设计模式

Outline

- 可修改性
- 设计模式
- 策略模式
- 抽象工厂模式
- 单键模式
- 迭代器模式

可修改性

- (M)实现的可修改性
  - 对已有实现的修改
  - 例如：修改现有促销策略
- (E)实现的可扩展性
  - 对新的实现的扩展
  - 例如：增加一条新的促销策略
- (C)实现的灵活性
  - 对实现的动态配置
  - 例如：动态修改更改某商品对应促销策略

如何实现可修改性？

- 接口与实现的分离

如何将接口与实现的分离 -- Java视角

- 通过接口与实现该接口的类，将接口与实现相分离
- 通过子类继承父类，将父类的接口与子类的实现相分离。

实现接口

- interface 定义了规约
- 实现class 实现了规约

```
public class Client {
    public static void main (String [] args){
        //创建
        Interface_A a = new Class_A1();

        //调用
        a.method_A();
    }
}
```

```
public interface Interface_A {
    // 接口
    public void method_A();
}
```

```
public class Class_A1 implements Interface_A {
    public void method_A(){
        // 实现
        System.out.println("Class_A1 's method_A()!");
    }
}
```

类图与依赖关系

- Client、 Interface_A、 Class_A1之间是什么关系？
- Client和Class_A1是否存在依赖关系？

继承

- “父类”定义了规约
- “子类”实现了规约

```
public class Client {
    public static void main (String [] args){
        // 创建
        Super_A a = new Sub_A1();

        // 调用
        a.method_A();
    }
}
```

```
public class Super_A{
    public void method_A(){
        // 父类的接口和父类的实现
        System.out.println(“Super_A ‘s method_A()!”);
    }
```

```
public class Sub_A1 extends Super_A{
    public void method_A(){
        // 子类的实现
        System.out.println(“Sub_A ‘s method_A()!”);
    }
}
```

类图与依赖关系

- Client、 Super_A、 Sub_A1之间是什么关系？
- Client和Sub_A1是否存在依赖关系？

(M)实现的可修改性

- 对于实现的可修改性，无论是Class_A1还是Sub_A1的method_A方法的实现的修改都和Client中的调用代码没有任何耦合性。

```
public class Class_A2 implements Interface_A {
        public void method_A(){
                System.out.println("Class_A2 's method_A()!");
        }
}
```

```
public class Sub_A2 extends Super_A {
        public void method_A(){
                System.out.println("Sub _A2 's method_A()!");
        }
}
```

扩展

(E)实现的可扩展性

- 对于实现的可扩展性，我们可以通过Class_A2还是Sub_A2的创建来实现。

```
public class Client {
        //创建
        Interface_A a = new Class_A1();
        // Interface_A a = new Class_A2();
        // Super_A a = new Sub_A1();
        // Super_A a = new Sub_A2();

        //调用的接口不变
        //但是当 a 指向不同的类的对象，就会动态的选择的不同实现。
        a.method_A();
    }
}
```

(C)实现的灵活性

继承的优点

- 虽然继承也能很好的完成接口与实现的分离，但是继承还有他独有的特征。
- 子类不但继承了父类的接口还继承了父类的实现，这可以更好的进行代码的重用。

继承的缺点

- 继承的父类与所有子类存在共有接口的耦合性。当父类接口发生改变的时候，子类的接口就一定会更改，这样就会影响到Client代码。
- 而且当子类创建对象的时候，就决定了其实现的选择，没法再动态的修改。

组合

- 而利用接口的组成关系，却能在实现接口和实现的前提下，体现更好的灵活性。前端类和后端类是组合关系。前段类重用了后端类的代码。

```
class Backend{
        public int method_2(){
        }
}
```

```
class Frontend{
        public Backend back = new Backend();
        public int method_2(){
                back.method_2();
        }
}
```

```
class Client{
        public static void main(String[] args){
                Frontend front = new Frontend();
                int i = front.method_2();
        }
}
```

组合的优点

- 前端和后端在接口上不存在耦合性。当后端接口发送改变的时候，并不会直接影响到Client代码。
- 后端类的实现亦可以动态创建、动态配置、动态销毁，非常灵活。

Outline

- 可修改性
- 设计模式
- 策略模式
- 抽象工厂模式
- 单键模式
- 迭代器模式

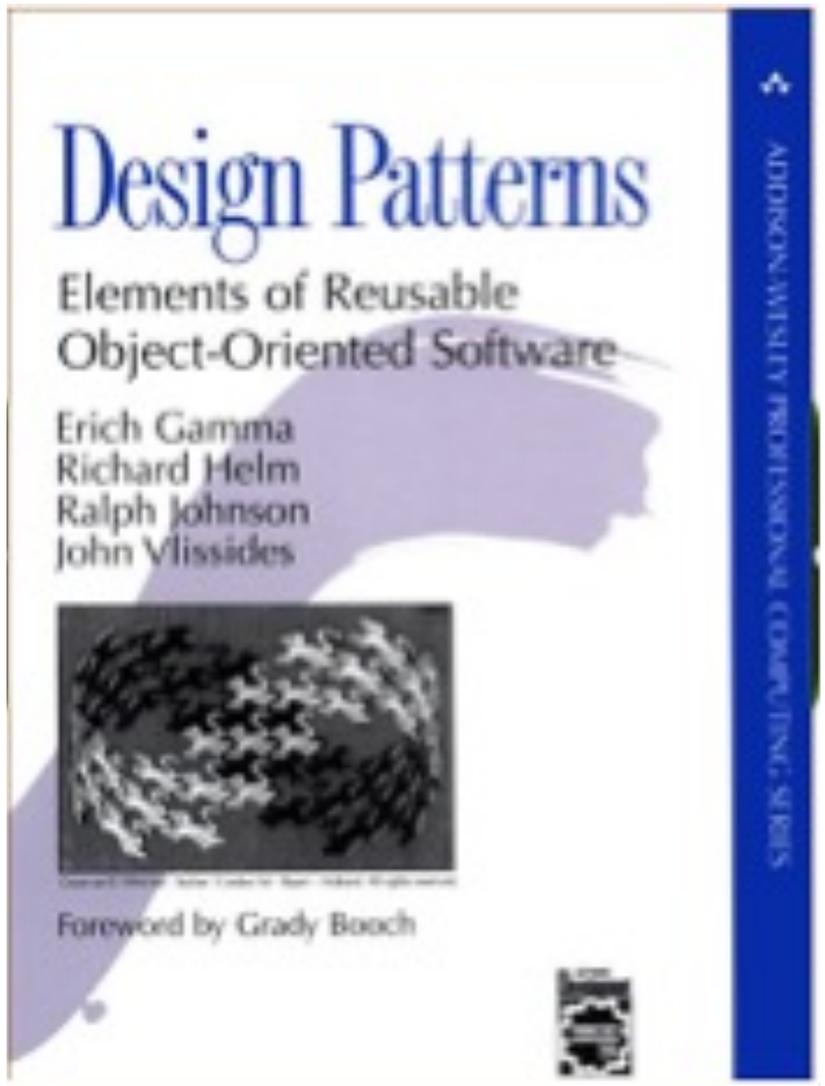
Why ?

- Designing OO software is hard
- Designing reusable OO software – harder
- Experienced OO designers make good design
- New designers tend to fall back on non-OO techniques used before
- Experienced designers know something – what is it?

- Expert designers know not to solve every problem from first principles
- They reuse solutions
- These patterns make OO designs more flexible, elegant, and ultimately reusable

What is a design pattern

- A design pattern
  - abstracts a recurring design structure
  - comprises class and/or object
    - dependencies,
    - structures,
    - interactions, or
    - conventions
  - distills design experience



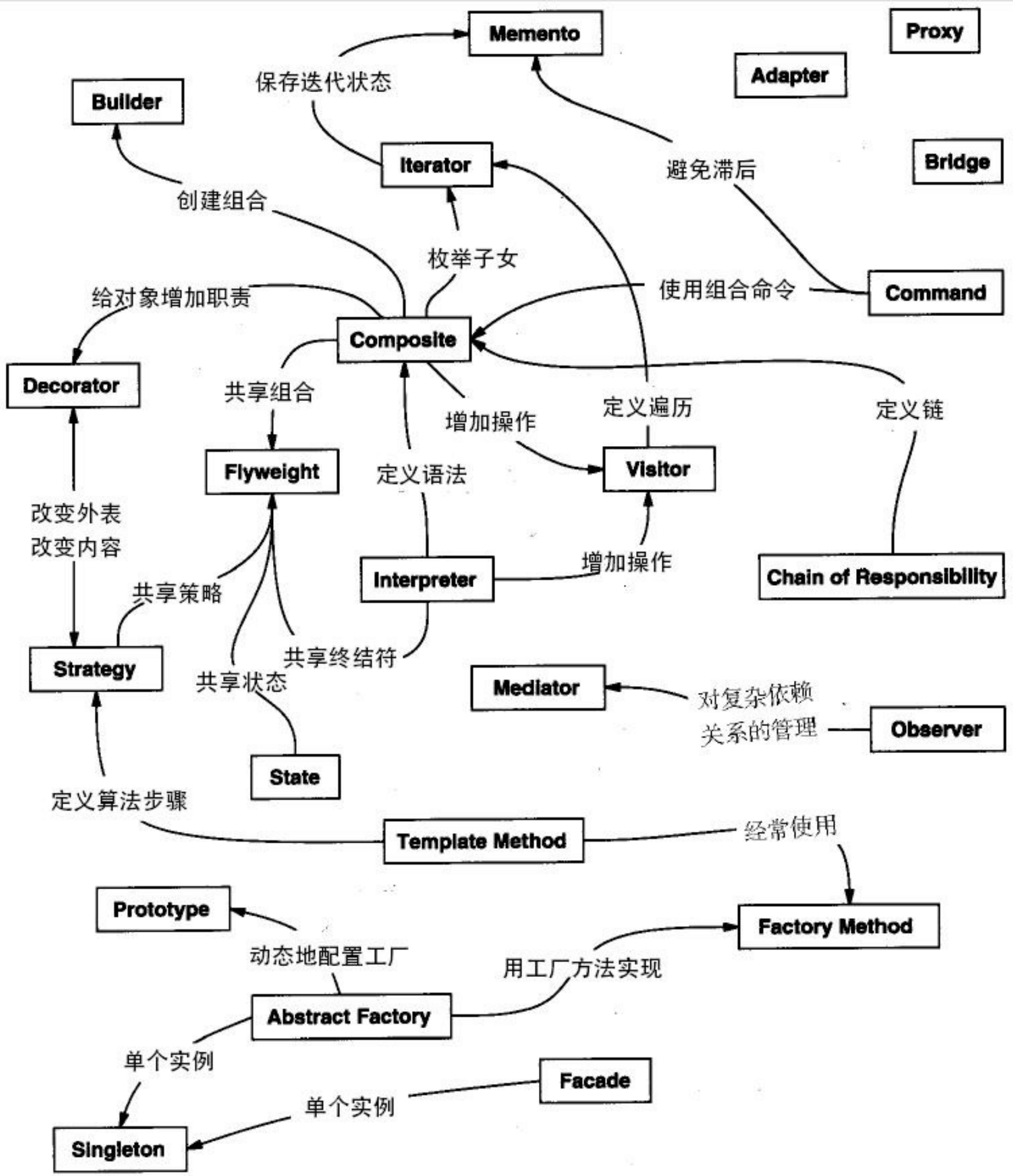

Memento
Proxy
Adapter
保存迭代状态
Builder
Iterator
避免滞后
Bridge
创建组合
枚举子女
给对象增加职责
使用组合命令
Command
Composite
Decorator
共享组合
定义遍历
定义链
增加操作
Flyweight
定义语法
Visitor
改变外表
改变内容
共享策略
Interpreter
增加操作
Chain of Responsibility
共享终结符
Strategy
共享状态
Mediator
对复杂依赖
关系的管理
Observer
State
定义算法步骤
Template Method
经常使用
Prototype
Factory Method
动态地配置工厂
用工厂方法实现
Abstract Factory
单个实例
Facade
单个实例
Singleton

图　设计模式之间的关系

模式

- 典型问题
- 设计分析
- 解决方案
- 案例

解决方案

- 组成与协作：
  - 描述了设计中涉及的各个类的组成成分，他们之间的相互关系及各自的职责和协作方式。
- 应用场景：
  - 描述了应该何时使用模式。它解释了设计模式所要解决的问题，以及解决这个问题时所面临的特点的环境、限制条件、场景等。这也是我们在应用某种模式之前，需要仔细去体察的。
- 使用注意点：
  - 因为模式只是一个模板，他可以应用与多种不同场合，所以解决方案并不描述一个具体的实现，而是提供解决方案的一个抽象模型。

Outline

- 可修改性
- 设计模式
- 策略模式
- 抽象工厂模式
- 单键模式
- 迭代器模式

典型问题

- 在一个大规模的连锁超市中雇员的薪水支付可以分为很多种。其中雇员的薪酬支付方式和支付频率就有好几种：
  - 有些雇员是钟点工，按时薪来支付。薪水=时薪*工作小时数。每周三支付。
  - 有些雇员按月薪支付。薪水=固定月薪。每月21日支付。
  - 有些雇员是提成制。薪水=销售额*提成比率。每隔一周的周三支付。

```
class PaymentStrategy{
  //拥有每个雇员的支付相关的数据
  ArrayList<DOUBLE> hourList = new ArrayList< DOUBLE >();
  ArrayList< DOUBLE > hourRateList = new ArrayList< DOUBLE >();
  ArrayList< DOUBLE > contractValueList = new ArrayList< DOUBLE >();
  ArrayList< DOUBLE > commissionRateList = new ArrayList< DOUBLE >();

  ...
  //计算需要支付的金额
  public double calculatePayment(int employeeID){
switch(e.getPaymentClassification()){
case HOURLY:
      return hourList.get (employeeID)* hourRateList.get (emplyeeID);
      break;
case COMMISSIONED:
      return contractValueList.get (employeeID)* commissionRateList.get (emplyeeID);
```

```
            break;
        case SALARIED: ...
}
  }
  public boolean isPayDay(int employeeID){
switch(e.getPaymentSchedule()){
        case MONTHLY: ...
        case WEEKLY: ...
        case BIWEEKLY: ...
}
  }
}
```

潜在的变化

- 钟点工可能两星期支付一次；
  - (M)实现的可修改性
- 现在是时薪以后可能会变为月薪；
  - (C)实现的灵活性
- 也有可能出现新的薪水支付方式和支付频率。
  - (E)实现的可扩展性

设计分析 -I

- 支付设计几个类？
- 各自有几个职责？

首先，可以把上下文和策略分割为不同的类实现不同的职责。上下文Context类负责通过执行策略实现自己职责；而策略类Strategy只负责复杂策略的实现。

设计分析 - 2

- 如何设计策略类？
- 接口和实现要不要分离？
- 如果要，如何分离？

组合还是继承？

- 其次，上下文类和策略类之间的关系是用组合比继承更加合适。
  - 组合使得上下文类和策略类之间的接口之间的耦合性会很低；
  - 策略类的接口和实现的修改都相对比较容易；
  - 此外，如果是继承关系，则上下文类只能在行为的n种实现里面n选一（对象创建时就选定了策略），而如果是组合关系，上下文类则可以维护一个策略队列，实现n选多，从而达到动态的配置。

- 最后，各种策略则在具体策略类（ConcreteStrategy）中提供，而向上下文类提供统一的策略接口。
  - 由于策略和上下文独立开来，策略的增减、策略实现的修改都不会影响上下文和使用上下文的客户。
- 当出现新的促销策略或现有的促销策略发生变化时，只需要实现新的具体策略类（实现策略的接口），由客户使用。

表 16-1 使用的设计原则和解释

使用的设计原则	解释
减少耦合	减少策略的使用类和策略的实现类直接的耦合
依赖倒置	策略的使用类依赖的是策略的接口，而非策略的实现类。

设计原则

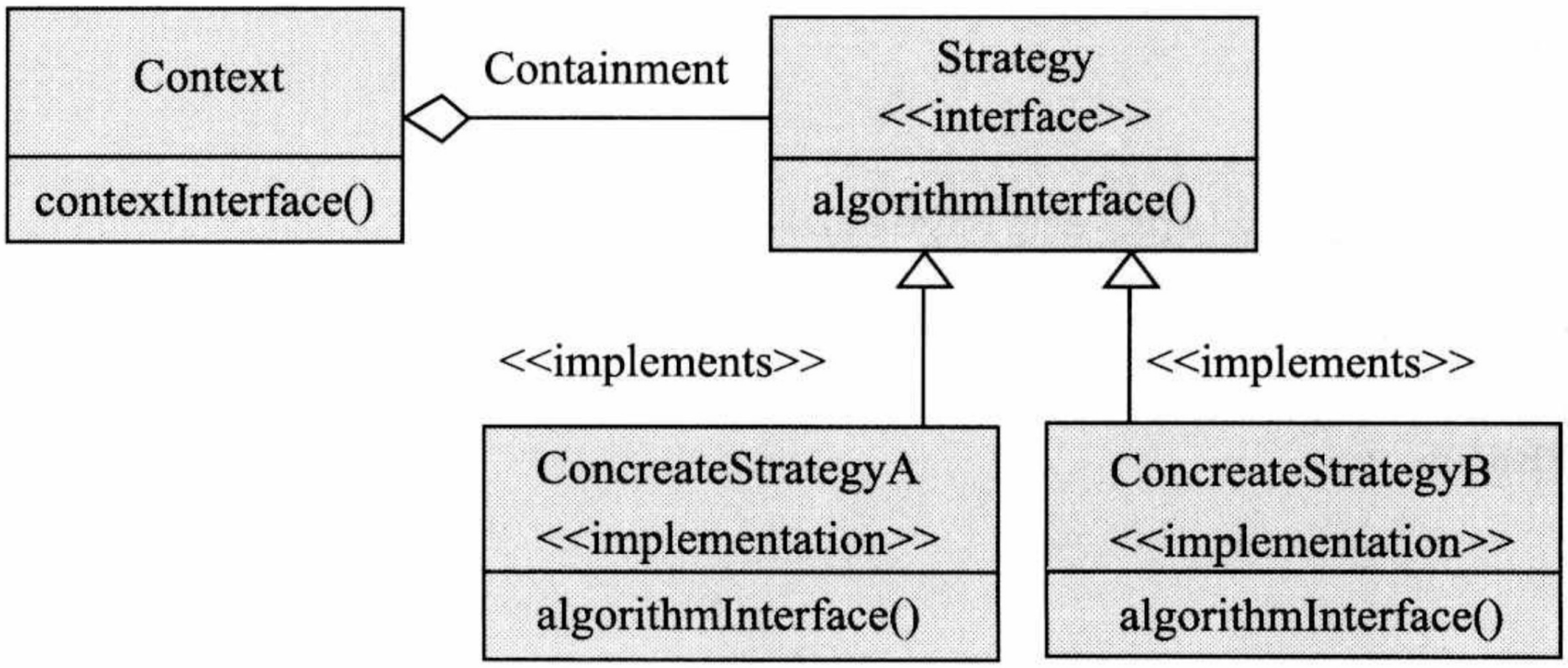


图 16-8　策略模式

策略模式：定义了算法族，分别封装起来，让他们之间可以互相替换，此模式让算法的变化独立于使用算法的客户。

策略模式类图

参与者

- 上下文（Contex）：1)被配置了具体策略ConcreteStrategy； 2)拥有Strategy对象的一个引用；

  3)实现了一些方法以供Strategy访问其数据。

- 策略（Strategy）：声明了所支持策略的接口。Context利用这些被ConcreteStrategy定义的接口。
- 具体策略（ConcreteStrategy）：实现了Strategy声明的接口，给出了具体的实现。

协作

- 上下文Context和Strategy的相互协作完成整个算法。Context可能会通过提供方法让Strategy访问其数据；甚至将自身的引用传给Strategy，供其访问其数据。Strategy会在需要的时候访问Context的成员变量。
- 上下文Context将一些对他的请求转发给策略类来实现，客户（Client）通常创建ConcreteStrategy的对象，然后传递给Context来灵活配置Strategy接口的具体实现；这样Client就有可以拥有一个Strategy接口的策略族，其中包含多种ConcreteStrategy的实现。

应用场景

- 当很多相关类只在它们的行为的实现上不一样。策略模式提供了一个很好的方式来配置某个类，让其具有上述多种实现之一。
- 当我们需要同一个行为的不同实现（变体）的时候。策略模式可以用作实现这些变体。
- 算法需要用到一些数据，而这些数据不应该被客户知道。我们可以通过策略模式隐藏复杂的算法和数据接口。
- 一个类定义了很多行为，这些行为作为一个switch选择语句的分支执行部分。策略模式可以消除这些分支选择。

注意点

- Strategy可以是接口，也可以是类。如果是类，则可以抽象所有具体算法中公共的实现部分。
- 当然，我们也可以直接通过Context的子类来实现不同的Context实现。不过这样算法的具体实现，就和算法的利用的实现项目交织在一起，不利于理解和维护。
- 策略模式消除了类似根据策略类型的Switch语句。
- 可以动态选择不同的策略
- 客户必须提前知晓各自不同的策略。
- Context和Strategy之间的通讯是有代价的。Context提供了对其成员变量的访问方式。可以有时候，对于某些具体的策略的实现ConcreteStrategy可能并不需要全部的访问，这会存在一定的隐患。
- 策略模式会创建出较多的对象。

- Client：　TestDrive 类；
- Context:　Employee 类；
- Strategy：　PaymentClassification 接口； PamentSchedule 接口；
- ConcreteStrategy : HourlyClassification 类、CommissionedClassification 类、SalariedClassification 类、WeeklySchedule 类、MonthlySchedule 类、BiweeklySchedule 类。

案例

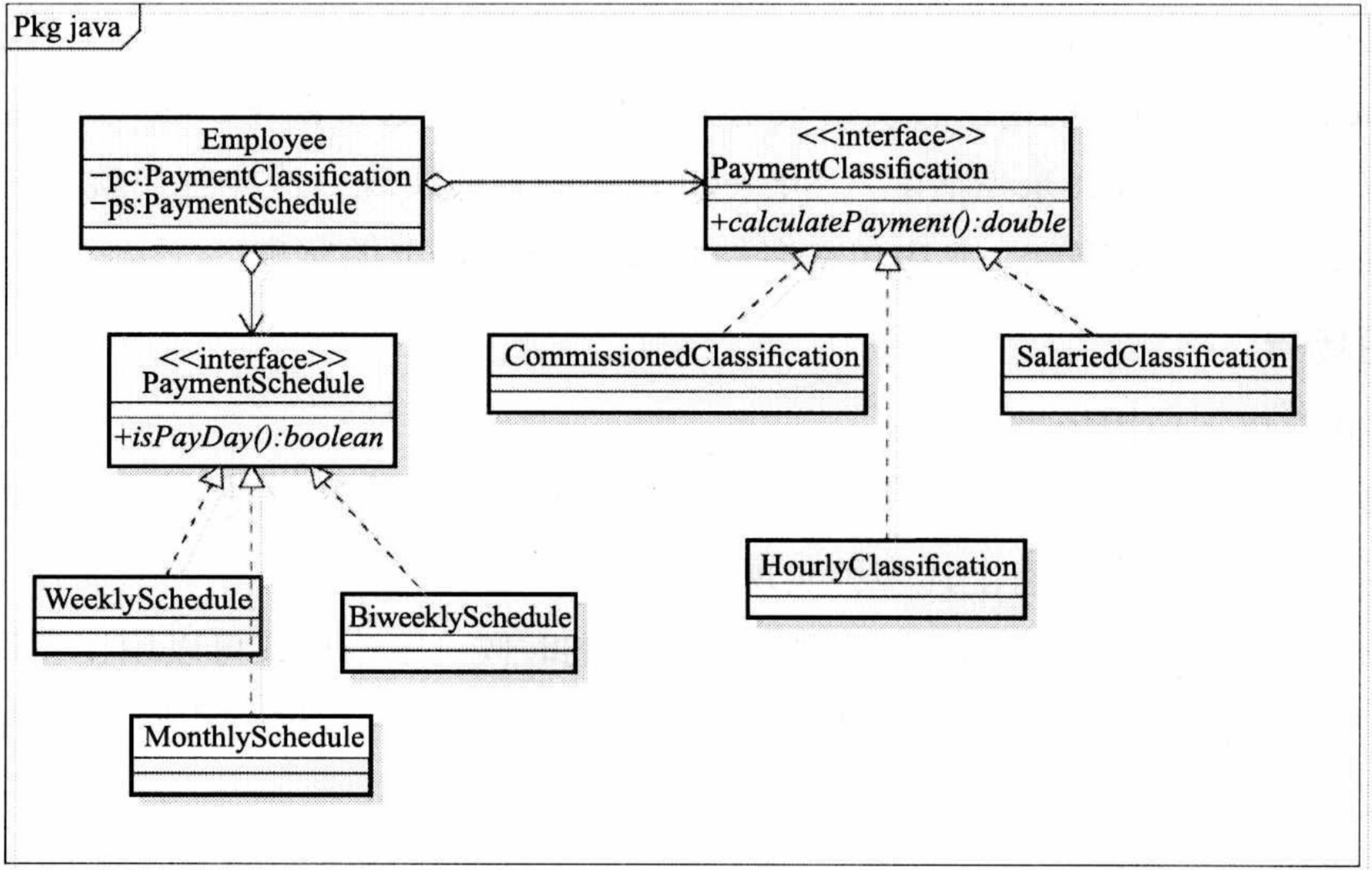


图 16-9　策略模式实例

案例

```
public class TestDrive {

    public static void main(String[] args){
        //创建 Employee 对象
        Employee tom = new Employee("tome",0);
        Employee jack = new Employee("jack",1);
        Employee kevin = new Employee("kevin",2);

        //创建不同的具体策略
        HourlyClassification hc = new HourlyClassification(10,40);
        CommissionedClassification cc=new CommissionedClassification(0.01,1000000);
        SalariedClassification sc = new SalariedClassification(3000);

        WeeklySchedule ws = new WeeklySchedule(2012, Calendar.FEBRUARY,22);
        BiweeklySchedule bs = new BiweeklySchedule(2012,Calendar.FEBRUARY,22);
        MonthlySchedule ms = new MonthlySchedule(21);
```

案例

```
//配置 Employee 对象
//也可以通过带参数的构造方法来配置
tom.setPaymentClassification(hc);
tom.setPaymentSchedule(ws);

jack.setPaymentClassification(cc);
jack.setPaymentSchedule(bs);

kevin.setPaymentClassification(sc);
kevin.setPaymentSchedule(ms);

//Employee 对象的使用
...
while(i.hasNext()){
        Employee e = i.next();
        if(e.isPayDay()){
                e.getPayment();
        }
}
...
```

案例

```
public class Employee {
	//add some codes here
	String name;
	int ID;
	PaymentClassification pc;
	PaymentSchedule ps;

	public Employee(String s, int id){
		name = s;
		ID = id;
	}
	//使用支付方式
	public void getPayment(){
		double payment = 0;
		payment = pc.calculatePayment();
		System.out.println(name + " get "+ payment +" dollars!");
	}
```

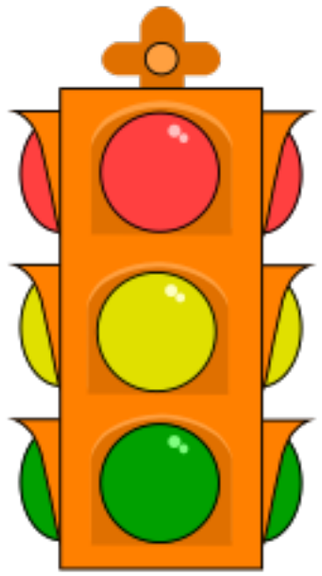
```
//使用支付频率策略
public boolean isPayDay(){
        boolean isPay = false;
        isPay = ps.isPayDay();
        if(isPay) {
                System.out.println("A payDay for " + name + "!");
        }
        return isPay;
}
//设置策略
public void setPaymentClassification(PaymentClassification paymentClassification){
        pc = paymentClassification;
        }
        public void setPaymentSchedule(PaymentSchedule paymentSchedule){
                ps = paymentSchedule;
        }
}
```

```
public interface PaymentClassification {
        //策略接口
        public double calculatePayment();
}
```

案例

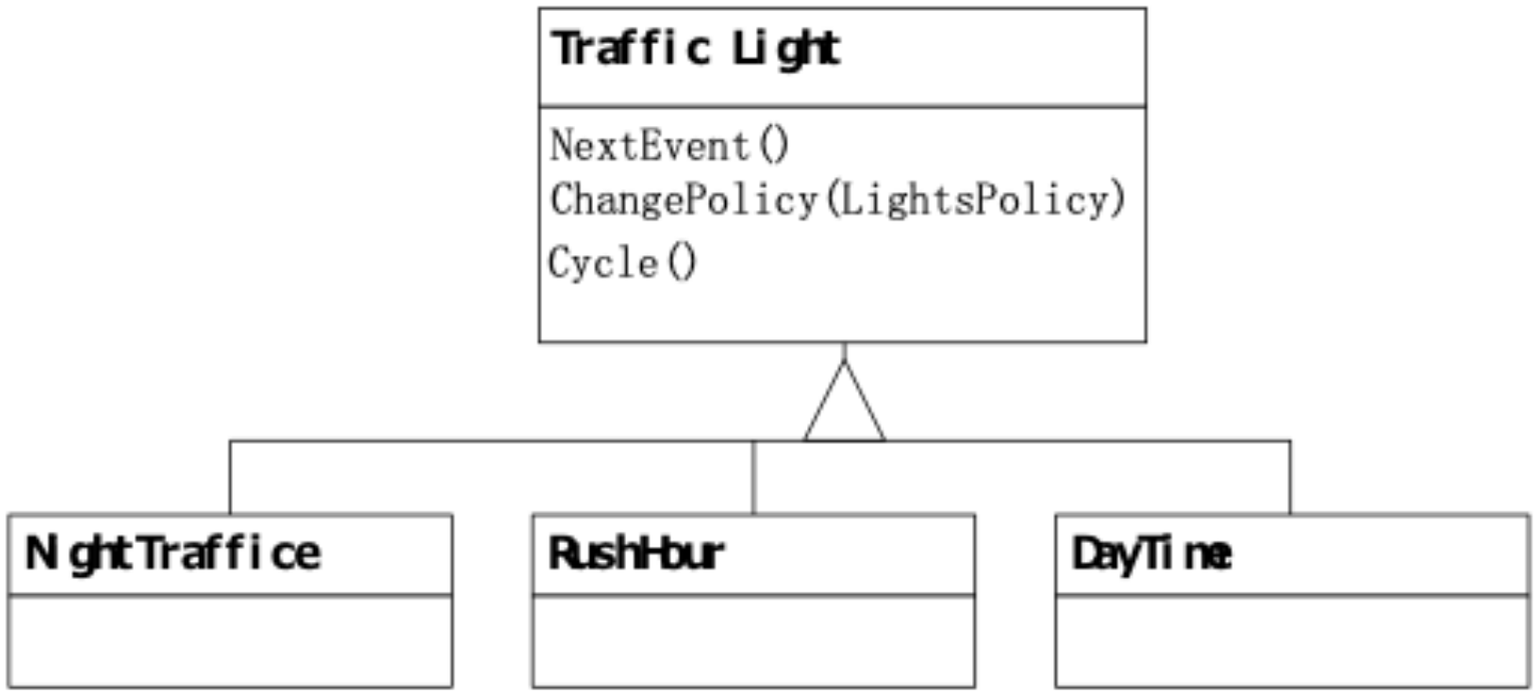
```
public class HourlyClassification implements PaymentClassification{
        int hourlyRate;
        int hours;

        public HourlyClassification(int rate, int h){
                hourlyRate = rate;
                hours = h;
        }
        //策略接口方法的实现
        public double calculatePayment(){
                int sum = 0;
                sum= hourlyRate*hours;
                return sum;
        }
}
```

红绿灯

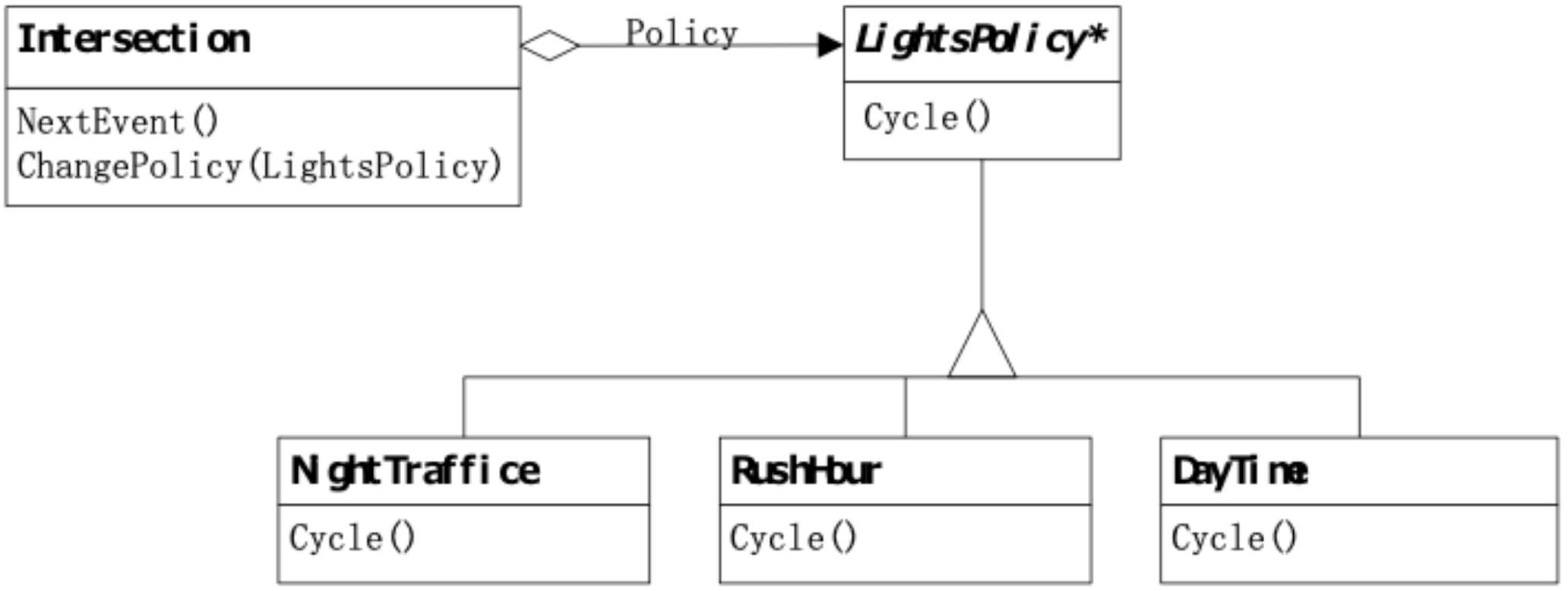


The “dumb” policy: change the green route every 5 seconds
Midnight policy: change to yellow always
Rush hour policy: double the “green time” in the busy route

```
next_green(the_green_ro

rent_policy) {

ht: next_time := time + cc

night: if (not the_green_r

}

n_hour: ...
```



Good but not enough


Intersection
NextEvent()
ChangePolicy(LightsPolicy)
Policy
LightsPolicy*
Cycle()
NightTraffice
Cycle()
RushHour
Cycle()
DayTime
Cycle()

Outline

- 可修改性
- 设计模式
- 策略模式
- 抽象工厂模式
- 单键模式
- 迭代器模式

先说说工厂模式

- 在软件系统中，对象的创建往往是一个比较复杂而且比较特殊的事情。往往我们会需要根据不同类型的对象。如果是普通的方法，我们可以通过多态的形式来体现不同的行为实现。而构造方法却无法多态。

```
class Client{
    public void doSomething1(int type){
        if(type == 1) {
            Class a = new ClassA1();
        }else {
            Class a = new ClassA2();
        }
        a.method1();
    }
    public void doSomething2(int type){
        if(type == 1) {
            Class a = new ClassA1();
        }else {
            Class a = new ClassA2();
        }
        a.method2();
    }

}
```

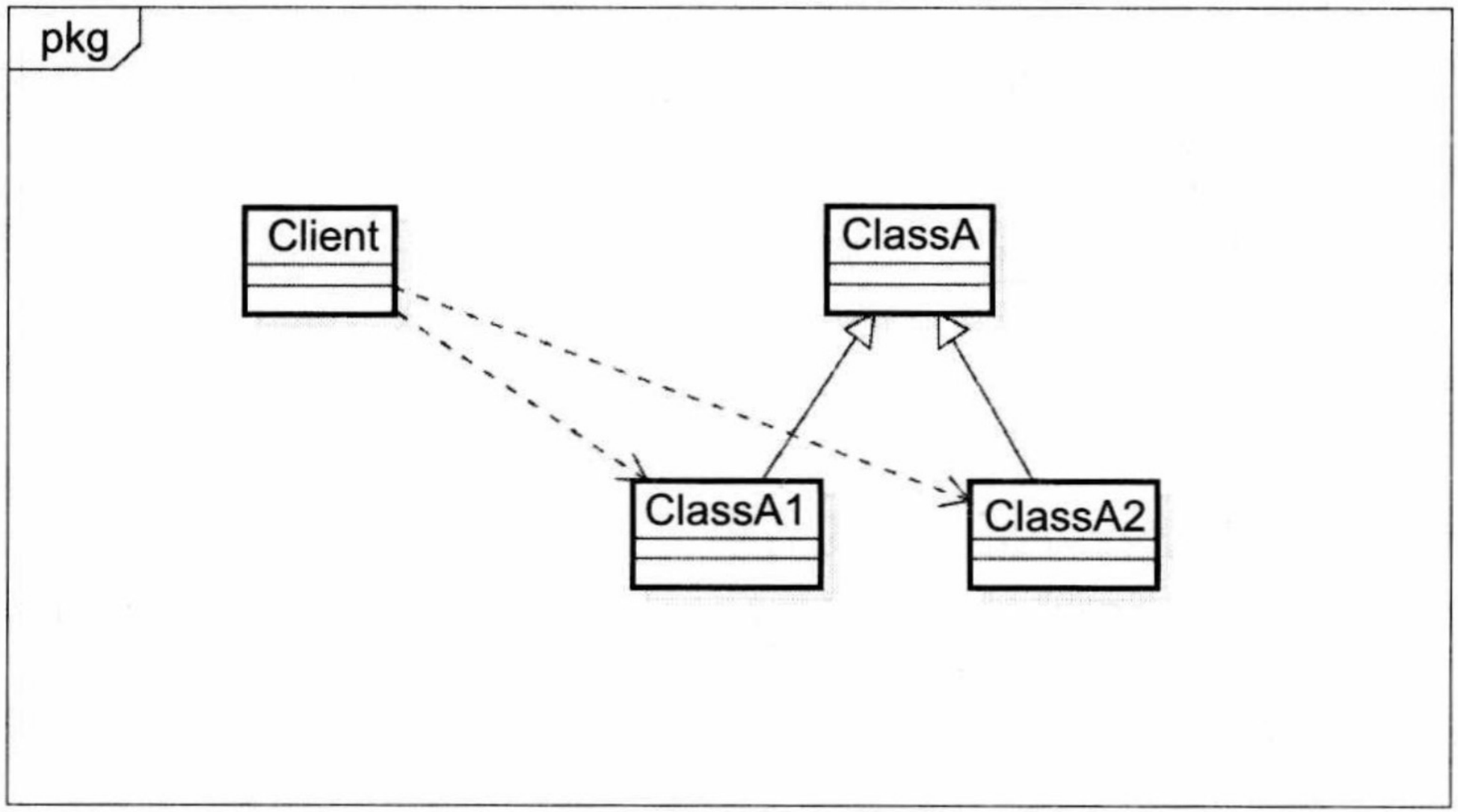


图 16-12　　Client 依赖具体类

问题来了

- 如图所示，Client严重依赖着具体类ClassA1和ClassA2。
- Client代码中到处分布着创建A对象的复杂判断。
- 当我们A的子类发生改变，或者创建对象的复杂逻辑发生改变，都会对Client代码造成很复杂的修改。

用“工厂”来解决

- 我们需要依赖一个专门类—工厂的创建方法。工厂模式就是为对象的创建提供一个接口，将具体创建的实现封装在接口之下，这样具体创建的实现的改变就不会对客户代码Client类产生影响。从而降低了Client类和ClassA1等多个具体类的耦合。

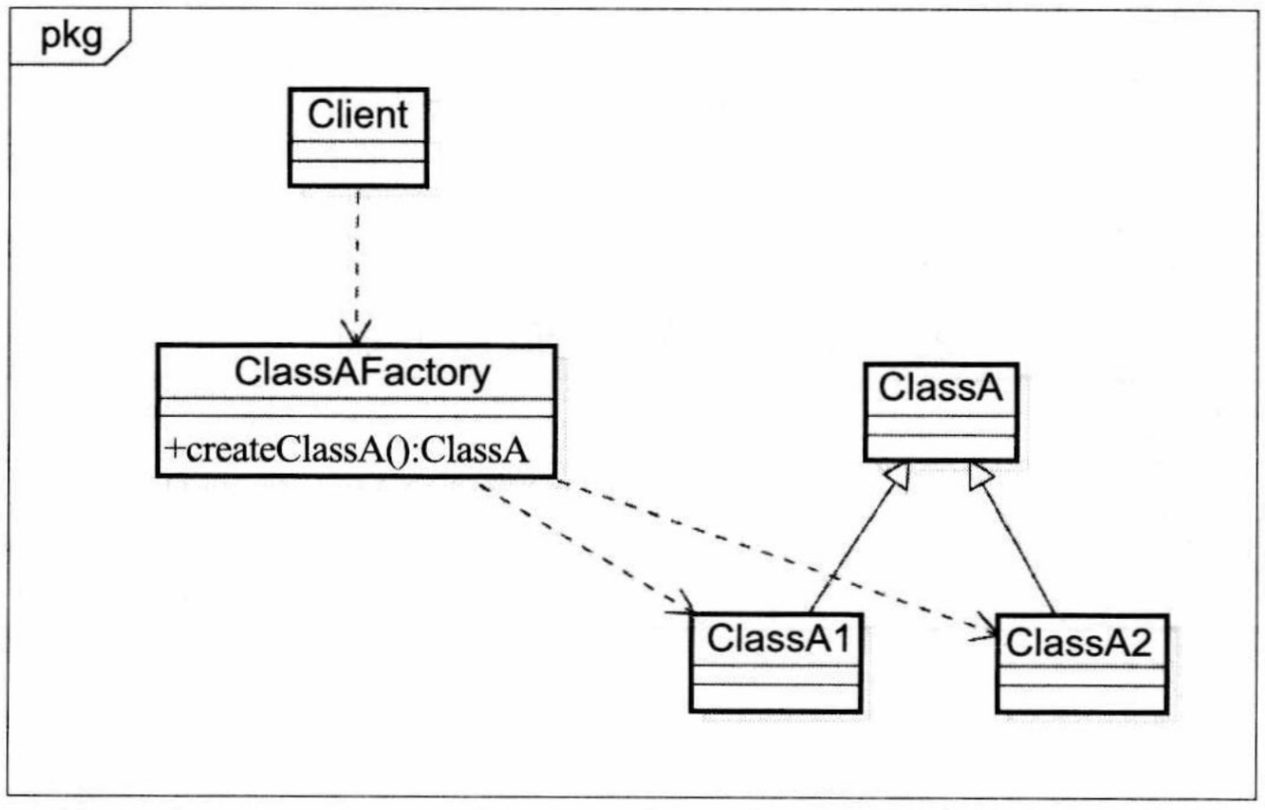


图 16-13　　工厂模式

```
class Client1 extends Client{
    public ClassA createClassA(){
        if(type == 1) {
            Class a = new ClassA1();
        }else {
            Class a = new ClassA2();
        }
      return a;
    }
}
```

图 16-14　工厂模式的代码

还有问题

- 而在软件系统中，经常面临着“多种对象”的创建工作，由于需求的变化，多种对象的具体实现有时候需要灵活组合。
- 比如汽车由引擎、轮胎、车身、车门等各部件组成。而每一部件都有很多种。一个汽车装配车间会依赖不同种的各个部件装配出不同型号的车。如果这时候我们为每一型号的车根据工厂模式创建一个工厂，由于部件的组合关系，我们就会遇到“组合爆炸“问题，对这个装配车间需要创建“无数”个工厂。这就对工厂模式提出更高要求。

设计分析

- 分析具体的需求，我们可以发现，对客户Client来说需要同时实现工厂的灵活性和产品的灵活性。所以，我们提供了两套接口：一是表现出稳定的工厂行为（创建不同的对象）的工厂接口，二是表现出稳定产品行为的产品接口。从而，实现了工厂多态和产品多态。
- 工厂接口既使得原本分布于代码各处的多种对象的实例化，现在变为集中到具体的工厂内部，又隔离了“对象实例化的组合”的变化。
- 客户Client通过抽象工厂接口的方法得到ProductA和ProductB的实例，再利用产品接口来灵活使用具体的产品。

表 16-2　使用的设计原则和解释

使用的设计原则	解　　释
职责抽象	抽象对于对象创建的职责
接口的重用	提供对于对象创建的接口

使用的原则

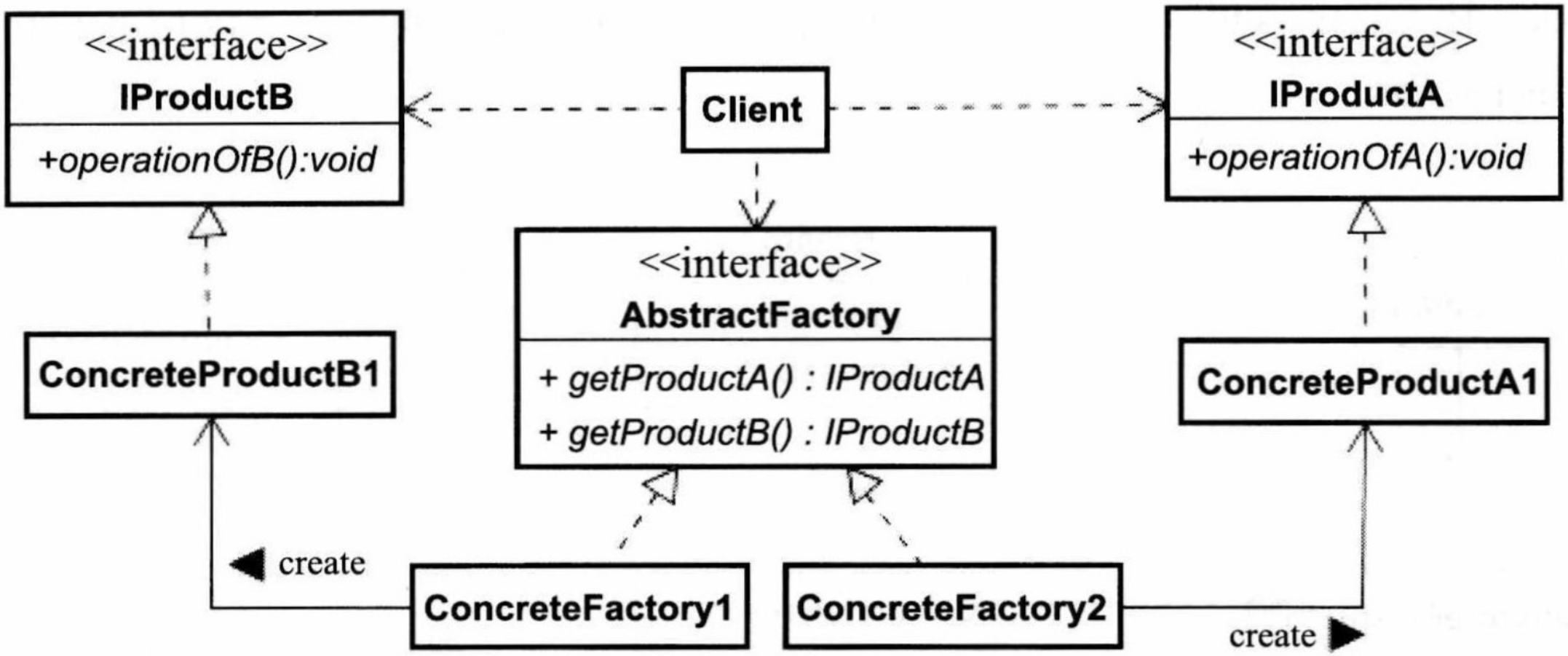
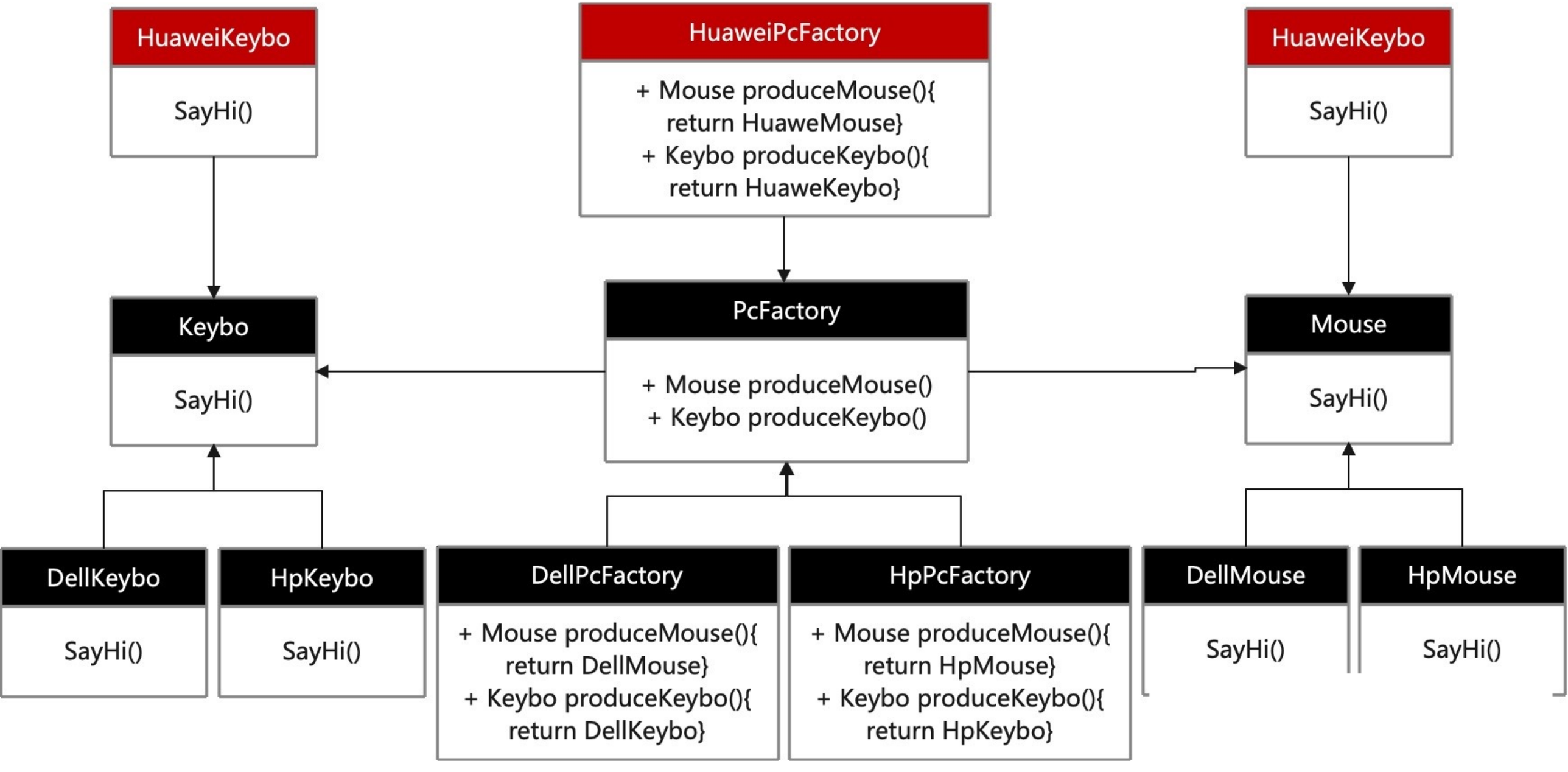


图 16-15　抽象工厂模式

抽象工厂模式：定义了一个创建对象的接口，由子类决定要使实例化哪一个类。工厂方法让类的实例化延迟到子类。

抽象工厂模式的类图


HuaweiKeybo
SayHi()
HuaweiPcFactory
+ Mouse produceMouse(){
return HuaweMouse}
+ Keybo produceKeybo(){
return HuaweKeybo}
HuaweiKeybo
SayHi()
Keybo
SayHi()
PcFactory
+ Mouse produceMouse()
+ Keybo produceKeybo()
Mouse
SayHi()
DellKeybo
SayHi()
HpKeybo
SayHi()
DellPcFactory
+ Mouse produceMouse(){
return DellMouse}
+ Keybo produceKeybo(){
return DellKeybo}
HpPcFactory
+ Mouse produceMouse(){
return HpMouse}
+ Keybo produceKeybo(){
return HpKeybo}
DellMouse
SayHi()
HpMouse
SayHi()

参与者

- 抽象工厂（AbstractFactory）声明了创建抽象产品的各个接口。
- 具体工厂（ConcreteFactory）实现了对具体产品的创建。
- 抽象产品（AbstractProduct）声明了一种产品的接口。
- 具体产品（ConcreteProduct）定义了具体工厂中创建出来的具体产品，实现了抽象产品的接口。
- 客户（Client）使用抽象工厂和抽象产品的类。使用抽象工厂的方法来创建产品。

协作

- 通常情况下，只有一个具体的工厂的实例被创建。这个具体工厂对于创建产品这个事情本身有具体的实现。对于创建不同的产品对象，客户应该用不同的具体工厂。
- 抽象工厂转移了产品的创建到其子类具体工厂类中间去。

应用场景

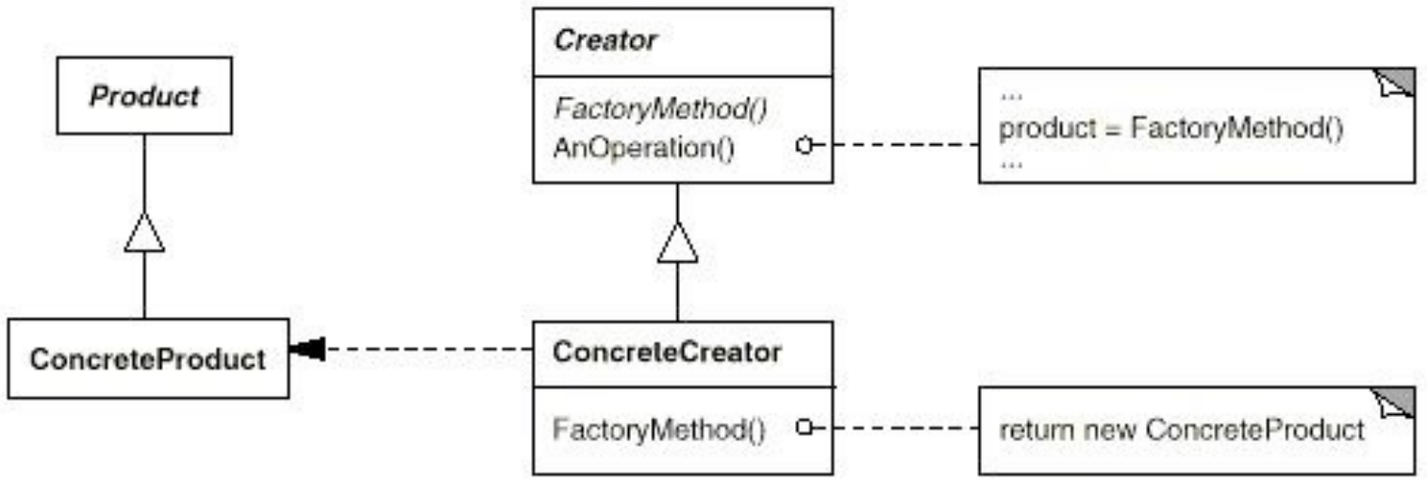
- 抽象工厂模式可以帮助系统独立于如何对产品的创建、构成、表现。
- 抽象工厂模式可以让系统灵活配置拥有某多个产品族中的某一个。
- 一个产品族的产品应该被一起使用，抽象工厂模式可以强调这个限制。
- 如果你想提供一个产品的库，抽象工厂模式可以帮助暴露该库的接口，而不是实现。

应用注意点

- 隔离了客户和具体实现。客户可见的都是抽象的接口。
- 使得对产品的配置变得更加灵活。
- 可以使得产品之间有一定一致性。同一类产品可以很容易一起使用。
- 但是限制是对于新的产品的类型的支持是比较困难。抽象工厂的接口一旦定义好，就不容易变更了。
- 而这个场景的“代价”，或者是“限制”，是一个工厂中具体产品的种类是稳定的。

工厂方法模式

- 当然，工厂接口可以通过抽象工厂模式的专门的接口来实现，另外也可以通过父类的工厂方法，来让子类继承相应的工厂接口，这就是工厂方法模式（Factory Method Pattern）

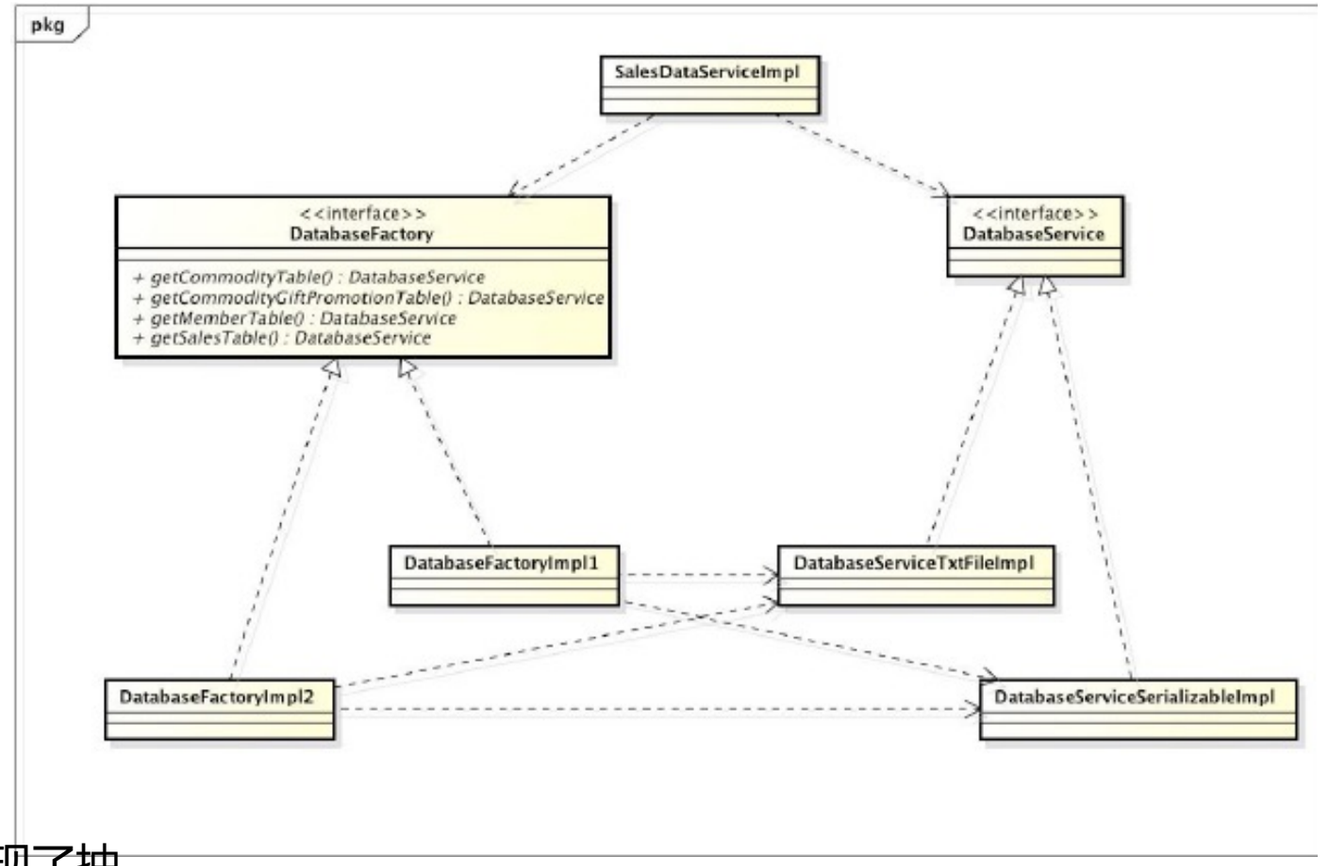


案例

- 对于各个数据库表格数据，我们可以有不同的实现：TXT文件存储，对象序列化存储，数据库存储，甚至是混合式存储。

案例

DatabaseFactory是抽象工厂。
DatabaseFactoryTxtFileImpl和
DatabaseFactorySeriablizableImpl是具体的工厂，即实现了抽象工厂的接口。
DatabaseFactoryTxtFileImpl实现中利用了
DatabaseServiceTxtFileImpl来创建不同的数据库表格，而提供DatabaseService的服务。
对于客户SalesDataServiceImpl来说，利用DatabaseFactory提供的接口创建不同的各式各样的数据库，对每个数据库享用DatabaseService提供的服务，从而达到很好的灵活性。



```
public interface DatabaseFactory {
        //数据库的抽象工厂
        //每个数据库中都有相同的数据表格
        //每个数据表格有不同的实现
        public DatabaseService getCommodityTable();
        public DatabaseService getCommodityGiftPromotionTable ();
        public DatabaseService getCommodityPricePromotionTable ();
        public DatabaseService getMemberTable ();
        public DatabaseService getSalesTable ();
}
```

```
public class DatabaseFactoryImpl1 implements DatabaseFactory {
    //单键
    //数据表格可以存在 txt 文件，也可以序列化保存。


    DatabaseService commodityDatabase = new
    DatabaseServiceTxtFileImpl("commodity.txt",DATATYPE.COMMODITY);
    …
    DatabaseService salesDatabase = new
    DatabaseServiceTxtFileImpl("sales.ser",DATATYPE.SALES);


     public DatabaseService getCommodityTable (){
          return commodityDatabase;
     }
     …
     public DatabaseService getSalesTable (){
          return salesDatabase;
     }

}
```

```
public class CommodityDataServiceImpl implements CommodityDataService {

    //用组成代替继承
    private static DatabaseService commodityDataBase;

    private static CommodityDataServiceImpl commodityDataServericeImpl = null;

    ArrayList<POJO> commodityList = new ArrayList<POJO>();
    ArrayList<Integer> commodityIndexList = new ArrayList<Integer>();

    private CommodityDataServiceImpl() {
        ArrayList<String> comList;
        DatabaseFactory factory = DatabaseFactoryTxtFileImpl.getInstance();
          //使用工厂获得产品的实例
        commodityDataBase =   factory.getCommodityDatabase();
            ...


        ...
}
```

Outline

- 可修改性
- 设计模式
- 策略模式
- 抽象工厂模式
- 单件模式
- 迭代器模式

典型问题

- 在有些场景中，对于某个类，在内存中只希望有唯一一个对象存在。每次想得到这个类的一个对象的引用的时候，都指向唯一的那个对象。无论我创建多少次这个类的对象，其实总共还是只创建了一个对象。

设计分析

- 为了实现只创建一个对象
  - 首先要让类的构造方法变为私有的；
  - 然后只能通过getInstance方法获得Singleton类型的对象的引用。
  - 这其中类的成员变量中拥有一个静态的Singleton类型的引用变量uniqueInstance。
  - getInstance方法返回引用变量uniqueInstance，如果uniqueInstance等于null，则说明首次创建，通过关键字new创建Singleton对象，并且将该对象的引用变量赋值给uniqueInstance；否则不等与null，则说明不是首次创建，每次只需要返回已创建的对象的引用uniqueInstance即可。

表 16-3　使用的设计原则和解释

使用的设计原则	解　　释
职责抽象	隐藏单件创建的实现

使用的原则

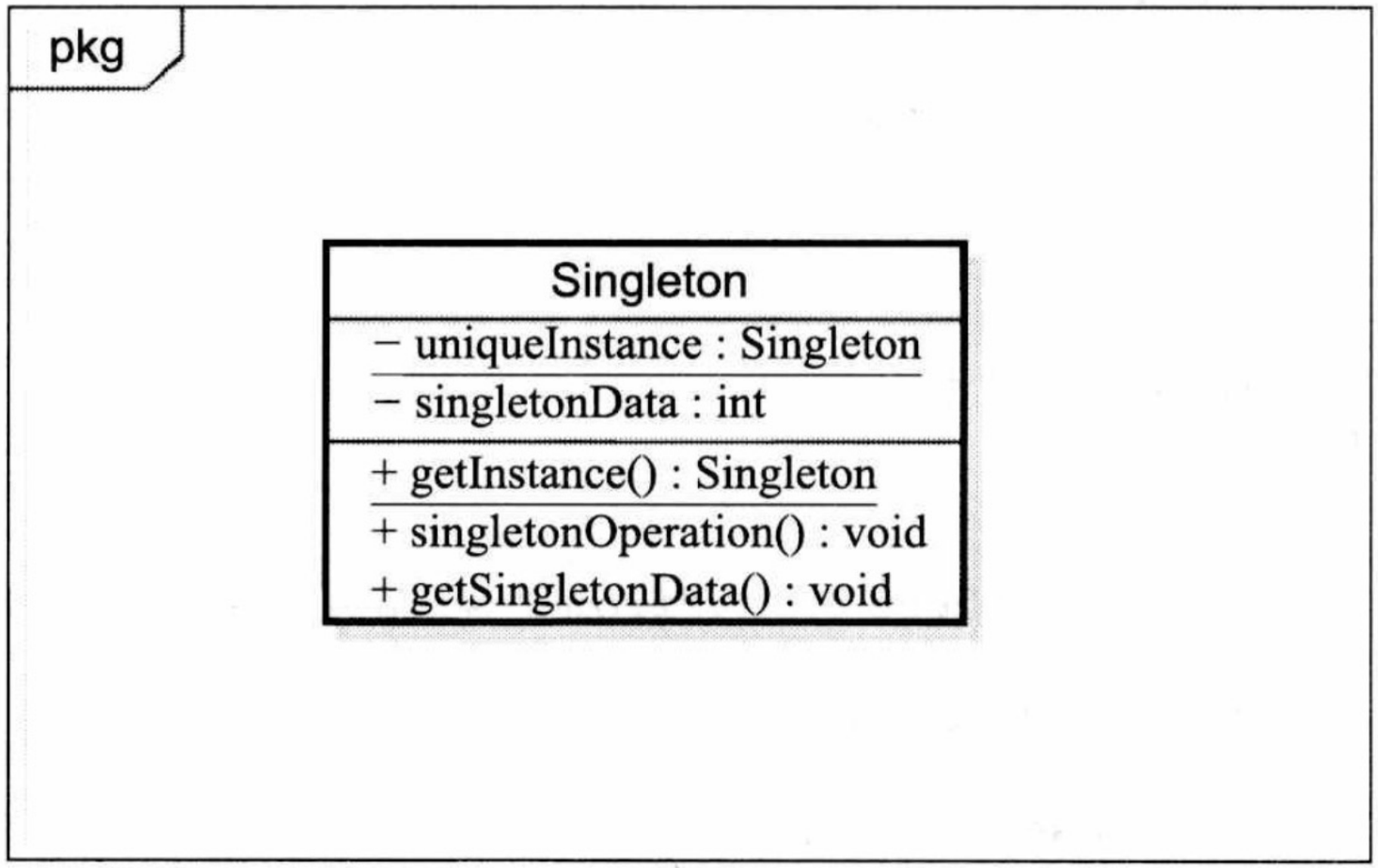


图 16-19　单件模式

单件模式确保一个类只有一个实例，并提供一个全局访问点。

单件模式类图

```
public class DatabaseFactoryTxtFileImpl implements DatabaseFactory {
	//单键
	private static DatabaseFactoryTxtFileImpl databaseFactoryTxtFile = null;

	private    DatabaseFactoryTxtFileImpl(){

	}

	public static DatabaseFactory getInstance(){
		if(databaseFactoryTxtFile==null)
			databaseFactoryTxtFile=new DatabaseFactoryTxtFileImpl();
		return databaseFactoryTxtFile;
	}

}
```

Outline

- 可修改性
- 设计模式
- 策略模式
- 抽象工厂模式
- 单键模式
- 迭代器模式

典型问题 - I

- 在这样一个场景中，对于某个方法f()，可能需要调用g()。g()有参数是一个链表集合的引用，并且完成对一个链表集合的操作。

```
f()
{
        LinkedList list = new LinkedList();
        //
        g(list);
}
g(LinkedList list)
{
        list.add(..);
        g2(list);
}
```

典型问题 - 2

- 如果我们的需求发生改变，需要对这个集合进行快速的查询，这个时候用链表就不太合适了，用散列集合就更加合适。所以，g()的参数如果是某个具体的集合类型，灵活性就不足。所以，我们可以改为一个抽象的类型，比如Collection。

```
f()
{
        Collection list = new LinkedList();
        //
        g(list);
}
g(Collection list)
{
        list.add(..);
        g2(list);
}
```

典型问题 - 3

- 如下所示，就可以很方便的替换为散列表。

```
f()
{
        Collection list = new HashSet();
        //
        g(list);
}
g(Collection list)
{
        for(Iterator i = c.iterator();i.hasNext();)
                do_something_with(i.next());
}
```

典型问题 - 4

- 对于g()的来说，往往可能只是希望挨个访问某个聚合结构。而且我们往往并不希望让g()知道到底是什么样的聚合结构，是LinkedList还是HashSet，是Collection还是Map。这个时候，迭代器模式就可以帮我们。
- 此外，对于集合类型作为参数，我们可以对集合类的修改会直接修改原集合，从而使得我们通常意义上的所向往的“值传递”失效，所以，大大增强了耦合性。而这个问题，迭代器也可以帮我们解决。

设计分析

- 为了满足前面说的需求，其实我们只需要对遍历操作进行抽象。而对于遍历这件事情来说，主要有两个行为：1）是否有下一个元素；2）得到下一个元素。所以，我们设计迭代器接口hasNext()和next()，分别对应与前面两个行为。有了这两个接口，就可以完成遍历操作。这样，g()的参数转换为一个迭代器的引用之后，就会具有更大的灵活性。
- 迭代器提供的方法只提供了对集合的访问的方法，却屏蔽了对集合修改的方法，这样就对我们把集合作为参数可以做到对集合的“值传递”的效果。

表 16-4 使用的设计原则和解释

使用的设计原则	解释
减少耦合	减少遍历的使用类和遍历的实现类直接的耦合
依赖倒置	遍历的使用类依赖的是策略的接口，而非遍历的实现类。

Programming to interfaces

使用的原则

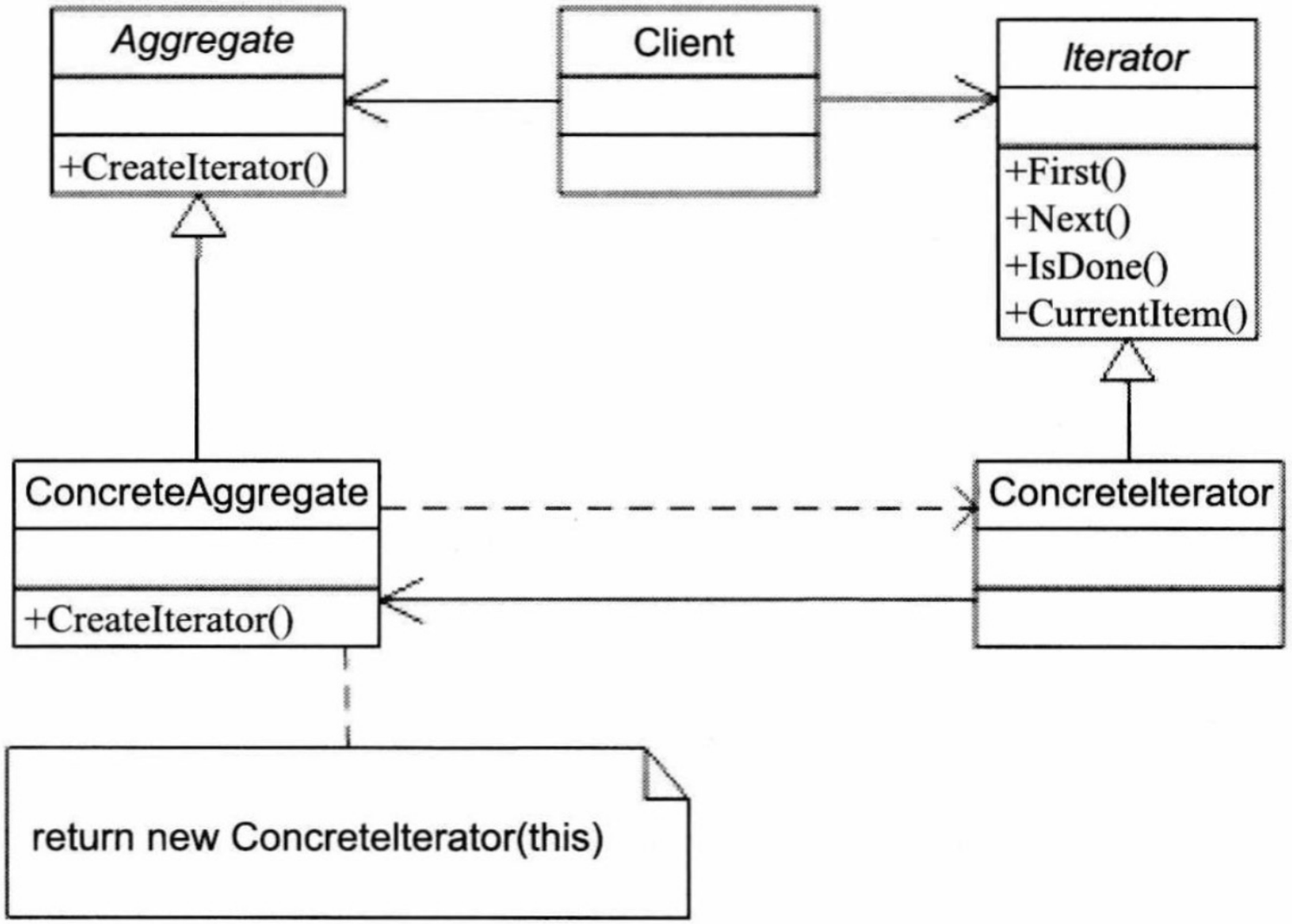


图 16-24　迭代器模式

迭代器模式提供一种顺序访问一个聚合对象的各个元素，而不暴露其内部表示。

迭代器模式类图

参与者

- 迭代器（Iterator）： 迭代器定义访问和遍历元素的接口
- 具体迭代器（ConcreteIterator）：具体迭代器实现迭代器接口。对该聚合遍历时跟踪当前位置。
- 聚合（Aggregate）：聚合定义创建相应迭代器对象的接口。
- 具体聚合（ConcreteAggregate）：具体聚合实现创建相应迭代器的接口，该操作返回ConcreteIterator的一个适当的实例。

协作

- 具体迭代器跟踪聚合中的当前对象，并能够计算出待遍历的后继对象。

应用场景

- 访问一个聚合对象的内容而无需暴露它的内部实现。
- 支持对聚合对象的多种遍历。
- 为遍历不同的聚合结构提供一个统一的接口。

应用的注意点

- 它支持以不同的方式遍历一个聚合。
- 迭代器简化了聚合的接口。
- 在同一个聚合上可以有多个遍历。

案例

- 再看看具体的集合的实现HashSet与g()之间的耦合

```
f()
{
        Collection list = new HashSet();
        //
        g(c.iterator());
}
g(Iterator i)
{
     while(i.hasNext())
           do_something_with(i.next());
}
```

习题 - 配对

- 设计原则
  - 接口重用
  - 职责抽象
  - 依赖倒置
- 设计模式
  - 策略模式
  - 抽象工厂模式
  - 单键模式
  - 迭代器模式