目錄

设计模式 **For** 小白

这是一本给小白看的设计模式书。这并不是一本对《**Design Pattern for Dummies**》的中文翻译，而是我在阅读过程中的中文笔记。说是“笔记”其实也不恰当，应该是我对其中一些重点做的中文版记录吧。就我个人而言，在阅读中文书时，我喜欢在书上将重点内容用笔做下划线记录，这样以后我翻阅时，就会先看重点内容，算是“把书读薄”，可是在看英文技术书时，我发现这个方法就不适用了。。。面对满纸英文，即使我先去看那些有下划线的重点内容，可是阅读速度还是会比看中文书慢很多，而且记忆也不深刻，后来我总结出一个本方法，那就是先翻译成中文，这样以后再查阅时，我会先看自己的中文版，如果还是不理解或者忘记了，再翻看原版。所以你不要期望这是一本严格翻译的书，这只是我在阅读时将其中比较重要的内容做的中文记录而已。

编程就是借助于计算机来解决现实问题，每当遇到一个新的问题，我们总是先对其进行分析，甚至对问题归类，经常，我们碰到的新问题本质上和以前解决过的老问题是同一类型，仅仅是新瓶装旧酒而已。

既然很多问题本质上都相同，我们当然不想从零开始解决问题，换句话说，我们不想每次都重复造轮子！设计模式就是帮助咱们重用以前的轮子滴。这里的“轮子”指的是对于一类问题的解决方案。我们不但可以自己实现设计模式也可以借用别人的设计模式。

前人已经总结了一系列的常见设计模式，这本书就是学习这些常用的设计模式的，这样以后遇到一个新问题，我们分析分析，就可以套用某个现成的设计模式来解决啦。

本书的一大特点是在讲解每一个设计模式时，都配有可运行的Java示例代码，帮助理解，你无需担心自己的Java水平，本书的示例都很简单。

目录

本书的章节可分为三大部分。

Part 1 认识设计模式

Part 1 引入(设计)模式的概念以及如何使用它们。在这一部分我们将学习策略(Strategy)、工厂(Factory)、观察者(Observer)和单例(Singleton)等模式。

Part 2 成为OOP小工

(设计)模式非常依赖面向对象编程(OOP)，在这一部分，你将会看到如何利用设计模式打磨你的OOP能力。比如使用子类和模板方法(Template Method)模式重定义算法的步骤；使用适配器(Adapter)模式将一个对象的接口转换成一个完全不同的接口；使用迭代器(Iterator)和组成(Composite)模式着手解决对象集合；使用命令(Command)和中介者(Mediator)模式协调对象。相信Part2过后，你将从OOP小白成为OOP小工。

Part 3 现代设计模式

除了标准的23个设计模式，这一部分还将介绍一些新的设计模式，要知道设计模式是在不断发展滴。希望Part3过后，你也能从一些列新问题中提炼出新的设计模式。

恭喜你！你的问题被前人解决过

本章，

- 介绍设计模式
- 了解设计模式如何帮助我们解决问题
- 扩展OOP
- 学习几个具体的设计模式

一个设计模式，就是对于一类问题的经过测试的解决方法。如果你对本书介绍的设计模式都熟悉，那么以后你遇到一个新的问题，你就可以在大脑中思索，然后运用合适的设计模式来解决它，既节省了时间又保证了正确性。

Erich Gamma在1995年出版的《Design Patterns: Elements of Reusable Object-Oriented Software》中介绍了23种标准地设计模式，这23种模式也被程序员称为Gang of Four，或者GoF。

不要被23吓到，其中最常用的模式也就几个，但是多了解一些总没坏处。本书也会花不少笔墨讲解一些近几年出现的新的设计模式。

中介者(Mediator)模式

图1-1展示了一个具体的实际场景，某一个购物网站，只有四个网页。由于网页之间可以进行跳跃，比如当前在“Welcome”页面，你点击某个超链接可以跳转到“Store”页面。那么，这就有一个问题，每个页面必须包含能够跳转到其他页面的代码，而这些代码很可能是重复性的，比如“Welcome”和“Goodbye”页面都包含能够跳转到“Store”的代码。

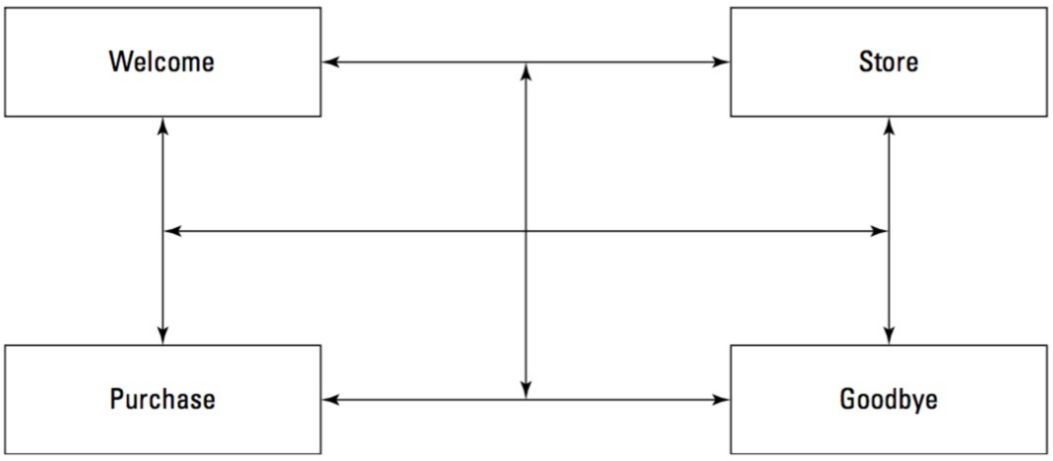


Figure 1-1: In this illustration of the basic navigation of a typical online store, the Mediator pattern mediates between objects.

对于这个问题，可以通过中介者模式来解决：使用一个中介者对象来包含所有的页面跳转间的代码。每个页面只需要将是否进行跳转的状态告诉中介者对象，而页面之间的跳转工作则交给中介者对象来进行。看下图1-2：

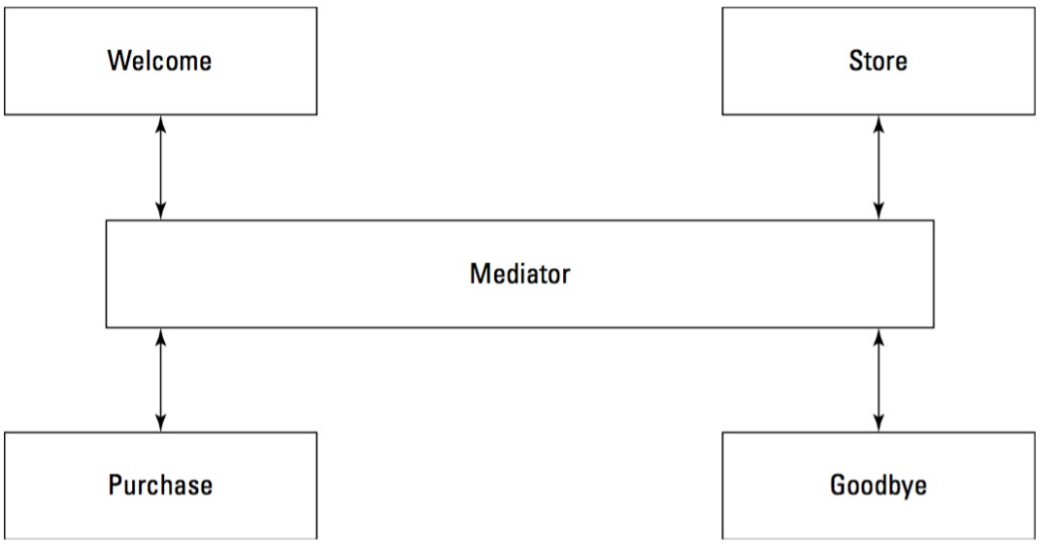


Figure 1-2: The mediator directs Web site traffic.

中介者模式，用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显示地相互引用，在本例中中介对象将跳转代码和具体的页面内容进行了分离，减少了页面之间的耦合性，如果以后网站增加了一些页面，我们也只需修改中介者对象，而无需修改现有的页面内容。

那么哪些情况适合使用中介者模式呢？

- 一组对象以定义良好但是复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。
- 一个对象引用其他很多对象并且直接与这些对象通信，导致难以复用该对象。
- 想定制一个分布在多个类中的行为，而又不想生成太多的子类。

适配器(Adapter)模式

秋高气爽适合出游，不差钱的你打算去美帝腐败两天，旅游得带上手机吧，可是问题来了，咱们国家的标准电压有效值是220V，美帝的是110V，怎么充电呢？你到Google一搜，发现有一种东西叫做“转换器插头”，

你再一搜，发现原来针对不同国家的电压标准，就有不同的转换器插头,

这个转换器插头就运用了我们要讲的适配器模式，它的作用就是将一个类的接口转换成客户希望的另一个接口。也被称作包装器(Wrapper)。

适配器模式产生的动机很直接，某些为了复用的工具类有时候无法复用，这不是因为功能不相符，而仅仅是因为它的接口与应用的接口不匹配。如果这时重新对工具类造轮子就太浪费时间和精力了。我们只需要构造一个适配器对象：

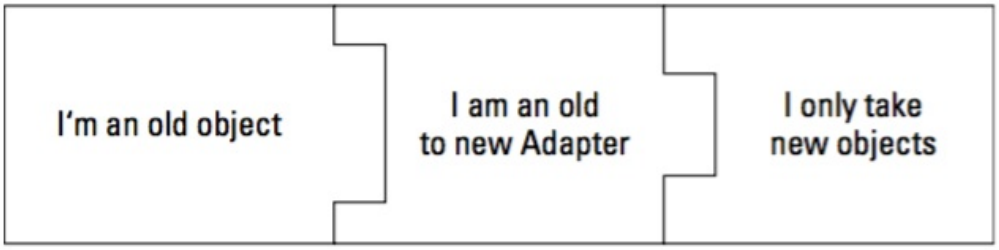


Figure 1-5: Old objects work with new objects via an adapter.

代理(Proxy)模式

代理模式，为其他对象提供一种代理以控制对这个对象的访问。

基本原理：

- 代理类和实体类都继承同一个接口（或抽象基类），具有相同的功能接口。
- 代理类内部维护一个实体类对象，眞正的功能实现是调用改对象的接口。

```
import java.util.*;


interface Image {
 public void displayImage();
}


//on System A class
RealImage implements Image {
 private String filename;
 public RealImage(String filename) {
 this.filename = filename;
 loadImageFromDisk();
 }
```

```
 private void loadImageFromDisk() {
 System.out.println("Loading " + filename);
 }

 public void displayImage() {
 System.out.println("Displaying " + filename);
 }
}

//on System B class
ProxyImage implements Image {
 private String filename;
 private Image image;

 public ProxyImage(String filename) {
 this.filename = filename;
 }

 public void displayImage() {
 if(image == null)
```

```
  image = new RealImage(filename);

  image.displayImage();

  }

 }



 class ProxyExample {

  public static void main(String[] args) {

  Image image1 = new ProxyImage("HiRes_10MB_Photo1");

  Image image2 = new ProxyImage("HiRes_10MB_Photo2");

  image1.displayImage(); // loading necessary

  image2.displayImage(); // loading necessary

  }

 }
```

代理模式是一种比较常见的模式，比较典型的应用场景：

- 远程代理(Remote Proxy)：为一个对象在不同的地址空间提供局部代表，比如**RPC**调用。
- 虚代理(Virtual Proxy)：根据需要创建开销大的对象，提高性能。
- 保护代理(Protection Proxy)：用来控制真实对象访问是的饿权限。
- 智能指针(Smart Reference)：取代了简单的指针，它在访问对象时执行一些附加操作，比如**C++11**中的智能指针。

观察者(Observer)模式

观察者模式面向的需求是：A 对象（观察者）对 B 对象（被观察者）的某种变化高度敏感，需要在 B 变化的一瞬间做出反应。举个例子，新闻里喜闻乐见的警察抓小偷，警察需要在小偷伸手作案的时候实施抓捕。在这个例子里，警察是观察者，小偷是被观察者，警察需要时刻盯着小偷的一举一动，才能保证不会漏过任何瞬间。程序的观察者模式和这种真正的『观察』略有不同，观察者不需要时刻盯着被观察者（例如 A 不需要每过 2ms 就检查一次 B 的状态），而是采用注册(Register)或者称为订阅(Subscribe)的方式，告诉被观察者：我需要你的某某状态，你要在它变化的时候通知我。 典型的使用场景就是图形化界面编程时为按钮设置监听器。比如，开发者添加了一个按钮(Button)，希望用户用鼠标点击此按钮后会弹出警告框。在这个场景中，用户点击按钮就是一个事件(event)，可是用户何时会点击按钮确实不确定的，这怎么办呢？通用的做法是给按钮添加一个监听方法，这个方法负责监听用户的点击行为，也就是观察者，一旦点击事件发生，观察者会产生一个警告框。

所以，观察者模式又被称为发布-订阅(publish-subscribe)模式。

观察者模式产生的动机，将一个系统分割成一系列相互协作的类有一个常见的副作用：需要维护相关对象间的一致性。我们不希望为了维护一致性而使各类紧密耦合，因为这样降低了它们的可重用性。

注意，对于同一个被观察者，可以有多个观察者，好比一个小偷可以被多个警察同时监控，一旦小偷作案，监控的警察都会行动起来进行抓捕：

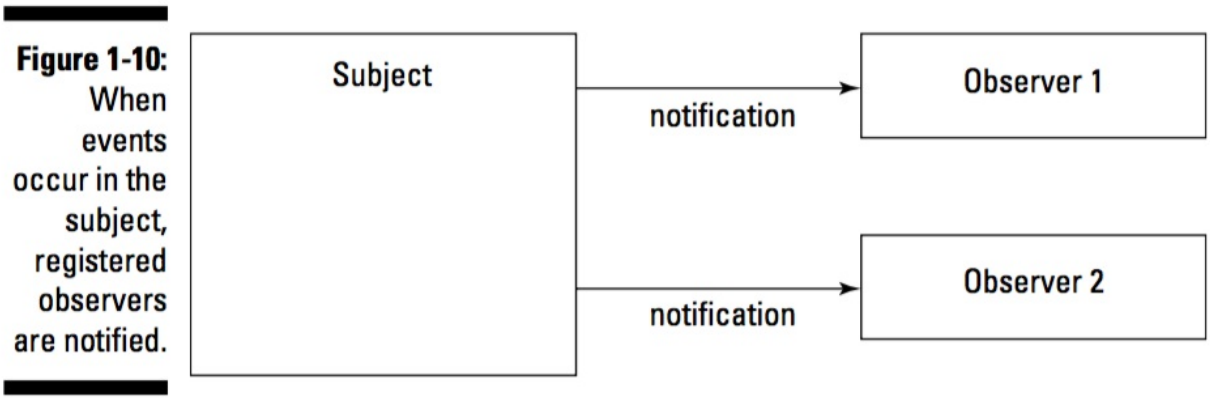


Figure 1-10: When events occur in the subject, registered observers are notified.

适用性：

- 当一个抽象模型有两个方面，其中一个方面依赖于另一个方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。
- 当对一个对象的改变需要同时改变其他对象，而不知道具体有多少对象有待改

变。

- 当一个对象必须通知其他对象，而它又不能假定其他对象是谁。换言之，你不希望这些对象是紧密耦合的。

策略(Strategy)模式，从理论到实践

本章，

- 扩展OOP
- 认识抽象、封装、多态和继承
- 从“is-a”到“has-a”
- 使用算法解决问题
- 使用策略设计模式

初级程序员往往认为OOP是万能的，还需要设计模式做啥？实际上，设计模式恰恰可看作是对OOP的扩展。

扩展OOP

设计模式圣经《Design Patterns: Elements of Reusable Object-Oreiented Software》的题目中包含了设计模式的两个重要属性，一个是reuse，即重复性；另一个就是OOP，设计模式产生的背景就是协助OOP解决问题的。

OOP(面向对象编程)诞生的背景是程序越来越复杂，通过引入对象来对程序各部分进行抽象继而实现了对程序的分割。本质上属于分治(divide-conquer)的思想。

OOP有四大属性：抽象(abstraction)、封装(encapsulation)、多态(polymorphism)和继承(inheritance)。

抽象并不是一种具体的编程技巧，它应用在分析问题的过程中，一个好的程序员必须要具有对问题进行抽象的能力，这样才能达到事半功倍。啥是抽象？抽象就是你在脑中对问题进行分割，思考用OOP如何去解决的过程。

只有做到心中有数，才能码代码如有神。

抽象工作完成后，下一步就要对对象进行封装了，这个对象要包含哪些方法？包含哪些数据？public还是private？怎么设计对外接口？说白了，脏活累活都要在封装完成，对于使用者来说，他只看到了对象的接口，而对象内部复杂的逻辑则完全隐藏。

设计模式在封装阶段发挥了重要作用，一个最基本的设计思路是将变化最多或者最需要维护的那部分进行封装。比如中介者(Mediator)模式将页面跳转部分进行封装。

OOP的另两个基石是多态和继承，不过对于设计模式来说，它更偏爱组合(composition)而不是继承。继承是一种“is-a”关系，比如正方形是一个长方形。而组合是一种“has-a”关系，比如汽车类中包含轮子对象、方向盘对象。由于“has-a”关系的存在，一个对象会包含多个不同类型的对象，我们又期望尽可能的用相同的代码来操作不同的对象，所以多态在设计模式中占有重要地位。

可能你有疑问了，为啥要用组合而不是继承呢？看一个具体的例子吧，你所在的公司接到了一个单子，和宝驴汽车公司合作设计车型，CTO说这好办，就用继承解决，先创建一个汽车基类：

```
public abstract class Vehicle {
    public Vehicle() {
    }
    public void go() {
        System.out.println("Now I'm driving.");
    }
}
```

然后想要啥车型，说吧，甲方说先来一个街车赛车，CTO说这好办，

```
public class StreetRacer extends Vehicle {
    public StreetRacer() {
    }
}

public class Run {
    public static void main(String[] args) {
        StreetRacer streetRacer = new StreetRacer();
        streetRacer.go();
    }
}
```

程序输出：

```
Now I'm driving.
```

甲方说再来一个F1赛车呢？CTO说这也好办，

```
public class FormulaOne extends Vehicle {
    public FormulaOne() {
    }
}


public class Run {
    public static void main(String[] args) {
        StreetRacer streetRacer = new StreetRacer();
        FormulaOne formulaOne = new FormulaOne();

        streetRacer.go();
        formulaOne.go();
    }
}

输出：
Now I'm driving.
Now I'm driving.
```

CTO很开心，继承就是好用啊。过几天甲方又说，能造直升飞机吗？CTO一拍脑袋，好办啊，

```
public class Helicopter extends Vehicle {
    public Helicopter() {
    }
}

public class Run {
    public static void main(String[] args) {
        StreetRacer streetRacer = new StreetRacer();
        FormulaOne formulaOne = new FormulaOne();
        Helicopter helicopter = new Helicopter();

        streetRacer.go();
        formulaOne.go();
        helicopter.go();
    }
}
```

输出：

```
Now I'm driving.
Now I'm driving.
Now I'm driving.
```

咦，好像哪里不对，直升机不会开啊，它可是会飞的呀。没想到过了两天，甲方又说我们还想造喷气式飞机，如果还继承Vehicle，

```
public class Jet extends Vehicle {
    public Jet() {
    }
}

public class Run {
    public static void main(String[] args) {

        StreetRacer streetRacer = new StreetRacer();
        FormulaOne formulaOne = new FormulaOne();
        Helicopter helicopter = new Helicopter();
        Jet jet = new Jet();

        streetRacer.go();
        formulaOne.go();
        helicopter.go();
        jet.go(); }}

输出：
Now I'm driving.
Now I'm driving.
Now I'm driving.
Now I'm driving.
```

情况越来越糟糕了，CTO说这好办，修改Helicopter的go方法就好了，

```
public class Helicopter extends Vehicle {
    public Helicoper(){

    }
    @Override
    public void go() {
        System.out.println("Now I'm flying.");
    }
}
```

或者创建一个接口，这个接口包含一个go方法，每个类都必须实现自己的go方法。

```
public class Helicoper implements IFly {
    public Helicoper(){
    }

    public void go() {
        System.out.println("Now I'm flying.");
    }

}
```

你想，如果车型很多，这样做代码就得不到重用了，这个方法也不好。

从“is-a”到“has-a”

相对于严格的继承关系，组合就显得灵活多了，特别是对于经常变换的部分，用组合实现要好太多，因为继承属于硬编码，直接将变化部分编码到基类，每一个子类都会得到相同的继承。

上一节的例子中，经常变换的部分就是go方法，我们需要把它独立出来。对于汽车和飞机来说，go的实现是不同的，而每一种对go的实现又被称为一种算法(algorithm)。每一个算法针对一类具体的对象/任务。

为了让所有的算法都实现相同的方法，我们需要创建一个接口来保护此方法，

```
public interface GoAlgorithm {
    public void go();
}
```

然后我们需要实现不同的算法，每一个算法都针对一类车型，

```
public class GoByDrivingAlgorithm implements GoAlgorithm {
    public void go() {
        System.out.println("Now I'm driving.");
    }
}

public class GoByFlyingAlgorithm implements GoAlgorithm {
    public void go() {
        System.out.println("Now I'm flying.");
    }
}

public class GoByFlyingFastAlgorithm implements GoAlgorithm {
    public void go() {
        System.out.println("Now I'm flying fast.");
    }
}
```

有了上面的算法，我们就可以结合“has-a”来实现各个车型了。还是来创建一个基类，

```
public abstract class Vehicle{
    private GoAlgorithm goAlgorithm;

    public Vehicle(){
    }

    public void setGoAlgorithm(GoAlgorithm algorithm){
        goAlgorithm = algorithm;
    }

    public void go(){
        goAlgorithm.go();
    }

}
```

为了实现各个具体的车型，在继承基类时，我们只需要设置具体的算法即可，

```
public class StreetRacer extends Vehicle{
    public StreetRacer(){
        setGoAlgorithm(new GoByDrivingAlgorithm());
    }
}

public class FormulaOne extends Vehicle{
    public FormulaOne(){
        setGoAlgorithm(new GoByDrivingAlgorithm());
    }
}

public class Helicopter extends Vehicle{
    public Helicopter(){
        setGoAlgorithm(new GoByFlyingAlgorithm());
    }
}

public class Jet extends Vehicle{
    public Jet(){
        setGoAlgorithm(new GoByFlyingFastAlgorithm());
    }
}
```

测试一下吧，

```
public class StartTheRace {
    public static void main(String[] args){

        StreetRacer streetRacer = new StreetRacer();
        FormulaOne formulaOne = new FormulaOne();
        Helicopter helicopter = new Helicopter();
        Jet jet = new Jet();

        streetRacer.go();
        formulaOne.go();
        helicopter.go();
        jet.go();
    }
}
```

输出：

```
Now I'm driving.
Now I'm driving.
Now I'm flying.
Now I'm flying fast.
```

使用策略设计模式

上一节的例子其实就使用了策略(Strategy)设计模式，如果你使用继承来解决一个问题，

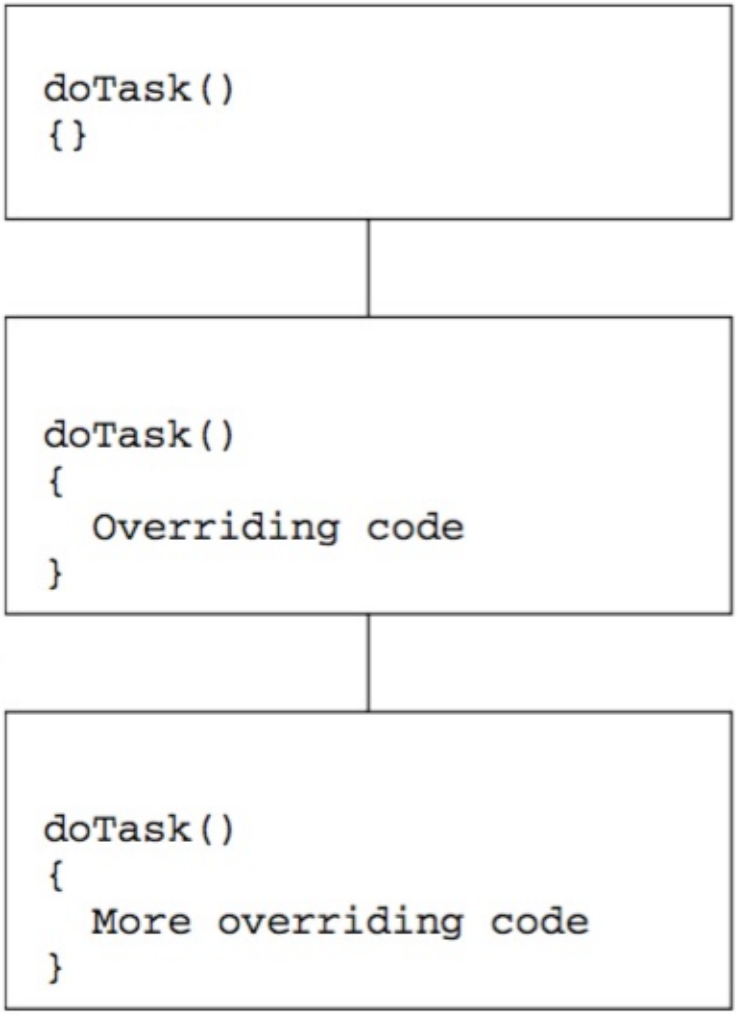


Figure 2-2: Adding tasks requires rewriting code.

随着程序规模的复杂，特例越来越多，继承关系也会随之变得很复杂。

策略模式则将易变的部分抽取出来进行单独封装（上一节中的go接口），然后在你需要的时候选择使用哪些封装好的对象。

通过组合方式来定义需要的类，

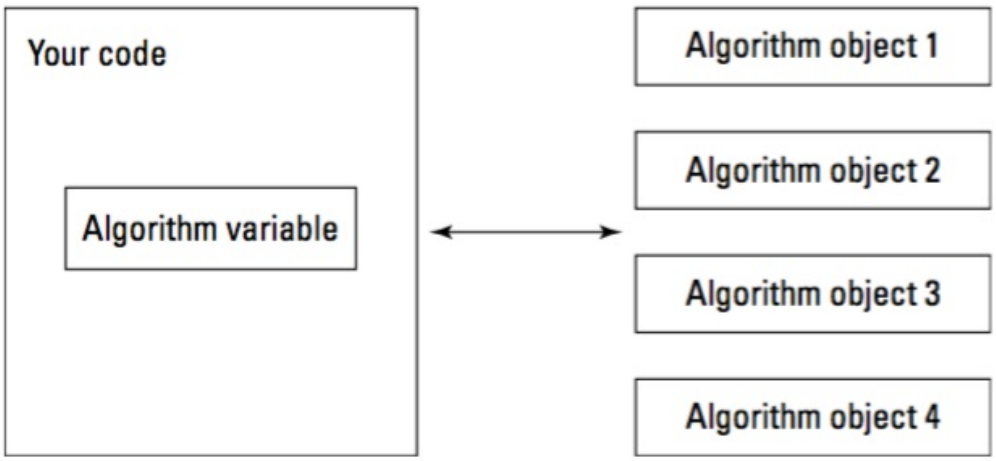


Figure 2-3: The Strategy pattern saves you time in the long run.

通常，每一个算法被称为一个策略。

使用装饰器和工厂模式扩展对象

本章，

- 介绍装饰器模式
- 创建你自己的装饰器
- 用装饰器包装对象
- 通过工厂模式扩展new操作
- 用工厂封装对象创建过程
- 使用工厂方法设计模式

本章主要介绍两个设计模式：装饰器(Decorator)模式和工厂(Factory)模式。装饰器模式用于扩展一个类的功能，对于一个写好的类，如果你要添加功能，可以借助于装饰器模式而不必修改现有的类哦。工厂模式则为创建对象提供了不同于new的新方式。

不修改，只增添

有编程经验的人都知道，做一个完整的项目，大部分时间都花在代码修改上面，要么是不断修改需求，要么就是前期设计不合理要进行重构。设计模式的出现，就是希望借助于某些固定形式，减少花在代码修改上面的时间和精力。所以，我们会看到前面介绍的几个模式都重点针对经常变换的代码部分。

需求总是不完善的，怎样做到不改变现有代码，而只添加功能来满足需求呢？继承可以做到但不够灵活，另一种方式就是下面要介绍的装饰器模式，别名Extender/Augmenter/Wrapper。看名字就知道，它的作用就是保持现有代码不修改，只增添代码来完善功能需求。

看一个具体的例子吧，有如下一个电脑类，它的description方法会返回对电脑的硬件描述，

```
public class Computer {

    public Computer() {
    }

    public String description() {
        return "You're getting a computer.";
    }
}
```

现在，有人觉得description方法返回的信息太少了，为啥不返还硬盘信息呢？于是你做出如下修改，

```
public class Computer {

    public Computer() {

     }

    public String description() {

         return "You're getting a computer and a disk.";

    }

}
```

可是电脑还有显示器啊，你又进行了修改，

```
public class Computer {

    public Computer() {

    }

    public String description() {
        return "You're getting a computer, a disk and a monitor.
";

    }

}
```

你嫌这样太麻烦了，一直修改现有的代码，早晚会出现问题哦，记住，对于现有的测试过的代码，应尽可能减少对它的修改。

怎么办呢？啊哈，可以使用前面学过的策略模式啊，把description方法看做前面的go方法，可以啊。不过，在这里我们用一个更直接的模式，装饰器模式，简单粗暴就是硬扩展，

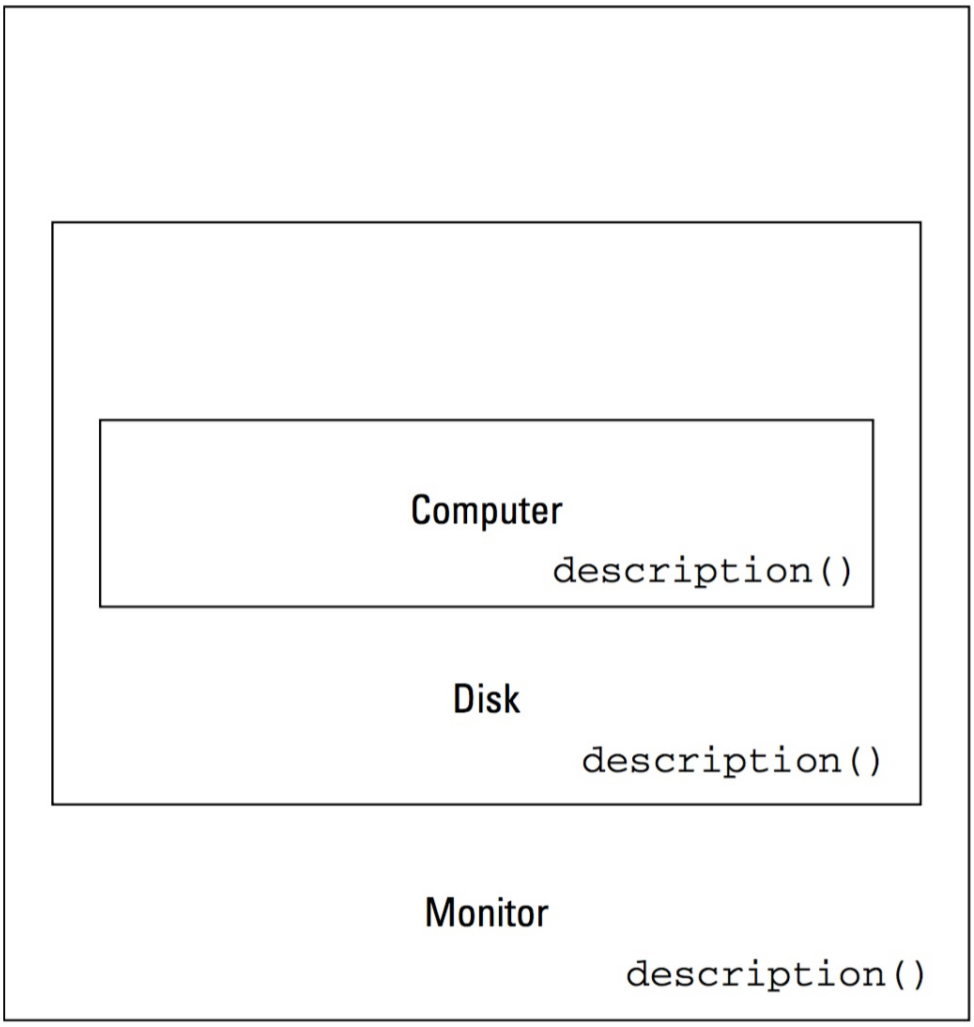


Figure 3-3: Finally, you add a monitor to the computer.

我们先创建核心组件，即不需要改变的部分，

```
public class Computer {

    public Computer() {
    }

    public String description() {
        return "computer.";
    }
}

public abstract class ComponentDecorator extends Computer {
    public abstract String description();
}
```

现在我们为电脑类添加一个硬盘，注意由于是包装(Wrapper)，Disk的构造器需要一个包装的对象，

```
public class Disk extends ComponentDecorator {
    Computer computer;

    public Disk(Computer c) {
        computer = c;
    }

    public String description() {
        return computer.description() + " add a disk.";
    }

}
```

再添加一个CD，

```
public class CD extends ComponentDecorator {
    Computer computer;

    public CD(Computer c) {
        computer = c;
    }

    public String description() {
        return computer.description() + " add a CD";
    }
}
```

再添加一个显示器，

```
public class Monitor extends ComponentDecorator {
    Computer computer;

    public Monitor(Computer c) {
        computer = c;
    }

    public String description() {
        return computer.description() + " add a monitor";
    }
}
```

细心的你应该看到了，Disk、CD和Monitor的构造器中形参都是Computer，这里就是利用多态特性。

好了，测试一下吧，

```
public class Test {
    public static void main(String[] args) {
        Computer computer = new Computer();

        computer = new Disk(computer);
        computer = new Monitor(computer);
        computer = new CD(computer);
        computer = new CD(computer);

        System.out.println("You're getting a " + computer.descri
ption() + '.');
    }
}
```

输出：

```
You're getting a computer. add a disk. add a monitor add a CD ad
d a CD.
```

通过工厂模式扩展**new**操作

由于你工作出色，宝驴项目进展顺利，现在进入后台开发阶段，CTO跑过来说，“你搞定数据库连接部分吧”。这对你来说，很简单，先创建连接Oracle数据库的类OracleConnection，然后就可以连接了，

```
Connection connection = new OracleConnection();
```

CTO思考了一会，“可以再连接微软SQL Server么？”你想了一会，有创建了一个SqlServerConnection类，

```
Connection connection = new SqlServerConnection();
```

“厉害哦，MySQL呢？我觉得默认连接MySQL比较合适”。CTO又说道，你又坐下来敲了一会代码，创建了MySqlConnection类，

```
Connection connection = new MySqlConnection();
```

由于现在有了三种连接方式，你决定再增加一个变量，让用户决定具体连接哪种数据库，

```
Connection connection;

if (type.equals("Oracle")) {
    connection = new OracleConnection();
}
else if (type.equals("SQL Server")) {
    connection = new SqlServerConnection();
}
else {
    connection = new MySqlConnection();
}
```

不赖哦，你心里想，不过现在系统里至少有200处连接数据库的地方，还是将连接数据库代码块放到一个单独的方法中吧，

```
public Connection createConnection(String type) {
    if (type.equals("Oracle")) {
        connection = new OracleConnection();
    }
    else if (type.equals("SQL Server")) {
        connection = new SqlServerConnection();
    }
    else {
        connection = new MySqlConnection();
    }

}
```

Wow，搞定了，好开心哦。 突然，CTO又跑过来说，“不光是MySql，还有MongoDB、PostgreSQL、LevelDB、Redis、HBase啊”。不知怎么地，你感觉自己的手不听话了，想打人。。。

其实不管添加多少数据库连接方式，都只需要修改createConnection方法就可以，但是由于此方法处于系统中的核心代码块，每次改写，都可能影响到系统的正常运行，甚至崩溃。这可太危险了。

想一想设计模式吧，把最容易改变的代码分离出来，也就是这里的createConnection方法，然后将其封装成一个对象。这里我们采用工厂模式，也就是把方法封装成一个工厂对象，用于创建数据库连接。

整个的过程如下：

- 1. 最开始，你使用new来创建Oracle数据库连接OracleConnection
- 1. 随着需求的修改，你又使用new创建SQL Server和MySQL连接。这部分代码快又在整个系统中重复200多次。
- 1. 所以你决定将数据库连接部分放到一个方法中createConnection，减少代码的重复编辑。
-

1. 然而createConnection方法还是由于需求的修改需要经常变换，再加上此方法位于系统核心代码块，你静下心来思考，决定使用工厂模式将此方法封装成对象，然后和系统核心代码分离，这样以后再对它修改也不会影响到核心代码块。

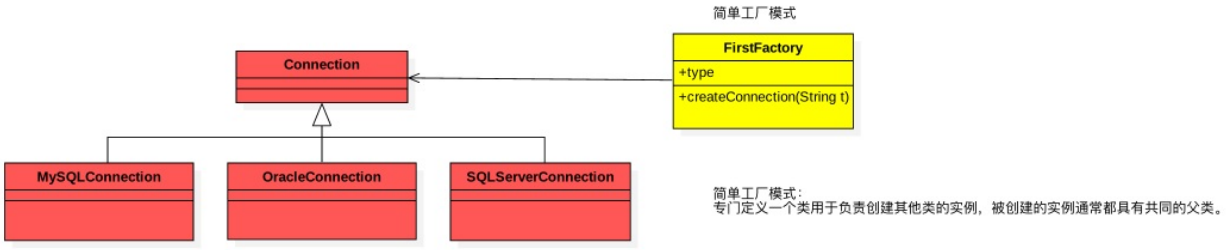
简单工厂(Simple Factory)模式

使用FirstFactory类封装连接创建过程，通过给其构造器传递type参数来决定创建何种数据库连接，

```
public class FirstFactory {
    protected String type;

    public FirstFactory(String t) {
        type = t;
    }

    public Connection createConnection() {
        if (type.equals("Oracle")) {
            return new OracleConnection();
        }
        else if (type.equals("SQL Server")) {
            return new SqlServerConnection();
        }
        else {
            return new MySqlConnection();
        }
    }
}
```

如下是创建数据库连接的过程，

```
FirstFactory factory;
factory = new FirstFactory("Oracle");

Connection connection = factory.createConnection();
```



接着上面的例子，

```
FirstFactory factory;
factory = new FirstFactory("Oracle");

Connection connection = factory.createConnection();

connection.setParams("username", "Steve");
connection.setParams("password", "Open the door!!!");
connection.initialize();
connection.open();

...
```

由于具体的连接方式有多重，为了保持对外API一致，我们将Connection设定为抽象类，

```
public abstract class Connection {
    public Connection() {
    }

    public String description() {
        return "Generic";
    }
}
```

再来创建几个具体的连接类，

```
public class OracleConnection extends Connection {
    public OracleConnection() {
    }

    public String description() {
        return "Oracle";
    }
}

public class SqlServerConnection extends Connection {
    public SqlServerConnection() {
    }

    public String description() {
        return "SQL Server";
    }
}

public class MySqlConnection extends Connection {
    public MySqlConnection() {
    }

    public String description() {
        return "MySQL";
    }
}
```

对于简单工厂类，有的人喜欢将其创建为final，即不允许继承，至于哪种方法好，则见仁见智了。

工厂方法(Factory Method)模式

CTO跑过来说，“宝驴公司对目前的数据库连接设计并不满意，每次我们这边更新FirstFactory，都可能会影响到他们那边的应用代码，能不能只给他们一个接口，至于具体的数据库连接实现，交给他们自己呢？”这是一个很实际的想法，然后，你想到了工厂方法设计模式。定义一个用于创建对象的接口或抽象类，让子类决定实例化哪一个类，使每一个类的实例化延迟到其子类。

换句话说，作为类库设计者只需要对外暴露接口，至于接口如何实现，则完全由类库调用者自己实现。

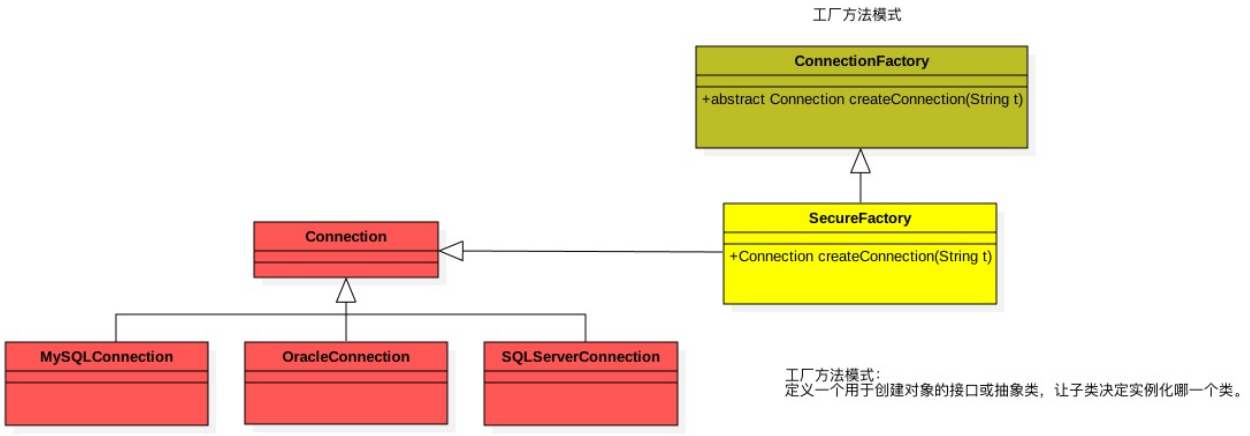
这里，我们用抽象类实现工厂方法模式，

```
public abstract class ConnectionFactory {
    public ConnectionFactory() {
    }

    public abstract Connection createConnection(String type);

}
```

然后，宝驴公司的开发人员就可以自己实现他们的连接类了，

```
public class SecureFactory extends ConnectionFactory {
    public Connection createConnection(String type) {
        if (type.equals("Oracle")) {
            return new SecureOracleConnection();
        }
        else if (type.equals("SQL Server")) {
            return new SecureSqlServerConnection();
        }
        else {
            return new SecureMySqlConnection();
        }
    }
}
```

"Wow, 这样安全多了。"宝驴的人很开心。。。。。。



抽象工厂(Abstract Factory)模式

在工厂方法模式中具体工厂负责生产具体的产品，每一个具体工厂对应一种具体产品，工厂方法也具有唯一性，一般情况下，一个具体工厂中只有一个工厂方法或者一组重载的工厂方法。但是有时候我们需要一个工厂可以提供多个产品对象，而不是单一的产品对象。

这句话可能难理解，以刚才连接数据库的例子说明，我们创建的工厂方法虽然可以为不同的数据库做连接，但说到底，这个工厂仅仅能用作连接数据库，更进一步的抽象，我们把“工厂”这个概念也抽象出来，有连接数据库的工厂，也有操作XML文件的工厂，或者是处理IO流的工厂。

使用观察者和职责链模式进行监控

本章，

- 学习观察者设计模式
- 使用Java Observable类
- 使用Java Observer接口
- 使用职责链设计模式

有一天，CTO走进你的办公室，“有人在修改我们的核心数据库，老板对此很生气，他说以后要第一时间告诉他”。你说，这好办，我用观察者模式搞定，以后一旦有人修改数据库，老板都会收到通知。

这一章，学习两个用于监控目标的设计模式，观察者(Observer)模式和职责链(Chain of Responsibility)模式。

观察者模式

对于核心数据库的修改，不但要第一时间通知老板、CTO，还要通知运维人员。也就是说，有多个对象需要及时获取通知。用观察者模式很好解决。

观察者模式的更多细节，可以翻阅前面的1.4节。

看一个Java中的例子，在用户界面编程时，观察者监听用户的点击行为，

```
JButton button = new JButton("Check Spelling");
JTextField text = new JTextField(30);

public void init() {
    Container contentPane = getContentPane();
    contentPane.setLayout(new FlowLayout());
    contentPane.add(button);
    contentPane.add(text);

    button.addActionListener(new ActionListener() { //使用匿名类
        public void actionPerformed(ActionEvent event) {
            text.setText("Good job");
        }
    });
}
```

通常，我们会为被观察对象创建一个接口，包含三个方法，分别是观察者注册方法、观察者移除方法和通知方法。

```
public interface Subject {

    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

再来创建一个观察者接口，

```
public interface Observer {
    public void update(String operation, String record);
}
```

接口都有了，再来创建具体的类，

```
import java.util.*;

public class Database implements Subject {
    private Vector<Observer> observers;
    private String operation;
    private String record;

    public Database() {
        observers = new Vector<Observer>();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    public void notifyObservers() {
        for (int loopIndex=0; loopIndex<observers.size(); loopIn
dex++) {
            Observer observer = (Observer)observers.get(loopInde
x);
            observer.update(operation, record);
        }
    }

    public void editRecord(String operation, String record) {
        this.operation = operation;
        this.record = record;
        notifyObservers();
    }

}
```

我们再来创建Observer类，

```
public class Archiver implements Observer {
    public Archiver() {
    }

    public void update(String operation, String record) {
        System.out.println("The Archiver says a " + operation +
" operation was performed on " + record);
    }

}
```

本章介绍的两个设计模式都是用于对其他对象发送通知的，属于弱耦合(loose coupling)技术，使用弱耦合比硬编码进行连接好处多的不要不要的。

上面的代码有一个很大的缺点，Observer直接操作subject的引用，这可是很大的隐患。

Java中一般通过继承**Observable**来创建subject类，

```
import java.util.Observable;
import java.util.Observer;

public class Database extends Observable {
    private String operation;
    private String record;

    public Database() {
    }

    public void editRecord(String operation, String record) {
        this.operation = operation;
        this.record = record;
        setChanged(); //必须先调用此方法，才能调用notifyObservers方法
        notifyObservers();
    }

    public String getRecord() {
        return record;
    }

    public String getOperation() {
        return operation;
    }

}
```

再来实现新版本的Observer，也就是实现**java.util.Observer**接口，

```
import java.util.Observer;

public class Archiver implements Observer {

    public Archiver() {
    }

    public void update(Observable obs, Object record) {
        System.out.println("The archiver says a " + ((Database)o
bs).getOperation() + " operation was performed on " + ((Database
)obs).getRecord());
    }

}
```

注意Observable是一个类而不是接口，这就限制了subject的灵活性，你可以自己设计Observable而无需非要继承java中的Observable。

职责链模式

职责链模式是一种对象的行为模式。在职责链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织和分配责任。

从击鼓传花谈起

击鼓传花是一种热闹而又紧张的饮酒游戏。在酒宴上宾客依次坐定位置，由一人击鼓，击鼓的地方与传花的地方是分开的，以示公正。开始击鼓时，花束就开始依次传递，鼓声一落，如果花束在某人手中，则该人就得饮酒。

比如说，贾母、贾赦、贾政、贾宝玉和贾环是五个参加击鼓传花游戏的传花者，他们组成一个环链。击鼓者将花传给贾母，开始传花游戏。花由贾母传给贾赦，由贾赦传给贾政，由贾政传给贾宝玉，又贾宝玉传给贾环，由贾环传回给贾母，如此往复，如下图所示。当鼓声停止时，手中有花的人就得执行酒令。

击鼓传花便是职责链模式的应用。职责链可能是一条直线、一个环链或者一个树结构的一部分。

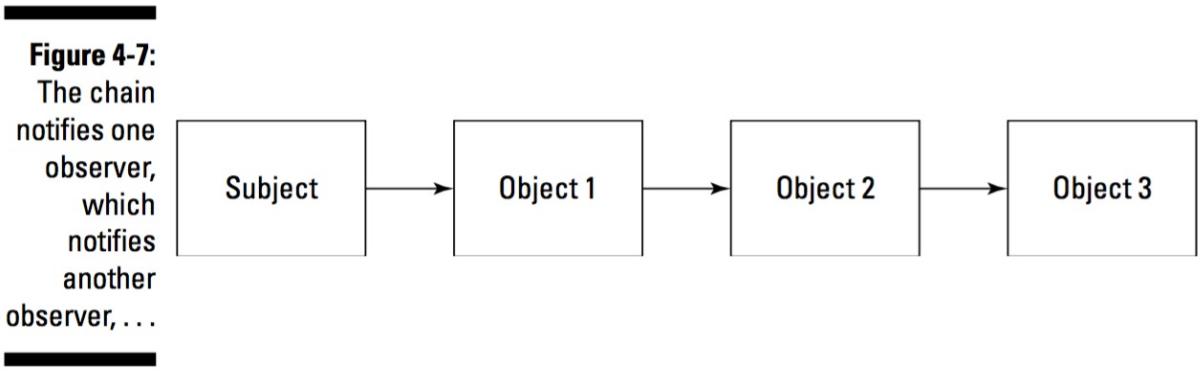


Figure 4-7: The chain notifies one observer, which notifies another observer, . . .

链上的某个对象决定处理此请求后，它后面的对象就收不到通知了。 因此职责链模式适用于那些等级不一致的观察者对象。即如果优先级低的能处理，就无需通知优先级高的观察者了，理论上，每个对象都有机会处理请求。

看一个具体的例子，用户右击UI界面中的某一个元素寻求帮助，如果应用前端能够处理此请求就处理，如果不能，将请求传递给应用中间层处理，如果中间层也不能处理，将请求传递给应用对象本身，应用会发送一个默认消息作为处理结果。

为了控制职责链上的所有对象行为一致，因此让他们都实现同一个接口。在这个例子中，将此接口命名为HelpInterface，它含有一个方法，getHelp，

```
interface HelpInterface {
    public void getHelp(int helpconstant);
}
```

然后接着创建职责链上的三个类，

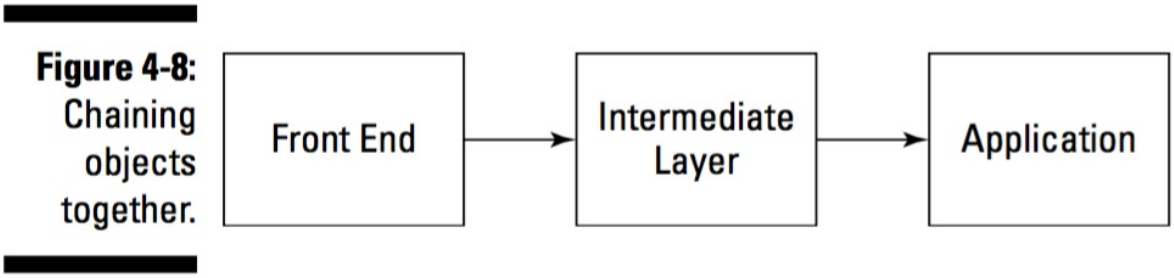


Figure 4-8: Chaining objects together.

为了将类对象串联起来，做法是将链上每个类的后一个对象作为形参传给它的构造器，请起来有点别扭，具体做法就是把一个**IntermediateLayer**对象作为参数传给**FrontEnd**构造器，

```
public class FrontEnd implements HelpInterface {
    HelpInterface successor; //后一个对象
    final int FRONT_END_HELP = 1;

    public FrontEnd(HelpInterface s) {
        successor = s;
    }

    public void getHelp(int helpConstant) {
        if (helpConstant != FRONT_END_HELP) {
            successor.getHelp(helpConstant);
        } else {
            System.out.println("This is the front end. Don't you
 like it?");
        }
    }

}
```

```
public class IntermediateLayer implements HelpInterface {
    final int INTERMEDIATE_LAYER_HELP = 2;
    HelpInterface successor;

    public IntermediateLayer(HelpInterface s) {
        successor = s;
    }

    public void getHelp(int helpConstant) {
        if (helpConstant != INTERMEDIATE_LAYER_HELP) {
            successor.getHelp(helpConstant);
        } else {
            System.out.println("This is the intermediate layer.
Nice, eh?");
        }
    }

}
```

最后一个类是Application，作为职责链上的门将，

```
public class Application implements HelpInterface {
    public Application() { //空
    }

    public void getHelp(int helpConstant) {
        System.out.println("This is the application.");
    }

}
```

测试一下上面的代码，

```
public class TestHelp {

    public static void main(String[] args) {
        final int FRONT_END_HELP = 1;
        final int INTERMEDIATE_LAYER_HELP = 2;
        final int GENERAL_HELP = 3;

        Application app = new Application();
        IntermediateLayer intermediateLayer = new IntermediateLa
yer(app);
        FrontEnd frontEnd = new FrontEnd(intermediateLayer);

        frontEnd.getHelp(GENERAL_HELP);
    }
}


输出：

This is the application.
```

适用性，

- 有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。
- 你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
- 可处理一个请求的对象集合应被动态指定。

从一个到多个：单例模式和享元模式

本章，

- 使用单例模式
- 创建单例
- 通过同步避免多线程问题
- 解决多线程问题的更好的方式
- 使用享元模式

有一天，CTO又跑来向你抱怨，“系统性能好差啊，感觉什么都运行的很慢，这是咋回事啊”，你说，我早就注意到系统中有一个很大的数据库对象，20M啊，你知道系统运行时创建多少个数据库对象？“219个”，天啊，系统运行时有219个20M的对象？怪不得系统性能差，光这些对象就要耗尽太多系统资源了，必须要有这么多对象吗？CTO说不出来，好吧，我看要用单例模式解决这个问题了。

本章讨论的是怎样控制代码中的对象数量，有两个模式可以做到：单例(Singleton)模式和享元(Flyweight)模式。

单例模式，每个类只创建一个对象；享元模式，一般也只为类创建一个对象，但看起来却好像有多个对象存在。

单例模式

单例模式，保证一个类只能实例化一个对象。看一下单例模式和new的区别，

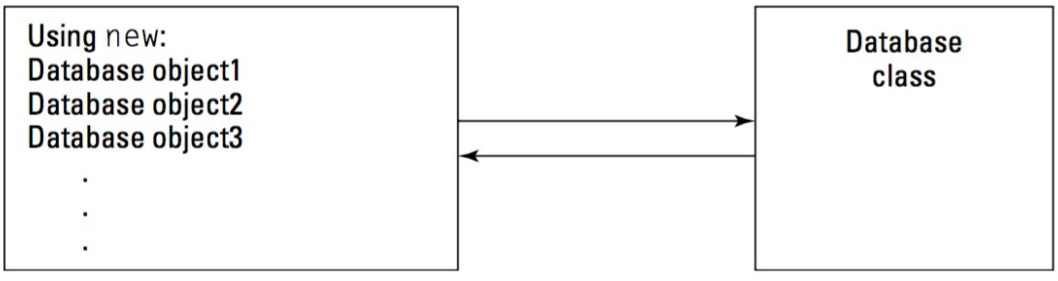


Figure 5-1: Creating object1, object2, object3, and many more.

单例模式不但可应用于减少系统资源使用，共享数据，还有利于多线程编程，比如一个数据库对象，如果每个数据库连接线程都创建一个数据库对象，在进行写操作时，容易造成数据不一致。

看具体的例子，先来创建一个数据库类，

```
public class Database {
    private int record;
    private String name;

    public Database(String n) {
        name = n;
        record = 0;
    }

    public void editRecord(String operation) {
        System.out.println("Performing a " + operation + " opera
tion on record " + record + " in database " + name);
    }

    public String getName() {
        return name;
    }


}
```

现在，不论谁使用new都可以创建数据库对象，还能创建任意多个。怎样阻止用new创建数据库对象呢？方法就是把构造器设置为private，

```
private Database(String n) {
    name = n;
    record = 0;
}
```

这样，除了Database内部的方法能调用构造器，其他代码都不能使用new了。

现在，就要在类内部创建一个调用构造器的方法了，这个方法返回一个Database对象，但是为了保证仅有一个Database对象存在，方法在创建对象时要先判断这个对象是否已经存在，同时，方法和对象都要为static，很好理解，

```
public class Database {

    private static Database singleObject;
    private int record;
    private String name;

    private Database(String n) {
        name = n;
        record = 0;
    }

    public static Database getInstance(String n) {
        if (singleObject == null ) {
            singleObject = new Database(n);
        }

        return singleObject;
    }

...

}
```

Wow，看起来不错哦。 不论你调用getInstance多少次，都只会创建一个Database对象，测试一下，

public class TestSingleton {

```
public static void main(String[] args) {
    Database database;
    database = Database.getInstance("products");

    System.out.println("This is the " + database.getName() + " d
atabase.");

    database = Database.getInstance("employees");
    System.out.println("This is the " + database.getName() + " d
atabase.");
}
```

}

输出，

This is the products database.

This is the products database.

```
```

多线程问题

前一节我们实行了一个单例模式的例子，目前一切都很完美。

```
public static Database getInstance(String n) {
    if (singleObject == null) {
        singleObject = new Database(n);
    }

    return singleObject;
}
```

准确地说，单线程模式下getInstance是完美滴，可是在多线程情境下就有一点问题了。首先，不管是单线程还是多线程，单例模式都要保证仅能实例化一个类对象。那么，多线程情境下有啥问题呢？问题就出在要先判断singleObject是不是null上，如果两个线程同时调用getInstance并且都先对singleObject进行测试，并且现在singleObject为null，那么两个线程可能都会单独创建一个Database对象。

这个问题怎么解决呢？对多线程熟悉的同学，知道一个简单的方法是使用Java的synchronized关键词，来限制每次只有一个线程能访问getInstance方法。

```
public class DatabaseSynchronized {

    private static DatabaseSynchronized singleObject;
    private int record;
    private String name;

    private DatabaseSynchronized(String n) {
        name = n;
        record = 0;
    }

    public static synchronized DatabaseSynchronized getInstance(
String n) {
        if (singleObject == null) {
            singleObject = new DatabaseSynchronized(n);
        }

        return singleObject;
    }

    public void editRecord(String operation) {
        System.out.println("Performing a " + operation + " opera
tion on record " + record + " in database " + name);
    }

    public String getName() {
        return name;
    }

}
```

测试一下吧，

```
public class TestSingletonSynchronized implements Runnable {
    Thread thread;

    public static void main(String[] args) {
        TestSingletonSynchronized t = new TestSingletonSynchroni
zed();
    }

    public TestSingletonSynchronized() {
        DatabaseSynchronized database;
        database = DatabaseSynchronized.getInstance("products");

        thread = new Thread(this, "second");
        thread.start();

        System.out.println("This is the " + database.getName() +
 " database.");
    }

    public void run() {
        DatabaseSynchronized database = DatabaseSynchronized.get
Instance("employees");

        System.out.println("This is the " + database.getName() +
 " database.");
    }

}
```

输出：

```
This is the products database.
This is the products database.
```

享元模式

享元模式，重在共享(share)。要分析能不能从现有对象中抽取出他们的共性，然后为共性创建对象，把共性对象作为可共享的。

比如，建立一个学生共享对象，

```
public class Student {

    String name;
    int id;
    int score;
    double averageScore;

    public Student(double a) {
        averageScore = a;
    }

    public void setName(String n) {
        name = n;
    }

    public void setId(int i) {
        id = i;
    }

    public void setScore(int s) {
        score = s;
    }

    public String getName() {
        return name;
    }

    public int getID() {
        return id;
    }
```

```
    public int getScore() {
        return score;
    }

    public double getStanding() {
        return (((double) score) / averageScore - 1.0) * 100.0;
    }



}
```

看看怎么使用，

```
public class TestFlyweight {

    public static void main(String[] args) {
        String[] names = {"Ralph", "Alice", "Sam"};
        int[] ids = {1001, 1002, 1003};
        int[] scores = {45, 55, 66};

        double total = 0;
        for (int loopIndex=0; loopIndex<scores.length; loopIndex
++) {
            total += scores[loopIndex];
        }

        double averageScore = total / scores.length;

        Student student = new Student(averageScore);

        for (int loopIndex=0; loopIndex<scores.length; loopIndex
++) {
            student.setName(names[loopIndex]);
            student.setId(ids[loopIndex]);
            student.setScore(scores[loopIndex]);

            System.out.println("Name: " + student.getName());
```

```
            System.out.println("Standing: " + Math.round(student
.getStanding()));
            System.out.println(" ");
        }
    }
}
```

输出：

```
Name: Ralph
Standing: -19

Name: Alice
Standing: -1

Name: Sam
Standing: 19
```

为了避免多线程时的问题，并且本例子只需要一个Student对象，所以不妨JVM加载本类时直接创建唯一的Student对象(单例模式)。

```
public class StudentThreaded {

    String name;
    int id;
    int score;
    double averageScore;
    private static StudentThreaded singleObject = new StudnetThr
eaded();

    private StudentThreaded() {
    }

...

}
```

让圆形钉子扎进方形洞，使用适配器模式和外观模式

本章，

- 使用适配器模式
- 创建适配器
- 解决适配器存在的问题
- 使用外观模式

这一章继续学习俩设计模式，适配器(Adapter)模式和外观(Facade)模式。

适配器模式,修复连接问题

适配器模式，能让你在不修改两个需要连接的对象内部情况下，让他们进行适配。这是非常重要的，因为很多情况下，你并不了解某个类的内部实现细节。

照常，咱们举一个例子，公司里正在使用某个库的API(类对象)，API包含四个方法，

- setFirstName
- setLastName
- getFirstName
- getLastName

有一天，CTO决定替换掉这个库，“经过我的调研，我发现了一个更好的库”，可是新的库函数API不同于以前，只包含两个方法，

- setName
- getName

CTO说，“这好办，把我们现在的系统代码改一改就可以啦”，呵呵，但是在领导面前不能有过激表现，现在的系统和setFirstName四个方法交互良好，根本没有修改的必要。其实嘛，不用修改现在的代码，用适配器模式就好了。“是吗？行，这个问题就交给你去做，我很看好你哦”。

你想了想，使用适配器模式大致如下，

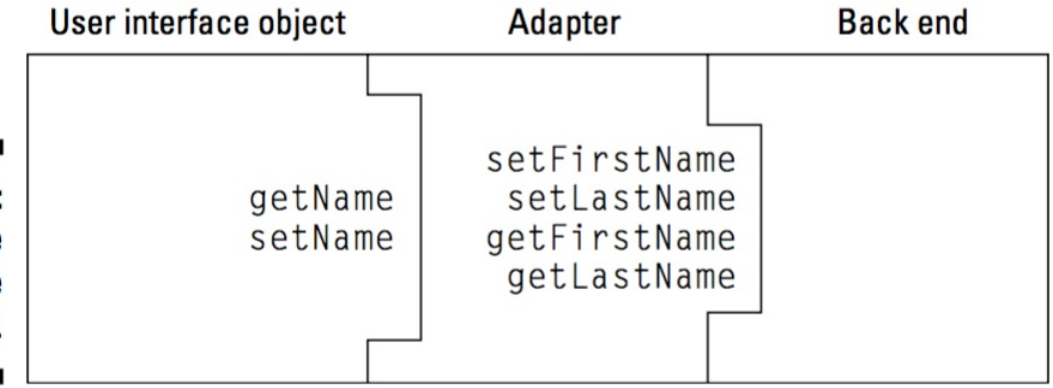


Figure 6-4: The Ace and Acme adapter.

适配器类型分两类，一类是对象适配器，一类是类适配器。 这里的例子是面向对象的。

先看一下原来的API接口，接口是一个AcmeClass对象，

```
public interface AcmeInterface {

     public void setFirstName(String f);
     public void setLastName(String l);
     public String getFirstName();
     public String getLastName();

}


public class AcmeClass implements AcmeInterface {

     String firstName;
     String lastName;

     public void setFirstName(String f) {
         firstName = f;
     }

     public void setLastName(String l) {
         lastName = l;
     }

     public String getFirstName() {
         return firstName;
     }

     public String getLastName() {
         return lastName;
     }
}
```

新的库函数API接口，

```
public interface AceInterface {

     public void setName(String n);
     public String getName();

}

public class AceClass implements AceInterface {

     String name;

     public void setName(String n) {
         name = n;
     }

     public String getName() {
         return name;
     }

}
```

现在公司的系统只接受Acme类对象，你现在要做的是，创建一个适配器，让现有系统可以接受Ace对象。

创建对象适配器，组合方式

对象适配器，也就是两个类对象之间的适配器，以组合的方式实现。适配器对象包含一个Ace对象，对外暴露Acme的方法。 然后用适配器对象和现有系统交互，在系统看来，它还是和Acme对象在打交道。

“旧瓶装新酒”。组合新类，

```
public class AceToAcmeAdapter implements AcmeInterface {

    AceClass aceObject;
    String firstName;
    String lastName;

    public AceToAcmeAdapter(AceClass s) {
        aceObject = s;
        firstName = aceObject.getName().split(" ")[0];
        lastName = aceObject.getName().split(" ")[1];
    }

    public void setFirstName(String f) {
        firstName = f;
    }

    public void setLastName(String l) {
        lastName = l;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

}
```

测试一下，

```
public class TestAdapter {

    public static void main(String[] args) {
        AceClass aceObject = new AceClass();
        aceObject.setName("Cary Grant");

        AceToAcmeAdapter adapter = new AceToAcmeAdapter(aceObjec
t);
        System.out.println("Customer's first name: " + adapter.g
etFirstName());
        System.out.println("Customer's last name: " + adapter.ge
tLastName());
    }

}
```

输出：

```
Customer's first name: Cary
Customer's last name: Grant
```

创建类适配器，继承方式

对象适配器通过组合的方式解决问题，类适配器则通过继承方式。

看一个例子，公司的UI团队说，“我们想用AWT，可是目前系统有些代码是用的Swing，AWT中有一个方法getState，Swing中对应的方法是isSelected，怎么办？”你告诉他，使用类适配器可以解决。

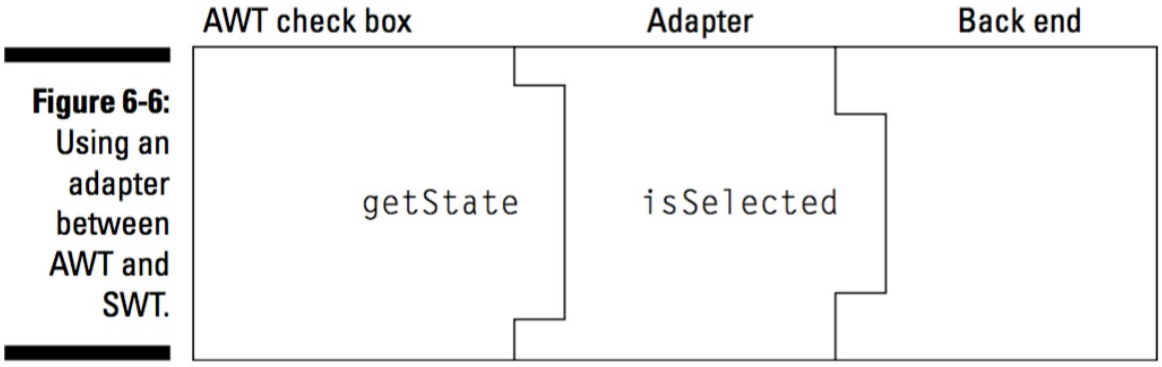


Figure 6-6: Using an adapter between AWT and SWT.

继承新类，

```
public class CheckboxAdapter extends Checkbox {
    public CheckboxAdapter(String n) {
        super(n);
    }

    public boolean isSelected() {
        return getState();
    }

}
```

对象适配器由于采用组合方式，要比类适配器更灵活。适配器模式不仅可以修改类或对象的行为，也可以像装饰器模式一样增加类或对象的行为哦。

外观模式

外观(Facade)模式将现有代码进行封装，目的是简化代码的调用。

举个例子，CTO笑嘿嘿的走到你面前，“我设计了一个打印机类，好好用。”你眼皮都没抬，怎么使用啊？”第一步，调用初始化方法；第二步，调用turnFanOn方法；第三步，调用热身方法warmUp；第四步，调用获取数据方法getData；第五步，调用数据格式化方法formatData；第“，你已经没有耐心听了，赶忙打住，我觉得可以用外观模式简化一下打印步骤，“哦？啥是外观模式”外观模式，用于简化API接口的调用啊。你想啊，一个不容易上手，不好用的软件，计算功能再强大，也会没市场的。“有道理，那你帮我改一下吧。”

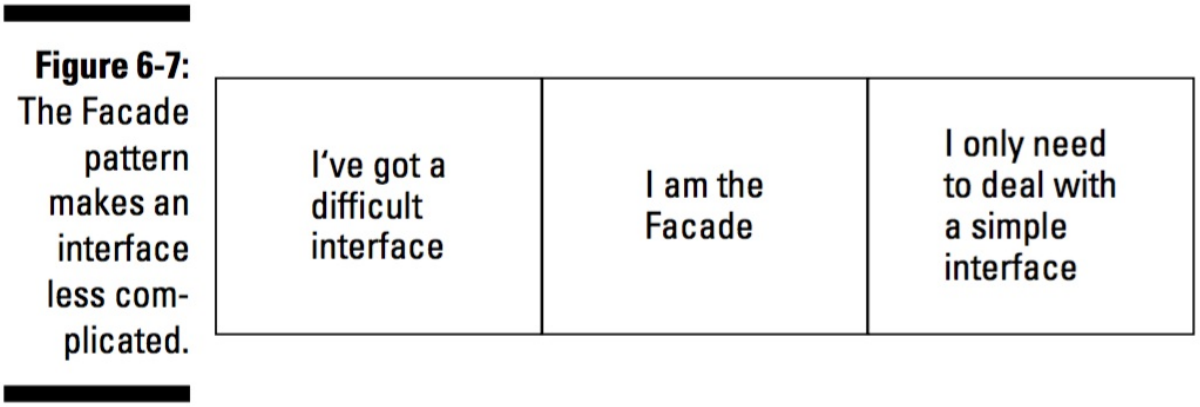


Figure 6-7: The Facade pattern makes an interface less complicated.

如果你不想重构代码，那么试一试外观模式吧，不过要记住，外观模式不是万能的，它只是对现有代码加了一层封装，一旦现有代码作出修改，外观层也要随之做出改变，这无疑会带来一定的工作量。

除了可以简化使用，外观模式也可以用于隐藏/混淆 实现细节，它只暴露最少的接口给用户。

来看一个比较复杂的类，

```
public class DifficultProduct {

    char nameChars[] = new char[7];

    public DifficultProduct() {
    }

    public void setFirstNameCharacter(char c) {
```

```
        nameChars[0] = c;
    }

    public void setSecondNameCharacter(char c) {
        nameChars[1] = c;
    }

    public void  setThirdNameCharacter(char c) {
        nameChars[2] = c;
    }

    public void setFourthNameCharacter(char c) {
        nameChars[3] = c;
    }

    public void setFifthNameCharacter(char c) {
        nameChars[4] = c;
    }

    public void setSixthNameCharacter(char c) {
        nameChars[5] = c;
    }

    public void setSixthNameCharacter(char c) {
        nameChars[6] = c;
    }

    public void setSeventhNameCharacter(char c) {
        nameChars[7] = c;
    }

    public String getName() {
        return new String(nameChars);
    }

}
```

看看创建DifficultProduct对象的过程，

```
DifficultProduct difficultProduct = new DifficultProduct();

difficultProduct.setFirstNameCharacter('p');
difficultProduct.setSecondNameCharacter('r');
difficultProduct.setThirdNameCharacter('i');
difficultProduct.setFourthNameCharacter('n');
difficultProduct.setFifthNameCharacter('t');
difficultProduct.setSixthNameCharacter('e');
difficultProduct.setSeventhNameCharacter('r');
```

好吧，实在无语了，再看一个简化版本，只对外暴露两个简单的方法，setName和getName，

```
public class SimpleProductFacade {

    DifficultProduct difficultProduct;

    public SimpleProductFacade() {
        difficultProduct = new DifficultProduct();
    }

    public void setName(String n) {
        char chars[] = n.toCharArray();

        if (chars.length > 0) {
            difficultProduct.setFirstNameCharacter(chars[0]);
        }

        if (chars.length > 1) {
            difficultProduct.setSecondNameCharacter(chars[1]);
        }

        if (chars.length > 2) {
            difficultProduct.setThirdNameCharacter(chars[2]);
        }

        if (chars.length > 3) {
            difficultProduct.setFourthNameCharacter(chars[3]);
        }
```

```
        if (chars.length > 4) {
            difficultProduct.setFifthNameCharacter(chars[4]);
        }

        if (chars.length > 5) {
            difficultProduct.setSixthNameCharacter(chars[5]);
        }

        if (chars.length > 6) {
            difficultProduct.setSeventhNameCharacter(chars[6]);
        }
    }

    public String getName() {
        return difficultProduct.getName();
    }

}
```

模板方法模式和生成器模式

本章，

- 使用模板方法设计模式
- 理解生成器模式和模板方法模式的区别
- 使用生成器模式

公司里接到了一个建造汽车机器人的合同，程序员们加班加点设计了一个Robot类，

```
public class Robot {

    public Robot() {
    }

    public void go() {
        start();
        getParts();
        assemble();
        test();
        stop();
    }

    public void start() {
        System.out.println("Starting...");
    }

    public void getParts() {
        System.out.println("Getting  a carburetor...");
    }

    public void assemble() {
        System.out.println("Installing the carburetor...");
    }

    public void test() {
        System.out.println("Reving the engine...");
    }

    public void stop() {
        System.out.println("Stopping...");
    }

}
```

运行一下，

```
public class TestRobot {
    public static void main(String[] args) {
        Robot robot = new Robot();
        robot.go();
    }

}
```

输出：

```
Starting...
Getting a carburetor...
Installing the carburetor...
Reving the engine...
Stopping...
```

通过模板方法模式创建机器人

“好消息，我们接到了机器人公司的另一单子，设计做饭机器人。”CTO很开心，可是程序员们很痛苦，他们不得不对汽车机器人类里面的方法完全重写，按照剧情，你出场了，用模板方法模式吧，因为两个机器人类包含的方法都相同，只不过方法实现略有区别。

先创建一个机器人模板类，这里方法go就是模板方法，

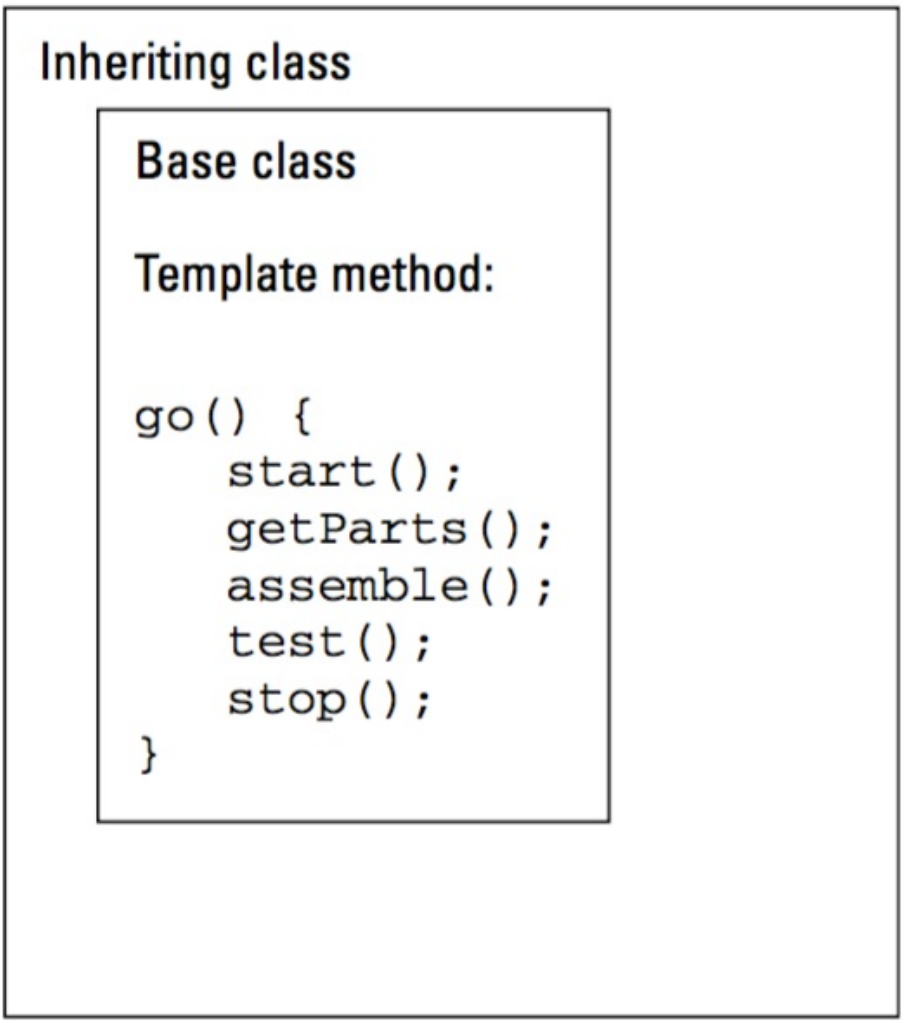


Figure 7-3: Inheriting from the template base class.

然后在创建具体的机器人类型时继承模板类，重写某些方法即可。

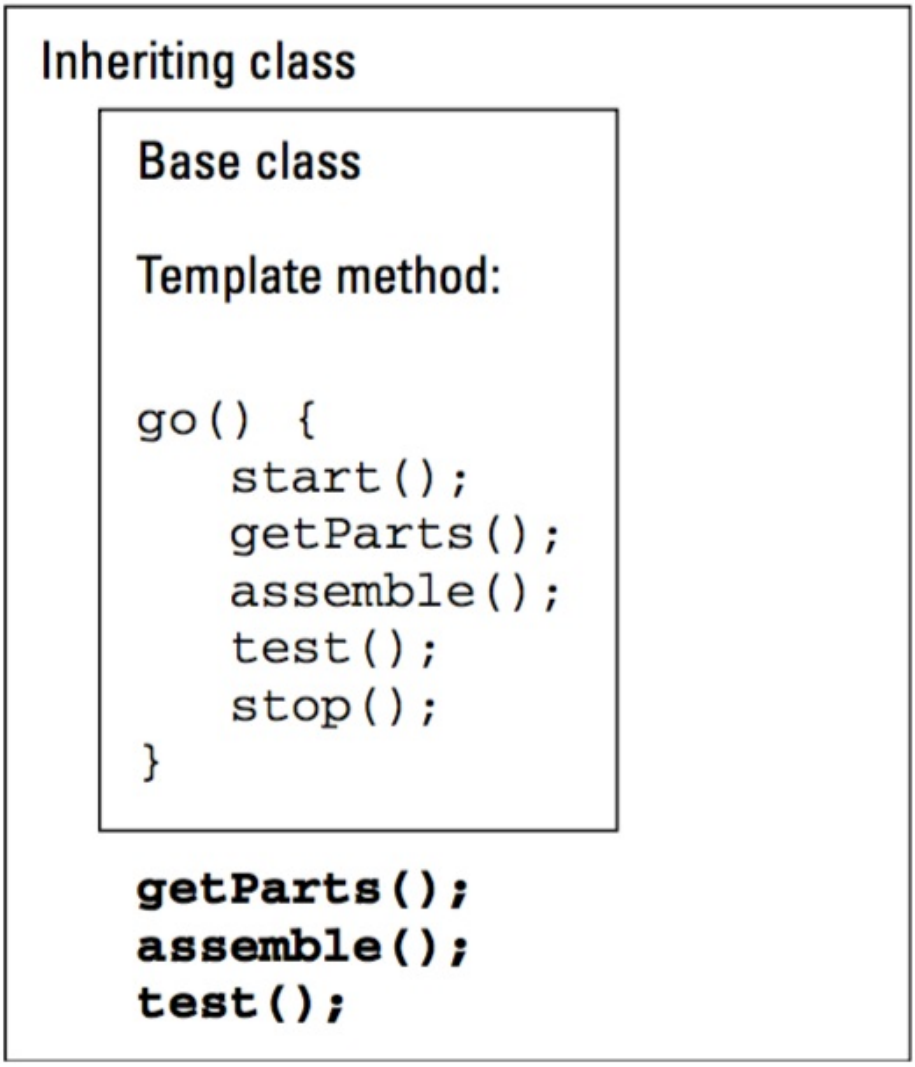


Figure 7-4: Modifying the inherited methods.

```
public abstract class RobotTemplate {
    public final void go() { //禁止重写
        start();
        getParts();
        assemble();
        test();
        stop();
    }

    public void start() {
        System.out.println("Starting...");
    }

    public void getParts() {
        System.out.println("Getting parts...");
    }

    public void assemble() {
        System.out.println("Assembling...");
    }

    public void test() {
        System.out.println("Testing...");
    }

    public void stop() {
        System.out.println("Stopping...");
    }

}
```

下面通过继承来创建汽车机器人，

```
public class AutomotiveRobot extends RobotTemplate {

    String name;

    public AutomotiveRobot(String n) {
        name = n;
    }

    public void getParts() {
        System.out.println("Getting a carburetor...");
    }

    public void asssemble() {
        System.out.println("Installing the carburetor...");
    }

    public void test() {
        System.out.println("Revving the engine...");
    }

    public String getName() {
        return name;
    }

}
```

再来创建做法机器人，

```
public class CookieRobot extends RobotTemplate {
    private String name;

    public CookieRobot(String n) {
        name = n;
    }

    public void getParts() {
        System.out.println("Getting flour and sugar...");
    }

    public void assemble() {
        System.out.println("Baking a cookie...");
    }

    public void test() {
        System.out.println("Crunching a cookie...");
    }

    public String getName() {
        return name;
    }

}
```

测试一下，

```
public class TestTemplate {

    public static void main(String[] args) {
        AutomotiveRobot automotiveRobot = new AutomotiveRobot("A
utomotive Robot");

        CookieRobot cookieRobot = new CookieRobot("Cookie Robot"
);
        System.out.println(automotiveRobot.getName() + ":");
        automotiveRobot.go();

        System.out.println();
        System.out.println(cookieRobot.getName() + ":");
        cookieRobot.go();

    }

}
```

输出，

```
Automotive Robot:
Starting...
Getting a carburetor...
Assembling...
Revving the engine...
Stopping...

Cookie Robot:
Starting...
Getting flour and sugar...
Baking a cookie...
Crunching a cookie...
Stopping...
```

钩子方法

如果你想控制模板方法中某一个方法的执行，可以通过钩子(hook)方法。看一个具体的例子，testOK方法就是钩子方法，

```
public abstract class RobotHookTemplate {

    public final void go() {
        start();
        getParts();
        assemble();
        if (testOK()) {
            test();
        }
        stop();
    }

    public void start() {
        System.out.println("Starting...");
    }

    public void getParts() {
        System.out.println("Getting parts...");
    }

    public void assemble() {
        System.out.println("Assembling...");
    }

    public void test() {
        System.out.println("Testing...");
    }

    public void stop() {
        System.out.println("Stopping...");
    }

    public boolean testOK() {
        return true;
    }

}
```

默认情况下，忽略钩子方法testOK，当然你可以根据实际情况重写testOK方法，

```
public class CookieHookRobot extends RobotHookTemplate {

    private String name;

    public CookieHookRobot(String n) {
        name = n;
    }

    public void getParts() {
        System.out.println("Getting flour and sugar...");
    }

    public void assemble() {
        System.out.println("Baking a cookie...");
    }

    public String getName() {
        return name;
    }

    public boolean testOK() {
        return false;
    }

}
```

适用性，

- 一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。
- 各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。
- 控制子类扩展。模板方法只在特定点用钩子操作，这样就只允许在这些点进行扩展。

生成器模式

“我刚和甲方沟通过，他们想对机器人执行流程拥有更高的控制权balabala”，现在的需求是对于go方法内的一系列方法，甲方想自己可以更改执行顺序并且可以添加方法，并且他们还可以自己控制创建何种机器人。

生成器(Builder)设计模式就是做这个的，将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

生成器模式和工厂模式比较像，不过工厂模式中创建流程通常是单步，即只有一个方法，而生成器模式的创建对象过程则有多个方法，是一个流程式的。

我们可以创建一个接口，让甲方自己去实现，

```
public interface RobotBuilder {
    public void addStart();
    public void addGetParts();
    public void addAssemble();
    public void addTest();
    public void addStop();
    public RobotBuilder getRobot();

}

public class CookieRobotBuilder implements RobotBuilder {
    CookieRobotBuildable robot;
    ArrayList<Integer> actions;

    public CookieRobotBuilder() {
        robot = new CookieRobotBuildable();
        actions = new ArrayList<Integer>();
    }

    public void addStart() {
        actions.add(new Integer(1));
    }

    public void addGetParts() {
        actions.add(new Integer(2));
```

```
    }

    public void addAssemble() {
        actions.add(new Integer(3));
    }

    public void addTest() {
        actions.add(new Integer(4));
    }

    public void addStop() {
        actions.add(new Integer(5));
    }

    public RobotBuildable getRobot() {
        robot.loadActions(actions);
        return robot;
    }

}
```

甲方通过配置actions来设置创建机器人的流程。

每个类型的机器人都要有一个go方法，看一看机器人接口，

```
public interface RobotBuildable {
    public void go();
}


public class CookieRobotBuildable implements RobotBuildable {
    ArrayList<Integer> actions;

    public CookieRobotBuildable() {
    }

    public void loadActions(ArrayList a) {
        actions = a;
    }
```

```
    public final void go() {
        Iterator itr = actions.iterator();
        while(itr.hasNext()) {
            switch((Integer)itr.next()) {
                case 1:
                    start();
                    break;
                case 2:
                    getParts();
                    break;
                case 3:
                    assemble();
                    break;
                case 4:
                    test();
                    break;
                case 5:
                    stop();
                    break;
        }
    }

    public void start() {
        System.out.println("Starting...");
    }

    public void getParts() {

}
```