42
43    {
44       _document.Undo();
45    }
46 }
47
48
49 /**/// <summary>
50
51 /// 重做命令
52
53 /// </summary>
54
55 public class RedoCommand：DocumentCommand
56
57 {
58    public RedoCommand(Document doc)
```

```
59
60        : base(doc)
61    {
62
63    }
64
65    public override void Execute()
66
67    {
68        _document.Redo();
69    }
70 }
```

现在还需要一个 Invoker 角色的类，这其实相当于一个中间角色，前面我曾经说过，使用这样的一个中间层也是我们经常使用的手法，即把 A 对 B 的依赖转换为 A 对 C 的依赖。如下：




```
1 /// <summary>
2
3 /// Invoker 角色
4
5 /// </summary>
6
7 public class DocumentInvoker
8
9 {
```

```
10    DocumentCommand _discmd;
11
12    DocumentCommand _undcmd;
13
14    DocumentCommand _redcmd;
15
16    public DocumentInvoker(DocumentCommand discmd,DocumentCommand u
ndcmd,DocumentCommand redcmd)
17    {
18
19        this._discmd = discmd;
20
21        this._undcmd = undcmd;
22
23        this._redcmd = redcmd;
24
25    }
26
27    public void Display()
28
29    {
30        _discmd.Execute();
31    }
32
33    public void Undo()
34
35    {
36        _undcmd.Execute();
37    }
38
39    public void Redo()
40
41    {
42        _redcmd.Execute();
```

```
43    }
44 }
45
46 现在再来看客户程序的调用代码：
47 class Program
48
49 {
50    static void Main(string[] args)
51
52    {
53
54       Document doc = new Document();
55
56
57       DocumentCommand discmd = new DisplayCommand(doc);
58
59       DocumentCommand undcmd = new UndoCommand(doc);
60
61       DocumentCommand redcmd = new RedoCommand(doc);
62
63
64       DocumentInvoker invoker = new DocumentInvoker(discmd,undcmd,redc
md);
65
66       invoker.Display();
67
68       invoker.Undo();
69
70       invoker.Redo();
71
72    }
73 }
```

可以看到在客户程序中，不再依赖于 Document 的 Display()，Undo()，Redo()命令，通过 Command 对这些命令进行了封装，使用它的一个关键就是抽象的 Command 类，它定义了一个操作的接口。同时我们也可以看到，本来这三个命令仅仅是三个方法而已，但是通过 Command 模式却把它们提到了类的层面，这其实是违背了面向对象的原则，但它却优雅的解决了分离命令的请求者和命令的执行者的问题，在使用 Command 模式的时候，一定要判断好使用它的时机。

Command 实现要点：

1．Command 模式的根本目的在于将“行为请求者”与“行为实现者”解耦，在面向对象语言中，常见的实现手段是“将行为抽象为对象”。

2．实现 Command 接口的具体命令对象 ConcreteCommand 有时候根据需要可能会保存一些额外的状态信息。

3．通过使用 Compmosite 模式，可以将多个命令封装为一个“复合命令”MacroCommand。

4．Command 模式与 C#中的 Delegate 有些类似。但两者定义行为接口的规范有所区别：Command 以面向对象中的“接口-实现”来定义行为接口规范，更严格，更符合抽象原则；Delegate 以函数签名来定义行为接口规范，更灵活，但抽象能力比较弱。

5．使用命令模式会导致某些系统有过多的具体命令类。某些系统可能需要几十个，几百个甚至几千个具体命令类，这会使命令模式在这样的系统里变得不实际。

Command 的优缺点：

命令允许请求的一方和接收请求的一方能够独立演化，从而且有以下的优点：

1.命令模式使新的命令很容易地被加入到系统里。

2.允许接收请求的一方决定是否要否决（Veto）请求。

3.能较容易地设计-个命令队列。

4.可以容易地实现对请求的 Undo 和 Redo。

5.在需要的情况下，可以较容易地将命令记入日志。

6.命令模式把请求一个操作的对象与知道怎么执行一个操作的对象分割开。

7.命令类与其他任何别的类一样，可以修改和推广。

8.你可以把命令对象聚合在一起，合成为合成命令。比如宏命令便是合成命令的例子。合成命令是合成模式的应用。

9.由于加进新的具体命令类不影响其他的类，因此增加新的具体命令类很容易。

命令模式的缺点如下：

1.使用命令模式会导致某些系统有过多的具体命令类。某些系统可能需要几十个，几百个甚至几千个具体命令类，这会使命令模式在这样的系统里变得不实际。

```
1 class Program
2
3 {
4    static void Main(string[] args)
5
6    {
7       Document doc = new Document();
8
9       doc.Display();
10
11       doc.Undo();
12
13       doc.Redo();
14    }
```

```
15  }
```

模板方法(Template Method)

无处不在的 Template Method

如果你只想掌握一种设计模式，那么它就是 Template Method!

动机(Motivate):

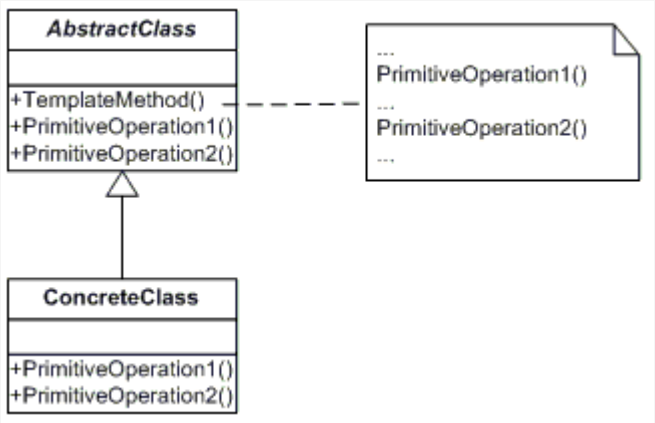
变化 -----是软件设计的永恒主题，如何管理变化带来的复杂性？设计模式的艺术性和复杂度就在于如何
分析，并发现系统中的变化和稳定点，并使用特定的设计方法来应对这种变化。

意图(Intent):

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

--------《设计模式》GOF

结构图(Struct):



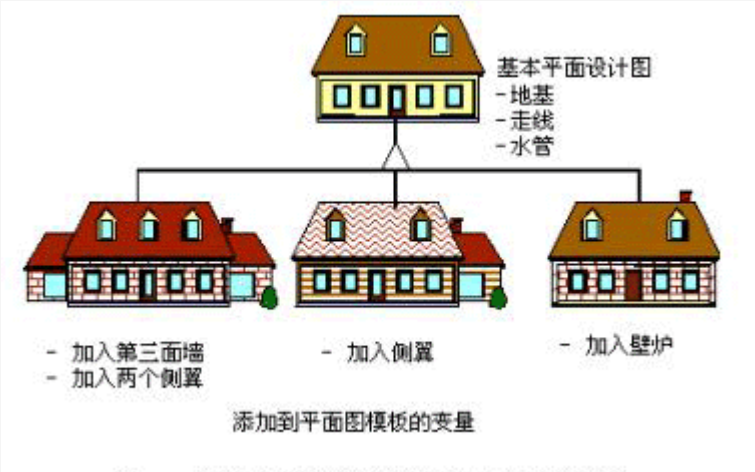
适用性：

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。首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。

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。

生活中的例子：

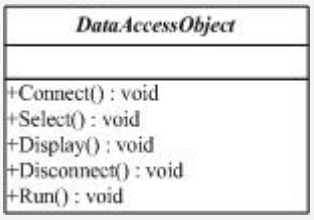


代码实现：

假如我们需要简单的读取 Northwind 数据库中的表的记录并显示出来。对于数据库操作，我们知道不管读取的是哪张表，它一般都应该经过如下这样的几步：

1. 连接数据库（Connect）
2. 执行查询命令（Select）
3. 显示数据（Display）
4. 断开数据库连接（Disconnect）

这些步骤是固定的，但是对于每一张具体的数据表所执行的查询却是不一样的。显然这需要一个抽象角色，给出顶级行为的实现。如下
图：



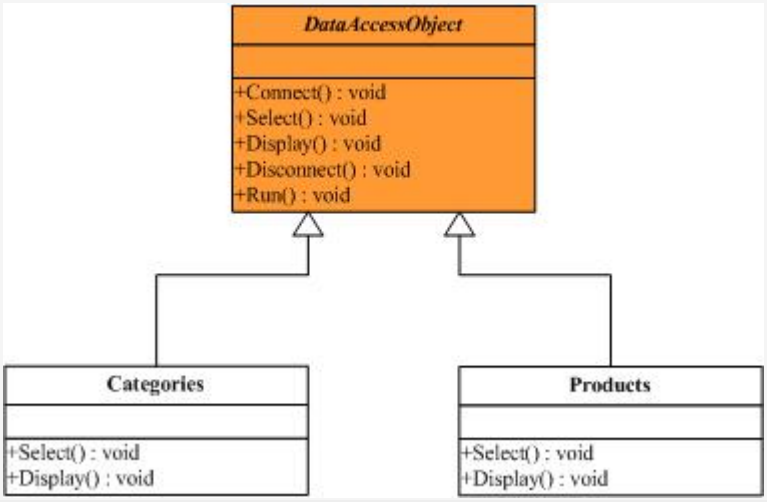
Template Method 模式的实现方法是从上到下，我们首先给出顶级框架 DataAccessObject 的实现逻辑：

```
1 public abstract class DataAccessObject
2
3 {
```

```
4   protected string connectionString;
5
6   protected DataSet dataSet;
7
8   protected virtual void Connect()
9
10  {
11      connectionString =
12
13          "Server=.;User Id=sa;Password=;Database=Northwind";
14
15  }
16
17  protected  abstract void Select();
18
19  protected abstract void Display();
20
21
22  protected virtual void Disconnect()
23
24  {
25      connectionString = "";
26  }
27
28  // The "Template Method"
29
30  public void Run()
31
32  {
33      Connect();
34
35      Select();
36
37      Display();
```

```
38
39          Disconnect();
40      }
41  }
```

显然在这个顶级的框架 DataAccessObject 中给出了固定的轮廓，方法 Run()便是模版方法，Template Method 模式也由此而得名。而对于 Select()和 Display()这两个抽象方法则留给具体的子类去实现，如下图：




```
1  class Categories : DataAccessObject
2
3  {
4      protected override void Select()
5      {
6          string sql = "select CategoryName from Categories";
7
8          SqlDataAdapter dataAdapter = new SqlDataAdapter(
9
10             sql, connectionString);
11
12         dataSet = new DataSet();
13
14         dataAdapter.Fill(dataSet, "Categories");
15
16     }
17
```

```
18    protected override void Display()
19
20    {
21
22        Console.WriteLine("Categories ---- ");
23
24        DataTable dataTable = dataSet.Tables["Categories"];
25
26        foreach (DataRow row in dataTable.Rows)
27
28        {
29
30            Console.WriteLine(row["CategoryName"].ToString());
31
32        }
33
34        Console.WriteLine();
35
36    }
37 }
```

```
1 class Products : DataAccessObject
2
3 {
4    protected override void Select()
5
6    {
7       string sql = "select top 10 ProductName from Products";
8
9       SqlDataAdapter dataAdapter = new SqlDataAdapter(
10
11          sql, connectionString);
12
13       dataSet = new DataSet();
```

```
14
15        dataAdapter.Fill(dataSet, "Products");
16
17    }
18
19    protected override void Display()
20
21    {
22
23        Console.WriteLine("Products ---- ");
24
25        DataTable dataTable = dataSet.Tables["Products"];
26
27        foreach (DataRow row in dataTable.Rows)
28
29        {
30            Console.WriteLine(row["ProductName"].ToString());
31
32        }
33
34        Console.WriteLine();
35
36    }
37
38 }
```

再来看看客户端程序的调用，不需要再去调用每一个步骤的方法：

```
1 public class App
2
3 {
4    static void Main()
5    {
6
7       DataAccessObject dao;
```

```
8
9
10          dao = new Categories();
11
12          dao.Run();
13
14
15          dao = new Products();
16
17          dao.Run();
18
19          // Wait for user
20
21          Console.Read();
22
23      }
24
25  }
```

在上面的例子中，需要注意的是：

1. 对于 Connect()和 Disconnect()方法实现为了 virtual，而 Select()和 Display()方法则为 abstract，这是因为如果这个方法有默认的实现，则实现为 virtual，否则为 abstract。

2. Run()方法作为一个模版方法，它的一个重要特征是：在基类里定义，而且不能够被派生类更改。有时候它是私有方法（private method），但实际上它经常被声明为 protected。它通过调用其它的基类方法（覆写过的）来工作，但它经常是作为初始化过程的一部分被调用的，这样就没必要让客户端程序员能够直接调用它了。

3. 在一开始我们提到了不管读的是哪张数据表，它们都有共同的操作步骤，即共同点。因此可以说 Template Method 模式的一个特征就是剥离共同点。

Template Mehtod 实现要点：

1．Template Method 模式是一种非常基础性的设计模式，在面向对象系统中有着大量的应用。它用最简洁的机制（虚函数的多态性）为很多应用程序框架提供了灵活的扩展点，是代码复用方面的基本实现结构。

2．除了可以灵活应对子步骤的变化外，“不用调用我，让我来调用你（Don’t call me ,let me call you)”的反向控制结构是 Template Method 的典型应用。“Don’t call me.Let me call you”是指一个父类调用一个子类的操作，而不是相反。

3．在具体实现方面，被 Template Method 调用的虚方法可以具有实现，也可以没有任何实现（抽象方法，纯虚方法），但一般推荐将它们设置为 protected 方法。

桥接模式（Bridge Pattern)

动机(Motivate):

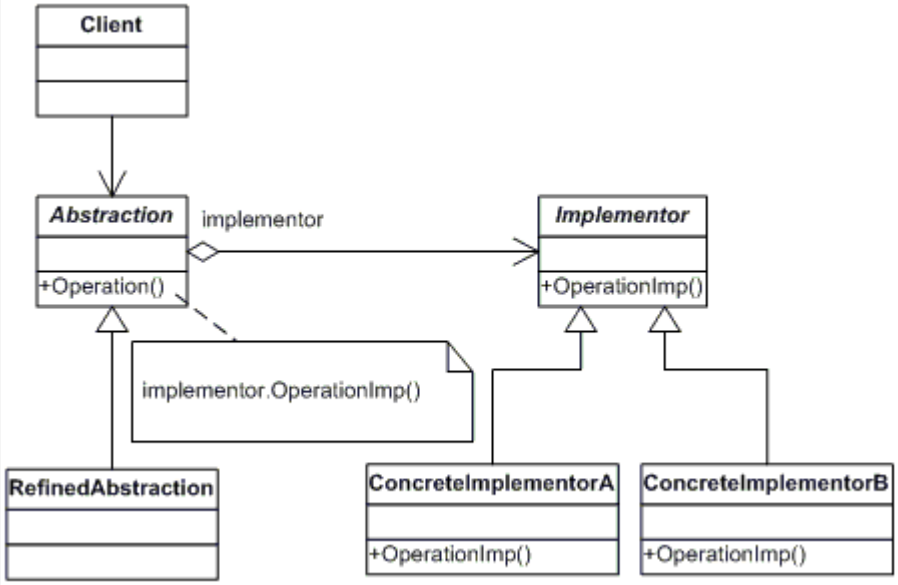
在软件系统中，某些类型由于自身的逻辑，它具有两个或多个维度的变化，那么如何应对这种“多维度的变化”？如何利用面向对象的技术来使得该类型能够轻松的沿着多个方向进行变化，而又不引入额外的复杂度？

意图(Intent):

将抽象部分与实现部分分离，使它们都可以独立的变化。

------《设计模式》GOF

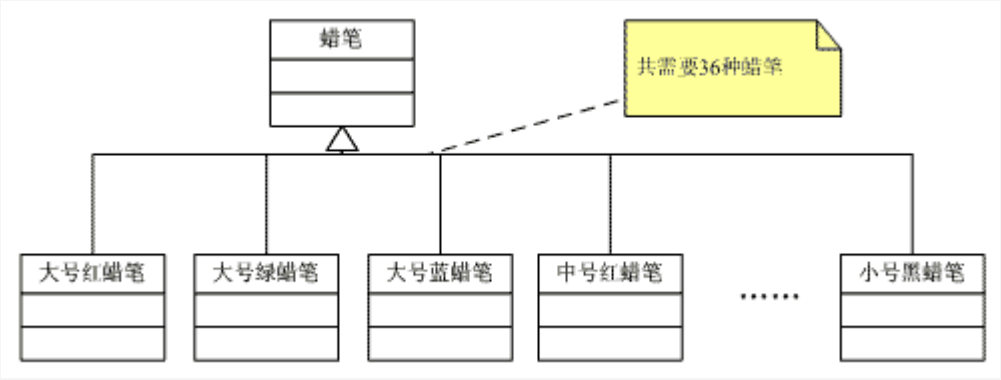
结构图(Struct):



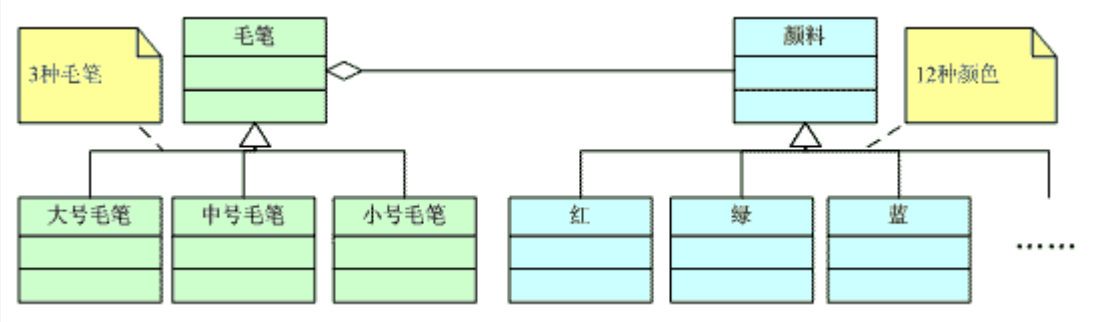
生活中的例子：

我想大家小时候都有用蜡笔画画的经历吧。红红绿绿的蜡笔一大盒，根据想象描绘出格式图样。而毛笔下的国画更是工笔写意，各展风采。而今天我们的故事从蜡笔与毛笔说起。

设想要绘制一幅图画，蓝天、白云、绿树、小鸟，如果画面尺寸很大，那么用蜡笔绘制就会遇到点麻烦。毕竟细细的蜡笔要涂出一片蓝天，是有些麻烦。如果有可能，最好有套大号蜡笔，粗粗的蜡笔很快能涂抹完成。至于色彩吗，最好每种颜色来支粗的，除了蓝天还有绿地呢。这样，如果一套 12 种颜色的蜡笔，我们需要两套 24 支，同种颜色的一粗一细。呵呵，画还没画，开始做梦了：要是再有一套中号蜡笔就更好了，这样，不多不少总共 36 支蜡笔。



再看看毛笔这一边，居然如此简陋：一套水彩 12 色，外加大中小三支毛笔。你可别小瞧这"简陋"的组合，画蓝天用大毛笔，画小鸟用小毛笔，各具特色。



呵呵，您是不是已经看出来了，不错，我今天要说的就是 Bridge 模式。为了一幅画，我们需要准备 36 支型号不同的蜡笔，而改用毛笔三支就够了，当然还要搭配上 12 种颜料。通过 Bridge 模式，我们把乘法运算 3×12＝36 改为了加法运算 3＋12＝15，这一改进可不小。那么我们这里蜡笔和毛笔到底有什么区别呢？

实际上，蜡笔和毛笔的关键一个区别就在于笔和颜色是否能够分离。【GOF95】桥梁模式的用意是"将抽象化（Abstraction)与实现化(Implementation)脱耦，使得二者可以独立地变化"。关键就在于能否脱耦。蜡笔的颜色和蜡笔本身是分不开的，所以就造成必须使用 36 支色彩、大小各异的蜡笔来绘制图画。而毛笔与颜料能够很好的脱耦，各自独立变化，便简化了操作。在这里，抽象层面的概念是： "毛笔用颜料作画"，而在实现时，毛笔有大中小三号，颜料有红绿蓝等 12 种，于是便可出现 3×12 种组合。每个参与者（毛笔与颜料）都可以在自己的自由度上随意转换。

蜡笔由于无法将笔与颜色分离，造成笔与颜色两个自由度无法单独变化，使得只有创建 36 种对象才能完成任务。Bridge 模式将继承关系转换为组合关系，从而降低了系统间的耦合，减少了代码编写量。

代码实现：

```
abstract class Brush
    {
        protected Color c;
        public abstract void Paint();

        public void SetColor(Color c)
        { this.c = c; }
    }
```

```
class BigBrush：Brush
    {
        public override void Paint()
        { Console.WriteLine("Using big brush and color {0} painting", c.color); }
    }
```

```
class SmallBrush：Brush
    {
        public override void Paint()
        { Console.WriteLine("Using small brush and color {0} painting", c.color); }
    }
```

```
class Color
    {
        public string color;
    }
```

```
class Red：Color
    {
```

```
3       public Red()
4       { this.color = "red"; }
5    }
```

```
1   class Green : Color
2    {
3       public Green()
4       { this.color = "green"; }
5    }
```

```
1   class Blue : Color
2    {
3       public Blue()
4       { this.color = "blue"; }
5    }
```

```
1    class Program
2    {
3   public static void Main()
4   {
5    Brush b = new BigBrush();
6    b.SetColor(new Red());
7    b.Paint();
8    b.SetColor(new Blue());
9    b.Paint();
10    b.SetColor(new Green());
11    b.Paint();
12
13    b = new SmallBrush();
14    b.SetColor(new Red());
15    b.Paint();
16    b.SetColor(new Blue());
17    b.Paint();
18    b.SetColor(new Green());
```

```
19    b.Paint();
20  }
```

适用性：

1．如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的联系。

2．设计要求实现化角色的任何改变不应当影响客户端，或者说实现化角色的改变对客户端是完全透明的。

3．一个构件有多于一个的抽象化角色和实现化角色，系统需要它们之间进行动态耦合。

4．虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，设计要求需要独立管理这两者。

Bridge 要点：

1．Bridge 模式使用“对象间的组合关系”解耦了抽象和实现之间固有的绑定关系，使得抽象和实现可以沿着各自的维度来变化。

2．所谓抽象和实现沿着各自维度的变化，即“子类化”它们，得到各个子类之后，便可以任意它们，从而获得不同平台上的不同型号。

3．Bridge 模式有时候类似于多继承方案，但是多继承方案往往违背了类的单一职责原则（即一个类只有一个变化的原因），复用性比较差。Bridge 模式是比多继承方案更好的解决方法。

4．Bridge 模式的应用一般在“两个非常强的变化维度”，有时候即使有两个变化的维度，但是某个方向的变化维度并不剧烈——换言之两个变化不会导致纵横交错的结果，并不一定要使用 Bridge 模式

适配器模式（Adapter Pattern)

适配（转换）的概念无处不在.....

适配，即在不改变原有实现的基础上，将原先不兼容的接口转换为兼容的接口。

例如：二转换为三箱插头，将高电压转换为低电压等。

动机(Motivate)：

在软件系统中，由于应用环境的变化，常常需要将“一些现存的对象”放在新的环境中应用，但是新环境要求的接口是这些现存对象所不满足的。

那么如何应对这种“迁移的变化”？如何既能利用现有对象的良好实现，同时又能满足新的应用环境所要求的接口？这就是本文要说的 Adapter 模式。

意图(Intent)：

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。

--------《设计模式》GOF

结构(Struct)：

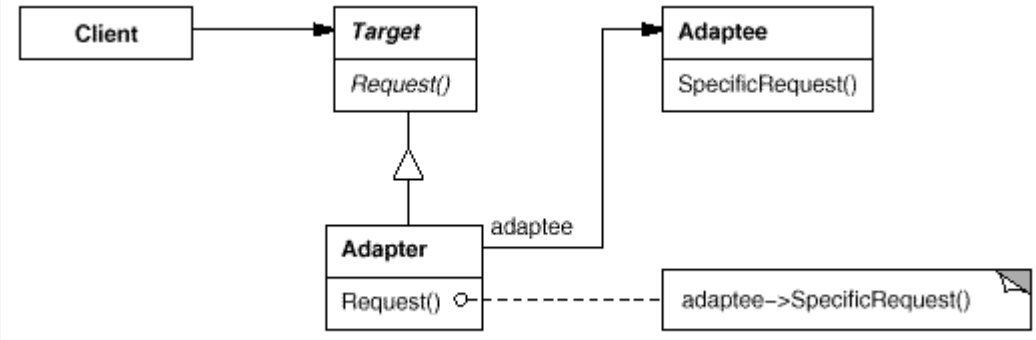


图 1：对象适配器

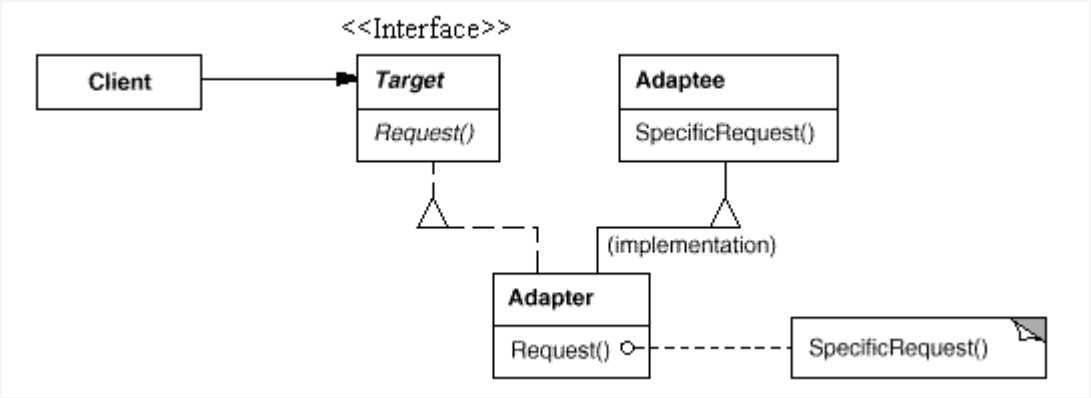
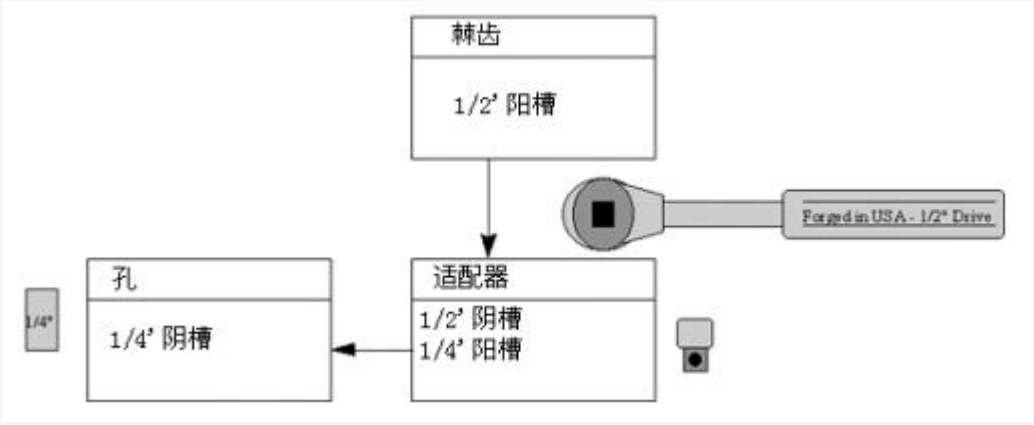


图 2：类适配器

生活中的例子：



适用性：

1．系统需要使用现有的类，而此类的接口不符合系统的需要。

2．想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。这些源类不一定有很复杂的接口。

3．（对对象适配器而言）在设计里，需要改变多个已有子类的接口，如果使用类的适配器模式，就要针对每一个子类做一个适配器，而这不太实际。

示意性代码实例：

```
interface IStack
{
    void Push(object item);
    void Pop();
    object Peek();
}
```

```
//对象适配器(Adapter 与 Adaptee 组合的关系)
public class Adapter : IStack //适配对象
{
    ArrayList adaptee;//被适配的对象
    public Adapter()
    {
        adaptee = new ArrayList();
    }
    public void Push(object item)
    {
        adaptee.Add(item);
    }
    public void Pop()
    {
        adaptee.RemoveAt(adaptee.Count - 1);
    }
    public object Peek()
    {
        return adaptee[adaptee.Count - 1];
    }
}
```

类适配器

```
public class Adapter :ArrayList, IStack
{
    public void Push(object item)
    {
      this.Add(item);
    }
```

```
7       public void Pop()
8       {
9           this.RemoveAt(this.Count - 1);
10      }
11     public  object Peek()
12      {
13          return this[this.Count - 1];
14      }
15  }
```

Adapter 模式的几个要点：

Adapter 模式主要应用于“希望复用一些现存的类，但是接口又与复用环境要求不一致的情况”，在遗留代码复用、类库迁移等方面非常有用。

GOF23 定义了两种 Adapter 模式的实现结构：对象适配器和类适配器。但类适配器采用“多继承”的实现方式，带来不良的高耦合，所以一般不推荐使用。对象适配器采用“对象组合”的方式，更符合松耦合精神。

Adapter 模式可以实现的非常灵活，不必拘泥于 GOF23 中定义的两种结构。例如，完全可以将 Adapter 模式中的“现存对象“作为新的接口方法参数，来达到适配的目的。

Adapter 模式本身要求我们尽可能地使用”面向接口的编程"风格，这样才能在后期很方便的适配。

.NET 框架中的 Adapter 应用：

(1)在.Net 中复用 com 对象：

Com 对象不符合.net 对象的接口

使用 tlbimp.exe 来创建一个 Runtime Callable Wrapper(RCW)以使其符合.net 对象的接口。

(2).NET 数据访问类（Adapter 变体）：

各种数据库并没有提供 DataSet 接口

使用 DBDataAdapter 可以将任何各数据库访问/存取适配到一个 DataSet 对象上。

(3)集合类中对现有对象的排序（Adapter 变体）；

现有对象未实现 IComparable 接口

实现一个排序适配器（继承 IComparer 接口），然后在其 Compare 方法中对两个对象进行比较。

外观模式（Facade Pattern)

动机(Motivate):

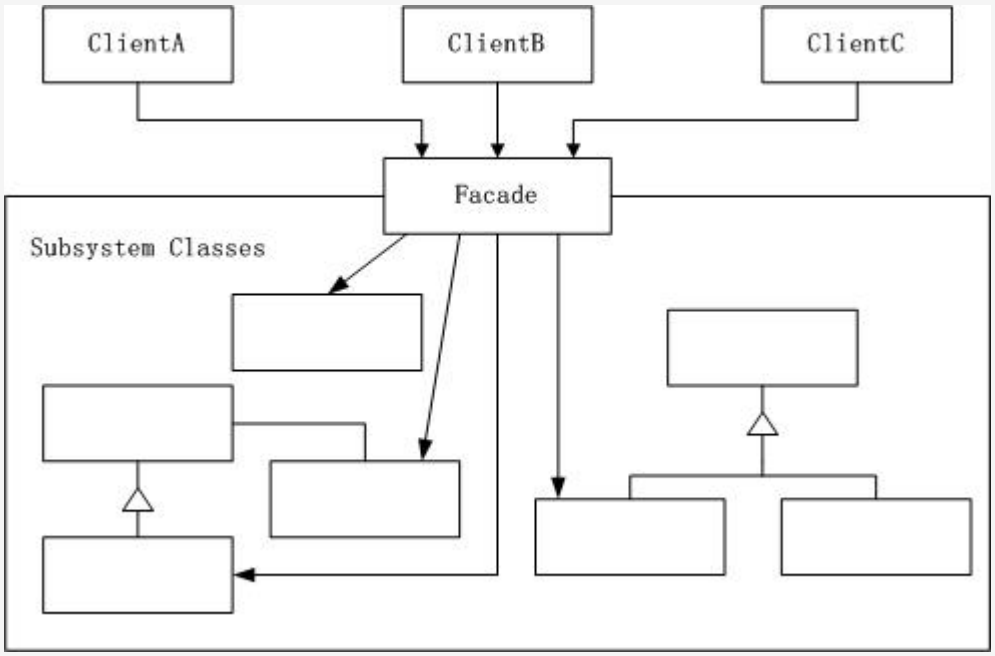
在软件开发系统中，客户程序经常会与复杂系统的内部子系统之间产生耦合，而导致客户程序随着子系统的变化而变化。那么如何简化客户程序与子系统之间的交互接口？如何将复杂系统的内部子系统与客户程序之间的依赖解耦？

意图(Intent):

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

---------《设计模式》GOF

结构图(Struct):



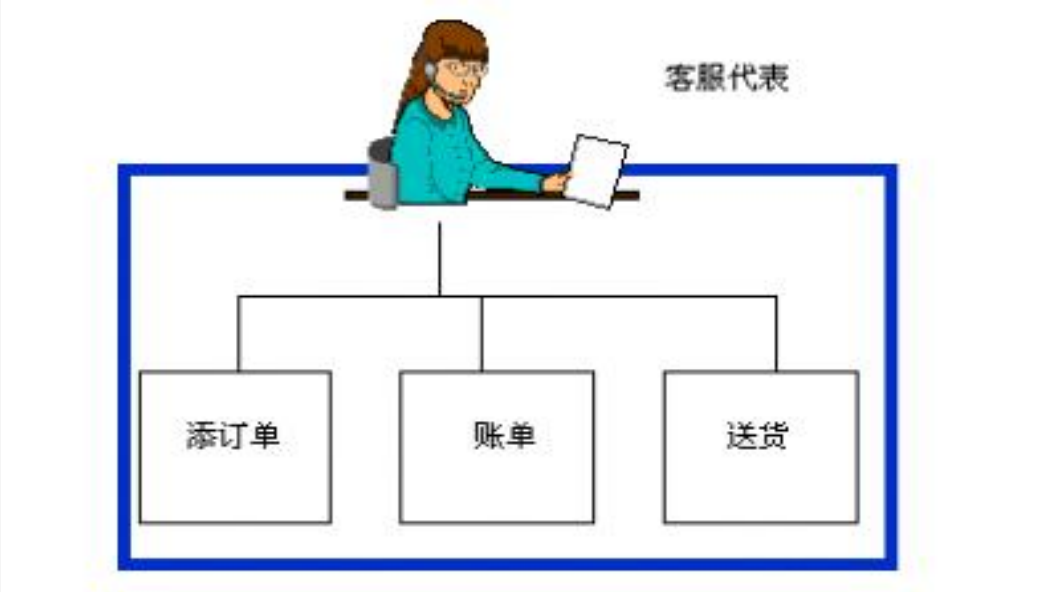
适用性：

1. 为一个复杂子系统提供一个简单接口。

2. 提高子系统的独立性。

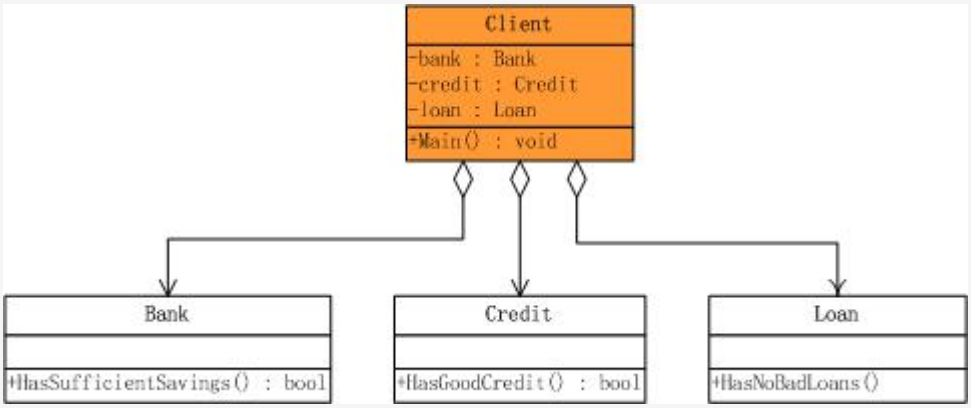
3. 在层次化结构中，可以使用 Facade 模式定义系统中每一层的入口。

生活中的例子：



代码实现：

我们平时的开发中其实已经不知不觉的在用 Façade 模式，现在来考虑这样一个抵押系统，当有一个客户来时，有如下几件事情需要确认：到银行子系统查询他是否有足够多的存款，到信用子系统查询他是否有良好的信用，到贷款子系统查询他有无贷款劣迹。只有这三个子系统都通过时才可进行抵押。我们先不考虑 Façade 模式，那么客户程序就要直接访问这些子系统，分别进行判断。类结构图下：



在这个程序中，我们首先要有一个顾客类，它是一个纯数据类，并无任何操作，示意代码：

```
1 //顾客类
2 public class Customer
3 {
4     private string _name;
5
6     public Customer(string name)
```

```
    {
        this._name = name;
    }

    public string Name
    {
        get { return _name; }
    }
}
```

下面这三个类均是子系统类，示意代码：

```
//银行子系统
public class Bank
{
    public bool HasSufficientSavings(Customer c, int amount)
    {
        Console.WriteLine("Check bank for " + c.Name);
        return true;
    }
}

//信用子系统
public class Credit
{
    public bool HasGoodCredit(Customer c)
    {
        Console.WriteLine("Check credit for " + c.Name);
        return true;
    }
}

//贷款子系统
public class Loan
{
```

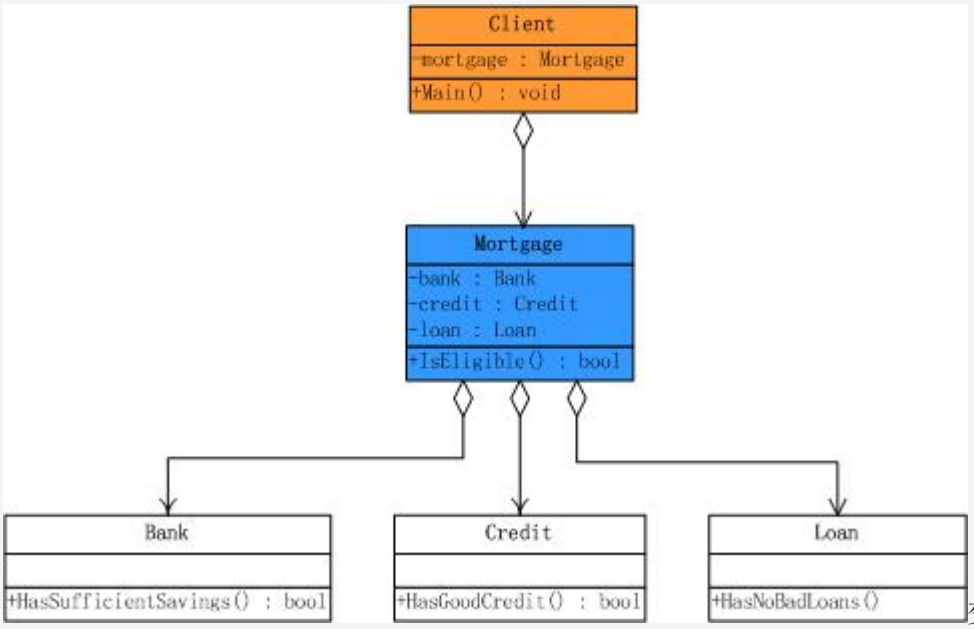
```
24    public bool HasNoBadLoans(Customer c)
25    {
26        Console.WriteLine("Check loans for " + c.Name);
27        return true;
28    }
29 }
```

看客户程序的调用：

```
 1 //客户程序
 2 public class MainApp
 3 {
 4    private const int _amount = 12000;
 5
 6    public static void Main()
 7    {
 8       Bank bank = new Bank();
 9       Loan loan = new Loan();
10        Credit credit = new Credit();
11
12        Customer customer = new Customer("Ann McKinsey");
13
14        bool eligible = true;
15
16        if (!bank.HasSufficientSavings(customer, _amount))
17        {
18            eligible = false;
19        }
20        else if (!loan.HasNoBadLoans(customer))
21        {
22            eligible = false;
23        }
24        else if (!credit.HasGoodCredit(customer))
25        {
26            eligible = false;
```

```
27        }
28
29        Console.WriteLine("\n" + customer.Name + " has been " + (eligible ? "A
pproved" : "Rejected"));
30        Console.ReadLine();
31    }
32 }
```

可以看到，在不用 Façade 模式的情况下，客户程序与三个子系统都发生了耦合，这种耦合使得客户程序依赖于子系统，当子系统化时，客户程序也将面临很多变化的挑战。一个合情合理的设计就是为这些子系统创建一个统一的接口，这个接口简化了客户程序的判断操作。看一下引入 Façade 模式后的类结构图：



变

外观类 Mortage 的实现如下：

```
1 /外观类
2 public class Mortgage
3 {
4     private Bank bank = new Bank();
5     private Loan loan = new Loan();
6     private Credit credit = new Credit();
7
```

```
8     public bool IsEligible(Customer cust, int amount)
9     {
10        Console.WriteLine("{0} applies for {1:C} loan\n",
11          cust.Name, amount);
12
13        bool eligible = true;
14
15        if (!bank.HasSufficientSavings(cust, amount))
16        {
17            eligible = false;
18        }
19        else if (!loan.HasNoBadLoans(cust))
20        {
21            eligible = false;
22        }
23        else if (!credit.HasGoodCredit(cust))
24        {
25            eligible = false;
26        }
27
28        return eligible;
29    }
30 }
```

顾客类和子系统类的实现仍然如下：

```
1 //银行子系统
2 public class Bank
3 {
4     public bool HasSufficientSavings(Customer c, int amount)
5     {
6         Console.WriteLine("Check bank for " + c.Name);
7         return true;
8     }
9 }
10
```

```
11 //信用证子系统
12 public class Credit
13 {
14     public bool HasGoodCredit(Customer c)
15     {
16         Console.WriteLine("Check credit for " + c.Name);
17         return true;
18     }
19 }
20
21 //贷款子系统
22 public class Loan
23 {
24     public bool HasNoBadLoans(Customer c)
25     {
26         Console.WriteLine("Check loans for " + c.Name);
27         return true;
28     }
29 }
30
31 //顾客类
32 public class Customer
33 {
34     private string name;
35
36     public Customer(string name)
37     {
38         this.name = name;
39     }
40
41     public string Name
42     {
43         get { return name; }
```

```
44     }
45 }
```

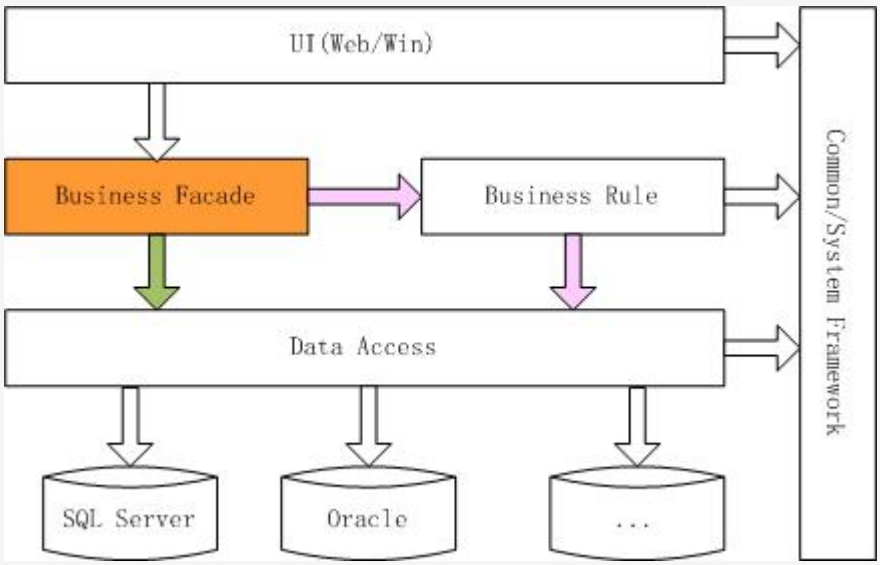
而此时客户程序的实现：

```
 1 //客户程序类
 2 public class MainApp
 3 {
 4     public static void Main()
 5     {
 6         //外观
 7         Mortgage mortgage = new Mortgage();
 8 
 9         Customer customer = new Customer("Ann McKinsey");
10         bool eligable = mortgage.IsEligible(customer, 125000);
11 
12         Console.WriteLine("\n" + customer.Name +
13             " has been " + (eligable ? "Approved" : "Rejected"));
14         Console.ReadLine();
15     }
16 }
```

可以看到引入 Façade 模式后，客户程序只与 Mortgage 发生依赖，也就是 Mortgage 屏蔽了子系统之间的复杂的操作，达到了解耦内部子系统与客户程序之间的依赖。

.NET 架构中的 Façade 模式

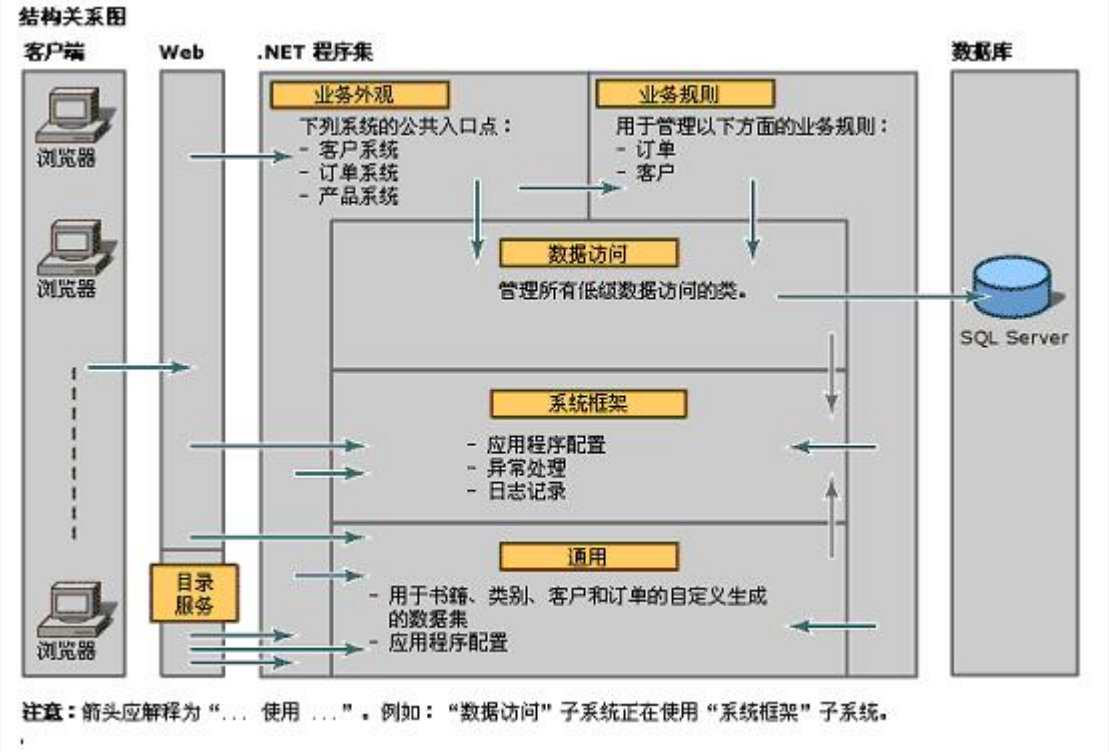
Façade 模式在实际开发中最多的运用当属开发 N 层架构的应用程序了，一个典型的 N 层结构如下：



在这个架构中，总共分为四个逻辑层，分别为：用户层 UI，业务外观层 Business Façade，业务规则层 Business Rule，数据访问层 Data Access。其中 Business Façade 层的职责如下：

- 从“用户”层接收用户输入
- 如果请求需要对数据进行只读访问，则可能使用“数据访问”层
- 将请求传递到“业务规则”层
- 将响应从“业务规则”层返回到“用户”层
- 在对“业务规则”层的调用之间维护临时状态

对这一架构最好的体现就是 Duwamish 示例了。在该应用程序中，有部分操作只是简单的从数据库根据条件提取数据，不需要经过任何处理，而直接将数据显示到网页上，比如查询某类别的图书列表。而另外一些操作，比如计算定单中图书的总价并根据顾客的级别计算回扣等等，这部分往往有许多不同的功能的类，操作起来也比较复杂。如果采用传统的三层结构，这些商业逻辑一般是会放在中间层，那么对内部的这些大量种类繁多，使用方法也各异的不同的类的调用任务，就完全落到了表示层。这样势必会增加表示层的代码量，将表示层的任务复杂化，和表示层只负责接受用户的输入并返回结果的任务不太相称，并增加了层与层之间的耦合程度。于是就引入了一个 Façade 层，让这个 Facade 来负责管理系统内部类的调用，并为表示层提供了一个单一而简单的接口。看一下 Duwamish 结构
图：



从图中可以看到，UI 层将请求发送给业务外观层，业务外观层对请求进行初步的处理，判断是否需要调用业务规则层，还是直接调用数据访问层获取数据。最后由数据访问层访问数据库并按照来时的步骤返回结果到 UI 层，来看具体的代码实现。

在获取商品目录的时候，Web UI 调用业务外观层：

```
1 productSystem = new ProductSystem();
2 categorySet   = productSystem.GetCategories(categoryID);
```

业务外观层直接调用了数据访问层：

```
 1 public CategoryData GetCategories(int categoryId)
 2 {
 3     //
 4     // Check preconditions
 5     //
 6     ApplicationAssert.CheckCondition(categoryId >= 0,"Invalid Category Id",Ap
plicationAssert.LineNumber);
 7     //
 8     // Retrieve the data
 9     //
```

```
10     using (Categories accessCategories = new Categories())
11     {
12         return accessCategories.GetCategories(categoryId);
13     }
14
15 }
```

在添加订单时，UI 调用业务外观层：

```
1 public void AddOrder()
2 {
3     ApplicationAssert.CheckCondition(cartOrderData != null, "Order requires dat
a", ApplicationAssert.LineNumber);
4
5     //Write trace log.
6     ApplicationLog.WriteTrace("Duwamish7.Web.Cart.AddOrder:\r\nCustomerId:
" +
7                               cartOrderData.Tables[OrderData.CUSTOMER_TABLE].Row
s[0][OrderData.PKID_FIELD].ToString());
8     cartOrderData = (new OrderSystem()).AddOrder(cartOrderData);
9 }
```

业务外观层调用业务规则层：

```
 1 public OrderData AddOrder(OrderData order)
 2 {
 3     //
 4     // Check preconditions
 5     //
 6     ApplicationAssert.CheckCondition(order != null, "Order is required", Applicat
ionAssert.LineNumber);
 7
 8     (new BusinessRules.Order()).InsertOrder(order);
 9     return order;
10 }
```

业务规则层进行复杂的逻辑处理后，再调用数据访问层：

```
1 public OrderData AddOrder(OrderData order)
2 {
3     //
4     // Check preconditions
5     //
6     ApplicationAssert.CheckCondition(order != null, "Order is required", Applicat
ionAssert.LineNumber);
7
8     (new BusinessRules.Order()).InsertOrder(order);
9     return order;
10 }
11
12
13 业务规则层进行复杂的逻辑处理后，再调用数据访问层：
14 public bool InsertOrder(OrderData order)
15 {
16     //
17     // Assume it's good
18     //
19     bool isValid = true;
20     //
21     // Validate order summary
22     //
23     DataRow summaryRow = order.Tables[OrderData.ORDER_SUMMARY_TABL
E].Rows[0];
24
25     summaryRow.ClearErrors();
26
27     if (CalculateShipping(order) != (Decimal)(summaryRow[OrderData.SHIPPIN
G_HANDLING_FIELD]))
28     {
29         summaryRow.SetColumnError(OrderData.SHIPPING_HANDLING_FIELD,
```

```
OrderData.INVALID_FIELD);
30          isValid = false;
31      }
32
33      if (CalculateTax(order) != (Decimal)(summaryRow[OrderData.TAX_FIELD]))
34      {
35          summaryRow.SetColumnError(OrderData.TAX_FIELD, OrderData.INVALI
D_FIELD);
36          isValid = false;
37      }
38      //
39      // Validate shipping info
40      //
41      isValid &= IsValidField(order, OrderData.SHIPPING_ADDRESS_TABLE, Orde
rData.SHIP_TO_NAME_FIELD, 40);
42      //
43      // Validate payment info
44      //
45      DataRow paymentRow = order.Tables[OrderData.PAYMENT_TABLE].Rows[0
];
46
47      paymentRow.ClearErrors();
48
49      isValid &= IsValidField(paymentRow, OrderData.CREDIT_CARD_TYPE_FIEL
D, 40);
50      isValid &= IsValidField(paymentRow, OrderData.CREDIT_CARD_NUMBER_FI
ELD,  32);
51      isValid &= IsValidField(paymentRow, OrderData.EXPIRATION_DATE_FIELD,
30);
52      isValid &= IsValidField(paymentRow, OrderData.NAME_ON_CARD_FIELD, 4
0);
53      isValid &= IsValidField(paymentRow, OrderData.BILLING_ADDRESS_FIELD,
 255);
54      //
```

```
55    // Validate the order items and recalculate the subtotal
56    //
57    DataRowCollection itemRows = order.Tables[OrderData.ORDER_ITEMS_TAB
LE].Rows;
58
59    Decimal subTotal = 0;
60
61    foreach (DataRow itemRow in itemRows)
62    {
63       itemRow.ClearErrors();
64
65       subTotal += (Decimal)(itemRow[OrderData.EXTENDED_FIELD]);
66
67       if ((Decimal)(itemRow[OrderData.PRICE_FIELD]) <= 0)
68       {
69          itemRow.SetColumnError(OrderData.PRICE_FIELD, OrderData.INVALI
D_FIELD);
70          isValid = false;
71       }
72
73       if ((short)(itemRow[OrderData.QUANTITY_FIELD]) <= 0)
74       {
75          itemRow.SetColumnError(OrderData.QUANTITY_FIELD, OrderData.IN
VALID_FIELD);
76          isValid = false;
77       }
78    }
79    //
80    // Verify the subtotal
81    //
82    if (subTotal != (Decimal)(summaryRow[OrderData.SUB_TOTAL_FIELD]))
83    {
84       summaryRow.SetColumnError(OrderData.SUB_TOTAL_FIELD, OrderData
.INVALID_FIELD);
```

```
85          isValid = false;
86      }
87
88      if ( isValid )
89      {
90          using (DataAccess.Orders ordersDataAccess = new DataAccess.Orders())
91          {
92              return (ordersDataAccess.InsertOrderDetail(order)) > 0;
93          }
94      }
95      else
96          return false;
97  }
```

Facade 模式的个要点：

从客户程序的角度来看，Facade 模式不仅简化了整个组件系统的接口，同时对于组件内部与外部客户程序来说，从某种程度上也达到了一种“解耦”的效果----内部子系统的任何变化不会影响到 Facade 接口的变化。

Facade 设计模式更注重从架构的层次去看整个系统，而不是单个类的层次。Facdae 很多时候更是一种架构
设计模式。

注意区分 Facade 模式、Adapter 模式、Bridge 模式与 Decorator 模式。Facade 模式注重简化接口，Adapter 模式注重转换接口，Bridge 模式注重分离接口（抽象）与其实现，Decorator 模式注重稳定接口的前提下为对象扩展功能。

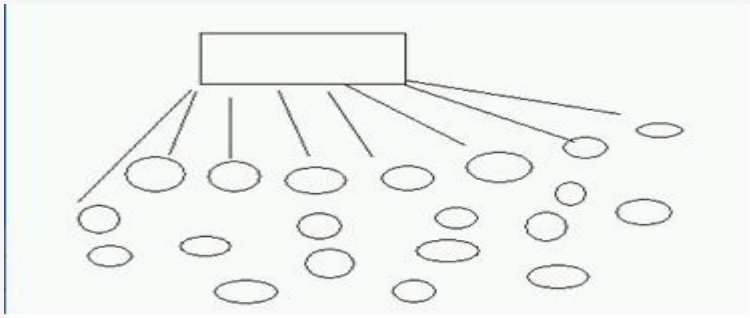
享元模式(Flyweight Pattern)

面向对象的代价

面向对象很好地解决了系统抽象性的问题，同时在大多数情况下，也不会损及系统的性能。但是，在

某些特殊的应用中下，由于对象的数量太大，采用面向对象会给系统带来难以承受的内存开销。比如:

图形应用中的图元等对象、字处理应用中的字符对象等。



动机(Motivate):

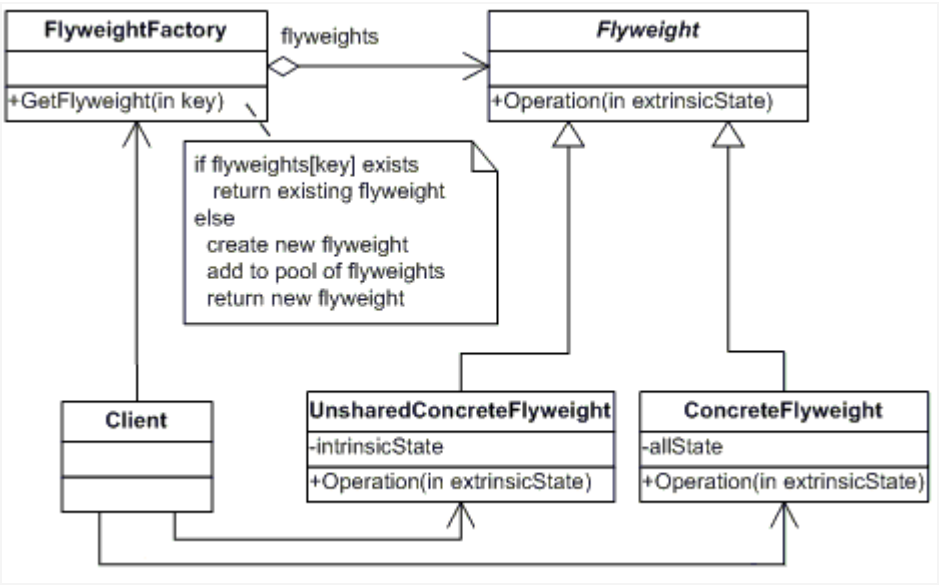
采用纯粹对象方案的问题在于大量细粒度的对象会很快充斥在系统中，从而带来很高的运行时代价--------主要指内存需求方面的代价。

如何在避免大量细粒度对象问题的同时，让外部客户程序仍然能够透明地使用面向对象的方式来进行操作?

意图(Intent):

运用共享技术有效地支持大量细粒度的对象。 --------《设计模式》GOF

结构(Struct):



适用性：

当以下所有的条件都满足时，可以考虑使用享元模式：

1、 一个系统有大量的对象。

2、 这些对象耗费大量的内存。

3、 这些对象的状态中的大部分都可以外部化。

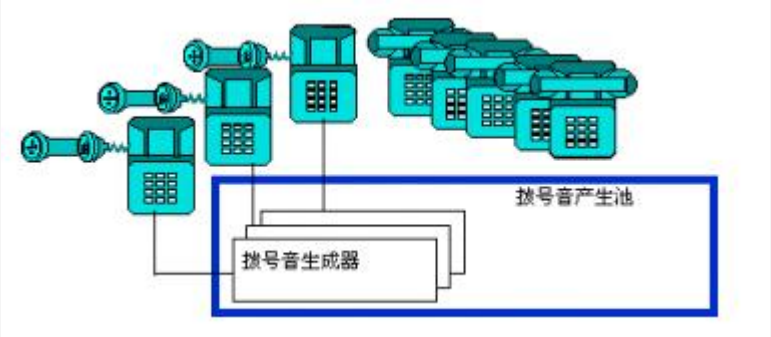
4、 这些对象可以按照内蕴状态分成很多的组，当把外蕴对象从对象中剔除时，每一个组都可以仅用一个对象代替。

5、 软件系统不依赖于这些对象的身份，换言之，这些对象可以是不可分辨的。

满足以上的这些条件的系统可以使用享元对象。最后，使用享元模式需要维护一个记录了系统已有的所有享元的表，而这需要耗费资源。因此，应当在有足够多的享元实例可供共享时才值得使用享元模式。

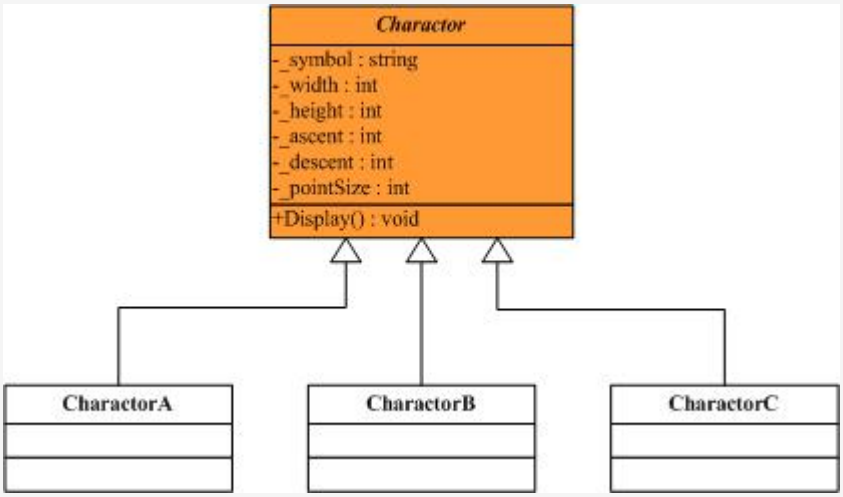
生活中的例子：

享元模式使用共享技术有效地支持大量细粒度的对象。公共交换电话网（PSTN）是享元的一个例子。有一些资源例如拨号音发生器、振铃发生器和拨号接收器是必须由所有用户共享的。当一个用户拿起听筒打电话时，他不需要知道使用了多少资源。对于用户而言所有的事情就是有拨号音，拨打号码，拨通电话。



代码实现：

Flyweight 在拳击比赛中指最轻量级，即“蝇量级”，这里翻译为“享元”，可以理解为共享元对象（细粒度对象）的意思。提到 Flyweight 模式都会一般都会用编辑器例子来说明，这里也不例外，但我会尝试着通过重构来看待 Flyweight 模式。考虑这样一个字处理软件，它需要处理的对象可能有单个的字符，由字符组成的段落以及整篇文档，根据面向对象的设计思想和 Composite 模式，不管是字符还是段落，文档都应该作为单个的对象去看待，这里只考虑单个的字符，不考虑段落及文档等对象，于是可以很容易的得到下面的结构图：



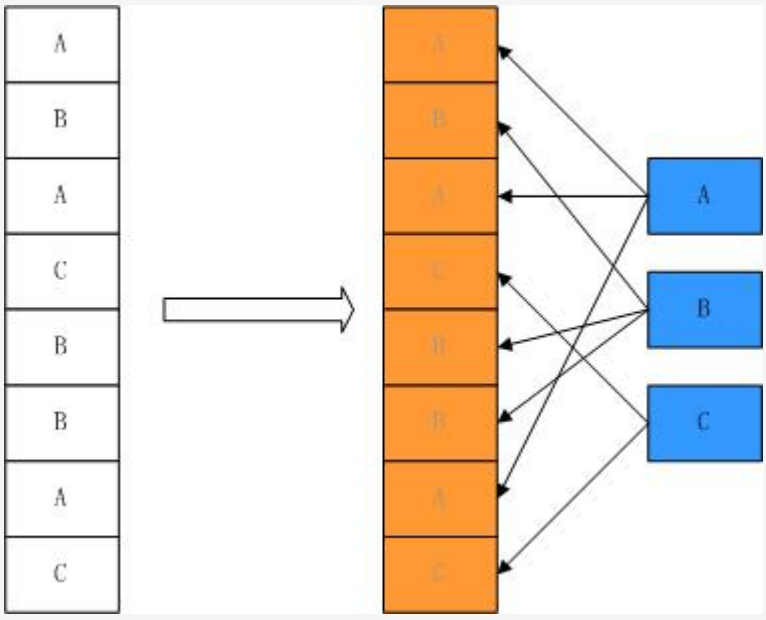

```
1 // "Charactor"
2 public abstract class Charactor
3 {
4     //Fields
5     protected char _symbol;
6
7     protected int _width;
8
```

```
 9    protected int _height;
10
11    protected int _ascent;
12
13    protected int _descent;
14
15    protected int _pointSize;
16
17    //Method
18    public abstract void Display();
19 }
20
21 // "CharactorA"
22 public class CharactorA : Charactor
23 {
24    // Constructor
25    public CharactorA()
26    {
27      this._symbol = 'A';
28      this._height = 100;
29      this._width = 120;
30      this._ascent = 70;
31      this._descent = 0;
32      this._pointSize = 12;
33    }
34
35    //Method
36    public override void Display()
37    {
38       Console.WriteLine(this._symbol);
39    }
40 }
41
42 // "CharactorB"
```

```
43 public class CharactorB : Charactor
44 {
45     // Constructor
46     public CharactorB()
47     {
48         this._symbol = 'B';
49         this._height = 100;
50         this._width = 140;
51         this._ascent = 72;
52         this._descent = 0;
53         this._pointSize = 10;
54     }
55 
56     //Method
57     public override void Display()
58     {
59         Console.WriteLine(this._symbol);
60     }
61 }
62 
63 // "CharactorC"
64 public class CharactorC : Charactor
65 {
66     // Constructor
67     public CharactorC()
68     {
69         this._symbol = 'C';
70         this._height = 100;
71         this._width = 160;
72         this._ascent = 74;
73         this._descent = 0;
74         this._pointSize = 14;
75     }
76 
```

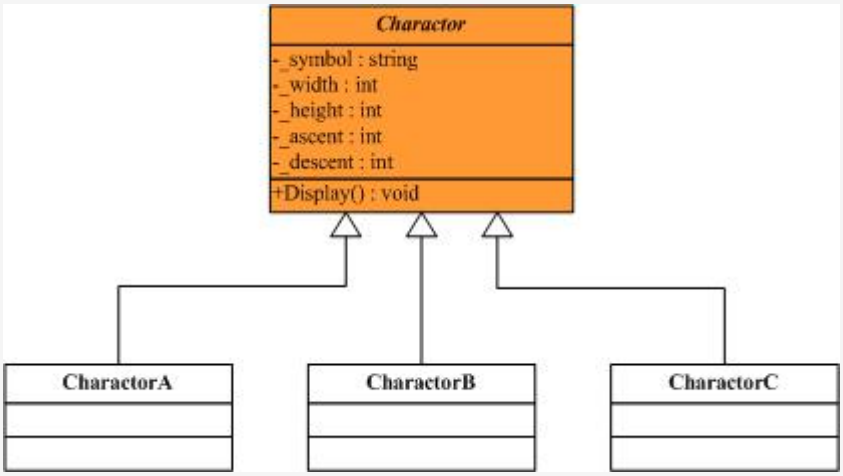
```
77    //Method
78    public override void Display()
79    {
80        Console.WriteLine(this._symbol);
81    }
82 }
```

好了，现在看到的这段代码可以说是很好地符合了面向对象的思想，但是同时我们也为此付出了沉重的代价，那就是性能上的开销，可以想象，在一篇文档中，字符的数量远不止几百个这么简单，可能上千上万，内存中就同时存在了上千上万个 Charactor 对象，这样的内存开销是可想而知的。进一步分析可以发现，虽然我们需要的 Charactor 实例非常多，这些实例之间只不过是状态不同而已，也就是说这些实例的状态数量是很少的。所以我们并不需要这么多的独立的 Charactor 实例，而只需要为每一种 Charactor 状态创建一个实例，让整个字符处理软件共享这些实例就可以了。看这样一幅示意图：



现在我们看到的 A，B，C 三个字符是共享的，也就是说如果文档中任何地方需要这三个字符，只需要使用共享的这三个实例就可以了。然而我们发现单纯的这样共享也是有问题的。虽然文档中的用到了很多的 A 字符，虽然字符的 symbol 等是相同的，它可以共享；但是它们的 pointSize 却是不相同的，即字符在文档中中的大小是不相同的，这个状态不可以共享。为解决这个问题，首先我们将不可共享的状态从类里面剔除出去，即去掉 pointSize 这个状态（只是

暂时的☺），类结构图如下所示：

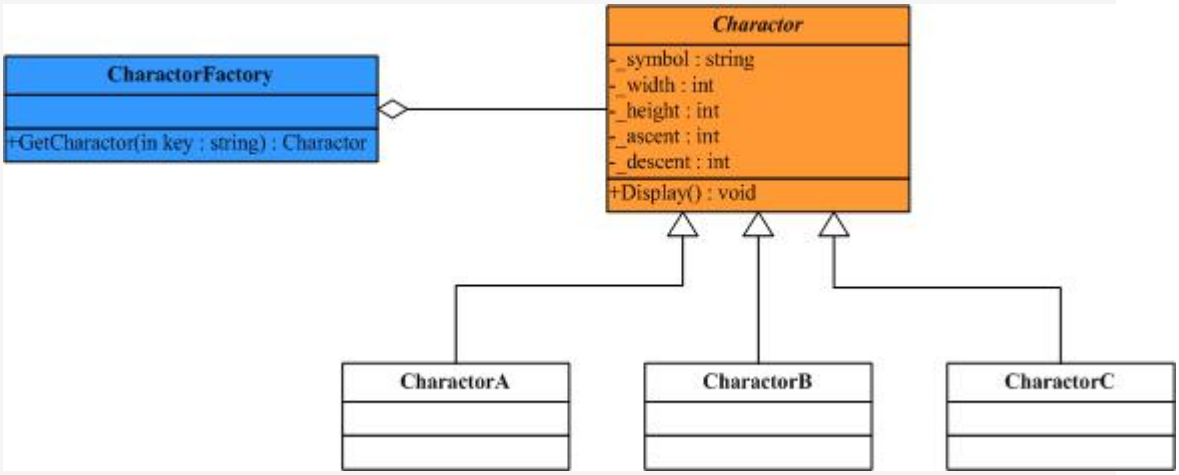



```
1 // "Charactor"
2 public abstract class Charactor
3 {
4     //Fields
5     protected char _symbol;
6
7     protected int _width;
8
9     protected int _height;
10
11     protected int _ascent;
12
13     protected int _descent;
14
15     //Method
16     public abstract void Display();
17 }
18
19 // "CharactorA"
20 public class CharactorA : Charactor
21 {
22     // Constructor
23     public CharactorA()
24     {
```

```
25        this._symbol = 'A';
26        this._height = 100;
27        this._width = 120;
28        this._ascent = 70;
29        this._descent = 0;
30    }
31
32    //Method
33    public override void Display()
34    {
35        Console.WriteLine(this._symbol);
36    }
37 }
38
39 // "CharactorB"
40 public class CharactorB : Charactor
41 {
42    // Constructor
43    public CharactorB()
44    {
45        this._symbol = 'B';
46        this._height = 100;
47        this._width = 140;
48        this._ascent = 72;
49        this._descent = 0;
50    }
51
52    //Method
53    public override void Display()
54    {
55        Console.WriteLine(this._symbol);
56    }
57 }
58
```

```
59 // "CharactorC"
60 public class CharactorC : Charactor
61 {
62     // Constructor
63     public CharactorC()
64     {
65         this._symbol = 'C';
66         this._height = 100;
67         this._width = 160;
68         this._ascent = 74;
69         this._descent = 0;
70     }
71 
72     //Method
73     public override void Display()
74     {
75         Console.WriteLine(this._symbol);
76     }
77 }
```

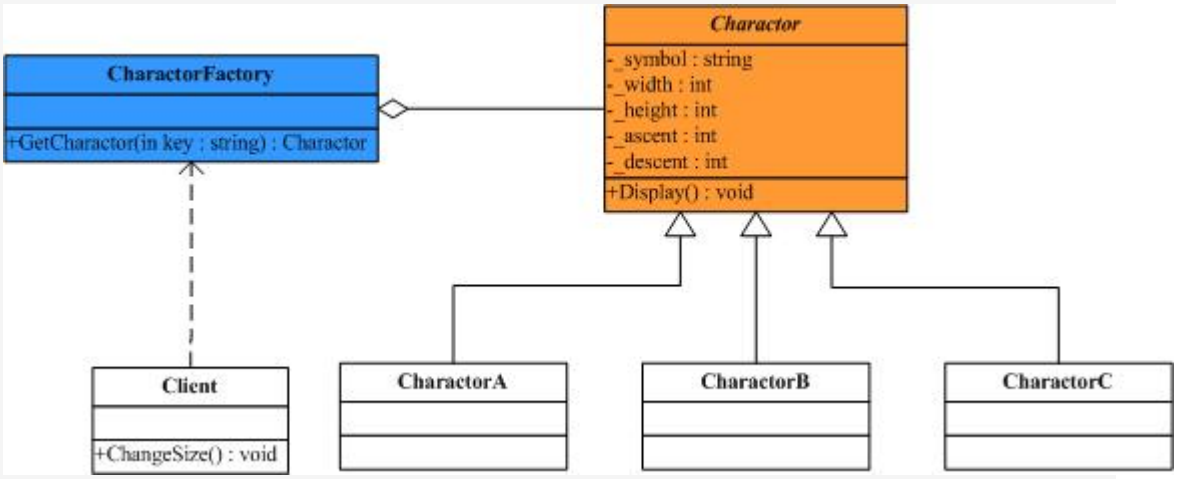
好，现在类里面剩下的状态都可以共享了，下面我们要做的工作就是控制 Charactor 类的创建过程，即如果已经存在了“A”字符这样的实例，就不需要再创建，直接返回实例；如果没有，则创建一个新的实例。如果把这项工作交给 Charactor 类，即 Charactor 类在负责它自身职责的同时也要负责管理 Charactor 实例的管理工作，这在一定程度上有可能违背类的单一职责原则，因此，需要一个单独的类来做这项工作，引入 CharactorFactory 类，结构图如下：




```
1 // "CharactorFactory"
2 public class CharactorFactory
3 {
4     // Fields
5     private Hashtable charactors = new Hashtable();
6
7     // Constructor
8     public CharactorFactory()
9     {
10        charactors.Add("A", new CharactorA());
11        charactors.Add("B", new CharactorB());
12        charactors.Add("C", new CharactorC());
13    }
14
15    // Method
16    public Charactor GetCharactor(string key)
17    {
18        Charactor charactor = charactors[key] as Charactor;
19
20        if (charactor == null)
21        {
22            switch (key)
23            {
24                case "A": charactor = new CharactorA(); break;
```

```
25              case "B": charactor = new CharactorB(); break;
26              case "C": charactor = new CharactorC(); break;
27              //
28          }
29          charactors.Add(key, charactor);
30      }
31      return charactor;
32    }
33 }
```

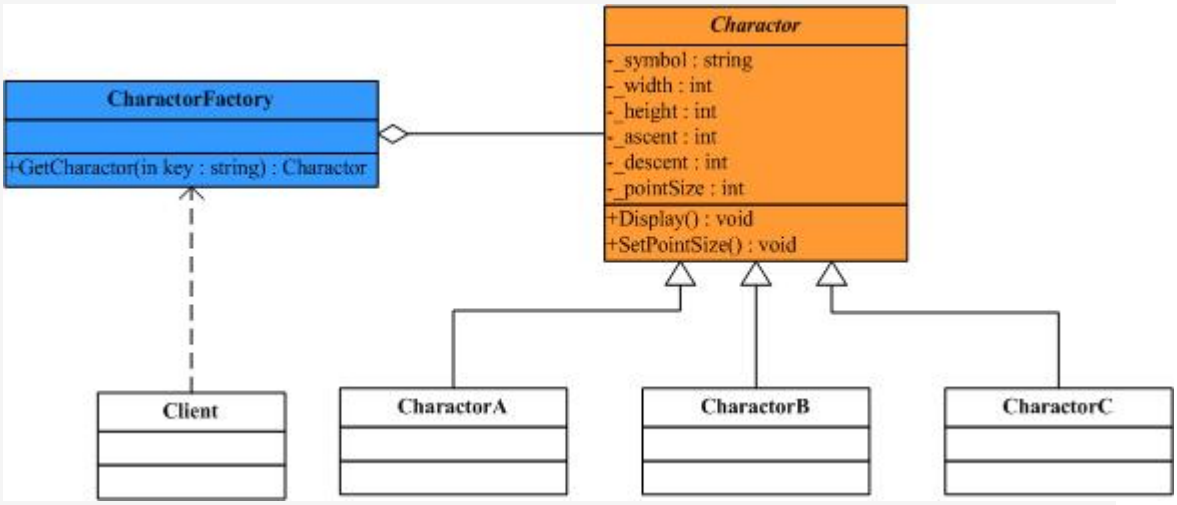
到这里已经完全解决了可以共享的状态（这里很丑陋的一个地方是出现了 switch 语句，但这可以通过别的办法消除，为了简单期间我们先保持这种写法）。下面的工作就是处理刚才被我们剔除出去的那些不可共享的状态，因为虽然将那些状态移除了，但是 Charactor 对象仍然需要这些状态，被我们剥离后这些对象根本就无法工作，所以需要将这些状态外部化。首先会想到一种比较简单的解决方案就是对于不能共享的那些状态，不需要去在 Charactor 类中设置，而直接在客户程序代码中进行设置，类结构图如下：




```
1 public class Program
2 {
3   public static void Main()
4   {
5     Charactor ca = new CharactorA();
6     Charactor cb = new CharactorB();
7     Charactor cc = new CharactorC();
8
```

```
9         //显示字符
10
11         //设置字符的大小 ChangeSize();
12     }
13
14     public void ChangeSize()
15     {
16         //在这里设置字符的大小
17     }
18 }
```

按照这样的实现思路，可以发现如果有多个客户端程序使用的话，会出现大量的重复性的逻辑，用重构的术语来说是出现了代码的坏味道，不利于代码的复用和维护；另外把这些状态和行为移到客户程序里面破坏了封装性的原则。再次转变我们的实现思路，可以确定的是这些状态仍然属于 Charactor 对象，所以它还是应该出现在 Charactor 类中，对于不同的状态可以采取在客户程序中通过参数化的方式传入。类结构图如下：




```
1 // "Charactor"
2 public abstract class Charactor
3 {
4     //Fields
5     protected char _symbol;
6
7     protected int _width;
```

```
 8
 9    protected int _height;
10
11    protected int _ascent;
12
13    protected int _descent;
14
15    protected int _pointSize;
16
17    //Method
18    public abstract void SetPointSize(int size);
19    public abstract void Display();
20 }
21
22 // "CharactorA"
23 public class CharactorA : Charactor
24 {
25    // Constructor
26    public CharactorA()
27    {
28       this._symbol = 'A';
29       this._height = 100;
30       this._width = 120;
31       this._ascent = 70;
32       this._descent = 0;
33    }
34
35    //Method
36    public override void SetPointSize(int size)
37    {
38       this._pointSize = size;
39    }
40
41    public override void Display()
```

```
42    {
43        Console.WriteLine(this._symbol +
44          "pointsize:" + this._pointSize);
45    }
46 }
47
48 // "CharactorB"
49 public class CharactorB : Charactor
50 {
51    // Constructor
52    public CharactorB()
53    {
54        this._symbol = 'B';
55        this._height = 100;
56        this._width = 140;
57        this._ascent = 72;
58        this._descent = 0;
59    }
60
61    //Method
62    public override void SetPointSize(int size)
63    {
64        this._pointSize = size;
65    }
66
67    public override void Display()
68    {
69        Console.WriteLine(this._symbol +
70          "pointsize:" + this._pointSize);
71    }
72 }
73
74 // "CharactorC"
75 public class CharactorC : Charactor
```

```
76 {
77     // Constructor
78     public CharactorC()
79     {
80         this._symbol = 'C';
81         this._height = 100;
82         this._width = 160;
83         this._ascent = 74;
84         this._descent = 0;
85     }
86 
87     //Method
88     public override void SetPointSize(int size)
89     {
90         this._pointSize = size;
91     }
92 
93     public override void Display()
94     {
95         Console.WriteLine(this._symbol +
96           "pointsize:" + this._pointSize);
97     }
98 }
99 
100 // "CharactorFactory"
101 public class CharactorFactory
102 {
103     // Fields
104     private Hashtable charactors = new Hashtable();
105 
106     // Constructor
107     public CharactorFactory()
108     {
109         charactors.Add("A", new CharactorA());
```

```
110         charactors.Add("B", new CharactorB());
111         charactors.Add("C", new CharactorC());
112     }
113 
114     // Method
115     public Charactor GetCharactor(string key)
116     {
117         Charactor charactor = charactors[key] as Charactor;
118 
119         if (charactor == null)
120         {
121             switch (key)
122             {
123                 case "A": charactor = new CharactorA(); break;
124                 case "B": charactor = new CharactorB(); break;
125                 case "C": charactor = new CharactorC(); break;
126                 //
127             }
128             charactors.Add(key, charactor);
129         }
130         return charactor;
131     }
132 }
133 
134 public class Program
135 {
136     public static void Main()
137     {
138         CharactorFactory factory = new CharactorFactory();
139 
140         // Charactor "A"
141         CharactorA ca = (CharactorA)factory.GetCharactor("A");
142         ca.SetPointSize(12);
143         ca.Display();
```

```
144
145         // Charactor "B"
146         CharactorB cb = (CharactorB)factory.GetCharactor("B");
147         ca.SetPointSize(10);
148         ca.Display();
149
150         // Charactor "C"
151         CharactorC cc = (CharactorC)factory.GetCharactor("C");
152         ca.SetPointSize(14);
153         ca.Display();
154     }
155 }
```

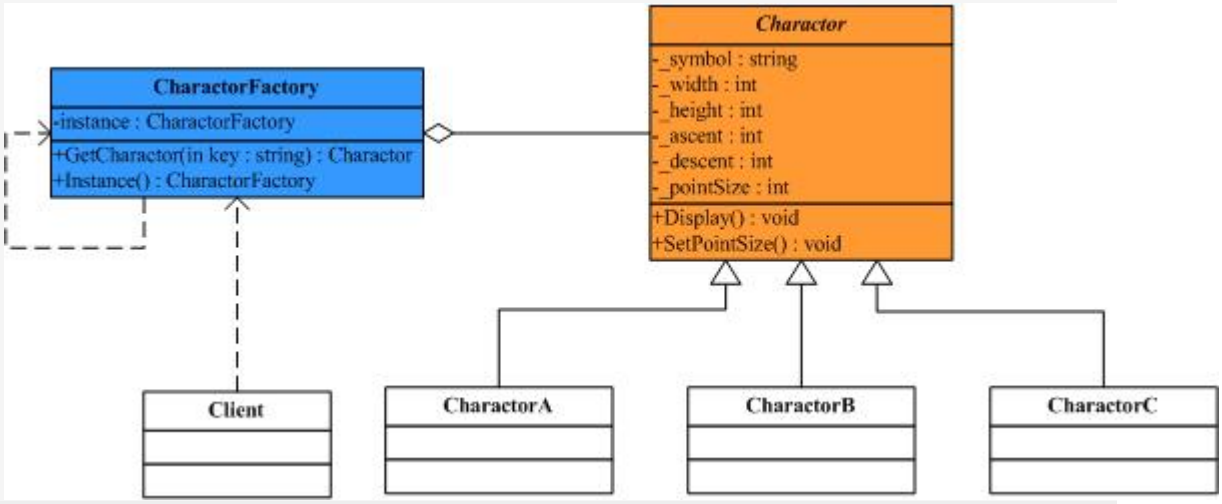
可以看到这样的实现明显优于第一种实现思路。好了，到这里我们就到到了通过 Flyweight 模式实现了优化资源的这样一个目的。在这个过程中，还有如下几点需要说明：

1．引入 CharactorFactory 是个关键，在这里创建对象已经不是 new 一个 Charactor 对象那么简单，而必须用工厂方法封装起来。

2．在这个例子中把 Charactor 对象作为 Flyweight 对象是否准确值的考虑，这里只是为了说明 Flyweight 模式，至于在实际应用中，哪些对象需要作为 Flyweight 对象是要经过很好的计算得知，而绝不是凭空臆想。

3．区分内外部状态很重要，这是享元对象能做到享元的关键所在。

到这里，其实我们的讨论还没有结束。有人可能会提出如下问题，享元对象（Charactor）在这个系统中相对于每一个内部状态而言它是唯一的，这跟单件模式有什么区别呢？这个问题已经很好回答了，那就是单件类是不能直接被实例化的，而享元类是可以被实例化的。事实上在这里面真正被设计为单件的应该是享元工厂（不是享元）类，因为如果创建很多个享元工厂的实例，那我们所做的一切努力都是白费的，并没有减少对象的个数。修改后的类结构图如下：




```
1 // "CharactorFactory"
2 public class CharactorFactory
3 {
4     // Fields
5     private Hashtable charactors = new Hashtable();
6
7     private CharactorFactory instance;
8     // Constructor
9     private CharactorFactory()
10    {
11        charactors.Add("A", new CharactorA());
12        charactors.Add("B", new CharactorB());
13        charactors.Add("C", new CharactorC());
14    }
15
16    // Property
17    public CharactorFactory Instance
18    {
19        get
20        {
21            if (instance != null)
22            {
23                instance = new CharactorFactory();
24            }
```

```
25          return instance;
26      }
27    }
28
29    // Method
30    public Charactor GetCharactor(string key)
31    {
32      Charactor charactor = charactors[key] as Charactor;
33
34      if (charactor == null)
35      {
36        switch (key)
37        {
38          case "A": charactor = new CharactorA(); break;
39          case "B": charactor = new CharactorB(); break;
40          case "C": charactor = new CharactorC(); break;
41          //
42        }
43        charactors.Add(key, charactor);
44      }
45      return charactor;
46    }
47 }
```

.NET 框架中的应用：

Flyweight 更多时候的时候一种底层的设计模式，在我们的实际应用程序中使用的并不是很多。在.NET 中的 String 类型其实就是运用了 Flyweight 模式。可以想象，如果每次执行 string s1 = “abcd”操作，都创建一个新的字符串对象的话，内存的开销会很大。所以.NET 中如果第一次创建了这样的一个字符串对象 s1，下次再创建相同的字符串 s2 时只是把它的引用指向“abcd”，这样就实现了“abcd”在内存中的共享。可以通过下面一个简单的程序来演示 s1 和 s2 的引用是否一致：

```
1 public class Program
2 {
3    public static void Main(string[] args)
4    {
5       string s1 = "abcd";
6       string s2 = "abcd";
7
8       Console.WriteLine(Object.ReferenceEquals(s1,s2));
9
10       Console.ReadLine();
11    }
12 }
```

Flyweight 实现要点：

1．面向对象很好的解决了抽象性的问题，但是作为一个运行在机器中的程序实体，我们需要考虑对象的代价问题。Flyweight 设计模式主要解决面向对象的代价问题，一般不触及面向对象的抽象性问题。

2．Flyweight 采用对象共享的做法来降低系统中对象的个数，从而降低细粒度对象给系统带来的内存压力。在具体实现方面，要注意对象状态的处理。

3．享元模式的优点在于它大幅度地降低内存中对象的数量。但是，它做到这一点所付出的代价也是很高的：享元模式使得系统更加复杂。为了使对象可以共享，需要将一些状态外部化，这使得程序的逻辑复杂化。另外它将享元对象的状态外部化，而读取外部状态使得运行时间稍微变长。

装饰模式(Decorator Pattern)

子类复子类，子类何其多

假如我们需要为游戏中开发一种坦克，除了各种不同型号的坦克外，我们还希望在不同场合中为其增加以下一种或多种功能;比如红外线夜视功能，比如水陆两栖功能，比如卫星定位功能等等。

按类继承的作法如下：

```
1  //抽象坦克
2  public abstract class Tank
3     {
4        public abstract void Shot();
5        public abstract void Run();
6     }
```

各种型号：

```
1 //T50 型号
2    public class T50:Tank
3     {
4        public override void Shot()
5        {
6           Console.WriteLine("T50 坦克平均每秒射击 5 发子弹");
7        }
8        public override void Run()
9        {
10          Console.WriteLine("T50 坦克平均每时运行 30 公里");
11       }
12    }
```

```
1 //T75 型号
2  public class T75 : Tank
3     {
4        public override void Shot()
5        {
6           Console.WriteLine("T75 坦克平均每秒射击 10 发子弹");
7        }
```

```
8        public override void Run()
9        {
10           Console.WriteLine("T75 坦克平均每时运行 35 公里");
11       }
12    }
```

```
1 //T90 型号
2  public class T90 :Tank
3    {
4       public override void Shot()
5       {
6          Console.WriteLine("T90 坦克平均每秒射击 10 发子弹");
7       }
8       public override void Run()
9       {
10          Console.WriteLine("T90 坦克平均每时运行 40 公里");
11       }
12    }
```

各种不同功能的组合：比如 IA 具有红外功能接口、IB 具有水陆两栖功能接口、IC 具有卫星定位功能接口。

```
1 //T50 坦克各种功能的组合
2 public class T50A:T50,IA
3 {
4   //具有红外功能
5 }
6 public class T50B:T50,IB
7 {
8  //具有水陆两栖功能
9 }
10 public class T50C:T50,IC
11 {
12
13 }
14 public class T50AB:T50,IA,IB
```

```
15 {}
18 public class T50AC:T50,IA,IC
19 {}
20 public class T50BC:T50,IB,IC
21 {}
22 public class T50ABC:T50,IA,IB,IC
23 {}
```

```
1
2  //T75 各种不同型号坦克各种功能的组合
3  public class T75A:T75,IA
4  {
5    //具有红外功能
6  }
7  public class T75B:T75,IB
8  {
9   //具有水陆两栖功能
10  }
11 public class T75C:T75,IC
12 {
13   //具有卫星定位功能
14 }
15 public class T75AB:T75,IA,IB
16 {
17  //具有红外、水陆两栖功能
18 }
19 public class T75AC:T75,IA,IC
20 {
21  //具有红外、卫星定位功能
22 }
23 public class T75BC:T75,IB,IC
24 {
25 //具有水陆两栖、卫星定位功能
26 }
```

```
27  public class T75ABC:T75,IA,IB,IC
28  {
29   //具有红外、水陆两栖、卫星定位功能
30 }
```

```
 1
 2  //T90 各种不同型号坦克各种功能的组合
 3  public class T90A:T90,IA
 4  {
 5    //具有红外功能
 6  }
 7  public class T90B:T90,IB
 8  {
 9   //具有水陆两栖功能
10  }
11  public class T90C:T90,IC
12  {
13    //具有卫星定位功能
14  }
15  public class T90AB:T90,IA,IB
16  {
17   //具有红外、水陆两栖功能
18 }
19  public class T90AC:T90,IA,IC
20  {
21   //具有红外、卫星定位功能
22 }
23  public class T90BC:T90,IB,IC
24  {
25  //具有水陆两栖、卫星定位功能
26 }
27  public class T90ABC:T90,IA,IB,IC
28  {
```

```
29   //具有红外、水陆两栖、卫星定位功能
30 }
```

由此可见，如果用类继承实现，子类会爆炸式地增长。

动机(Motivate)：

上述描述的问题根源在于我们“过度地使用了继承来扩展对象的功能”，由于继承为类型引入的静态物质，使得这种扩展方式缺乏灵活性;并且随着子类的增多（扩展功能的增多），各种子类的组合（扩展功能组合）会导致更多子类的膨胀（多继承）。

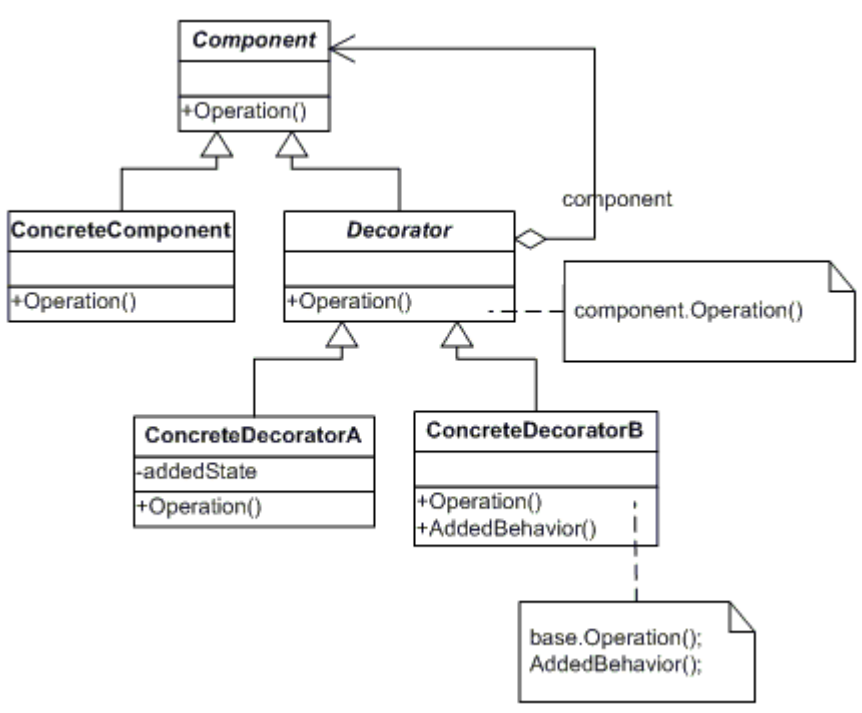
如何使“对象功能的扩展”能够根据需要来动态地实现？同时避免“扩展功能的增多”带来的子类膨胀问题？从而使得任何“功能扩展变化”所导致的影响将为最低？

意图(Intent)：

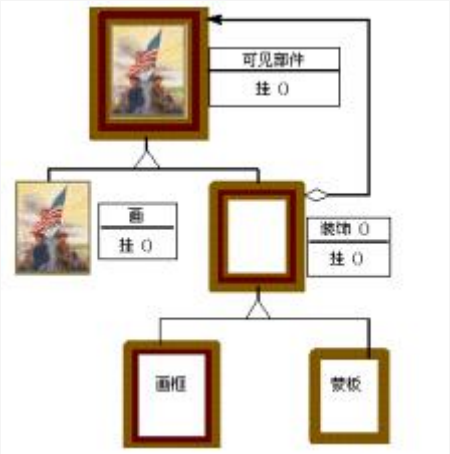
动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。

------《设计模式》GOF

结构图(Struct)：



生活中的例子：



适用性：

需要扩展一个类的功能，或给一个类增加附加责任。

需要动态地给一个对象增加功能，这些功能可以再动态地撤销。

需要增加由一些基本功能的排列组合而产生的非常大量的功能，从而使继承关系变得不现实。

实现代码：

```
1 namespace Decorator
2 {
3   public abstract class Tank
4     {
5       public abstract void Shot();
6       public abstract void Run();
7     }
8 }
```

```
1 namespace Decorator
2 {
3   public class T50:Tank
4     {
5       public override void Shot()
```

```
6       {
7           Console.WriteLine("T50 坦克平均每秒射击 5 发子弹");
8       }
9       public override void Run()
10      {
11          Console.WriteLine("T50 坦克平均每时运行 30 公里");
12      }
13   }
14 }
```

```
1 namespace Decorator
2 {
3    public class T75 : Tank
4    {
5       public override void Shot()
6       {
7          Console.WriteLine("T75 坦克平均每秒射击 10 发子弹");
8       }
9       public override void Run()
10      {
11          Console.WriteLine("T75 坦克平均每时运行 35 公里");
12      }
13   }
14 }
```

```
1 namespace Decorator
2 {
3    public class T90 :Tank
4    {
5       public override void Shot()
6       {
7          Console.WriteLine("T90 坦克平均每秒射击 10 发子弹");
8       }
9       public override void Run()
```

```
10        {
11            Console.WriteLine("T90 坦克平均每时运行 40 公里");
12        }
13    }
14 }
```

```
1 namespace Decorator
2 {
3   public abstract class Decorator :Tank //Do As 接口继承 非实现继承
4    {
5      private Tank tank; //Has a  对象组合
6      public Decorator(Tank tank)
7      {
8         this.tank = tank;
9      }
10      public override void Shot()
11      {
12         tank.Shot();
13      }
14      public override void Run()
15      {
16         tank.Run();
17      }
18    }
19 }
20
```

```
1
2 namespace Decorator
3 {
4   public class DecoratorA :Decorator
5    {
6      public DecoratorA(Tank tank) : base(tank)
7      {
```

```
8       }
9      public override void Shot()
10     {
11        //Do some extension //功能扩展 且有红外功能
12         base.Shot();
13     }
14    public override void Run()
15    {
16
17       base.Run();
18    }
19    }
20 }
```

```
1 namespace Decorator
2 {
3    public class DecoratorB :Decorator
4    {
5     public DecoratorB(Tank tank) : base(tank)
6      {
7      }
8     public override void Shot()
9     {
10       //Do some extension //功能扩展 且有水陆两栖功能
11       base.Shot();
12     }
13    public override void Run()
14    {
15
16       base.Run();
17    }
18   }
19 }
20
```

```
namespace Decorator
{
   public class DecoratorC :Decorator
   {
     public DecoratorC(Tank tank) : base(tank)
     {
     }
    public override void Shot()
     {
        //Do some extension //功能扩展 且有卫星定位功能
        base.Shot();
     }
   public override void Run()
   {

      base.Run();
   }

  }
}
```

```
  class Program
   {
      static void Main(string[] args)
      {
         Tank tank = new T50();
         DecoratorA da = new DecoratorA(tank); //且有红外功能
         DecoratorB db = new DecoratorB(da);   //且有红外和水陆两栖功能
         DecoratorC dc = new DecoratorC(db);   //且有红外、水陆两栖、卫星定们三种功能
         dc.Shot();
          dc.Run();
```

```
11        }
12    }
```

Decorator 模式的几个要点：

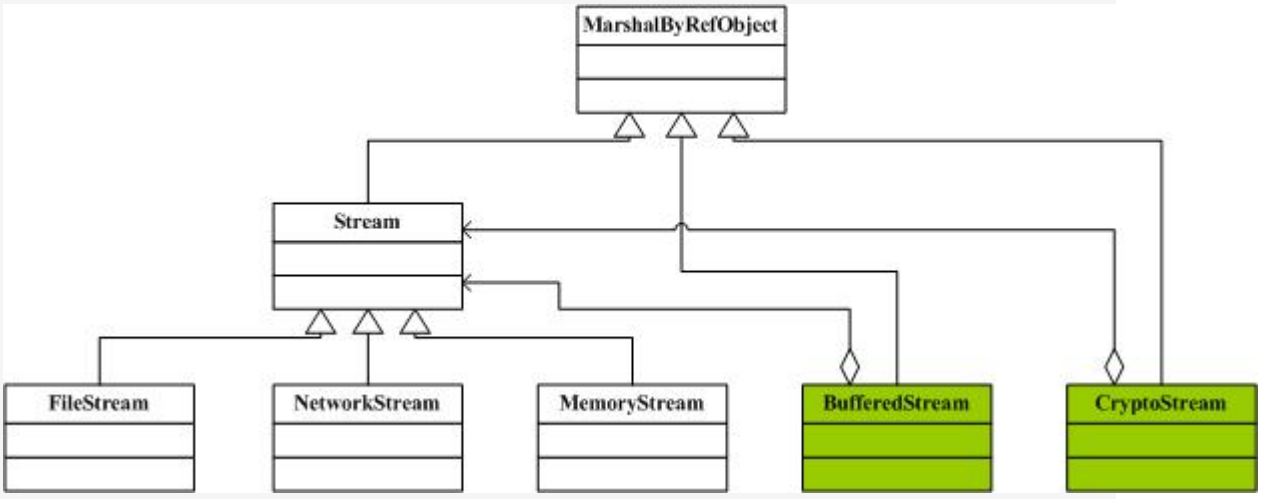
通过采用组合、而非继承的手法，Decorator 模式实现了在运行时动态地扩展对象功能的能力，而且可以
根据需要扩展多个功能。避免了单独使用继承带来的“灵活性差"和"多子类衍生问题"。

Component 类在 Decorator 模式中充当抽象接口的角色，不应该去实现具体的行为。而且 Decorator 类对于 Component 类应该透明---换言之 Component 类无需知道 Decorator 类，Decorator 类是从外部来扩展 Component 类的功能。

Decorator 类在接口上表现为 is-a Component 的继承关系，即 Decorator 类继承了 Component 类所且有的接口。但在实现上又表现 has a Component 的组合关系，即 Decorator 类又使用了另外一个 Component 类。我们可以使用一个或者多个 Decorator 对象来“装饰”一个 Component 对象，且装饰后的对象仍然是一个 Component 对象。

Decorator 模式并非解决”多子类衍生的多继承“问题，Decorator 模式应用的要点在于解决“主体
类在多个方向上的扩展功能”------是为“装饰”的含义。

Decorator 在.NET(Stream)中的应用：



可以看到， BufferedStream 和 CryptoStream 其实就是两个包装类，这里的 Decorator 模式省略了抽象装饰角色（Decorator），示例代码如下：

```
class Program

{

   public static void Main(string[] args)

   {

      MemoryStream ms =

         new MemoryStream(new byte[] { 100,456,864,222,567});



      //扩展了缓冲的功能

      BufferedStream buff = new BufferedStream(ms);



      //扩展了缓冲，加密的功能

      CryptoStream crypto = new CryptoStream(buff);

   }

}
```

通过反编译，可以看到 BufferedStream 类的代码（只列出部分），它是继承于 Stream 类：

```
public sealed class BufferedStream : Stream

{

   // Methods
```

```
6
7     private BufferedStream();
8
9     public BufferedStream(Stream stream);
10
11     public BufferedStream(Stream stream, int bufferSize);
12
13     // Fields
14
15     private int _bufferSize;
16
17     private Stream _s;
18
19 }
```

组合模式(Composite Pattern)

动机(Motivate):

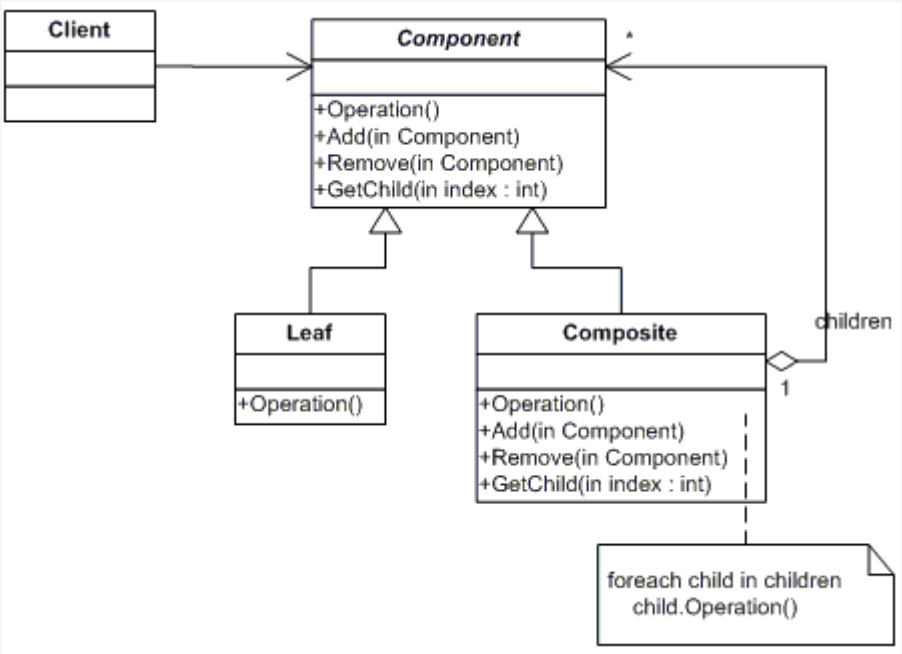
组合模式有时候又叫做部分-整体模式，它使我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以向处理简单元素一样来处理复杂元素, 从而使得客户程序与复杂元素的内部结构解耦。

意图(Intent):

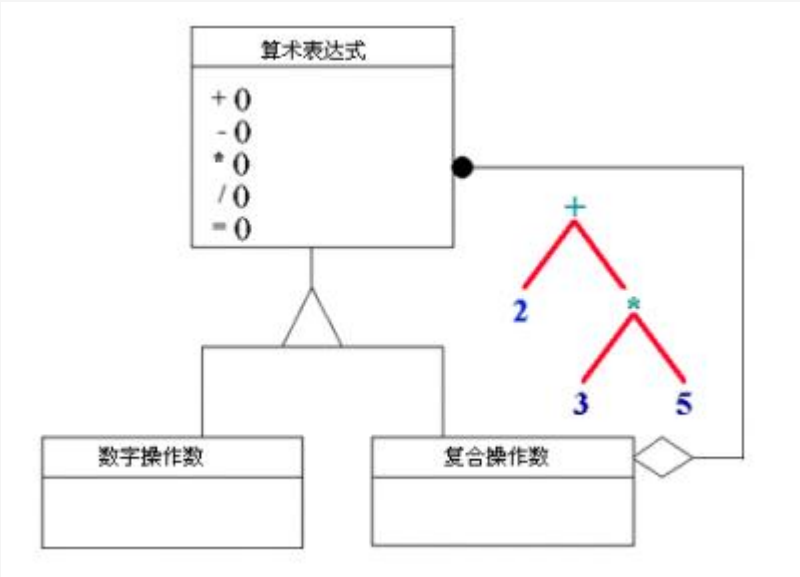
将对象组合成树形结构以表示“部分-整体”的层次结构。Composite 模式使得用户对单个对象和组合对象的使用具有一致性。

----------《设计模式》GOF

结构图(Struct):



生活中的例子：



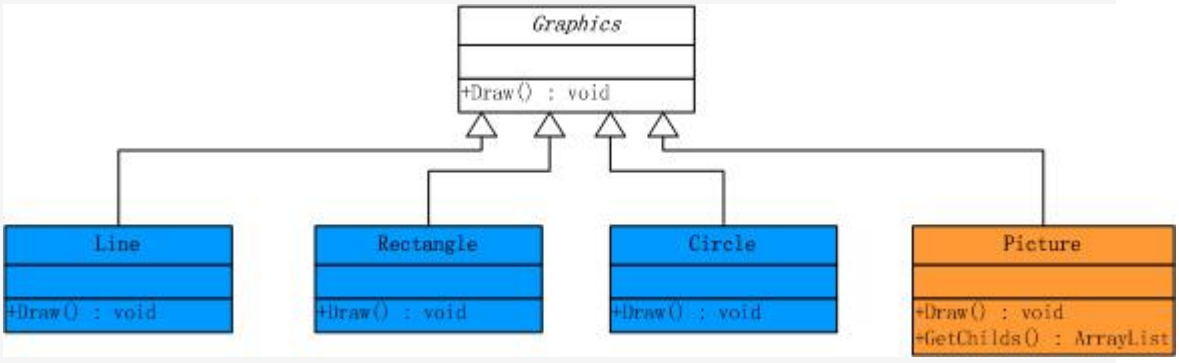
适用性：

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。

代码实现：

这里我们用绘图这个例子来说明 Composite 模式，通过一些基本图像元素（直线、圆等）以及一些复合图像元素（由基本图像元素组合而成）构建复杂的图形树。在设计中我们对每一个对象都配备一个 Draw()方法，在调用时，会显示相关的图形。可以看到，这里复合图像元素它在充当对象的同时，又是那些基本图像元素的一个容器。先看一下基本的类结构图：



图中橙色的区域表示的是复合图像元素。

示意性代码：

```
1 public abstract class Graphics
2 {
3     protected string _name;
```

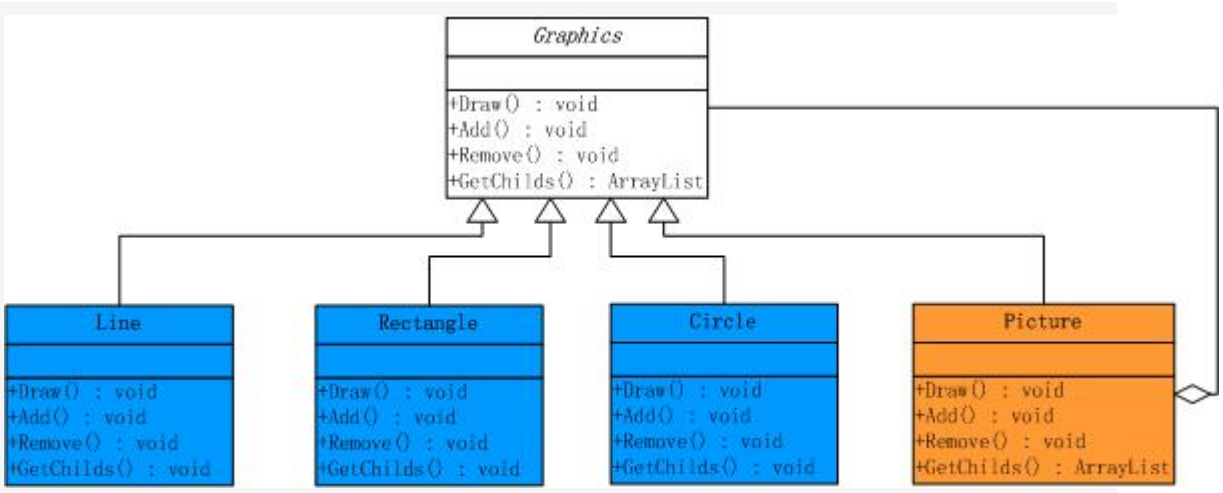
```
4
5    public Graphics(string name)
6    {
7        this._name = name;
8    }
9    public abstract void Draw();
10 }
11
12 public class Picture：Graphics
13 {
14    public Picture(string name)
15        : base(name)
16    { }
17    public override void Draw()
18    {
19        //
20    }
21
22    public ArrayList GetChilds()
23    {
24        //返回所有的子对象
25    }
26 }
```

而其他作为树枝构件，实现代码如下：

```
1 public class Line:Graphics
2 {
3    public Line(string name)
4        : base(name)
5    { }
6
7    public override void Draw()
8    {
9        Console.WriteLine("Draw a" + _name.ToString());
10    }
```

```
11 }
12
13 public class Circle：Graphics
14 {
15     public Circle(string name)
16         : base(name)
17     { }
18
19     public override void Draw()
20     {
21         Console.WriteLine("Draw a" + _name.ToString());
22     }
23 }
24
25 public class Rectangle：Graphics
26 {
27     public Rectangle(string name)
28         : base(name)
29     { }
30
31     public override void Draw()
32     {
33         Console.WriteLine("Draw a" + _name.ToString());
34     }
35 }
```

现在我们要对该图像元素进行处理：在客户端程序中，需要判断返回对象的具体类型到底是基本图像元素，还是复合图像元素。如果是复合图像元素，我们将要用递归去处理，然而这种处理的结果却增加了客户端程序与复杂图像元素内部结构之间的依赖，那么我们如何去解耦这种关系呢？我们希望的是客户程序可以像处理基本图像元素一样来处理复合图像元素，这就要引入 Composite 模式了，需要把对于子对象的管理工作交给复合图像元素，为了进行子对象的管理，它必须提供必要的 Add()，Remove()等方法，类结构图如下：



示意代码：

```
1 public abstract class Graphics
2 {
3     protected string _name;
4
5     public Graphics(string name)
6     {
7         this._name = name;
8     }
9     public abstract void Draw();
10     public abstract void Add();
11     public abstract void Remove();
12 }
13
14 public class Picture : Graphics
15 {
16     protected ArrayList picList = new ArrayList();
17
18     public Picture(string name)
19         : base(name)
20     { }
21     public override void Draw()
22     {
```

```
23        Console.WriteLine("Draw a" + _name.ToString());
24
25        foreach (Graphics g in picList)
26        {
27            g.Draw();
28        }
29    }
30
31    public override void Add(Graphics g)
32    {
33        picList.Add(g);
34    }
35    public override void Remove(Graphics g)
36    {
37        picList.Remove(g);
38    }
39 }
40
41 public class Line : Graphics
42 {
43    public Line(string name)
44        : base(name)
45    { }
46
47    public override void Draw()
48    {
49        Console.WriteLine("Draw a" + _name.ToString());
50    }
51    public override void Add(Graphics g)
52    { }
53    public override void Remove(Graphics g)
54    { }
55 }
56
```

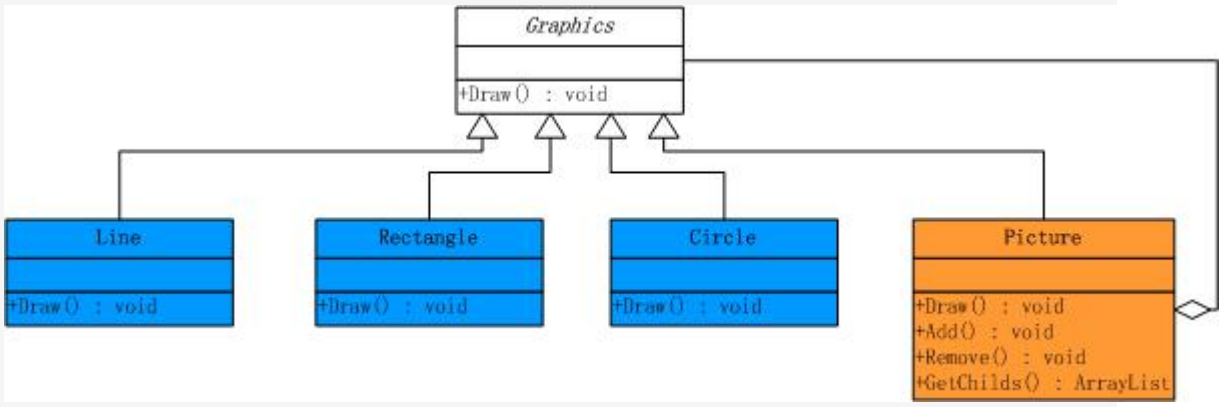
```
57 public class Circle：Graphics
58 {
59    public Circle(string name)
60       : base(name)
61    { }
62
63    public override void Draw()
64    {
65       Console.WriteLine("Draw a" + _name.ToString());
66    }
67    public override void Add(Graphics g)
68    { }
69    public override void Remove(Graphics g)
70    { }
71 }
72
73 public class Rectangle：Graphics
74 {
75    public Rectangle(string name)
76       : base(name)
77    { }
78
79    public override void Draw()
80    {
81       Console.WriteLine("Draw a" + _name.ToString());
82    }
83    public override void Add(Graphics g)
84    { }
85    public override void Remove(Graphics g)
86    { }
87 }
```

这样引入 Composite 模式后，客户端程序不再依赖于复合图像元素的内部实现了。然而，我们程序中仍然存在着问题，因为 Line，Rectangle，Circle 已经没有了子对象，它是一个基本图像元素，因此 Add()，Remove()的方法对于它来说没有任何意义，而且把这种错误

不会在编译的时候报错，把错误放在了运行期，我们希望能够捕获到这类错误，并加以处理，稍微改进一下我们的程序：

```
1 public class Line：Graphics
2 {
3    public Line(string name)
4       : base(name)
5    { }
6
7    public override void Draw()
8    {
9       Console.WriteLine("Draw a" + _name.ToString());
10    }
11    public override void Add(Graphics g)
12    {
13       //抛出一个我们自定义的异常
14    }
15    public override void Remove(Graphics g)
16    {
17       //抛出一个我们自定义的异常
18    }
19 }
```

这样改进以后，我们可以捕获可能出现的错误，做进一步的处理。上面的这种实现方法属于透明式的 Composite 模式，如果我们想要更安全的一种做法，就需要把管理子对象的方法声明在树枝构件 Picture 类里面，这样如果叶子节点 Line，Rectangle，Circle 使用这些方法时，在编译期就会出错，看一下类结构图：



示意代码：

```
public abstract class Graphics
{
    protected string _name;

    public Graphics(string name)
    {
        this._name = name;
    }
    public abstract void Draw();
}

public class Picture : Graphics
{
    protected ArrayList picList = new ArrayList();

    public Picture(string name)
        : base(name)
    { }
    public override void Draw()
    {
        Console.WriteLine("Draw a" + _name.ToString());

        foreach (Graphics g in picList)
        {
            g.Draw();
        }
    }

    public void Add(Graphics g)
    {
        picList.Add(g);
    }
```

```
33     public void Remove(Graphics g)
34     {
35         picList.Remove(g);
36     }
37 }
38
39 public class Line : Graphics
40 {
41     public Line(string name)
42         : base(name)
43     { }
44
45     public override void Draw()
46     {
47         Console.WriteLine("Draw a" + _name.ToString());
48     }
49 }
50
51 public class Circle : Graphics
52 {
53     public Circle(string name)
54         : base(name)
55     { }
56
57     public override void Draw()
58     {
59         Console.WriteLine("Draw a" + _name.ToString());
60     }
61 }
62
63 public class Rectangle : Graphics
64 {
65     public Rectangle(string name)
66         : base(name)
```

```
67    { }
68
69    public override void Draw()
70    {
71        Console.WriteLine("Draw a" + _name.ToString());
72    }
73 }
```

这种方式属于安全式的 Composite 模式，在这种方式下，虽然避免了前面所讨论的错误，但是它也使得叶子节点和树枝构件具有不一样的接口。这种方式和透明式的 Composite 各有优劣，具体使用哪一个，需要根据问题的实际情况而定。通过 Composite 模式，客户程序在调用 Draw()的时候不用再去判断复杂图像元素中的子对象到底是基本图像元素，还是复杂图像元素，看一下简单的客户端调用：

```
1 public class App
2 {
3    public static void Main()
4    {
5        Picture root = new Picture("Root");
6
7        root.Add(new Line("Line"));
8        root.Add(new Circle("Circle"));
9
10       Rectangle r = new Rectangle("Rectangle");
11       root.Add(r);
12
13       root.Draw();
```

Composite 模式实现要点：

1. Composite 模式采用树形结构来实现普遍存在的对象容器，从而将“一对多”的关系转化“一对一”的关系，使得客户代码可以一致地处理对象和对象容器，无需关心处理的是单个的对象，还是组合的对象容器。

2. 将“客户代码与复杂的对象容器结构”解耦是 Composite 模式的核心思想，解耦之后，客户代码将与纯粹的抽象接口——而非对象容器的复内部实现结构——发生依赖关系，从而更能“应对变化”。

3. Composite 模式中，是将“Add 和 Remove 等和对象容器相关的方法”定义在“表示抽象对象的 Component 类”中，还是将其定义在“表示对象容器的 Composite 类”中，是一个关乎“透明性”和“安全性”的两难问题，需要仔细权衡。这里有可能违背面向对象的“单一职责原则”，但是对于这种特殊结构，这又是必须付出的代价。ASP.NET 控件的实现在这方面为我们提供了一个很好的示范。

4. Composite 模式在具体实现中，可以让父对象中的子对象反向追溯；如果父对象有频繁的遍历需求，可使用缓存技巧来改善效率。