

附录 B

答案

第 2 章 接口型模式介绍

挑战 2.1 的答案（第 9 页）

一个没有抽象方法的抽象类从功能来看与接口相似。然而，需要注意如下区别：

- 一个类可以实现多个接口，但却只能继承最多一个抽象类。
- 抽象类可以包含具体方法；接口的所有方法都是抽象的。
- 抽象类可以声明和使用字段；接口则不能，但可以创建静态的 `final` 常量。
- 抽象类中的方法可以是 `public`、`protected`、`private` 或者默认的 `package`；接口的方法都是 `public`。
- 抽象类可以定义构造函数；接口不能。

挑战 2.2 的答案（第 9 页）

A. 正确。接口方法总是抽象的，不管你是否对此进行声明。

B. 正确。接口方法总是公开的，不管你是否对此进行声明。

- C. 错误。接口的可见性会被限制在它所在的包中。对于本例，接口应该被标记为 `public`，这样，`com.oozinoz.simulation` 包外的类才能够访问它。
- D. 正确。例如，`List` 和 `Set` 接口都继承自 `java.util` 的 `Collection` 接口。
- E. 错误。没有方法的接口称之为标记接口（marker interface）。有时候，一个方法处于类继承层次的高处，例如 `Object.clone()`，但它却不适用于所有子类。如果希望子类在此样式下能够有选择地实现，就可以创建一个标记接口。通过声明对 `Cloneable` 标记接口的实现，就能够要求子类也实现 `Object` 的 `clone()` 方法。
- F. 错误。接口虽然能够声明静态的 `static` 和 `final` 字段作为常量，但不能声明实例字段。
- G. 错误。虽然是个不错的主意，但 Java 接口却不能这样做，这会使得实现类必须提供一个特定的构造函数。

挑战 2.3 的答案（第 10 页）

一个例子被当做一个类被注册成为事件的侦听器时，这些侦听器类会收到它们关心的通知，而不是调用者。例如，我们需要在触发 `MouseListener.mouseDragged()` 方法时采取某个动作，但对于同一个侦听器而言，`MouseListener.mouseMoved()` 方法却是一个空的实现。

第 3 章 适配器（Adapter）模式

挑战 3.1 的答案（第 15 页）

解决方案如图 B.1 所示。

`OozinozRocket` 类的实例具有 `PhysicalRocket` 和 `RocketSim` 对象的功能。适配器模式能够将已经实现的方法适配为客户端需要的方法。

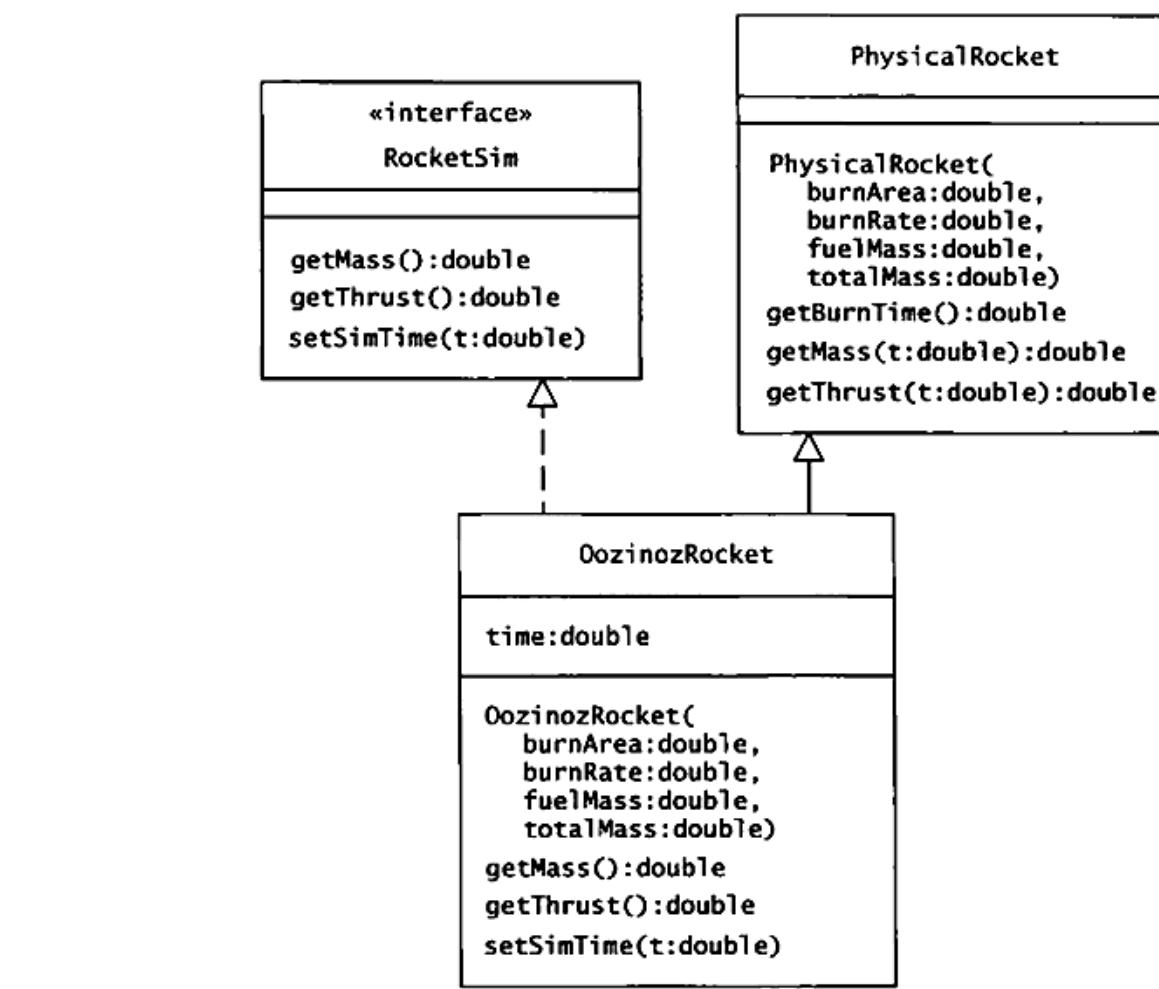


图 B.1 OozinozRocket 类将 PhysicalRocket 类适配为满足 RocketSim 接口的需要

挑战 3.2 的答案（第 16 页）

完整的类代码如下所示：

```
package com.oozinoz.firework;

import com.oozinoz.simulation.*;

public class OozinozRocket
    extends PhysicalRocket implements RocketSim {
    private double time;
    public OozinozRocket(
        double burnArea,
        double burnRate,
        double fuelMass,
        double totalMass) {
        super(burnArea, burnRate, fuelMass, totalMass);
    }
}
```

```

public double getMass() {
    return getMass(time);
}

public double getThrust() {
    return getThrust(time);
}

public void setSimTime(double time) {
    this.time = time;
}
}

```

你可以在本书提供的源代码 com.oozinoz.firework 包中找到这个类。

挑战 3.3 的答案 (第 19 页)

图 B.2 给出了解决方案。

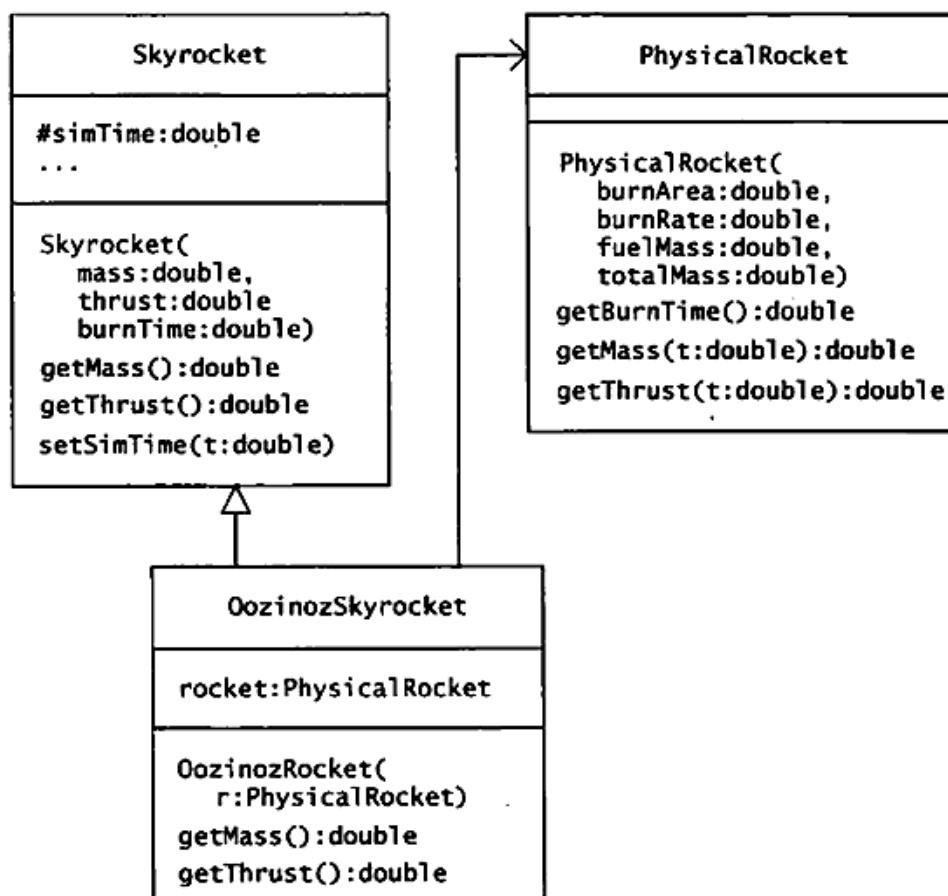


图 B.2 `OozinozSkyrocket` 对象是一个 `Skyrocket` 对象，但它的实现却是将调用转给 `PhysicalRocket` 对象

`OozinozSkyrocket` 类是一个对象适配器，它继承自 `Skyrocket`，因此它的实例拥有 `Skyrocket` 对象的功能。

挑战 3.4 的答案（第 20 页）

对象适配器使得 `OozinozSkyrocket` 类比采用类适配器更加脆弱的原因为：

- 没有 `OozinozSkyrocket` 类所提供的接口规范。由于 `Skyrocket` 的变化，可能在运行时出现编译时无法检测到的问题。
- `OozinozSkyrocket` 需要借助于访问其超类的 `simTime` 变量，但我们却无法保证该变量总是被声明为 `protected`，也不能保证处于 `Skyrocket` 类中的这一字段符合子类的意图。（我们不能期望提供者不会修改我们所依赖的 `Skyrocket` 代码，换言之，很难约束和控制它们所要做的事情。）

挑战 3.5 的答案（第 23 页）

代码实现大约如下所示：

```
package app.adapter;
import javax.swing.table.*;
import com.oozinoz.firework.Rocket;

public class RocketTable extends AbstractTableModel {
    protected Rocket[] rockets;
    protected String[] columnNames = new String[] {
        "Name", "Price", "Apogee" };

    public RocketTable(Rocket[] rockets) {
        this.rockets = rockets;
    }

    public int getColumnCount() {
        return columnNames.length;
    }

    public String getColumnName(int i) {
        return columnNames[i];
    }
}
```

```
public int getRowCount() {
    return rockets.length;
}

public Object getValueAt(int row, int col) {
    switch (col) {
        case 0:
            return rockets[row].getName();
        case 1:
            return rockets[row].getPrice();
        case 2:
            return new Double(rockets[row].getApogee());
        default:
            return null;
    }
}
```

`TableModel` 接口很好地展示了如何适应未来可能发生的变化。该接口以及实现了部分功能的 `AbstractTableModel`，减少了在标准 GUI 表控件中显示领域对象的实现工作。在接口的支持下，这种解决方案很容易进行适配，以应对未来的变化。

挑战 3.6 的答案（第 25 页）

- 一种观点：当用户单击鼠标时，我需要将 Swing 调用的结果转换或适配给对应的动作。换言之，当需要将 GUI 事件适配给应用程序的接口时，我使用了 Swing 的适配器类。我将一个接口转换为另一个，从而实现了适配器模式的意图。
- 一种争论：Swing 中的“适配器”类是桩（Stub），它们并没有真正转换或适配。在定义这些类的子类时，重写了你需要的方法。这些重写的方法和类才是适配器模式的例子。如果将“Adapter”改名为 `DefaultMouseListener`，就不会出现这样的争论了。

第 4 章 外观（Facade）模式

挑战 4.1 的答案（第 28 页）

示例与外观的区别如下：

- 示例通常是一个单独运行的应用程序，而外观不是。
- 示例通常包含了样本数据，而外观则没有。
- 外观通常是可配置的，示例不是。
- 外观的意图是为了重用，示例不是。
- 外观用在产品代码中，示例不是。

挑战 4.2 的答案（第 29 页）

`JOptionPane` 类是 Java 类库中少有的几个运用了外观模式的类。它属于产品代码，可配置，设计的目的是为了重用。除此之外，`JOptionPane` 类通过提供一个简单的接口使得对 `JDialog` 类的使用变得简单，满足外观模式的意图。你可能会认为该类是一个简化了的“子系统”，但事实上，一个独立的 `JDialog` 类并不能认为是子系统。但确切地讲，该类提供的丰富特性体现了外观类的价值。

Sun 公司在 JDK 中提供了多个示例程序。然而，这些类都不是 Java 类库的一部分。也就是说，它们并没有被放在 `java` 包中。外观属于 Java 类库，但示例不是。

`JOptionPane` 有多个静态方法，这使得它成为运用了外观模式的工具类。严格地讲，这样的类并不符合 UML 中工具类的定义，因为 UML 要求工具类只能处理静态方法。

挑战 4.3 的答案（第 29 页）

Java 类库之所以较少运用外观模式，可能存在如下几个合理但观点截然对立的理由：

- 作为 Java 开发者，通常要求对库中的工具做整体的了解。外观模式可能会限制这种运用系统的方式。它们可能会分散开发人员的注意力，并对类库提供的功能产生误解。
- 外观类介于丰富的工具包与特定应用程序之间。为了创建外观类，需要了解它所支持的应用程序类型。然而 Java 类库的用户如此之多，这种预先的支持是不可能的。
- Java 类库提供的外观类过少，这是一个缺陷。应该加入更多的外观类，提供更好的支持。

挑战 4.4 的答案（第 35 页）

最后的结果可能如图 B.3 所示。

注意，`createTitledBorder()` 方法不是静态方法。你给出的解决方案是否将这些方法声明为静态呢？不管是否声明为静态，都要分析其中的原因。

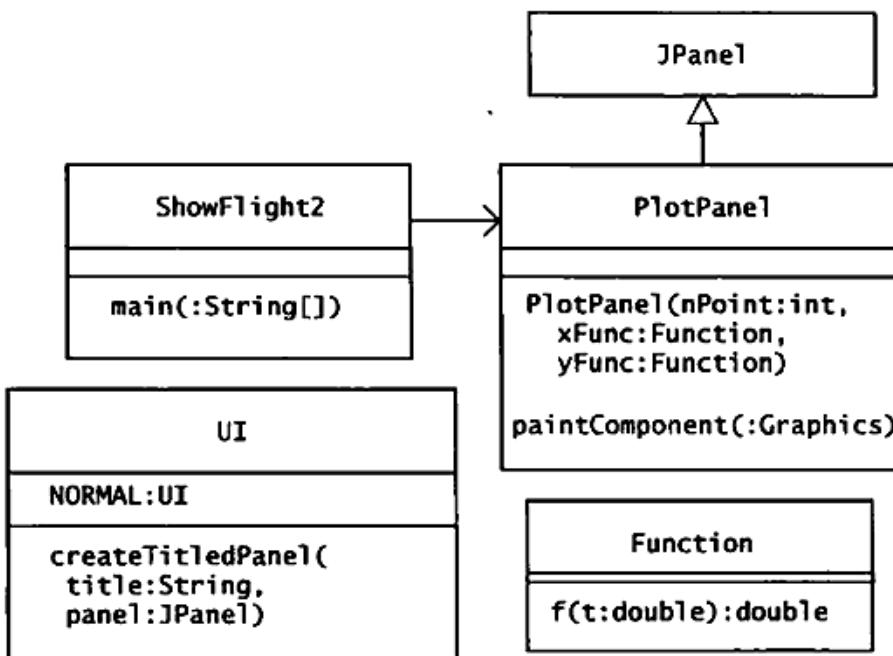


图 B.3 该图展示了将计算飞行路径的应用程序重构为多个类，每个类承担一个职责

本书的代码将 `UI` 方法声明为非静态的，这样 `UI` 的子类就可以重写它们，从而能够为用户界面的构建提供不同的工具实现。为了让标准的用户界面是可用的，设计参考了单例的 `NORMAL` 对象。

如下是与 `UI` 有关的代码：

```

public class UI {
    public static final UI NORMAL = new UI();
    protected Font font =
        new Font("Book Antiqua", Font.PLAIN, 18);

    // 省略部分实现

    public Font getFont() {
        return font;
    }

    public TitledBorder createTitledBorder(String title) {
        TitledBorder border =
            BorderFactory.createTitledBorder(
                BorderFactory.createBevelBorder(
                    BevelBorder.RAISED),
                title,
                TitledBorder.LEFT,
                TitledBorder.TOP);
        border.setTitleColor(Color.black);
    }
}
  
```

```
        border.setTitleFont(getFont());
        return border;
    }

    public JPanel createTitledPanel(
        String title, JPanel in) {
        JPanel out = new JPanel(); out.add(in);
        out.setBorder(createTitledBorder(title));
        return out;
    }
}
```

若要了解 GUI 工具包的更多内容，可以参考本书第 17 章抽象工厂模式中的内容，而第 8 章的单例模式则提供了关于单例的内容。

第 5 章 合成 (Composite) 模式

挑战 5.1 的答案（第 40 页）

设计合成类可用于维护合成对象的集合，使得合成对象既可以支持叶子对象，又能支持组合对象。

换言之，合成模式的设计使得我们能够将一种分组建模为另一种分组的集合。例如，我们可以将用户的系统权限定义为特定权限的集合或权限组。合成模式的另一个例子是对工作进程的定义，可以将其定义为进程步骤的集合以及其他进程。相比于将其定义为叶子对象集合的合成对象，这样的定义更为灵活。

若只支持叶子对象的集合，则合成对象只能有一层的深度。

挑战 5.2（第 41 页）

对于 Machine 类，getMachineCount() 方法的实现如下：

```
public int getMachineCount() {
    return 1;
}
```

类图显示 MachineComposite 使用了一个 List 对象，用以跟踪它的组件。要计算合成对

象中机器的数量，可以这样实现：

```
public int getMachineCount(){
    int count = 0;
    Iterator i = components.iterator();
    while (i.hasNext()) {
        MachineComponent mc = (MachineComponent) i.next();
        count += mc.getMachineCount();
    }
    return count;
}
```

如果使用 JDK 1.5，可以使用对 for 循环的扩展。

挑战 5.3 的答案（第 41 页）

答案见表 B.1 所示。

表 B.1 方法的定义

方 法	类	定 义
getMachineCount()	MachineComposite	返回组合中每个组件的数量和
	Machine	返回 1
isCompletelyUp()	MachineComposite	如果所有组件为“completely up”，则返回 true
	Machine	如果机器为“up”，则返回 true
stopAll()	MachineComposite	停止所有的组件
	Machine	停止机器
getOwners()	MachineComposite	创建一个 set，而非 list；为所有组件添加负责人，然后返回 set
	Machine	返回机器的负责人
getMaterial()	MachineComposite	返回组件中所有材料的集合
	Machine	返回当前的材料

挑战 5.4 的答案（第 45 页）

程序输出为：

```
Number of machines: 4
```

事实上，在 plant 工厂中，只有三台机器，但是 mixer 对象同时被 plant 与 bay 对象计数。这些对象包含的机器组件列表都引用了 mixer 对象。

结果出现了错误。工程师将添加的 `plant` 对象看做是 `bay` 合成对象的组件，调用 `getMachineCount()` 会导致死循环。

挑战 5.5 的答案（第 47 页）

`machineComposite.isTree()` 方法的合理实现为：

```
protected boolean isTree(Set visited) {  
    visited.add(this);  
    Iterator i = components.iterator();  
    while (i.hasNext()) {  
        MachineComponent c = (MachineComponent) i.next();  
        if (visited.contains(c) || !c.isTree(visited))  
            return false;  
    }  
    return true;  
}
```

挑战 5.6 的答案（第 49 页）

答案应显示如图 B.4 中的连线。

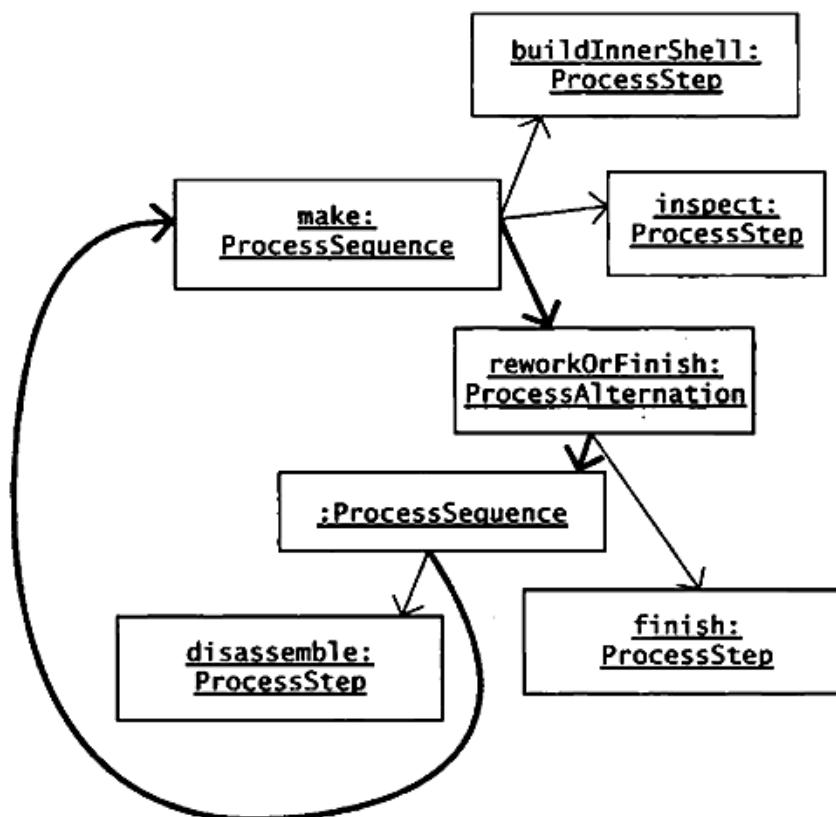


图 B.4 对象图环中的粗线表示这种循环是高空焰火弹生产流程与生俱来的

第 6 章 桥接 (Bridge) 模式

挑战 6.1 的答案 (第 53 页)

若要通过共同的接口控制多种机器，可以使用适配器模式，为每个控制器创建一个适配器类。每个适配器类都可以将标准的接口调用转换为对现有控制器的调用。

挑战 6.2 的答案 (第 54 页)

你所实现的代码可能会是这样：

```
public void shutdown() {  
    stopProcess();  
    conveyOut();  
    stopMachine();  
}
```

挑战 6.3 的答案 (第 56 页)

图 B.5 给出了解决方案。

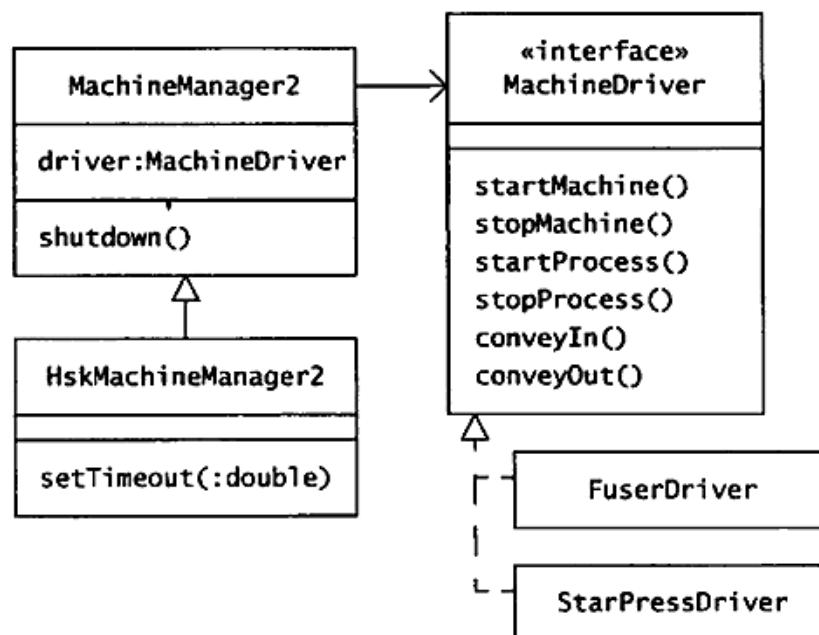


图 B.5 该图显示出了一个抽象 (MachineManager 的继承体系) 与实现 (抽象所使用的 MachineDriver 接口) 之间的分离

挑战 6.4 的答案（第 58 页）

图 B.6 给出了解决方案。

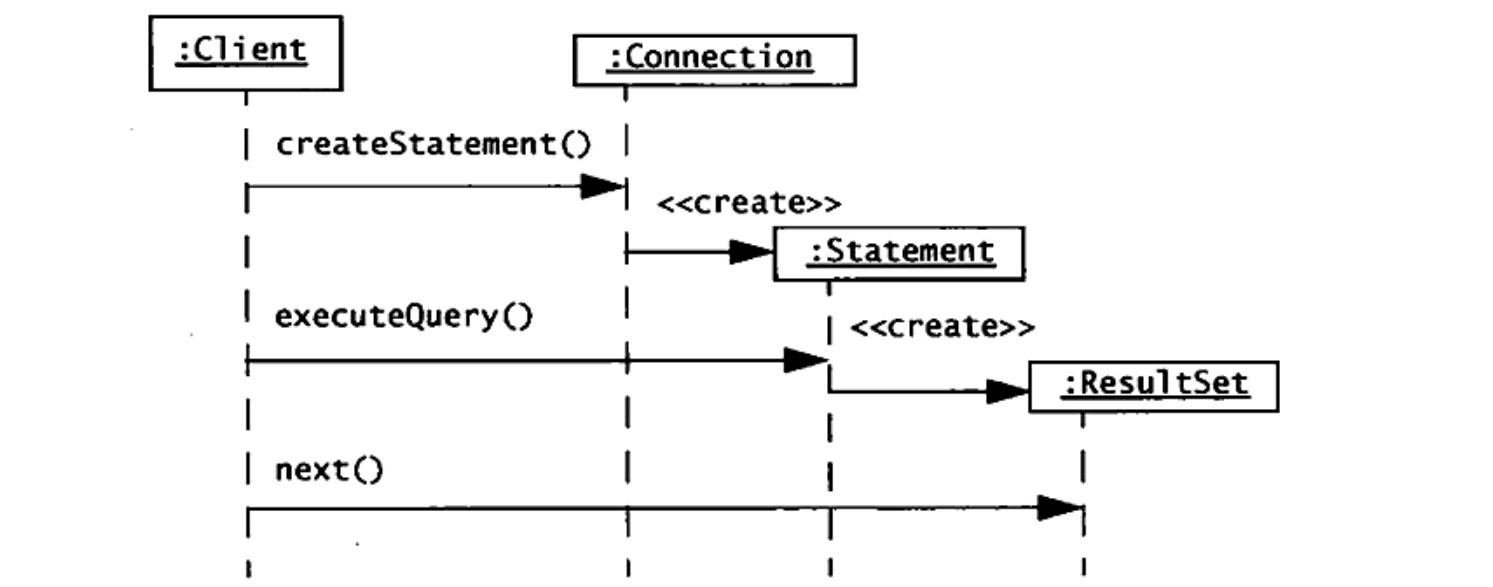


图 B.6 该图展示了在 JDBC 应用程序中常见的消息流

挑战 6.5 的答案（第 59 页）

为 SQL Server 编写专用代码有两种观点：

1. 我们不能预测未来，现在就耗费资金为将来可能永远不会发生的变化做准备，是一种常见的错误。我们现在只有 SQL Server，速度越快就意味着越短的响应时间，对目前而言就意味着节约了资金。
2. 因为仅使用 SQL Server，我们可以使用数据库提供的各种特性，而无须担心其他的数据库驱动器是否支持。

支持通用 SQL 驱动器的，也有两种观点：

1. 如果使用通用的 SQL 对象编写代码，当要改变数据库，例如使用 Oracle 时，就更容易修改。如果代码只能支持 SQL Server，就会导致无法从竞争激烈的数据库市场获利。
2. 使用通用的驱动器，使得我们可以编写一些试验性的代码，能够运行在一些便宜的数据库上，例如 MySql，而无须依赖于 SQL Server 数据库。

第 7 章 职责型模式介绍

挑战 7.1 的答案（第 62 页）

以下是该图存在的问题：

- `Rocket.thrust()`方法返回了 `Rocket` 对象，而不是数值类型或者物理数量值。
- `LiquidRocket` 类定义了 `getLocation()` 方法，但是，无论设计图还是问题域都没有说明火箭模型应该持有位置属性。即便需要该属性，也不能只为液态火箭提供该属性，而不管其他的火箭对象。
- `isLiquid()` 方法可以使用 `instanceof` 操作符来判断能否处理传入的对象，但是这样就必须在超类中定义一个返回 `false` 的 `isLiquid()` 方法。
- `CheapRockets` 的类名采用了复数形式，其他类的名称都是单数。
- `CheapRockets` 类实现了 `Runnable` 接口，但该接口与问题域中的 `CheapRocket` 对象没有任何关系。
- 可以将模型中的廉价特征作为属性建模，而无须为它单独创建一个类。
- 引入 `CheapRockets` 类使得我们无法区分火箭的燃料究竟是液态还是固态。例如，无法为廉价的液态火箭建模。
- 模型将 `Firework` 定义为 `LiquidRocket` 的子类，这就错误地表明所有焰火都属于液态火箭。
- 模型显示 `Reservation` 类与焰火类型存在直接关联，然而问题域中却不存在这样的关系。
- `Reservation` 类拥有 `city` 对象的副本，然而，事实上它可以从 `Location` 对象中获得 `city` 对象。
- `CheapRockets` 由 `Runnable` 对象组成，这有些奇怪。

挑战 7.2 的答案（第 63 页）

这一挑战的价值不在于获得正确的答案，而是锻炼你的思维，以便于设计出良好的类。比较你的定义，是否包含如下几点。

- 类的详细定义为：类是字段与方法的集合，字段持有数据值，而方法则操作这些数据值。
- 类创建了一个字段集合，即为对象定义了属性。这些属性的类型可以是其他类类型，也可以是基本数据类型，例如 `boolean` 和 `int`，还可以是接口类型。
- 类的设计者应该了解类的属性之间的关系。
- 类应该是内聚的。
- 类的名称能够反映类的意图，它既是属性的集合，又要符合类的行为。
- 类必须支持它所定义的所有行为，并实现它所继承（实现）的超类（接口）中的所有方法。（当然，是否实现超类或接口的所有方法，需依情况而定。）
- 类与其超类之间的关系必须满足继承关系。
- 类的每个方法名称都应该很好地表达方法要做的事情。

挑战 7.3 的答案（第 64 页）

有两种情况与操作调用的结果有关，要么它依赖于接收对象的状态，要么是接收对象的类。除此之外，可以使用其他人指定的名称。

方法的结果取决于对象状态的一个例子，可以参见第 6 章中 `MachineManager2` 类的 `stopMachine()` 方法。调用该方法的影响取决于 `MachineManager2` 对象使用哪个驱动程序。

当多态成为设计的一部分时，调用操作的结果部分或整体取决于接收对象的类。该原则在许多模式中都有体现，尤其是工厂方法模式、状态模式、策略模式、命令模式和解释器模式。例如，策略类的层次结构均实现了 `getRecommended()` 方法，使用不同的策略来推荐焰火。很容易预见到 `getRecommended()` 会推荐一种焰火，但并不知道接收 `getRecommended()` 调用的对象类型，只有在获知了使用哪种策略时，才会知晓。

第三种情况则是由于其他人已经定义了该名称。假设方法作为回调函数，而你已经重写了 `MouseListener` 类的 `mouseDown()` 方法。此时，就必须使用 `mouseDown()` 作为方法名，即使它的名称与意图不符。

挑战 7.4 的答案（第 65 页）

代码编译没有问题。访问权限被定义在类级别，而非对象级别。例如，一个 `Firework` 对象可以访问另外一个 `Firework` 对象的私有变量与方法。

第 8 章 单例 (Singleton) 模式

挑战 8.1 的答案（第 68 页）

为避免其他开发人员实例化你定义的类，可以创建唯一一个构造函数，并将其设置为私有的访问权限。注意，如果创建了其他非私有的构造函数，或者没有创建任何构造函数，其他对象都能够实例化该类。

挑战 8.2 的答案（第 68 页）

延迟实例化单例对象有两个原因：

1. 在静态初始化时，没有足够信息对单例对象进行初始化。例如，工厂单例就必须等待真正工厂的机器，才能建立通信通道。
2. 选择延迟初始化单例对象与获取资源有关，例如数据库连接，尤其是在一个特定会话中，它包含的应用程序并不需要该单例对象时。

挑战 8.3 的答案（第 70 页）

当两个线程几乎同时调用 `recordWipMove()` 方法时，就可能发生混乱。我们的解决方案必须解决这一问题：

```
public void recordWipMove() {  
    synchronized (classLock) {  
        wipMoves++;  
    }  
}
```

在一个线程执行递加操作时，另外的线程可以被激活并执行吗？当然是的。并非所有的机器都使用一个单独的指令对变量进行递加操作，我们无法保证编译器是在同一种情况下使用它们。一种有效策略是在多线程应用中，严格限制对单例数据的访问。对于这样的应用，可以参考 *Concurrent Programming in Java™*（由 Lea 在 2000 年编写）一书获得更多信息。

挑战 8.4 的答案（第 71 页）

OurBiggestRocket: 类名不合适。应该将类似“biggest”这样的属性表现为模型的属性，而非类名。如果开发人员必须提供这个类，可以实现为单例类。

TopSalesAssociate: 该类存在与 **OurBiggestRocket** 相同的问题。

Math: 该类是一个工具类，只有静态方法而无实例方法。它不是单例类。但请注意，它包含了一个私有构造函数。

System: 同样是一个工具类。

PrintStreem: 虽然 **System.out** 对象是一个 **PrintStream** 对象，并且承担了独一无二的职责，但它并非是 **PrintStream** 的唯一实例，所以它不是单例类。

PrintSpooler: **PrintSpoolers** 可以包含一台或多台打印机，所以很明显它不是单例类。

PrintManager: 在 Oozinoz 公司，有多台打印机，可以通过 **PrintManager** 单例对象来查询这些打印机的位置。

第 9 章 观察者 (Observer) 模式

挑战 9.1 的答案（第 75 页）

参考答案如下：

```
public JSlider slider() {
    if (slider == null) {
        slider = new JSlider();
        sliderMax = slider.getMaximum();
        sliderMin = slider.getMinimum();
        slider.addChangeListener(this);
        slider.setValue(slider.getMinimum());
    }
    return slider;
}
```

```

public void stateChanged(ChangeEvent e) {
    double val = slider.getValue();
    double tp = (val - sliderMin) / (sliderMax - sliderMin);
    burnPanel().setTPeak(tp);
    thrustPanel().setTPeak(tp);
    valueLabel().setText(Format.formatToNPlaces(tp, 2));
}

```

上述代码假定存在一个辅助类 `Format`, 用以格式化数值标签。你也许使用了诸如 ““+tp 或者 `Double.toString(tp)` 等表达式方式。(使用固定数目的小数位会使得结果看起来更逼真。)

挑战 9.2 的答案 (第 75 页)

参考答案如图 B.7 所示。为了能够注册标签对象以侦听滑动条事件, 图 B.7 的设计创建了一个实现了 `ChangeListener` 接口的 `JLabel` 子类。新的设计允许依赖于滑动条的组件注册它们感兴趣的事件, 并根据事件更新自身的状态。这种改进存在争议, 但我们还可以将设计重构为 MVC 架构。

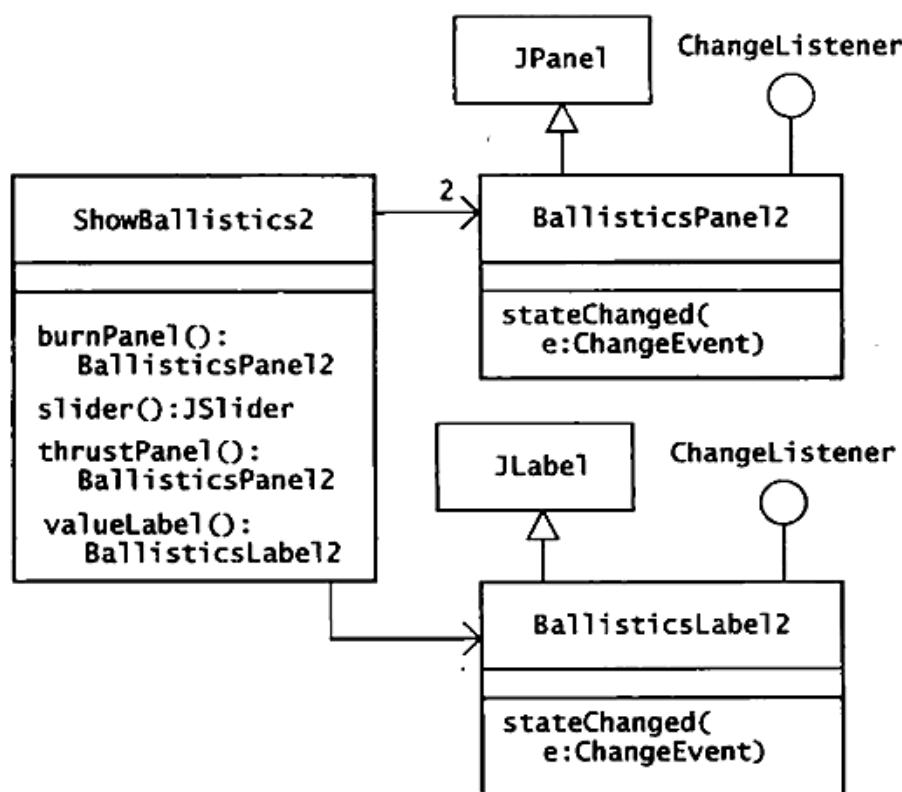


图 B.7 在本设计中, 依赖于滑动条的组件实现了 `ChangeListener`, 这样就可以注册滑动条事件

挑战 9.3 的答案（第 78 页）

图 B.8 给出了解决方案。

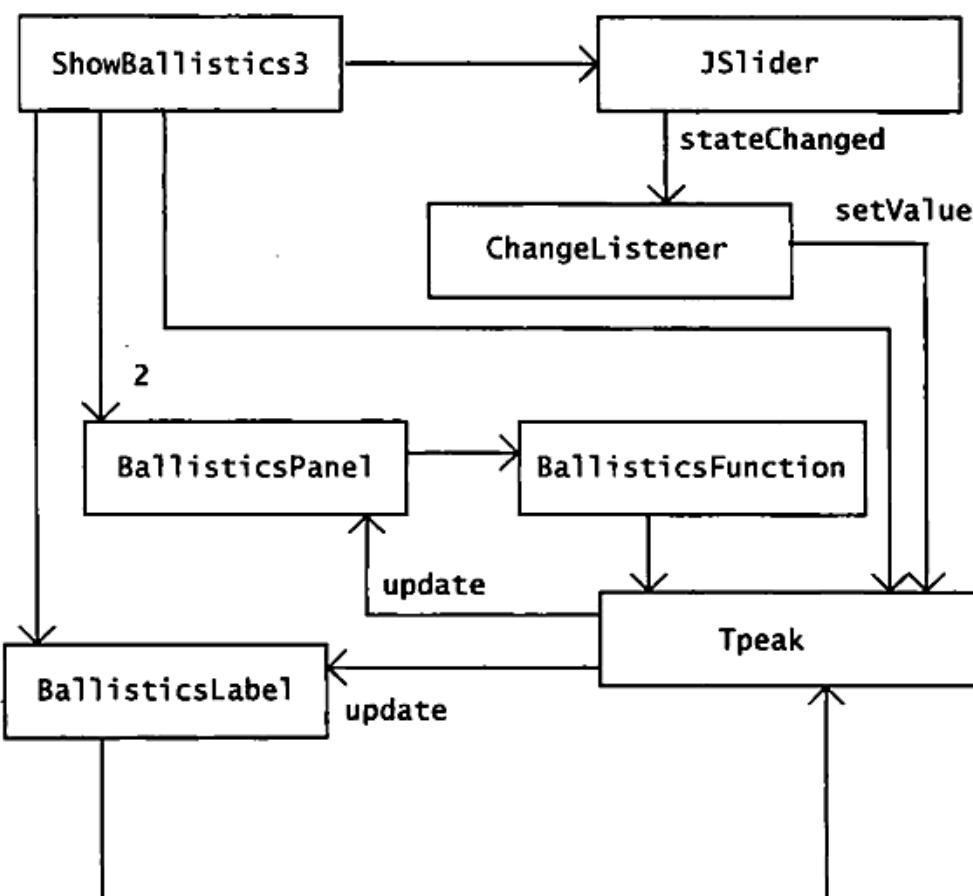


图 B.8 这一设计让应用程序观察滚动条；文本框和面板观察持有 tPeak 值的对象

持有峰值的 `Tpeak` 对象在本设计中起到了关键作用。`ShowBallistics3` 应用程序创建了 `Tpeak` 对象，只要移动了滑动条，该对象的值就会更新。显示组件（文本框与面板）被注册为 `Tpeak` 对象的观察者，实现对 `Tpeak` 对象的“侦听”。

挑战 9.4 的答案（第 80 页）

参考答案为：

```
package app.observer.ballistics3;
import javax.swing.*;
import java.util.*;

public class BallisticsLabel extends JLabel
    implements Observer {
```

```
public BallisticsLabel(Tpeak tPeak) {  
    tPeak.addObserver(this);  
}  
  
public void update(Observable o, Object arg) {  
    setText("") + ((Tpeak) o).getValue());  
    repaint();  
}  
}
```

挑战 9.5 的答案(第 80 页)

参考答案为:

```
protected JSlider slider() {  
    if (slider == null) {  
        slider = new JSlider();  
        sliderMax = slider.getMaximum();  
        sliderMin = slider.getMinimum();  
        slider.addChangeListener(new ChangeListener() {  
            public void stateChanged(ChangeEvent e) {  
                if (sliderMax == sliderMin) return;  
                tPeak.setValue(  
                    (slider.getValue() - sliderMin)  
                    / (sliderMax - sliderMin));  
            }  
        });  
        slider.setValue(slider.getMinimum());  
    }  
    return slider;  
}
```

挑战 9.6 的答案(第 81 页)

图 B.9 给出了当用户移动弹道应用程序中的滑动条时产生的调用流程。

挑战 9.7 的答案(第 83 页)

图 B.10 给出了可能的解决方案。注意，我们可以借助 `Observer` 和 `Observable` 来实现相同的设计。设计的关键在于 `Tpeak` 类，它可以通过维护一个具有侦听能力的对象，使得自己成

为可观察的对象。

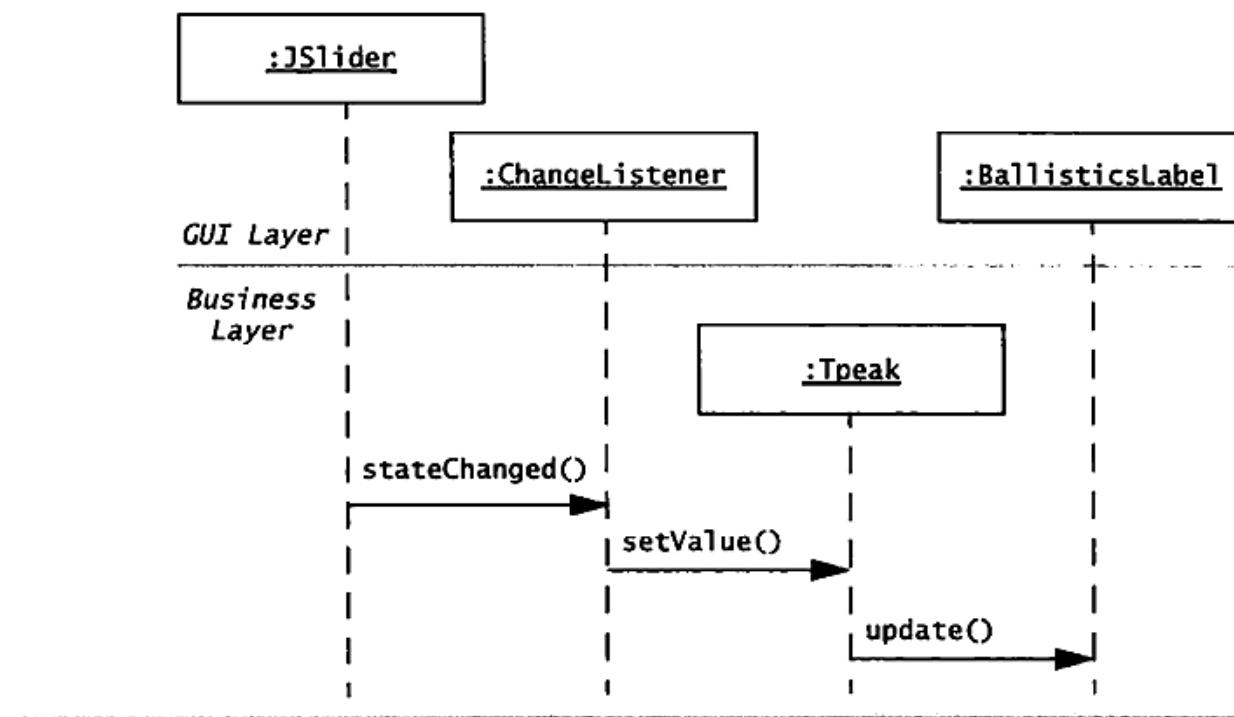


图 B.9 MVC 使得变更路径穿越了业务层

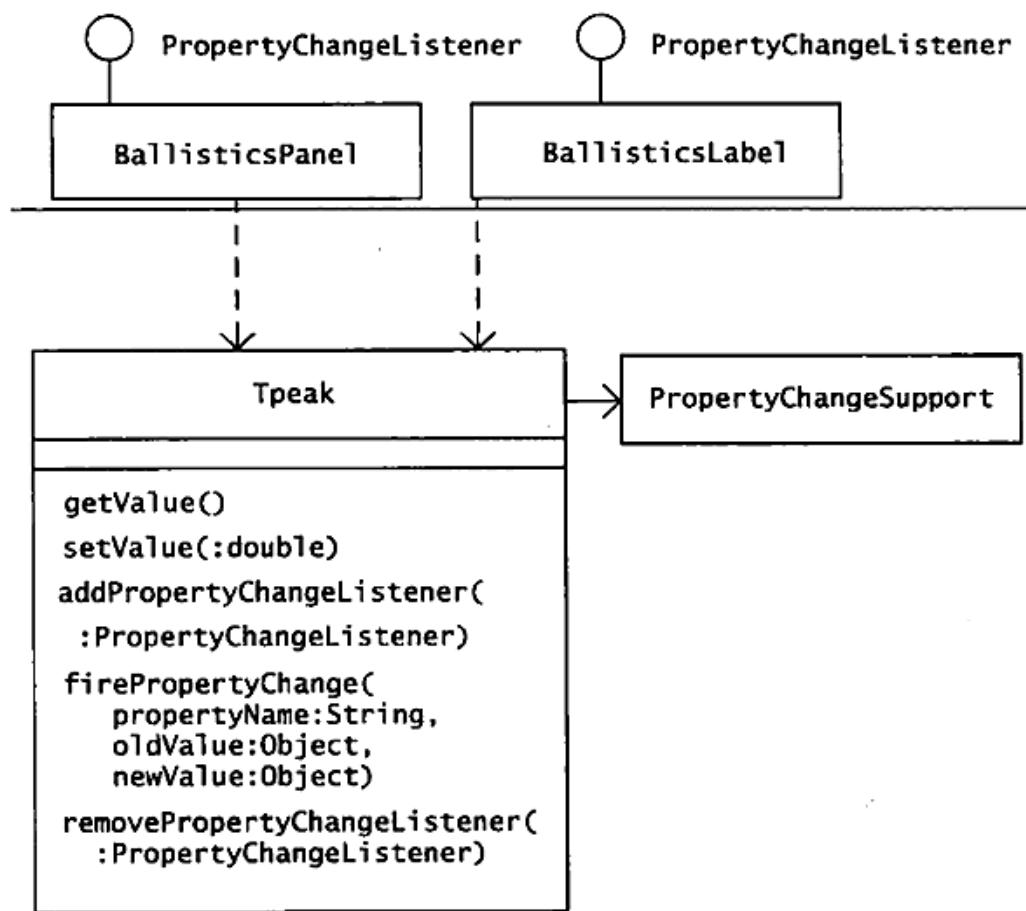
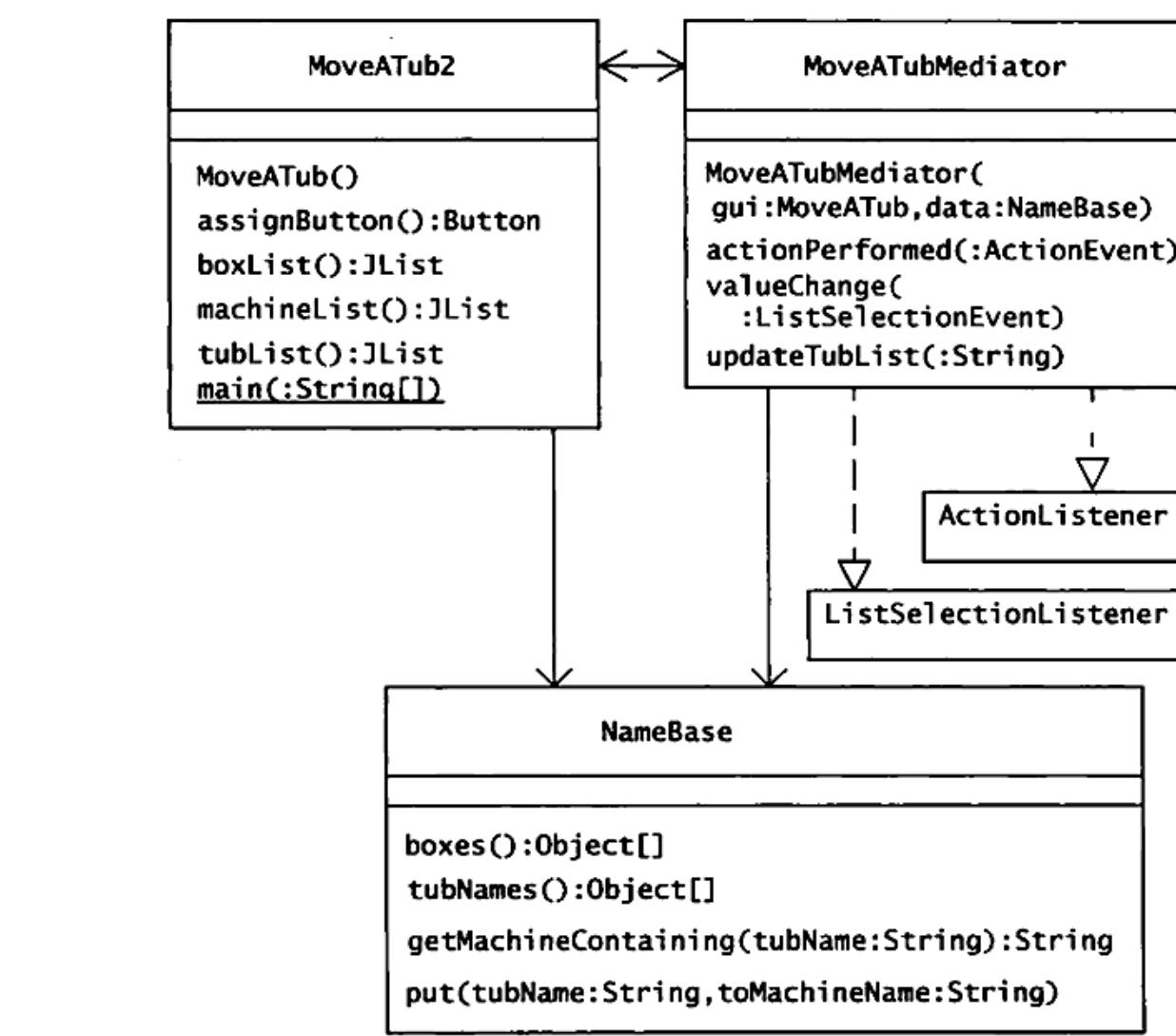


图 B.10 通过将面向侦听的调用委派给 PropertyChangeSupport 对象, Tpeak 类就能够增加侦听行为

第10章 调停者(Mediator)模式

挑战10.1的答案(第88页)

图B.11给出了一个解决方案。



图B.11 MoveATub类处理了组件的构建，MoveATubMediator类负责处理事件

在这一设计中，调停者类表现出了由Fowler等人在1999年提出的“特性依恋(feature envy)”特征。看起来，该类对GUI类的兴趣远远超过了对自身的兴趣，如valueChanged()方法所示：

```

public void valueChanged(ListSelectionEvent e) {
    // ...
  
```

```
gui.assignButton().setEnabled(  
    ! gui.tubList().isSelectionEmpty()  
    && !gui.machineList().isSelectionEmpty());  
}
```

某些开发人员并不喜欢这样的设计，但是，你可能会倾向于让一个类负责 GUI 组件的构造与布局，而将负责交互的组件以及用例流程单独分开。

挑战 10.2 的答案（第 88 页）

图 B.12 给出了一种解决方案。

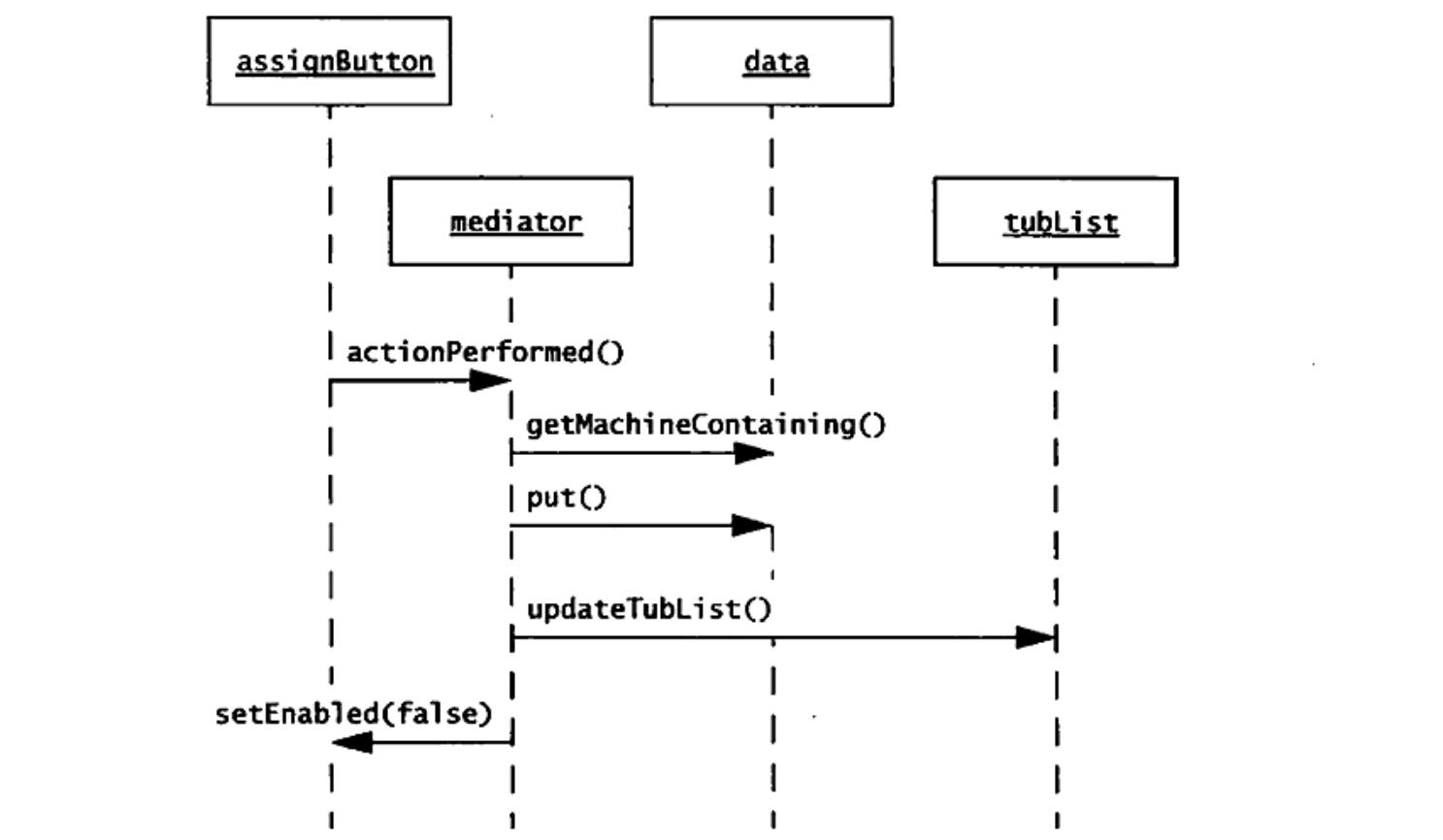


图 B.12 该图凸显了调停者的核心角色

这里给出的解决方案将调停者的角色定位为分发器，能够接收事件，并负责更新所有受到影响的对象。

挑战 10.3 的答案（第 91 页）

图 B.13 给出了被更新对象的关系图。

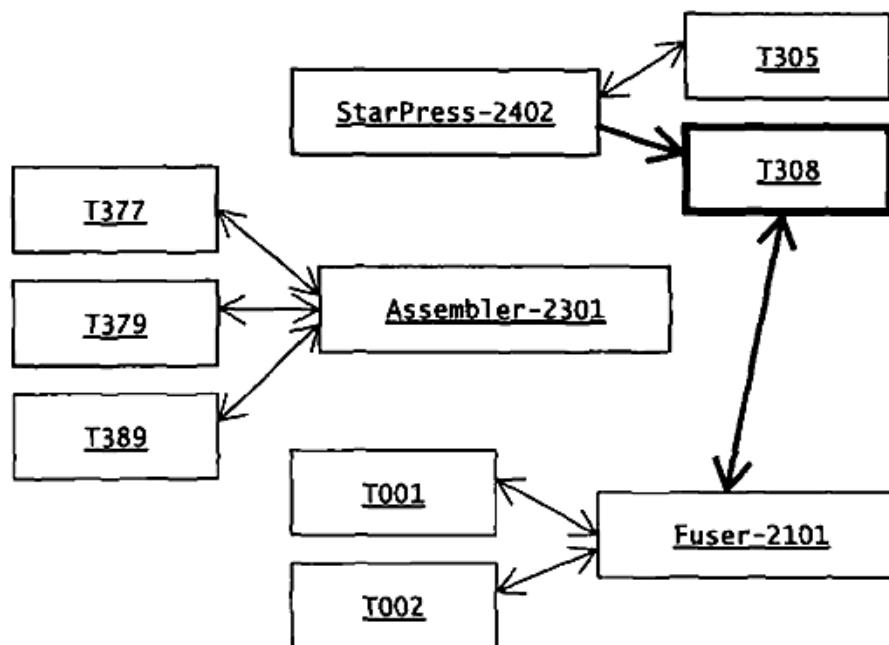


图 B.13 两台机器认为它们包含了材料箱 T308。无论是关系表还是实际应用，都不允许对象模型出现这样的情形（图中的粗线重点突出了这一问题）

开发人员的代码引入了这样的问题：StarPress-2402 仍然认为它拥有材料箱 T308。在关系表中，只要改变某一行的机器属性，就会自动地将材料箱从优先级高的机器中移除。如果关系被分散到一个分布式对象模型中，则这种自动移除工作将不会发生。材料箱/机器的关系模型需要独立的逻辑，能够转移到独立的调停者对象中。

挑战 10.4 的答案 (第 95 页)

TubMediator 类的完整代码如下所示：

```

package com.oozinoz.machine;
import java.util.*;

public class TubMediator {
    protected Map tubToMachine = new HashMap();

    public Machine getMachine(Tub t) {
        return (Machine) tubToMachine.get(t);
    }

    public Set getTubs(Machine m) {
        Set set = new HashSet();
    }
}
  
```

```
Iterator i = tubToMachine.entrySet().iterator();
while (i.hasNext()) {
    Map.Entry e = (Map.Entry) i.next();
    if (e.getValue().equals(m))
        set.add(e.getKey());
}
return set;
}

public void set(Tub t, Machine m) {
    tubToMachine.put(t, m);
}

}
```

挑战 10.5 的答案（第 95 页）

- 外观模式有助于对大型应用程序的重构。
- 桥接模式会将抽象的操作移到接口中。
- 观察者模式可以用于对 MVC 架构的支持。
- 享元模式会将对象中不变的部分分离出来，使之能够被共享。
- 构建者模式将构造对象的逻辑从类中分离出来。
- 工厂方法模式通过将类层次结构中的某方面行为转移到平行的类层次结构中，以减少类层次结构的职责范围。
- 状态模式和策略模式可以将与状态或策略有关的行为转移到单独的类中。

第 11 章 代理 (Proxy) 模式

挑战 11.1 的答案（第 101 页）

参考答案为：

```
public int getIconHeight() {
    return current.getIconHeight();
}
```

```
public int getIconwidth() {
    return current.getIconwidth();
}

public synchronized void paintIcon(
    Component c, Graphics g, int x, int y) {
    current.paintIcon(c, g, x, y);
}
```

挑战 11.2 的答案(第 101 页)

该设计存在的问题如下所示。

- 仅仅将部分调用转发给底层的 `ImageIcon` 对象是很危险的。`ImageIconProxy` 类从 `ImageIcon` 类继承了许多字段以及至少 25 个方法。要成为真正的代理，`ImageIconProxy` 对象需要转发大多数甚至所有的调用。彻底地调用转发需要许多可能存在错误的方法，随着 `ImageIcon` 类以及其超类发生变化，还需要做相应的调整。
- 你可能会质疑“缺失”的图片以及所需的图片在设计时是否真正到位。最好将图片传递过来，而不是让类负责去查找。

挑战 11.3 的答案(第 103 页)

`load()`方法将图像设置为“Loading…”，而 `run()`方法则执行在一个单独线程中，用以加载需要的图片：

```
public void load(JFrame callbackFrame) {
    this.callbackFrame = callbackFrame;
    setImage(LOADING.getImage());
    callbackFrame.repaint();
    new Thread(this).start();
}

public void run() {
    setImage(new ImageIcon(
        ClassLoader.getSystemResource(filename))
        .getImage());
    callbackFrame.pack();
}
```

挑战 11.4 的答案（第 108 页）

如类图所示，`RocketImpl` 构造函数将接收价格和最高点作为参数：

```
Rocket biggie = new RocketImpl(29.95, 820);
```

你可以将 `biggie` 声明为 `RocketImpl` 类型。然而，重要的是，客户端期望的是实现了 `Rocket` 接口的类型。

挑战 11.5 的答案（第 109 页）

完整的类图如图 B.14 所示。可以将图中 `getApogee()` 方法的接收器类型改为 `Rocket`。实际上，服务器和客户端都将这两个对象作为 `Rocket` 接口的实例进行引用。

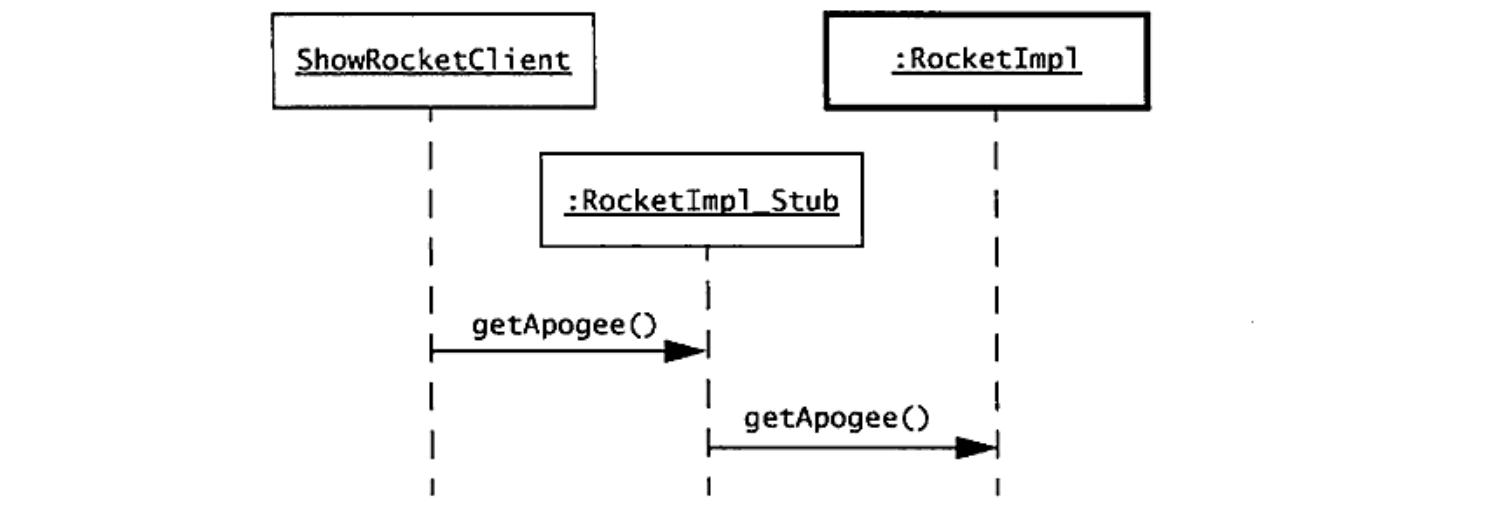


图 B.14 代理会转发客户端调用，使得远程对象看起来就像本地对象

第 12 章 职责链（Chain of Responsibility）模式

挑战 12.1 的答案（第 116 页）

在 Oozinoz 的设计中，使用了职责链去寻找负责机器的工程师，这存在一些潜在的问题。

- 我们没有指定如何设置职责链，让机器可以知道其父节点。实际上，很难确保父节点永远都不为空。

- 可以想象的是，对父对象的搜索可能会陷入到无限循环中，这取决于父对象是如何建立的。
- 并非所有对象都需要提供这些新方法所表示的行为。（例如，最顶端的元素就没有父对象。）
- 当前设计局限于细节，关心系统如何知道是哪些工程师正在工厂，并是否可用。该设计并不清楚完成责任需要的“实际时间”。

挑战 12.2 的答案（第 119 页）

你的类图应该与图 B.15 相似。

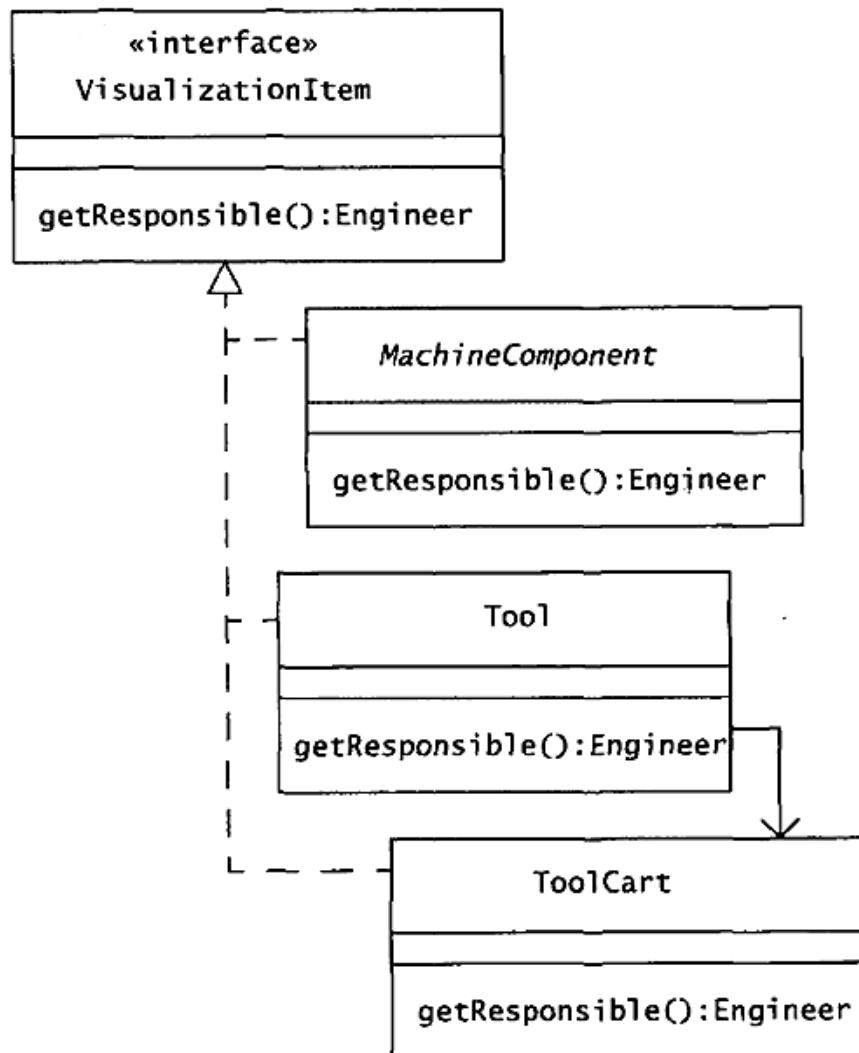


图 B.15 每个 `VisualizationItem` 对象都会告知其承担职责的工程师。从内部看，`VisualizationItem` 对象会将请求转发给其他父对象

在这个设计中，任何模拟对象的客户端都可以通过该模拟对象获知责任工程师的信息。这种方式可以让客户不必了解那个对象的职责，从而将负担转嫁给实现了 `VisualizationItem` 接口的对象。

挑战 12.3 的答案（第 119 页）

A. `MachineComponent` 对象拥有一个显式分配的责任人。如果没有，就将请求转交给它的父对象。

```
public Engineer getResponsible() {  
  
    if (responsible != null)  
        return responsible;  
    if (parent != null)  
        return parent.getResponsible();  
    return null;  
}
```

B. `Tool.Responsible` 类的代码反映了“工具总是指派给对应的工具车”原则。

```
public Engineer getResponsible() {  
    return toolCart.getResponsible();  
}
```

C. `ToolCart` 类的代码反映了“工具车都有一名责任工程师”原则。

```
public Engineer getResponsible() {  
    return responsible;  
}
```

挑战 12.4 的答案（第 120 页）

解决方案应该如图 B.16 所示。

我们提供的构造函数能够保证无论是否提供指定的工程师，都能够实例化 `Machine` 和 `MachineComposite` 对象。无论 `MachineComponent` 对象是否拥有指定的工程师，都能够从它的父对象获得责任工程师。

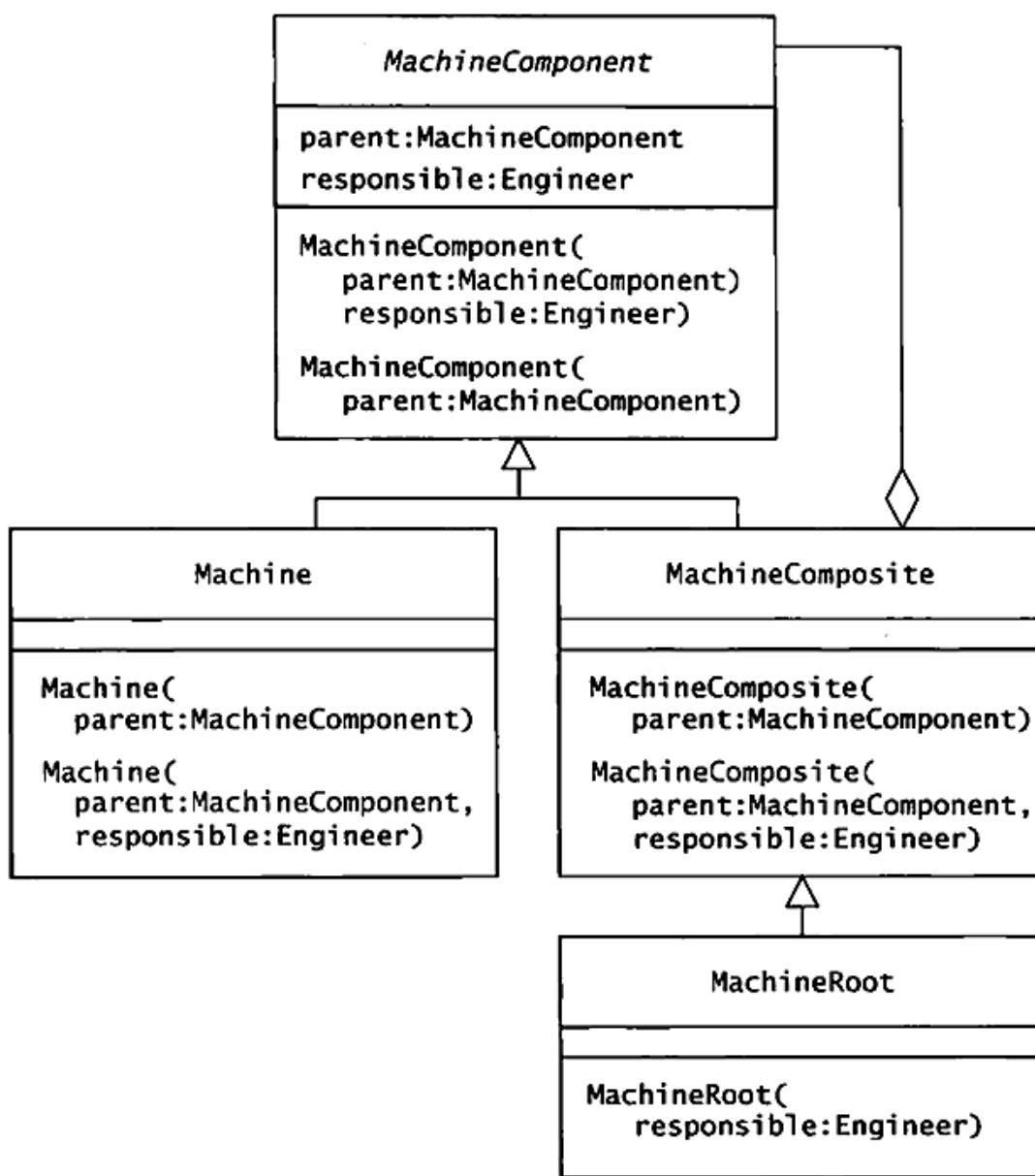


图 B.16 MachineComponent 类层次结构的构造函数必须支持两条原则：MachineRoot 对象必须拥有一个责任工程师；除了根对象的每个 MachineComponent 对象都必须具有父对象

挑战 12.5 的答案 (第 121 页)

职责链可以用于不具有组合结构的对象模型中，如：

- 一群接线工程师组成一个环状链条，他们遵守轮流提供客户服务的标准。如果当前的接线工程师在规定时间内无法回答与产品有关的问题，就通知系统进行切换，由下一个工程师提供帮助。
- 用户输入事件日期之类的信息，解析器组成一个链条对用户文本依次进行解码。

第 13 章 享元 (Flyweight) 模式

挑战 13.1 的答案（第 123 页）

支持字符串为不变类型的正方观点：在实际应用中，字符串常常被调用者共享。如果字符串能够被修改，则调用者对字符串不经意的修改就会影响到其他人。例如，当方法返回字符串类型的客户名时，它仍然持有对该名称值的引用。如果调用者将字符串转换为大写并放入哈希表中，客户名就会发生变化。在 Java 语言中，如果为字符串生成一个大写版本，就必须是一个新对象，而不能修改原来字符串的值。字符串的不变性使得它们在被多个调用者共享时，是安全的。而且，不变的字符串有助于避免系统出现安全风险。

反方观点：字符串的不变性虽然能够避免错误，但却因此付出了较大的代价。首先，即便确定需要修改字符串值，也无法修改。其次，在计算机语言中加入特殊规则，就使得语言变得难以学习和使用。Java 语言与 Smalltalk 语言一样强大，但前者学习起来更难。最后，任何计算机语言都无法避免使用者犯错。如果学习语言更加容易，就可以有更多时间来了解如何创建以及学习测试框架。

挑战 13.2 的答案（第 124 页）

可以将 Substance 中不变的部分，包括名称、符号和原子量转移到 Chemical 类中，如图 B.17 所示。

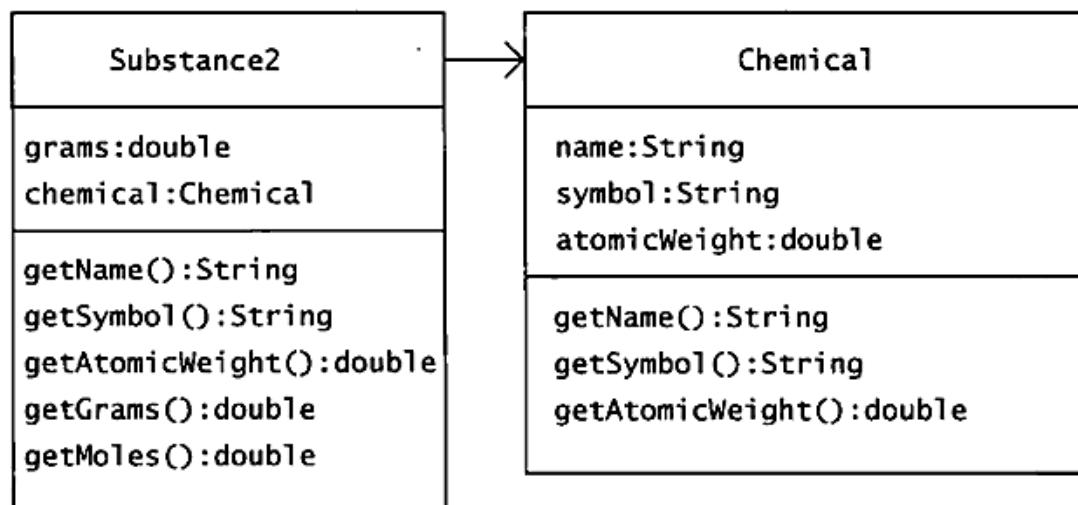


图 B.17 该类图体现了将原来属于 Substance 类中不变的部分提取到单独的类 Chemical 中

现在，类 Substance2 保持了对 Chemical 对象的引用。因此，类 Substance2 能够提供与之前 Substance 类相同的访问器。从内部来讲，这些访问器依赖于 Chemical 类，如下面的 Substance2 方法所示：

```
public double getAtomicweight() {  
    return chemical.getAtomicweight();  
}  
  
public double getGrams() {  
    return grams;  
}  
  
public double getMoles() {  
    return grams / getAtomicWeight();  
}
```

挑战 13.3 的答案（第 126 页）

一种方法是将 Chemical 的构造函数定义为私有的。这就可以防止 ChemicalFactory 类实例化 Chemical 类对象。

若要防止开发人员实例化 Chemical 类自身，可以将 Chemical 和 ChemicalFactory 类放在同一个包中，然后将 Chemical 类的构造函数设定为默认的访问限制（即包级别的访问限制）。

挑战 13.4 的答案（第 127 页）

使用嵌套类的方式过于复杂，但能够彻底确保只有 ChemicalFactory2 类才能实例化新的享元对象。最终的代码如下所示：

```
package com.oozinoz.chemical2;  
import java.util.*;  
  
public class ChemicalFactory2 {  
    private static Map chemicals = new HashMap();  
  
    class ChemicalImpl implements Chemical {  
        private String name;  
        private String symbol;  
        private double atomicweight;
```

```
ChemicalImpl(  
    String name,  
    String symbol,  
    double atomicWeight) {  
    this.name = name;  
    this.symbol = symbol;  
    this.atomicWeight = atomicWeight;  
}  
  
public String getName() {  
    return name;  
}  
  
public String getSymbol() {  
    return symbol;  
}  
  
public double getAtomicWeight() {  
    return atomicWeight;  
}  
  
public String toString() {  
    return name + "(" + symbol + ")" + "  
        atomicWeight + "]";  
}  
}  
  
static {  
    ChemicalFactory2 factory = new ChemicalFactory2();  
    chemicals.put("carbon",  
        factory.newChemicalImpl("Carbon", "C", 12));  
    chemicals.put("sulfur",  
        factory.newChemicalImpl("Sulfur", "S", 32));  
    chemicals.put("salt peter",  
        factory.newChemicalImpl(  
            "Salt peter", "KN03", 101));  
    //...  
}  
  
public static Chemical getChemical(String name) {
```

```
    return (Chemical) chemicals.get(
        name.toLowerCase());
    }
}
```

这段代码解决了如下三个问题。

1. `ChemicalImpl` 嵌套类应该是私有的，这样就只有 `ChemicalFactory2` 类才能使用该类。注意，嵌套类的访问范围必须是包级别或公有的，这样包含的类才可以实例化嵌套类。即使将构造函数定义为公有的，如果嵌套类本身被标记为私有，就没有其他类可以调用该构造函数。
2. `ChemicalFactory2` 的构造函数使用了一个静态的初始化器，以确保类只会构建一次化学药品列表。
3. `getChemical()` 方法可以在类的哈希表中根据名称查找化学药品。示例代码使用了小写的化学品名来存储与查询化学品。

第 14 章 构造型模式介绍

挑战 14.1 的答案（第 130 页）

与构造函数有关的特定规则包括如下几项：

- 如果不为类提供一个构造函数，Java 会提供一个默认的。
- 构造函数的名称必须与类名相匹配。（这就是说，构造函数的名称通常都以大写字母开始，而与方法名称的规定不同。）
- 构造函数可以通过 `this()` 和 `super()` 方法调用其他构造函数，只要该调用是放在构造函数的第一句。
- 构造函数的结果是该类的实例，而常规方法的返回值可以是任何类型。
- 可以使用 `new` 关键字或反射来调用构造函数。

挑战 14.2 的答案（第 131 页）

当把如下代码放到 `Fuse.java` 和 `QuickFuse.java` 中时，编译会失败：

```
package app.construction;
public class Fuse {
    private String name;
    public Fuse(String name) { this.name = name; }
}
```

和

```
package app.construction;
public class QuickFuse extends Fuse { }
```

编译器报告的错误大约如下所示：

```
Implicit super constructor Fuse() is undefined for default
constructor. Must define an explicit constructor.
```

当编译器遇到 QuickFuse 类时，就会发生错误，并为其提供一个默认的构造函数。默认的构造函数没有参数，并在默认情况下会调用其超类的无参构造函数。然而，Fuse() 构造函数却接受了一个字符串参数，这意味着编译器不再为 Fuse 类提供默认构造函数。QuickFuse 的默认构造函数无法调用超类的无参构造函数，因为该构造函数并不存在。

挑战 14.3 的答案（第 132 页）

程序会输出：

```
java.awt.Point[x=3,y=4]
```

（它成功地找到了接收两个参数的构造函数，并根据给定的参数值创建了一个新的 point 对象。）

第 15 章 构建者（Builder）模式

挑战 15.1 的答案（第 137 页）

要使解析器更加灵活，可以允许它接收逗号后的多个空格。要做到这一点，需要修改 split() 的调用格式：

```
s.split(", *");
```

或者，如果不接收逗号后的空格，也可以通过初始化如下的 Regex 对象，使它能够接收各种类型的空格：

```
s.split(",\\s*")
```

\s 字符表示为正则表达式中的“字符类”。注意，所有这些解决方案都假定字段中没有包含逗号。

为了让正则表达式更加灵活，或许你会质疑整个方法。特别的，你可能希望旅游局能够发送 XML 格式的预订信息。这就可以建立一系列的标签，在 XML 解析器中使用和读取这些标签。

挑战 15.2 的答案（第 139 页）

如果 `UnforgivingBuilder` 类的任何一个属性都是无效的，`build()` 方法就会抛出一个异常，否则返回一个有效的 `Reservation` 对象。如下是其中一种实现：

```
public Reservation build() throws BuilderException {
    if (date == null)
        throw new BuilderException("Valid date not found");

    if (city == null)
        throw new BuilderException("Valid city not found");

    if (headcount < MINHEAD)
        throw new BuilderException(
            "Minimum headcount is " + MINHEAD);

    if (dollarsPerHead.times(headcount)
        .isLessThan(MINTOTAL))
        throw new BuilderException(
            "Minimum total cost is " + MINTOTAL);

    return new Reservation(
        date,
        headcount,
        city,
        dollarsPerHead,
        hasSite);
}
```

代码会检查日期与城市值是否设置，并检查总人数以及人均费用的值是否合理。`ReservationBuilder` 超类定义了 `MINHEAD` 和 `MINTOTAL` 常量。

如果构造函数没有问题，就会返回一个有效的 `Reservation` 对象。

挑战 15.3 的答案（第 139 页）

如前所示，如果预订信息没有指定城市或日期，就会抛出异常，这是因为它无法预测这些值。至于缺失总人数与人均费用的值，需要注意如下几点：

- 如果预订请求没有指明总人数和人均费用，就将总人数设置为最小值，而将人均费用设置为最小金额除以总人数。
- 如果没有设置总人数，但却设置了人均费用，则将总人数设置为最小值，并能够保证总费用足以支持此次活动。
- 如果设置了总人数但却没有人均费用，就将人均费用的值设置为能够保证维持活动费用的最大值。

挑战 15.4 的答案（第 140 页）

参考答案如下所示：

```
public Reservation build() throws BuilderException {
    boolean noHeadcount = (headcount == 0);
    boolean noDollarsPerHead = (dollarsPerHead.isZero());

    if (noHeadcount&&noDollarsPerHead) {
        headcount = MINHEAD;
        dollarsPerHead = sufficientDollars(headcount);
    } else if (noHeadcount) {
        headcount = (int) Math.ceil(
            MINTOTAL.dividedBy(dollarsPerHead));
        headcount = Math.max(headcount, MINHEAD);
    } else if (noDollarsPerHead) {
        dollarsPerHead = sufficientDollars(headcount);
    }

    check();
}
```

```
    return new Reservation(
        date,
        headcount,
        city,
        dollarsPerHead,
        hasSite);
}
```

这段代码依赖于 `check()` 方法，它与 `UnforgivingBuilder` 类的 `build()` 方法相似：

```
protected void check() throws BuilderException {
    if (date == null)
        throw new BuilderException("Valid date not found");

    if (city == null)
        throw new BuilderException("Valid city not found");

    if (headcount < MINHEAD)
        throw new BuilderException(
            "Minimum headcount is " + MINHEAD);

    if (dollarsPerHead.times(headcount)
        .isLessThan(MINTOTAL))
        throw new BuilderException(
            "Minimum total cost is " + MINTOTAL);
}
```

第 16 章 工厂方法 (Factory Method) 模式

挑战 16.1 的答案 (第 142 页)

一个好的答案或许是你不必关心 `iterator()` 方法的返回值。重要的是，你必须知道迭代器支持什么样的接口，才能遍历集合中的元素。然而，如果你需要知道当前迭代的元素属于哪个类，可以用下面这行代码输出类名：

```
System.out.println(iter.getClass().getName());
```

结果如下：

```
java.util.AbstractList$Itr
```

Itr 类是 AbstractList 类的内部类。在使用 Java 进行编程的时候，是不会看到这个类的。

挑战 16.2 的答案（第 142 页）

有很多可能的答案，但是 `toString()` 方法可能是创建新对象时最常用的方法。例如，下面的代码创建了一个新的 `String` 对象：

```
String s = new Date().toString();
```

很多时候，`toString()` 的过程都是隐式完成的。例如：

```
System.out.println(new Date());
```

这行代码通过调用 `Date` 对象的 `toString()` 方法，创建了该对象的一个 `String` 对象。

另一种常用的创建新对象的方法是 `clone()`，该方法用于返回调用对象的一个副本。

挑战 16.3 的答案（第 143 页）

工厂方法模式的意图是让对象的提供者来决定创建哪个类。比较而言，`BorderFactory` 类的使用者通常都知道他们要创建的类型。`BorderFactory` 类使用了享元模式来有效地支持大量边界的共享使用。`BorderFactory` 类无须使用者管理对象的复用，而工厂方法模式使用户无须了解实例化哪个类。

挑战 16.4 的答案（第 144 页）

图 B.18 展示了 `CreditCheck` 接口的两个赊购审查类。工厂类提供了一个方法，用来返回 `CreditCheck` 对象。客户类在调用 `createCreditCheck()` 方法时，并不知道会实例化哪个类。

`createCreditCheck()` 方法是一个静态方法，因此客户类不需要为了获得一个 `CreditCheck` 对象而去实例化 `CreditCheckFactory` 类。如果不想让开发人员实例化该类，可以将该类定义成抽象类，或者为该类定义一个静态构造函数。

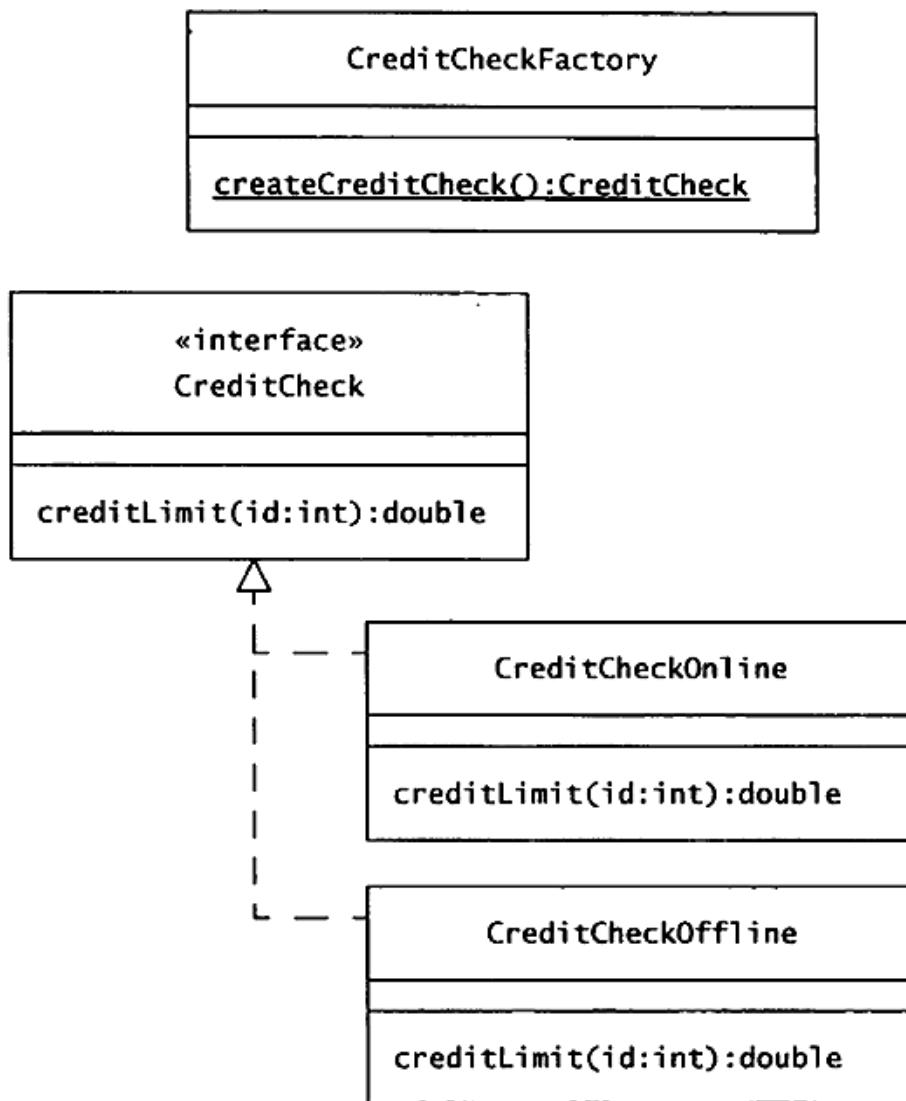


图 B.18 两个类实现了 CreditCheck 接口。实例化哪个类取决于服务提供者，而不需要客户类决定

挑战 16.5 的答案（第 145 页）

如果假定静态方法 `isAgencyUp()` 可以反映实际情况，`createCreditCheck()` 代码就可以简化成这样：

```

public static CreditCheck createCreditCheck() {
    if (isAgencyUp()) return new CreditCheckOnline();
    return new CreditCheckOffline();
}
  
```

挑战 16.6 的答案（第 146 页）

图 B.19 展示了一张合理的 Machine/MachinePlanner 并行结构图。

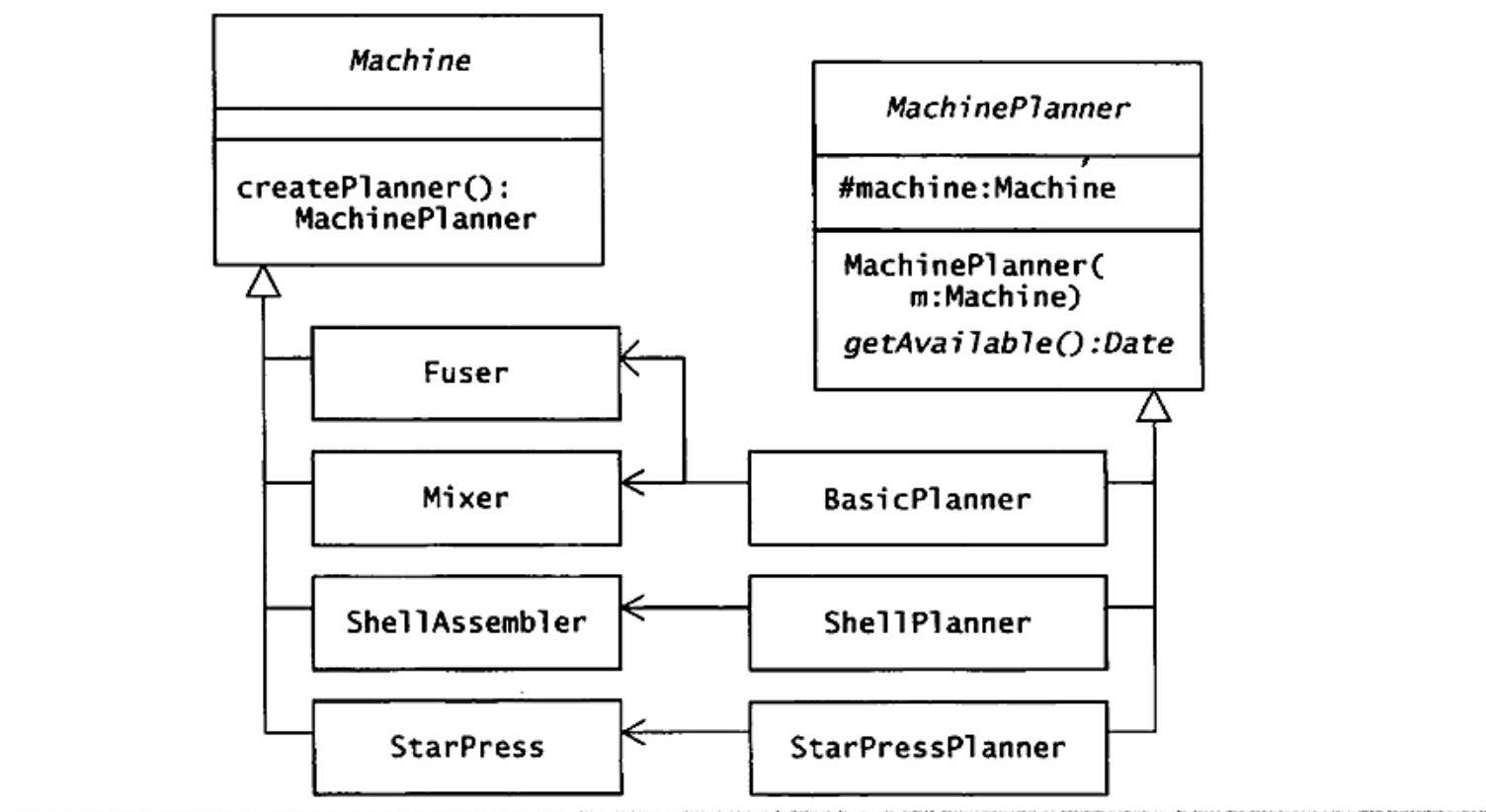


图 B.19 现在的逻辑规划被放在了一个独立的层次结构中。Machine 类的每个子类的 createPlanner() 方法都知道应该实例化哪个计划类

该图说明了 MachinePlanner 类的子类必须实现 getAvailable() 方法，还说明了 MachinePlanner 层次结构中的所有类都必须有一个接收 Machine 对象的构造函数。这样就可以让计划类获得对象的计划信息，比如机器的位置、当前处理的材料数量等。

挑战 16.7 的答案（第 147 页）

Machine 类的 createPlanner() 方法看起来如下所示：

```

public MachinePlanner createPlanner() {
    return new BasicPlanner(this);
}

```

无论 ShellAssembler 类和 StarPress 类是否需要重写该方法，Fuser 类和 Mixer 类都只能继承自该方法。对于 StarPress 类，createPlanner() 方法可以如下所示：

```

public MachinePlanner createPlanner() {
    return new StarPressPlanner(this);
}

```

这些方法用到了工厂方法模式。当我们需要一个计划对象时，就会调用机器的

`createPlanner()`方法。该方法返回什么样的计划对象，取决于具体的机器。

第 17 章 抽象工厂 (Abstract Factory) 模式

挑战 17.1 的答案 (第 151 页)

一种解决方案如下所示

```
public class BetaUI extends UI {
    public BetaUI () {
        Font oldFont = getFont();
        font = new Font(
            oldFont.getName(),
            oldFont.getStyle() | Font.ITALIC,
            oldFont.getSize());
    }

    public JButton createButtonOk() {
        JButton b = super.createButtonOk();
        b.setIcon(getIcon("images/cherry-large.gif"));
        return b;
    }

    public JButton createButtonCancel() {
        JButton b = super.createButtonCancel();
        b.setIcon(getIcon("images/cherry-large-down.gif"));
        return b;
    }
}
```

这段代码尽可能多地使用了超类提供的方法。

挑战 17.2 的答案 (第 152 页)

有一种更为灵活的设计，在接口中指定期待创建的方法和 GUI 属性，如图 B.20 所示。

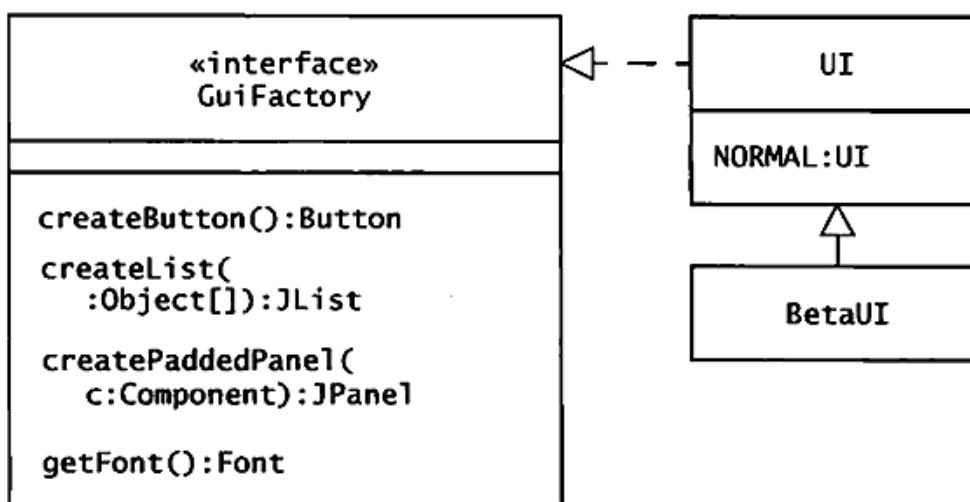


图 B.20 该图使用抽象工厂设计 GUI 控件，在 UI 类中减少了子类关于方法访问修饰符的依赖

挑战 17.3 的答案（第 155 页）

图 B.21 展示了一种解决方案，用来在 `Credit.Credit.ca` 中提供具体类以实现 `Credit` 的接口和抽象类。

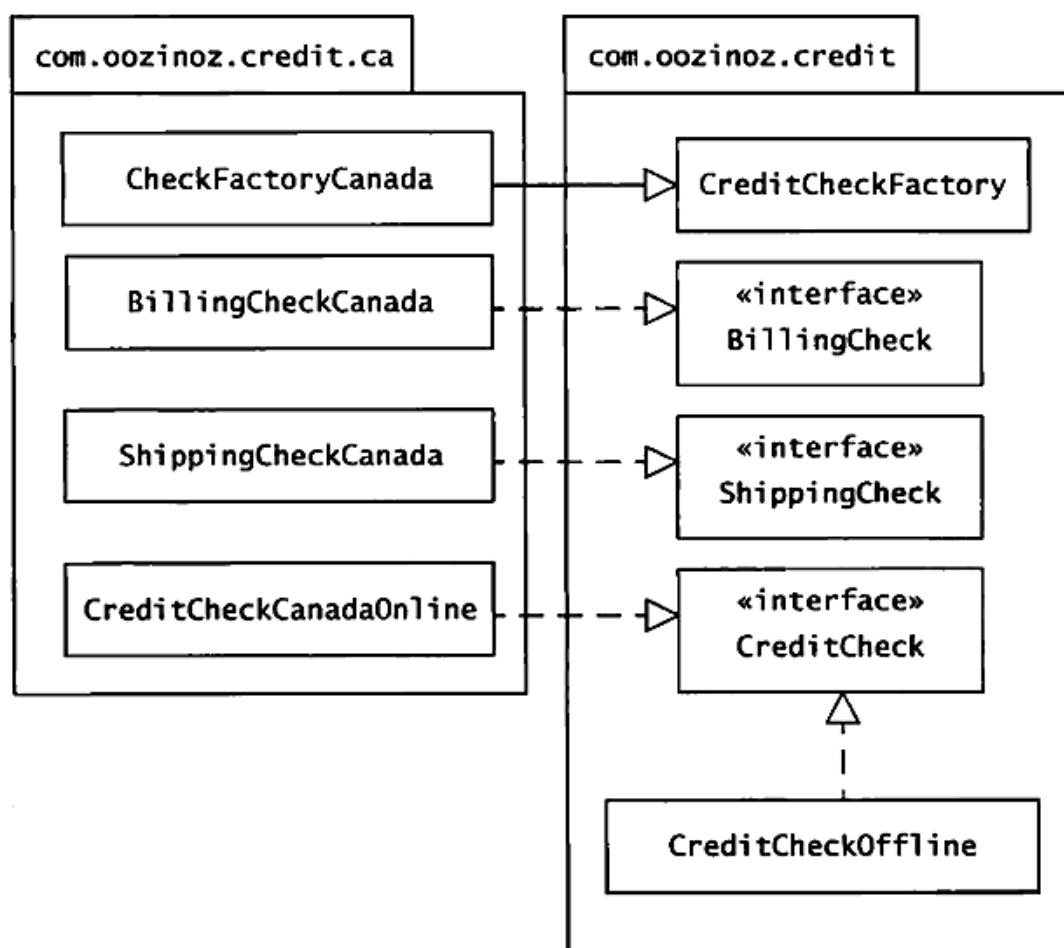


图 B.21 `com.oozinoz.credit.ca` 包提供了一组具体类，用来对加拿大来电进行各种审查

有一点需要注意的是，你只需一个具体类来进行离线电话审查。因为在 Oozinoz 公司，对来自美国和加拿大的离线电话审查是一致的。

挑战 17.4 的答案（第 156 页）

解决方案如下所示：

```
package com.oozinoz.credit.ca;
import com.oozinoz.check.*;

public class CheckFactoryCanada extends CheckFactory {
    public BillingCheck createBillingCheck() {
        return new BillingCheckCanada();
    }

    public CreditCheck createCreditCheck() {
        if (isAgencyUp())
            return new CreditCheckCanadaOnline();
        return new CreditCheckoffline();
    }

    public ShippingCheck createShippingCheck() {
        return new ShippingCheckCanada();
    }
}
```

你的方案应该满足下列要求：

- 实现继承自抽象类 `CreditCheckFactory` 的所有 `create-` 方法。
- 拥有每个 `create-` 方法返回值的合适类型。
- 如果代理离线，返回一个 `CreditCheckoffline` 对象。

挑战 17.5 的答案（第 157 页）

当前做法的好处是：将各个国家相关的类放在独立的包中，可以帮助 Oozinoz 公司的开发者来管理软件和开发成本。还可以将该包和其他包一样独立管理。我们可以很自信地说，美国相关的类不会影响到加拿大相关的类。我们也可以很容易地为新增的国家提供支持。例如，如果要开展墨西哥的业务，可以创建一个新包，来提供针对该国家所需要的审查服务。

这样做还有一个好处，就是可以将 `credit.mx` 包分配给一个有墨西哥数据服务经验的开发者来做。

当前做法的不妥之处在于：尽管理论上这种分离是很好的，但是实际上很难实施。我更加愿意把所有的类打包到一个包，直到出现 9 个或者更多的国家需要服务。当需要对包内容做出调整时，这种做法需要对每个包都进行调整，这增加了对这些包的管理任务。

第 18 章 原型（Prototype）模式

挑战 18.1 的答案（第 159 页）

这种设计的优点包括如下方面：

- 可以不创建新的类而直接创建工厂，甚至可以在运行时创建一个新的 GUI 工具箱。
- 可以通过复制一个旧的工厂，并进行轻微的调整来创建一个新的工厂。例如，可以创建一个与旧的 GUI 工具集几乎一致的新 GUI 工具集，而只存在字体上的差别。原型模式可以让新工具集从旧的工具集集成而来，比如颜色等。

缺点包括如下方面：

- 原型模式可以让我们更新属性值，比如颜色或者字体。但是却不允许为每个工厂创建不同的行为。
- 防止 UI 工具集类增长的动机并不明确，为什么这种增长是一个问题？我们必须将工具集初始化软件放在某些位置，有可能是 UI 工具类的静态方法。这种方式并没有真正减少我们需要管理的代码。

正确的答案是什么呢？类似的情况下，动手试验是很有帮助的：写出这两种设计的代码，然后再来评估这两种设计。同事们经常会在采用哪种方案上产生分歧，这是好事，实践后得出的结论更有说服力。（如果最后还是没能达成一致，你可能要引入架构师或者第三方来一起讨论。）

挑战 18.2 的答案（第 160 页）

可以这样概括 `clone()` 方法：全新的对象，但是字段相同。`clone()` 方法用原始类的类型

和字段来创建一个新的对象。新对象的字段值也都和原始类的字段值一致。如果这些字段是基本类型，比如整型，则进行值复制。但如果字段是引用类型，则进行引用复制。

`clone()`方法进行的是“浅复制”，它会在复制对象与原始对象之间共享子对象。“深复制”会包含对父对象属性的完整复制。例如，你克隆一个指向 B 对象的 A 对象，浅复制会创建一个新的 A' 对象指向原始的 B 对象，而深复制会创建一个新的 A' 对象指向新的 B' 对象。如果你想要实现一个深复制，需要自己实现复制方法。

注意，在使用 `clone()` 方法时，必须声明你的类实现 `Cloneable` 接口。这个标记接口没有方法，但是用来说明你有意支持 `clone()` 方法。

挑战 18.3 的答案（第 160 页）

这段代码将会产生三个对象，如图 B.22 所示。

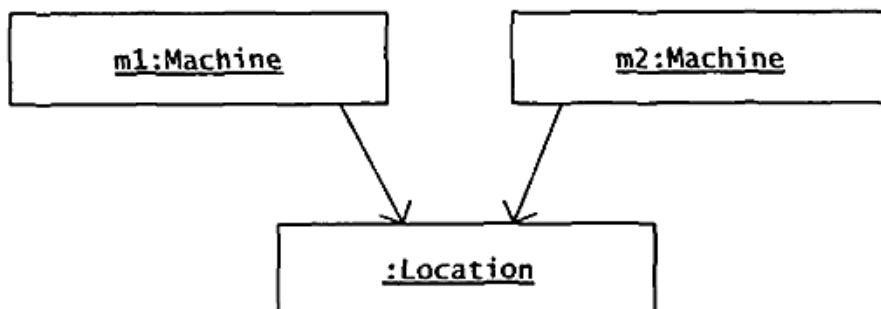


图 B.22 设计不足的克隆会创建一个不完整的复制，还是会和原始对象共享一些对象

`Machinesimulator` 类当前版本的 `clone()` 方法调用了 `super.clone()` 方法，该方法是 `Object` 实现的。这个方法用相同的属性创建了一个新的对象。基本类型，比如 `int`，都是直接复制。而对象类型，比如 `Machinesimulator` 类的 `Location` 属性，也是直接复制。注意是引用复制，而不是对象复制。`Object.clone()` 方法产生的效果如图 B.22 所示。

假设你改变了第二个机器的位置坐标和车间，因为只有一个 `Location` 对象，因此此类更改将会对第一个机器的位置造成影响。

挑战 18.4 的答案（第 162 页）

一个合理的方法如下所示：

```
public OzPanel copy2() {  
    OzPanel result = new OzPanel();
```

```
    result.setBackground(this.getBackground());
    result.setForeground(this.getForeground());
    result.setFont(this.getFont());
    return result;
}
```

`copy()`方法和`copy2()`方法都使得`OzPanel`的客户类不需要调用构造函数，并且支持原型模式。然而，手工的`copy2()`方法可能更加安全。这个方法知道需要复制哪些重要的属性，而避免了复制不清楚的属性。

第 19 章 备忘录 (Memento) 模式

挑战 19.1 的答案 (第 167 页)

下面是一个`FactoryModel`类的`undo()`方法的实现：

```
public boolean canUndo() {
    return mementos.size() > 1;
}
public void undo() {
    if (!canUndo()) return;
    mementos.pop();
    notifyListeners();
}
```

这段代码会在栈恢复到只剩一个备忘录的初始状态时，忽略`undo()`请求。栈顶始终是当前状态，因此`undo()`方法仅仅是从栈顶弹出之前的备忘录。

定义`createMemento()`方法时，必须告诉自己或者同事要求该方法返回重建对象所需全部信息。在这个例子中，机器模拟器可以在克隆过程中重建自己，而工厂模拟器可以在这些机器模拟器的克隆中重建自己。

挑战 19.2 的答案 (第 170 页)

一个方案如下：

```
public void stateChanged(ChangeEvent e) {
```

```
machinePanel().removeAll();

List locations = factoryModel.getLocations();

for (int i = 0; i < locations.size(); i++) {
    Point p = (Point) locations.get(i);
    machinePanel().add(createPictureBox(p));
}

undoButton().setEnabled(factoryModel.canUndo());
repaint();
}
```

每次状态改变时，这段代码都会从头重建 `machinePanel()` 中的机器列表。

挑战 19.3 的答案（第 170 页）

把备忘录存成一个对象的假设是，当用户想恢复原始对象时，应用程序始终是在运行的。而必须将备忘录进行持久化存储的理由如下：

- 在系统崩溃后，依然有能力恢复对象的状态。
- 程序允许用户重新进入系统后恢复之前尚未完成的任务。
- 你需要在另一台机器上重建某个对象。

挑战 19.4 的答案（第 173 页）

一个方案如下：

```
public void restore(Component source) throws Exception {
    JFileChooser dialog = new JFileChooser();
    dialog.showOpenDialog(source);

    if (dialog.getSelectedFile() == null)
        return;

    FileInputStream out = null;
    ObjectInputStream s = null;
    try {
        out = new FileInputStream(dialog.getSelectedFile());
        s = new ObjectInputStream(out);
        ArrayList list = (ArrayList) s.readObject();
    }
}
```

```
    factoryModel.setLocations(list);
} finally {
    if (s != null)
        s.close();
}
}
```

这段代码几乎就是 `save()` 方法的一个镜像方法，尽管 `restore()` 方法要求将工厂模型放入被恢复的位置列表中。

挑战 19.5 的答案（第 173 页）

封装是为了限制对对象状态以及方法的访问权限。拿工厂位置坐标集合来说，如果用文本模式来暴露对象的状态，就会造成用户可以随意修改对象状态。因此用 XML 形式来存储对象在某种程度上来说是有悖封装原则的。

在持久化存储方面违反封装原则所造成的问题，可能还需要进一步验证。而这通常取决于应用本身。为了解决这个问题，一般都会限制对对象的访问，这种方式在关系型数据库中很常见。而在某些例子中，你可能会对数据进行加密，比如传输敏感的 HTML 文本。这个问题不仅在于设计是否使用了封装或者备忘录模式，关键在于是否保证了数据的完整性。

第 20 章 操作型模式介绍

挑战 20.1 的答案（第 177 页）

职责链模式的原理是将某个操作分发在对象链中完成。对象链中的每个方法或者直接实现该操作，或者将调用转移给链中的下一个对象。

挑战 20.2 的答案（第 177 页）

下面给出了一个完整的 Java 方法修饰符的定义：

- **public**: 允许所有客户类访问。
- **protected**: 允许该类在同一个包中的子类访问。
- **private**: 只允许在该类的内部访问。

- **abstract**: 不提供实现。
- **static**: 与整个类相关，与单独的对象无关。
- **final**: 不允许被重写。
- **synchronized**: 方法会获得对对象监控器的访问权，如果方法是静态的，则将获得对类的访问权。
- **native**: 实现与平台依赖相关的代码。
- **strictfp**: 严格按照 FP 规则计算的 `double` 和 `float` 表达式，因此要求中间结果按照 IEEE 标准是有效的。

尽管一些开发者可能会在一个方法定义中使用这些修饰符，但是一些规则限制了组合使用修饰符。*Java™ Language Specification*（由 Gosling 等人在 2005 年编写）一书的第 8.4.3 小节列出了这些限制。

挑战 20.3 的答案（第 178 页）

存在如下两种情况：

在 Java 5 之前的版本中：如果你用某种方式改变了 `Bitmap.clone()` 方法的返回值，这段代码的编译将无法通过。`clone()` 签名会匹配 `Object.clone()` 的签名，因此返回类型也必须一致。

在 Java 5 中：语言本身已经支持协变返回类型，因此子类可以声明更加明确的返回类型。

挑战 20.4 的答案（第 179 页）

不在方法头声明异常的正方观点是：我们首先要注意到 Java 不要求方法声明其可能抛出的所有异常。例如任何方法都可能遇到空指针异常。Java 也承认，让程序员去声明所有可能的异常是不切实际的。应用程序需要某种策略去处理所有的异常。要求开发者声明指定的异常不是异常处理策略的最终方案。

另一方面：程序员需要获得所有可能的帮助信息。因此应用程序的架构需要一个稳定的异常处理策略。强制开发者在每个方法中声明诸如空指针类的异常也是不现实的。但是有些异常，例如打开文件时出错，此时对异常的强制要求就是有用的。C#从方法头中完全消除了程序的异常。

挑战 20.5 的答案（第 179 页）

该图展示了一种算法：用来检测对象模型是否为树形结构（包含来自 `MachineComponent`

类的两个操作) 以及 4 个方法。

第 21 章 模板方法 (Template Method) 模式

挑战 21.1 的答案 (第 185 页)

完整的程序应该与下面这段代码类似:

```
package app.templateMethod;
import java.util.Comparator;
import com.oozinoz.firework.Rocket;

public class ApogeeComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Rocket r1 = (Rocket) o1;
        Rocket r2 = (Rocket) o2;
        return Double.compare(r1.getApogee(), r2.getApogee());
    }
}
```

以及:

```
package app.templateMethod;
import java.util.Comparator;
import com.oozinoz.firework.Rocket;

public class NameComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Rocket r1 = (Rocket) o1;
        Rocket r2 = (Rocket) o2;
        return r1.toString().compareTo(r2.toString());
    }
}
```

挑战 21.2 的答案 (第 188 页)

`markMoldIncomplete()` 方法的代码将一个未完成的模具信息传递给了原材料管理器。一

种方案如下：

```
package com.oozinoz.ozAster;
import aster.*;
import com.oozinoz.businessCore.*;

public class OzAsterStarPress extends AsterStarPress {
    public MaterialManager getManager() {
        return MaterialManager.getManager();
    }

    public void markMoldIncomplete(int id) {
        getManager().setMoldIncomplete(id);
    }
}
```

挑战 21.3 的答案（第 189 页）

你需要使用的其实是一个钩子。你可能会这样描述你的请求：在机器排完药之后和开始清洗机器前，我希望能在 `shutDown()` 方法中帮我增加一个调用。如果我们把这件事写成类似 `collectPaste()` 的方法，那么它就可以被 Oozinoz 公司所复用。

开发人员会和你协商共同定出该方法的名字。问题的关键在于在一个模板方法模式中调用该方法，这样不仅可以解决我们的问题，并且使得程序更加健壮。

挑战 21.4 的答案（第 191 页）

`Machine` 类的 `getPlanner()` 方法应该充分利用抽象的 `createPlanner()` 方法：

```
public MachinePlanner getPlanner() {
    if (planner == null)
        planner = createPlanner();
    return planner;
}
```

这段代码要求为 `Machine` 类增加一个 `planner` 字段。在给 `Machine` 类增加这个字段和 `getPlanner()` 方法后，就可以在子类中删除该字段以及方法了。

这次重构创建了一个模板方法。`getPlanner()` 方法延迟初始化了 `planner` 变量，该变量的初始化由其子类的 `createPlanner()` 方法来实现。

第 22 章 状态 (State) 模式

挑战 22.1 的答案（第 194 页）

正如状态机所示，当门打开时，我们按键，门的状态会变成 StayOpen，再次按键，门将关闭。

挑战 22.2 的答案（第 197 页）

你的代码看起来应如下所示：

```
public void complete() {
    if (state == OPENING)
        setState(OPEN);
    else if (state == CLOSING)
        setState(CLOSED);
}

public void timeout() {
    setState(CLOSING);
}
```

挑战 22.3 的答案（第 201 页）

你的代码看起来应该如下所示：

```
package com.oozinoz.carousel;

public class DoorClosing extends DoorState {
    public DoorClosing(Door2 door) {
        super(door);
    }

    public void touch() {
        door.setState(door.OPENING);
    }
}
```

```
public void complete() {  
    door.setState(door.CLOSED);  
}  
}
```

挑战 22.4 的答案（第 202 页）

图 B.23 显示了一个合理的图。

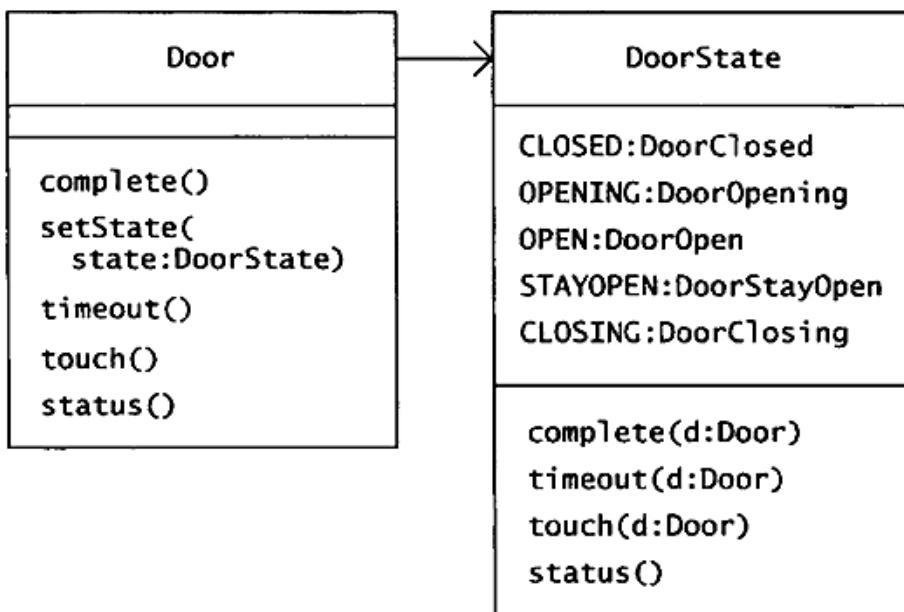


图 B.23 这个设计将 **DoorState** 对象设置成了常量。 **DoorState** 状态迁移方法会更新 **Door** 对象的状态，并且将 **Door** 对象作为方法传递的参数

第 23 章 策略（Strategy）模式

挑战 23.1 的答案（第 207 页）

图 B.24 展示了一种解决方案。

挑战 23.2 的答案（第 209 页）

GroupAdvisor 类和 **ItemAdvisor** 类都是 **Adapter** 类的实例，提供了客户期望的接口，并且利用了不同的接口来使用类的服务。

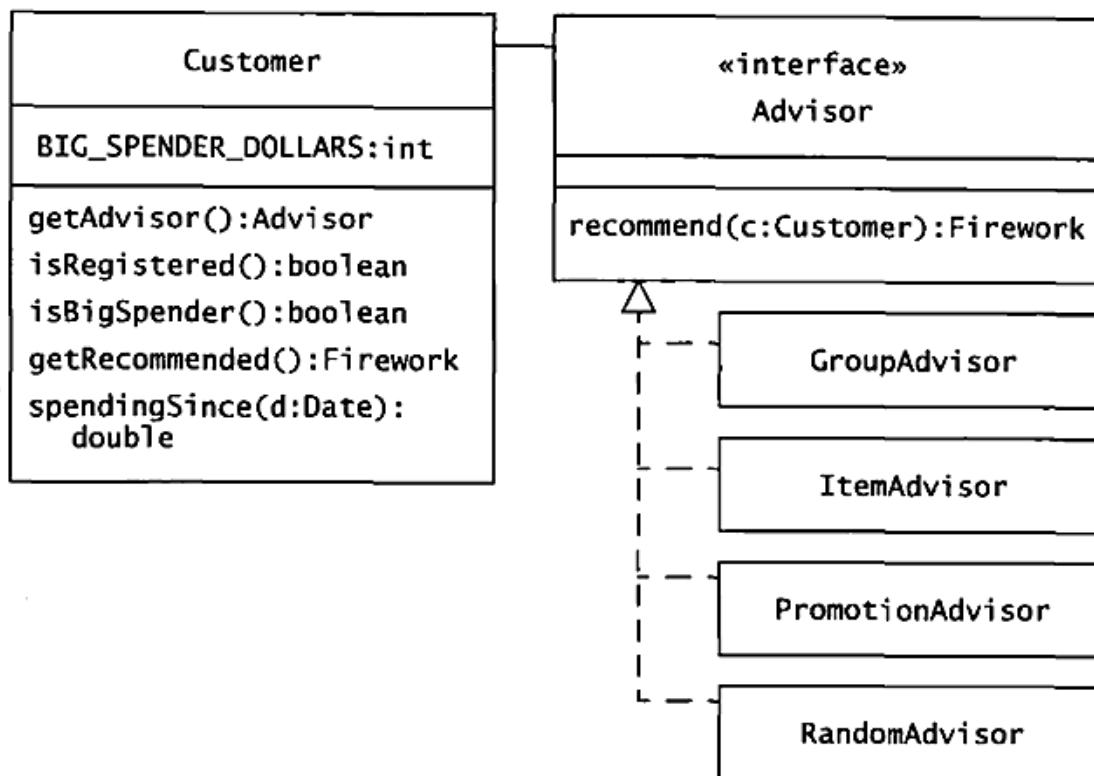


图 B.24 Oozinoz 公司的广告策略包含 4 个策略，这些策略都实现了 Advisor 接口

挑战 23.3 的答案（第 210 页）

你的代码应该看起来如下所示：

```

public Firework getRecommended() {
    return getAdvisor().recommend(this);
}
  
```

一旦知道了 advisor 对象，就可以使用多态来实现了。

挑战 23.4 的答案（第 212 页）

一个可重用的排序例子用的是模板方法模式还是策略模式？

使用策略模式：按照 *Design Patterns* 一书中的说法，模板方法的用意是让“子类”重新定义算法的具体执行步骤。但是 `Collections.sort()` 方法没有使用子类，而用的是一个 `Comparator` 实例。`Comparator` 类的每个实例都提供一个新的方法，因此也提供了一个新的算法和新的策略。所以，`sort()` 方法是策略模式的一个很好的例子。

使用模板方法模式：排序算法有很多，但是 `Collections.sort()` 仅仅用的是其中之一（快

速排序)。改变算法意味着将算法变成堆排序或者冒泡排序。策略模式的意图是让你使用多种算法，而在本例中没有体现出来。而模板方法模式的意图在于能让你将某一个步骤插入到特定算法中，而这正是 `sort()` 方法做的事情。

第 24 章 命令 (Command) 模式

挑战 24.1 的答案 (第 214 页)

很多 Java Swing 程序都使用了调停者模式，注册了一个单独的对象来接收所有的 GUI 事件。该对象负责协调组件间的交互，并且将用户输入转换成控制领域对象的命令。

挑战 24.2 的答案 (第 215 页)

你的代码看起来应该如下所示：

```
package com.oozinoz.visualization;
import java.awt.event.*;
import javax.swing.*;
import com.oozinoz.ui.*;

public class Visualization2 extends Visualization {
    public static void main(String[] args) {
        Visualization2 panel = new Visualization2(UI.NORMAL);
        JFrame frame = SwingFacade.launch(
            panel,
            "Operational Model");
        frame.setJMenuBar(panel.menus());
        frame.setVisible(true);
    }
    public Visualization2(UI ui) {
        super(ui);
    }
    public JMenuBar menus() {
        JMenuBar menuBar = new JMenuBar();
        JMenu menu = new JMenu("File");
        menuBar.add(menu);
        return menuBar;
    }
}
```

```
menuBar.add(menu);

JMenuItem menuItem = new JMenuItem("Save As...");
menuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        save();
    }
});
menu.add(menuItem);

menuItem = new JMenuItem("Restore From...");
menuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        restore();
    }
});
menu.add(menuItem);
return menuBar;
}

public void save() {
    try {
        mediator.save(this);
    } catch (Exception ex) {
        System.out.println("Failed save: " +
            ex.getMessage());
    }
}

public void restore() {
    try {
        mediator.restore(this);
    } catch (Exception ex) {
        System.out.println("Failed restore: " +
            ex.getMessage());
    }
}
}
```

尽管 actionPerformed()方法需要一个 ActionEvent 参数，你依然可以安全地忽略它。menus()方法为 Save 菜单和 Load 菜单都分别注册了一个匿名类的实例。当这些方法被调用时，我们也会很清楚事件的来源。

挑战 24.3 的答案(第 217 页)

testSleep()方法将 doze 命令传递给了工具类的 time()方法：

```
package app.command;

import com.oozinoz.robotInterpreter.Command;
import com.oozinoz.utility.CommandTimer;

import junit.framework.TestCase;
public class TestCommandTimer extends TestCase {
    public void testSleep() {
        Command doze = new Command() {
            public void execute() {
                try {
                    Thread.sleep(
                        2000 + Math.round(10 * Math.random()));
                } catch (InterruptedException ignored) {
                }
            }
        };
        long actual = CommandTimer.time(doze);

        long expected = 2000;
        long delta = 5;
        assertTrue(
            "Should be " + expected + " +/- " + delta + " ms",
            expected - delta <= actual
            && actual <= expected + delta);
    }
}
```

挑战 24.4 的答案（第 219 页）

你的代码应该看起来如下所示：

```
public void shutDown() {
    if (inProcess()) {
        stopProcessing();
        moldIncompleteHook.execute(this);
    }
    usherInputMolds();
    dischargePaste();
    flush();
}
```

注意，上面的代码并没有干扰对 `moldIncompleteHook` 的非空检查，因为它总是会被设置到一个真实的 Hook 对象上（初始情况下，它会被设置到一个什么都不做的 `NullHook` 对象上，但是用户可以设置到不同的钩子）。

你可能会这样使用它：

```
package app.templateMethod;

import com.oozinoz.businessCore.*;
import aster2.*;

public class ShowHook {
    public static void main(String[] args) {
        AsterStarPress p = new AsterStarPress();
        Hook h = new Hook() {
            public void execute(AsterStarPress p) {
                MaterialManager m = MaterialManager.getManager();
                m.setMoldIncomplete(p.getCurrentMoldID());
            }
        };
        p.setMoldIncompleteHook(h);
    }
}
```

挑战 24.5 的答案 (第 219 页)

在工厂方法模式下，客户类知道何时去创建一个新对象，但是不知道创建什么类型的对象。工厂方法模式将对象的创建移到了一个方法中，从而使用户不必关心应该去创建什么样的对象。这种方式也出现在抽象工厂模式中。

挑战 24.6 的答案 (第 220 页)

备忘录模式的意图是提供存储以及恢复对象的状态。通常来说，每当执行一个命令的时候，都可以将其加入到一个栈中，当需要恢复操作时，从栈中弹出这些备忘录并进行恢复即可。

第 25 章 解释器 (Interpreter) 模式

挑战 25.1 的答案 (第 228 页)

ForCommand 类的 execute() 方法的实现应该如下所示：

```
private void execute(MachineComponent mc) {
    if (mc instanceof Machine) {
        Machine m = (Machine) mc;
        variable.assign(new Constant(m));
        body.execute();
        return;
    }

    MachineComposite comp = (MachineComposite) mc;
    List children = comp.getComponents();
    for (int i = 0; i < children.size(); i++) {
        MachineComponent child =
            (MachineComponent) children.get(i);
        execute(child);
    }
}
```

execute() 方法遍历了整个机器组合结构。在遍历过程中如果遇到叶子（机器）节点时，

会将该机器分配给变量，并执行 FormMachine 对象的方法体。

挑战 25.2 的答案（第 232 页）

一种方案如下所示：

```
public void execute() {
    if (term.eval() != null)
        body.execute();
    else
        elseBody.execute();
}
```

挑战 25.3 的答案（第 232 页）

WhileCommand.java 的一种实现方式，如下所示：

```
package com.oozinoz.robotInterpreter2;

public class WhileCommand extends Command {
    protected Term term;

    protected Command body;

    public WhileCommand(Term term, Command body) {
        this.term = term;
        this.body = body;
    }

    public void execute() {
        while (term.eval() != null)
            body.execute();
    }
}
```

挑战 25.4 的答案（第 233 页）

一种答案是：解释器模式的意图是让你从类层级结构中组合可执行的对象，从而为某些公共操作提供各种解释行为。命令模式的意图仅仅是用来封装对象的请求。

解释器对象可否有命令对象的功能？当然可以！使用哪种模式取决于你的意图。你是想创建组合可执行对象的工具集，还是想封装对象的请求？

第 26 章 扩展型模式介绍

挑战 26.1 的答案（第 238 页）

从数学的角度来讲，圆是椭圆的一种特殊情况。然而在面向对象编程中，椭圆有一些圆所没有的行为。例如，椭圆的宽可能是高的两倍，而圆不可能是这样。如果这样的行为对你的程序很重要，椭圆就不可能是圆的对象，这将会违背 LSP 原则。

注意，如果你要用不可变对象，那么这样做未必适合。这只是一个简单的数学问题，而不是标准的类型层次结构。

挑战 26.2 的答案（第 239 页）

如果 `tub` 对象的 `Location` 属性有任何变化，`tub.getLocation().isUp()` 表达式可能会导致程序错误。例如，在箱子传输过程中，`location` 可能是 `null` 或者是机器人对象。此时计算 `tub.getLocation().isUp()` 的值可能会抛出异常。如果 `location` 是一个机器人对象，问题可能会变得更加糟糕，因为我们让机器人从自身收集原材料。这些潜在的问题是可以管理的，但是难道我们希望这些管理代码写在 `tub.getLocation().isUp()` 方法中吗？显然不是。这些代码可能在 `Tub` 类中都实现了。如果没有，就应该在 `Tub` 类中添加相关代码，否则就会导致我们在其他地方多次重复写这些代码。

挑战 26.3 的答案（第 240 页）

一个例子如下：

```
public static String getZip(String address) {  
    return address.substring(address.length() - 5);  
}
```

这段代码有一些坏味道，包括基本类型偏执（primitive obsession）（指一个字符串包括很多的属性）。

挑战 26.4 的答案（第 240 页）

一组解决方案如表 B.2 所示。

表 B.2 所使用的模式

例 子	使用的模式
某个焰火模拟器的设计者设计了一个接口，要求所建立的对象必须实现该接口，以便于参与模拟	适配器模式
允许在运行时组合可执行对象的工具集	解释器模式
父类提供了一个方法，要求子类来实现其中的一些步骤	模板方法模式
一个对象允许扩展它的行为，通过对象中封装的方法，并且在合适的时机来调用	命令模式
通过代码生成器来插入某些行为，以便形成模拟对象在本机执行的假象	代理模式
该设计允许注册一些回调方法，并且在对象发生变化时触发	观察者模式
该设计允许提供已经定义好接口的抽象操作，并且能够添加实现该接口的新驱动器	桥接模式

第 27 章 装饰器（Decorator）模式

挑战 27.1 的答案（第 248 页）

一种解决方案如下所示：

```
package com.oozinoz.filter;
import java.io.*;

public class RandomCaseFilter extends OozinozFilter {
    public RandomCaseFilter(Writer out) {
        super(out);
    }

    public void write(int c) throws IOException {
        out.write(Math.random() < .5
            ? Character.toUpperCase(c)
            : Character.toLowerCase(c));
    }
}
```

```
    ? Character.toLowerCase((char) c)
    : Character.toUpperCase((char) c));
}
}
```

随机大小写问题也非常有趣，看看下面的程序：

```
package app.decorator;
import java.io.BufferedWriter;
import java.io.IOException;

import com.oozinoz.filter.ConsoleWriter;
import com.oozinoz.filter.RandomCaseFilter;

public class ShowRandom {
    public static void main(String[] args)
        throws IOException {
        BufferedWriter w =
            new BufferedWriter(
                new RandomCaseFilter(new ConsoleWriter()));
        w.write("buy two packs now and get a "
            + "zippie pocket rocket -- free!");
        w.newLine();
        w.close();
    }
}
```

这段程序使用了第27章定义的 `ConsoleWriter` 类。运行该程序将产生如下输出：

```
buY tWO pACKS NOW AND gET A ZiPPIE PoCkEt RoCKEt -- frEE!
```

挑战 27.2 的答案(第 250 页)

一种方案如下所示：

```
package com.oozinoz.filter;
import java.io.Writer;

public class ConsoleWriter extends Writer {
    public void close() {}
    public void flush() {}
```

```
public void write(
    char[] buffer, int offset, int length) {
    for (int i = 0; i < length; i++)
        System.out.print(buffer[offset + i]);
}
```

挑战 27.3 的答案（第 256 页）

一种方案如下所示：

```
package com.oozinoz.function;

public class Exp extends Function {
    public Exp(Function f) {
        super(f);
    }

    public double f(double t) {
        return Math.exp(sources[0].f(t));
    }
}
```

挑战 27.4 的答案（第 257 页）

一种方案如下所示：

```
package app.decorator.brightness;

import com.oozinoz.function.Function;

public class Brightness extends Function {
    public Brightness(Function f) {
        super(f);
    }

    public double f(double t) {
        return Math.exp(-4 * sources[0].f(t))
            * Math.sin(Math.PI * sources[0].f(t));
    }
}
```

第 28 章 迭代器（Iterator）模式

挑战 28.1 的答案（第 265 页）

`display()`程序启动了一个线程，该线程可以在任何时候被唤醒。尽管 `sleep()`方法的调用保证了当 `display()` 正在休眠时，`run()` 方法才会被调用。输出结果表明了当程序运行时，`display()` 方法完整地运行了一个迭代，并且打印出了列表中索引 0 的数据项：

```
Mixer1201
```

此时，第二个线程被唤醒，并且将 `Fuser1101` 放到了列表的开头，并且将其他的机器名向下移动了一位。举例来说，`Mixer1201` 从索引 0 的位置移到了索引 1 的位置。

当主线程获得了控制权，`display()` 方法将打印出列表中剩下的部分，从索引 1 到结束。

```
Mixer1201
ShellAssembler1301
StarPress1401
UnloadBuffer1501
```

挑战 28.2 的答案（第 267 页）

反对使用 `synchronized()` 方法的理由是：当用 `for` 循环进行迭代时，`synchronized()` 方法会出错，甚至会使整个程序崩溃，除非你捕获了 `InvalidOperationException` 异常并写了相关的处理逻辑。

反对使用锁机制的理由是：线程安全的迭代设计依赖于访问集合线程间的协作。`synchronized()` 方法的关键点是捕获线程没有正常合作的情况。

`synchronized()` 方法和 Java 内置的锁机制，都不会简化多线程的开发。如果要进一步学习并发编程，请参考 *Concurrent Programming in Java™*（由 Lea 在 2000 年编写）这本书。

挑战 28.3 的答案（第 272 页）

正如 16 章所提到的，迭代器是工厂方法模式的一个经典例子。如果用户想要创建

`ProcessComponent` 类的迭代器，则必须知道创建迭代器的时机，接收类则需要知道实例化哪个类。

挑战 28.4 的答案（第 277 页）

一个方案如下所示：

```
public Object next() {
    if (peek != null) {
        Object result = peek;
        peek = null;
        return result;
    }

    if (!visited.contains(head)) {
        visited.add(head);
        if (shouldShowInterior()) return head;
    }

    return nextDescendant();
}
```

第 29 章 访问者（Visitor）模式

挑战 29.1 的答案（第 280 页）

区别在于 `this` 对象的类型不同。`accept()`方法调用 `MachineVisitor` 对象的 `visit()`方法，在 `Machine` 类内，`accept()`方法将会使用 `visit(:Machine)` 签名来查询 `visit()` 方法。而在 `MachineComposite` 类内，`accept()`方法将会使用 `visit(:MachineComposite)` 签名来查询 `visit()` 方法。

挑战 29.2 的答案（第 283 页）

一种方案如下所示：

```
package app.visitor;
```

```
import com.oozinoz.machine.MachineComponent;
import com.oozinoz.machine.OozinozFactory;

public class ShowFindVisitor {
    public static void main(String[] args) {
        MachineComponent factory = OozinozFactory.dublin();
        MachineComponent machine = new FindVisitor().find(
            factory, 3404);
        System.out.println(machine != null ?
            machine.toString() : "Not found");
    }
}
```

挑战 29.3 的答案(第 285 页)

一种方案如下所示:

```
package app.visitor;
import com.oozinoz.machine.*;
import java.util.*;

public class RakeVisitor implements MachineVisitor {
    private Set leaves;

    public Set getLeaves(MachineComponent mc) {
        leaves = new HashSet();
        mc.accept(this);
        return leaves;
    }

    public void visit(Machine m) {
        leaves.add(m);
    }

    public void visit(MachineComposite mc) {
        Iterator iter = mc.getComponents().iterator();
        while (iter.hasNext())
            ((MachineComponent) iter.next()).accept(this);
    }
}
```

挑战 29.4 的答案（第 290 页）

一个解决方案是给所有的 `accept()` 方法和 `visit()` 方法都增加一个 `set` 参数，这样就可以传递访问过的节点集合。`ProcessComponent` 类可以通过给 `accept()` 方法传递一个 `set` 对象来调用其抽象的 `accept()` 方法。

```
public void accept(ProcessVisitor v) {  
    accept(v, new HashSet());  
}
```

`ProcessAlternation`、`ProcessSequence` 和 `ProcessStep` 子类的 `accept()` 方法应该为：

```
public void accept(ProcessVisitor v, Set visited) {  
    v.visit(this, visited);  
}
```

现在访问者的开发人员必须创建包含 `visit()` 方法的类，其中 `visit()` 方法需要能接收 `visited` 集合。很明显，这时候非常适合使用 `set` 作为集合。尽管该访问者的开发人员需要公开对该集合的访问。

挑战 29.5 的答案（第 291 页）

访问者模式有几种替代方案：

- 将需要的行为加入到原来的类层级结构中。可以通过两种方式达到该目的：说服该类层级结构的开发者加入你所需要的代码；在代码集体所有制的前提下，我们可以更改该代码。
- 你可以让操作机器或者流程结构的类直接遍历该层次结构。如果想知道组合结构中子对象的类型，可以使用 `instanceof` 运算符，或者写一个返回 `Boolean` 类型的函数，例如 `isLeaf()` 或者 `isComposite()`。
- 如果想让增加的行为和现有的行为有很明显的差别，需要创建一个并行的类层次结构。例如工厂方法模式中的 `MachinePlanner` 类，其机器的行为就放在独立的层次结构中。