第9章

观察者（Observer）模式

客户端通常通过调用它所感兴趣的对象的方法来获取信息。然而，一旦感兴趣的对象发生改变，就会出现问题：客户端怎样才能知道它所依赖的对象信息发生了改变呢？

你可能遇到过这样的设计：当客户端感兴趣的对象某个方面发生了改变时，需要创建一个对象负责通知客户端。问题是，由于客户端自己知道它所感兴趣的对象的某些方面，因此这个对象自身却不应该承担更新客户端的职责。一个解决方案是当对象发生改变时通知客户端，让客户端自己去查询对象的新状态。

观察者模式的意图是在多个对象之间定义一对多的依赖关系，当一个对象的状态发生改变时，会通知依赖于它的对象，并根据新状态做出相应的反应。

经典范例：GUI 中的观察者模式

在观察者模式中，当某个对象发生改变时，会通知关注它的对象。该模式最常见的例子是用户图形界面。无论用户是在单击按钮，还是调整滑动条，程序中的很多对象都可能会对这些变化产生反应。Java 的设计者认为，用户需要了解 GUI 组件发生的变化，因而在 Swing 中广泛地运用了观察者模式。Swing 将客户端称为“侦听器（Listener）”，并且可以为组件的改变事件注册多个“侦听器”。

考虑图 9.1 中展示的 Oozinoz 公司一个典型的 GUI 应用程序。该应用程序可以让焰火引擎实验根据不同的参数进行可视化展示，从而确定火箭推力和燃料表面燃烧率的关系。

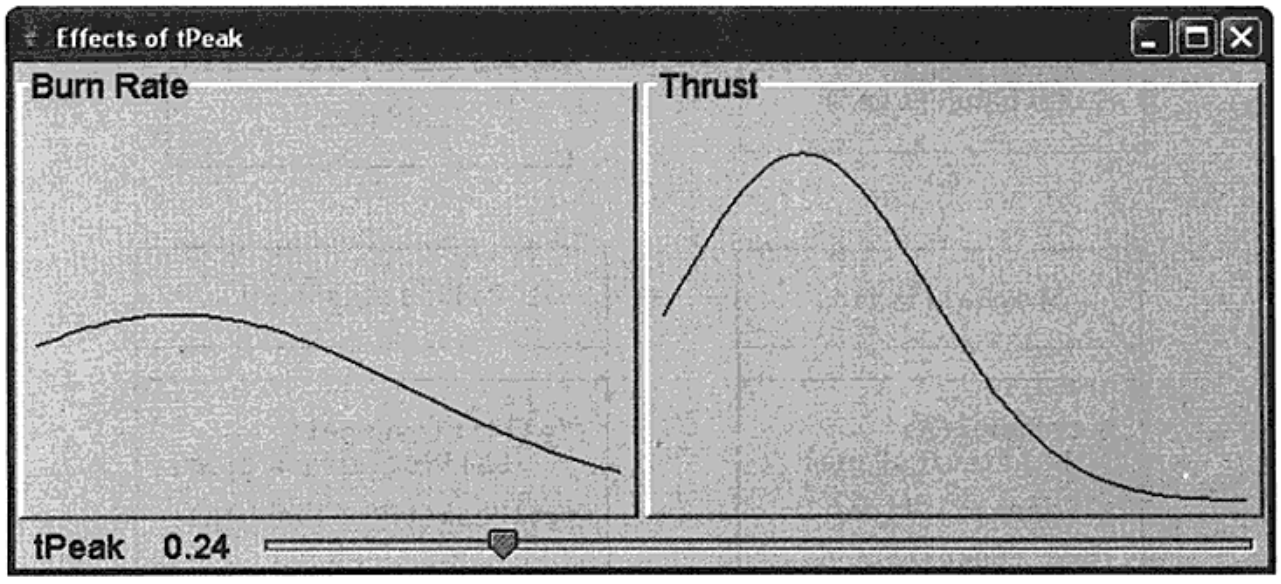

图 9.1　当用户调整滑动条的 tPeak 变量时，该曲线可以实时反映出变化

当固体火箭发动机点火时，燃料中暴露在空气中的部分将会燃烧，产生推动力。从点火到最大燃烧率的过程中，燃烧区间从点火区域扩大到整个燃料表面。最大燃烧率的峰值发生在 t_{peak} 时刻。随着燃料的燃烧，表面区域不断减少，直到燃料消耗完毕。这个弹道应用程序约定了时间标准，0 代表点火时刻，1 代表燃烧结束，因此 t_{peak} 是一个在 0～1 之间的数字。

Oozinoz 使用如下等式来计算燃烧率和推力：

$$rate = 25^{-(t-t_{peak})^2}$$

$$thrust = 1.7 \cdot \left(\frac{rate}{0.6}\right)^{1/0.3}$$

图 9.1 中的应用程序展示了 t_{peak} 是如何影响燃烧率和火箭推动力的。当用户移动滑动条时，t_{peak} 的值发生改变，曲线就呈现出新的形状。图 9.2 展示了该程序的一些主要类。

`ShowBallistics` 类和 `BallisticsPanel` 类都属于 `app.observer.ballistics` 包。`BallisticsFunction` 接口属于 `com.oozinoz.ballistics` 包，该接口定义了燃烧率和推力曲线。这个包也包含了 `Ballistics` 工具类，该类实现了 `BallisticsFunction` 接口。

当弹道应用程序初始化滑动条时，应用程序将自己注册成侦听器，以便接收滑动条事件。当滑动条滑动时，应用程序负责更新展示曲线的容器，并且更新用于显示 t_{peak} 的标签。

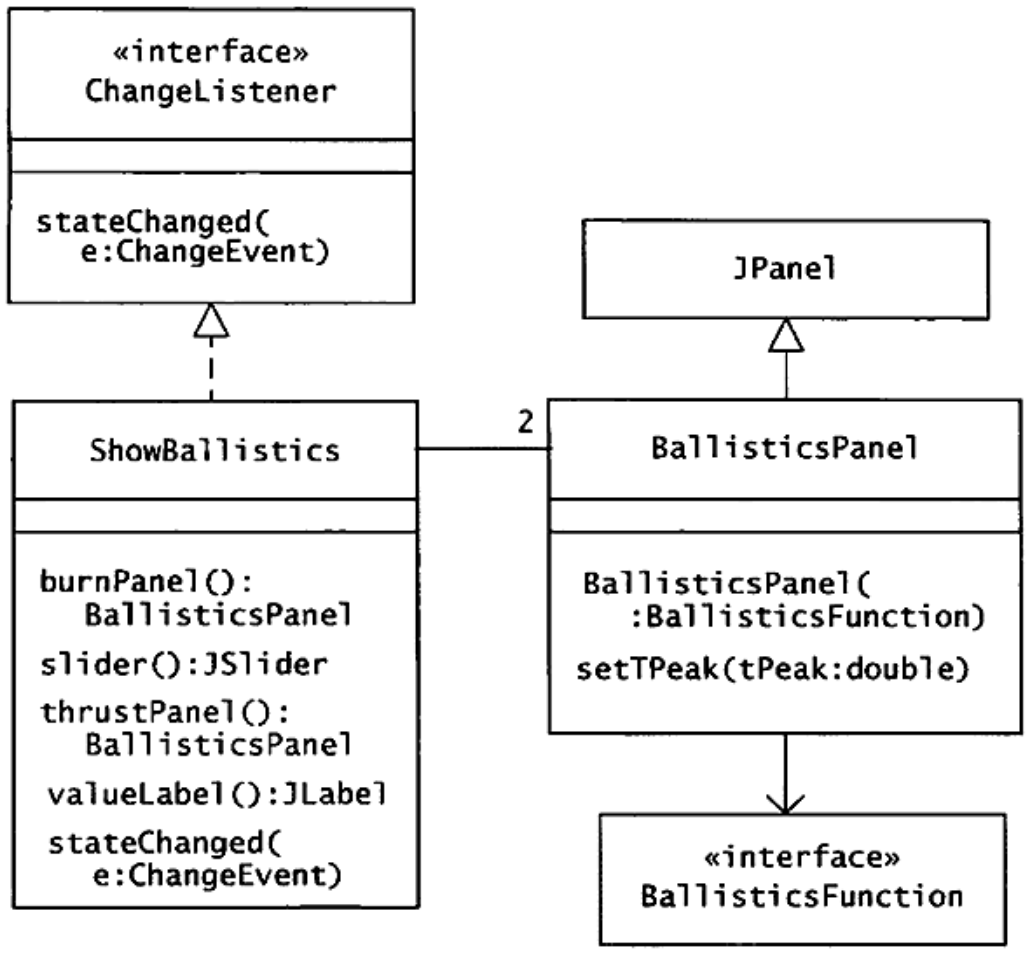

图 9.2 Ballistics 应用程序注册自身以便于接收滑动条事件

挑战 9.1

完成 ShowBallistics 的 slider()和 stateChanged()方法，以便弹道容器和 t_{peak} 标签可以反映出滑动条的值。

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

public void stateChanged(ChangeEvent e) {
```

```
        doubleval = slider.getValue();
        doubletp = (val - sliderMin) / (sliderMax - sliderMin);
        burnPanel(). ?? ( ?? );
        thrustPanel(). ?? ( ?? );
        valueLabel(). ?? ( ?? );
    }
```

答案参见第 313 页

ShowBallistics 类更新燃烧率与推力容器，以及依赖于滑动条值的 t_{peak} 标签。这种做法很常见，方案也还算不错。但需要注意的是，它完全曲解了观察者模式的意图。Swing 使用观察者模式的目的是为了让滑动条无须关心有哪些客户端对它有兴趣。但是 ShowBallistics 应用又将我们带回到之前极力避免的问题：一个独立对象（即应用程序）需要知道要更新哪些对象，并且负责进行相关的调用，而不是让每个独立的对象注册它们自身。

为了创建粒度更细的观察者模式，我们可以对代码进行少量修改，让每个感兴趣的对象注册它们自己，以便可以接收滑动条的改变事件。

挑战 9.2

请设计一个新的类图，可以让感兴趣的对象注册自身以便接收滑动条事件。不要忘记用于显示滑动条值的标签。

答案参见第 314 页

根据该设计，可以将 addChangeListener()方法的调用从 slider()方法中移出来，放到所依赖组件的构造函数中。

```
public BallisticsPanel2(
        BallisticsFunctionfunc,
        JSlider slider) {
    this.func = func;
    this.slider = slider;
    slider.addChangeListener(this);
}
```

当滑动条滑动时，BallisticsPanel2 对象会被通知。标签会重新计算 tPeak 值并重新绘制

自己。

```
public void stateChanged(ChangeEvent e) {
    double val = slider.getValue();
    double max = slider.getMaximum();
    double min = slider.getMinimum();
    tPeak = (val - min) / (max - min);
    repaint();
}
```

重构之后出现了一个新问题。这个设计调整了职责的分配，以便让感兴趣的对象向滑动条注册自身，并对滑动条的滑动事件做出自己的反应。这样的职责分配没有问题，但是，现在每个侦听滑动条的组件都需要重新计算 tPeak 的值。特别是如果使用挑战 9.2 中提供的 BallisticsLabel2 类的解决方案，它的 stateChanged()方法几乎和这里的 stateChanged()方法一样。为了消除重复代码，需要从当前设计中提取出一个潜在的领域对象来再次重构。

可以通过引入一个包含关键峰值的 Tpeak 类来简化系统。我们让应用程序侦听滑动条，并更新 Tpeak 对象，然后让其他所有感兴趣的对象去侦听这个对象。这就是模型/视图/控制器（MVC）模式。查阅 Buschmann 等人的著作 *Pattern Oriented Software Architecture*，其中对 MVC 有更深入的讨论[译注1]。

模型/视图/控制器

随着应用程序和系统的增长，需要将类和包的职责划分得足够细，以便于维护。模型/视图/控制器是指将对象（模型）从显示它的 GUI 元素中（视图/控制器）分离出来。Java 通过使用侦听器来支持这种分离，但是正如上一节提到的，并不是所有使用侦听器的设计都是 MVC 模式。

ShowBallistics 应用的早期版本将 GUI 程序与弹道学知识组合在一起。可以重构这段代码，按照 MVC 的思想去分解应用程序的职责。在这次重构过程中，修正过的 ShowBallistics 类应该将视图和控制器保留在 GUI 元素中。

译注1：Buschmann 等人的著作一共分为 5 卷，在架构模式的大前提下，每卷都有一个自己的主题。严格意义上讲，MVC 模式起源于 20 世纪 80 年代的 Smalltalk。现今，MVC 模式已经得到了相当普及的运用，诸多 MVC 框架如 Struts、Ruby On Rails、ASP.NET MVC 已经非常成熟。

MVC 的创建者期望将组件的外观（视图）和行为（控制器）分离。在实际应用中，组件的外观与用户交互的支持是紧耦合的，Swing 的典型应用没有将视图从控制器中分离出来。（如果深入 Swing 内部实现，可能会发现这种分离现象。）MVC 的价值在于：它将模型从应用程序中提了出来，形成了自己的领域。

ShowBallistics 应用程序的模型就是 tPeak 值。为了将其重构成 MVC，需要引入一个 Tpeak 类，该类拥有一个峰值，并且可以允许感兴趣的监听器去监听事件的变化。如下所示：

```
package app.observer.ballistics3;
import java.util.Observable;

public class Tpeak extends Observable {
    protected double value;
    public Tpeak(double value) {
        this.value = value;
    }

    public double getValue() {
        return value;
    }

    public void setValue(double value) {
        this.value = value;
        setChanged();
        notifyObservers();
    }
}
```

如果评审 Oozinoz 公司的这段代码，会发现一个关键点：代码与火箭引擎燃料燃烧率到达峰值的时间几乎没有关系。事实上，这段代码看起来更像是一个通用的工具类：该类可以控制值，且当该值发生改变时，会去通知侦听器。我们可以对这段代码进行重构，使其变得更为通用，不过，在重构前先来看看使用修正过的 Tpeak 类的设计，同样很有价值。

现在我们的设计让应用程序能够侦听滑动条，其余的则可以侦听 Tpeak 对象。当滑动条移动时，应用程序为 Tpeak 对象设置了一个新值。面板和文本框侦听 Tpeak 对象，当对象值改变时会更新它们自身。BurnRate 和 Thrust 类使用 Tpeak 对象来计算它们自己的功能，但是它们不需要通过注册来侦听事件。

挑战 9.3

创建一个类图，该图可以展示应用程序和滑动条的依赖关系，以及文本框和测绘容器与 Tpeak 对象的依赖关系。

答案参见第 315 页

该设计允许将滑动条的值同时转换为峰值。应用程序负责更新唯一的 Tpeak 对象，所有侦听（滑动条）值改变的 GUI 对象都可以查询 Tpeak 对象的新值。

然而，除了维持该值外，Tpeak 类再没有其他的价值了。因此我们希望分解出一个维护值的类。另外，像峰值这样的观察值，并不是独立的值，而是领域对象的一个属性。例如峰值是火箭引擎的一个属性。可以通过分离这些类来改善我们的设计，如使用维护值的类让 GUI 对象观察领域对象。

当你从领域对象或者业务对象中分解 GUI 对象时，可以创建一个代码层。这一层代码是拥有相似职责的一组类，通常会被放进一个单独的 Java 包中。像 GUI 层这样的高层，通常仅依赖于同层或者底层代码。层与层之间通常具有清晰定义的接口，就像 GUI 和它所代表的业务层。你要重新组织 ShowBallistics 代码的职责，使之构成系统的一层。就像图 9.3 所示。

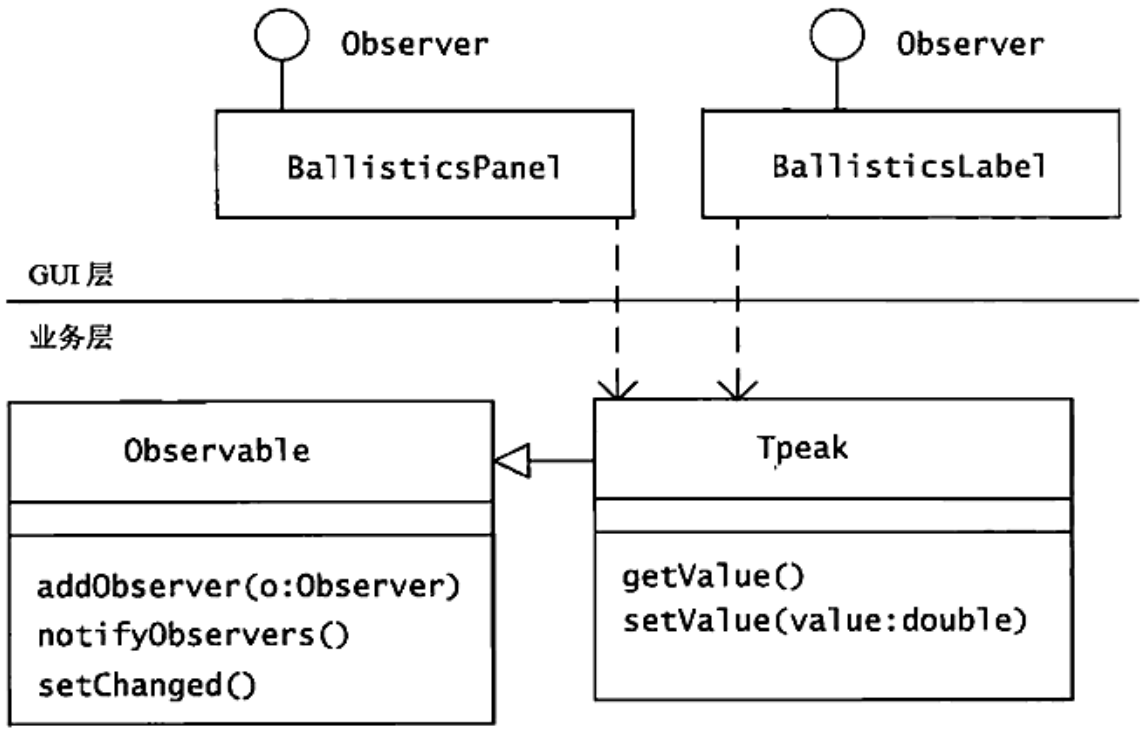

图 9.3　通过创建一个可观察的 Tpeak 类，可以将业务逻辑层从 GUI 层分离出来

图 9.3 的设计创建了一个 Tpeak 类对 t_{peak} 值进行建模，该值是应用程序显示弹道公式的一个关键值。BallisticsPanel 和 BallisticsLabel 类依赖于 Tpeak，为了避免让 Tpeak 对象负责去更新 GUI 元素，该设计应用了观察者模式，让感兴趣的对象可以注册到 Tpeak 类，以便在该类发生改变时得到通知。Java 类库中的 java.util 包中提供的 Observable 类和 Observer 接口，就是为了支持该设计。Tpeak 类继承自 Observable 类，它的数值发生改变时会通知其观察者。

```
public void setValue(double value) {
    this.value = value;
    setChanged();
    notifyObservers();
}
```

注意，你需要调用 setChanged()方法，以便使继承自 Observable 类的 notifyObservers()方法可以将改变广播出去。

notifyObservers()方法调用每个已经注册的观察者的 update()方法。如图 9.4 所示，Observer 接口的实现者都必须实现 update()方法。

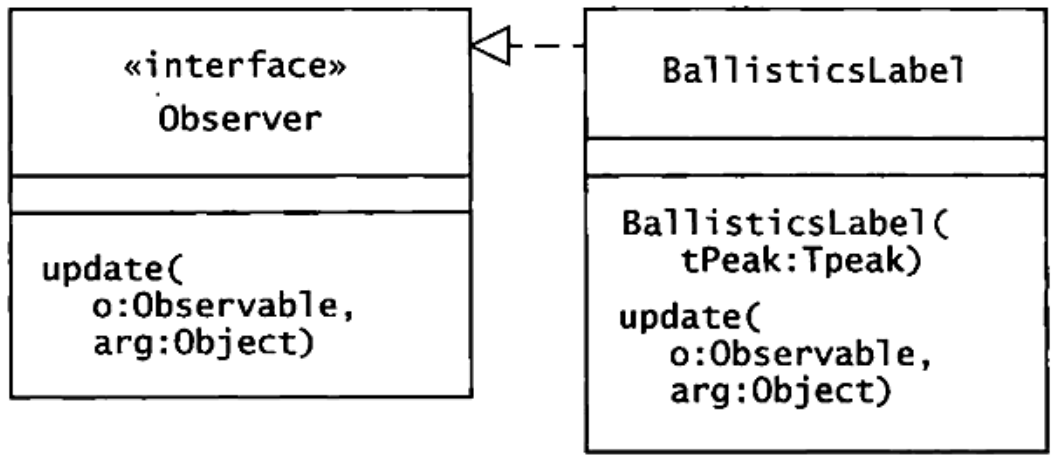

图 9.4　BallisticsLabel 类是一个观察者：它可以在 Observable 对象里注册它感兴趣的东西，当 Observable 对象发生改变时，BallisticsLabel 对象的 update()就会被调用

BallisticsLabel 对象无须保持对它所观察的 Tpeak 对象的引用。当然，在 BallisticsLabel 的构造函数中，该对象可以被注册成 Tpeak 对象的侦听器。该标签类的 update()方法将会接收一个参数类型为 Observable 的 Tpeak 对象，并且可以将参数转化为 Tpeak 类型，得到新值，然后更新标签的文本以及重绘曲线。

挑战 9.4

写出 BallisticsLabel.java 的完整代码。

答案参见第 315 页

弹道应用程序的新设计将业务对象从展现它的 GUI 元素中分离出来。这一设计包含两个关键步骤：

1. 观者者的实现类必须注册它们感兴趣的对象，并且自身能做出相应的变更，这通常包括重绘它们自己。

2. 当被观察者的实现类的值发生改变时，必须记得通知观察者。

在弹道应用程序中，这两个步骤几乎涵盖了需要跨层调用的代码。同时，还需要准备一个 Tpeak 对象，当应用程序的滑动条滑动时，该对象的值能随之变动。这可以通过准备一个 ChangeListener 的匿名类来实现。

挑战 9.5

假设 tPeak 是 Tpeak 的一个实例，并且是 ShowBallistics3 类的一个属性。请补充完成 ShowBallistics3.slider()的代码，以便滑动条的改变能更新到 tPeak：

```
public JSlider slider() {
    if (slider == null) {
        slider = new JSlider();
        sliderMax = slider.getMaximum();
        sliderMin = slider.getMinimum();
        slider.addChangeListener
            (
                new ChangeListener()
                {
                    // 挑战！
                }
            );
        slider.setValue(slider.getMinimum());
```

```
        }
        return slider;
    }
```

答案参见第 316 页

当你运用 MVC 时，事件的流向可能看起来有些迂回。弹道应用程序中滑动条的移动会驱动 ChangeListener 去更新 Tpeak 对象。反过来，Tpeak 对象的改变会通知应用程序的标签和面板容器对象，然后让这些对象重绘它们自己。变化的传播从 GUI 层到业务层，最后又回到 GUI 层。

挑战 9.6

请填写图 9.5 所示的消息。

答案参见第 316 页

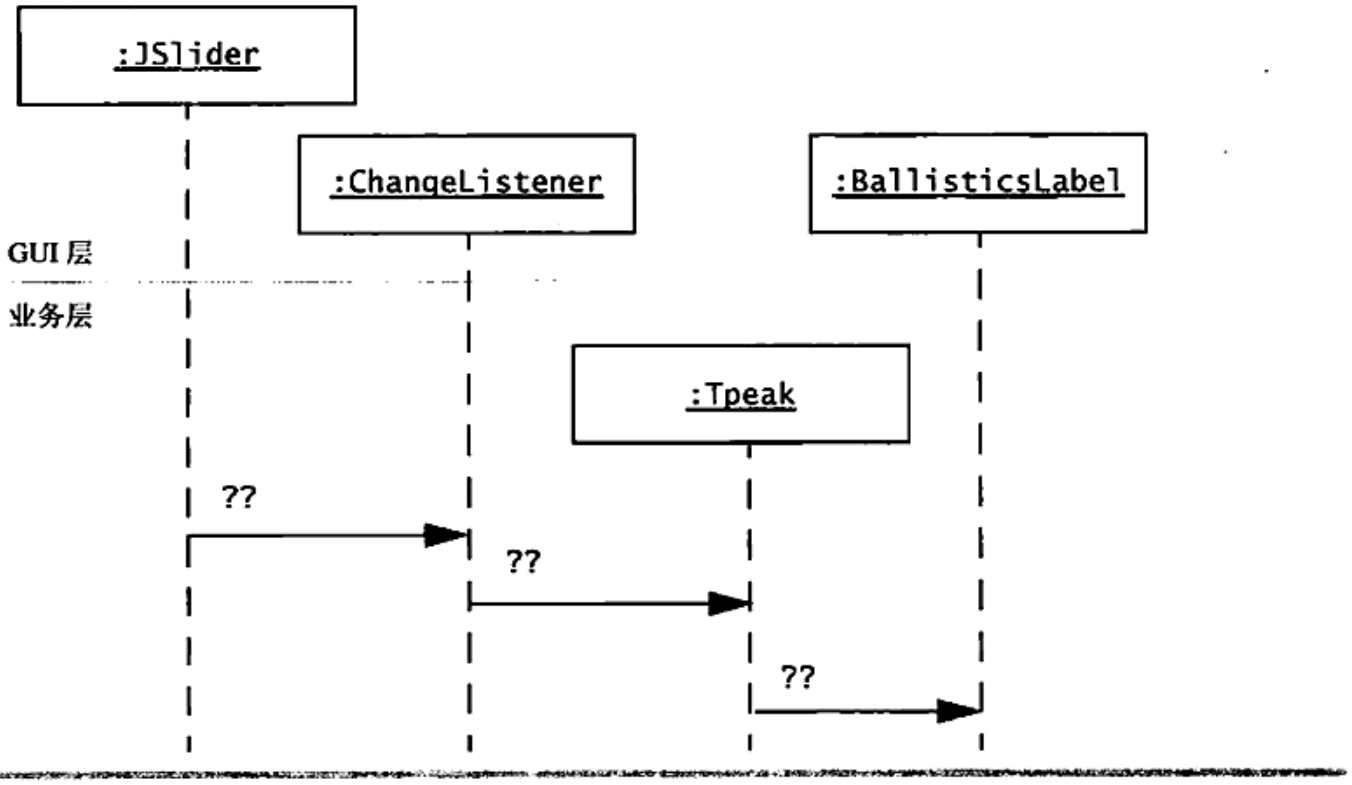

图 9.5　MVC 使得调用从 GUI 层到业务层，又从业务层回到 GUI 层

分层设计的好处在于使用接口能够分离不同的层。代码的分层就是职责的分层，这就使得代码更加容易维护。例如，在这个弹道应用程序中，你可以新增一个类似手持设备的 GUI，而不需要改变业务对象层的类。在业务对象层，你可以新增一个可以更新 Tpeak 对象的事件源，

而不需要更改 GUI。观察者模式提供的机制会自动更新 GUI 层的对象。

分层设计还提供了让不同层运行在不同计算机上的可能性，这样的系统称之为 *n* 层系统。一个 *n* 层软件设计可以大量减少运行在用户桌面上的程序的代码量。它还能使你在更改系统的业务类后无须升级安装用户机器上的软件。这极大地简化了部署。然而，在计算机之间传送消息是有代价的，你需要慎重实施 *n* 层系统的部署。例如，滚动条的来回滑动事件频繁在用户端和服务器之间发生，会消耗一定时间，这是用户难以接受的。此时，就必须让滑动事件发生在用户机器上，然后在达到新峰值时提交，从而达到分离用户行为的目的。

简而言之，观察者模式支持 MVC，该模式鼓励分层设计软件，这给软件开发和部署带来了很多实用的好处。

维护 Observable 对象

有时我们可能无法创建一个观察者类，该类继承自 `Observable` 类的子类。特别是当该类已经是其他类（不是 Java 内置的 `Object` 类）的子类时，我们可以提供一个拥有 `Observable` 对象的类，该类可以将关键的方法调用转发给 `Observable` 对象。`java.awt` 包中的 `Component` 类就使用了这种方法，但是它用了一个 `PropertyChangeSupport` 对象代替了 `Observable` 对象。

`PropertyChangeSupport` 类和 `Observable` 类非常相似，但前者属于 `java.beans` 包。JavaBeans API 支持创建可复用的组件，并在 GUI 组件中得到了广泛使用。当然，你可以在任何地方使用它。`Component` 类使用 `PropertyChangeSupport` 对象让感兴趣的观察者将自己注册到其中，以便在标签、面板容器或其他 GUI 组件发生改变时得到通知。图 9.6 展示了 `java.awt` 包中的 `Component` 类和 `PropertyChangeSupport` 类的关系。

`PropertyChangeSupport` 类描述了一个在使用观察者模式时需要解决的问题：被观察类需要向观察者提供多少改变的细节？该类使用了“推”模式，用模型给出了改变的细节（在 `PropertyChangeSupport` 中，通知给出了由旧值到新值的属性变化）；另一个方式是“拉”模式，它建立的模型会告诉观察者有改变发生，但是观察者需要查询模型来获取改变的具体信息。两种方式各有千秋。“推”模式需要更多的工作量，并且会将观察者与被观察者绑定起来，但是它提供了较好的性能。

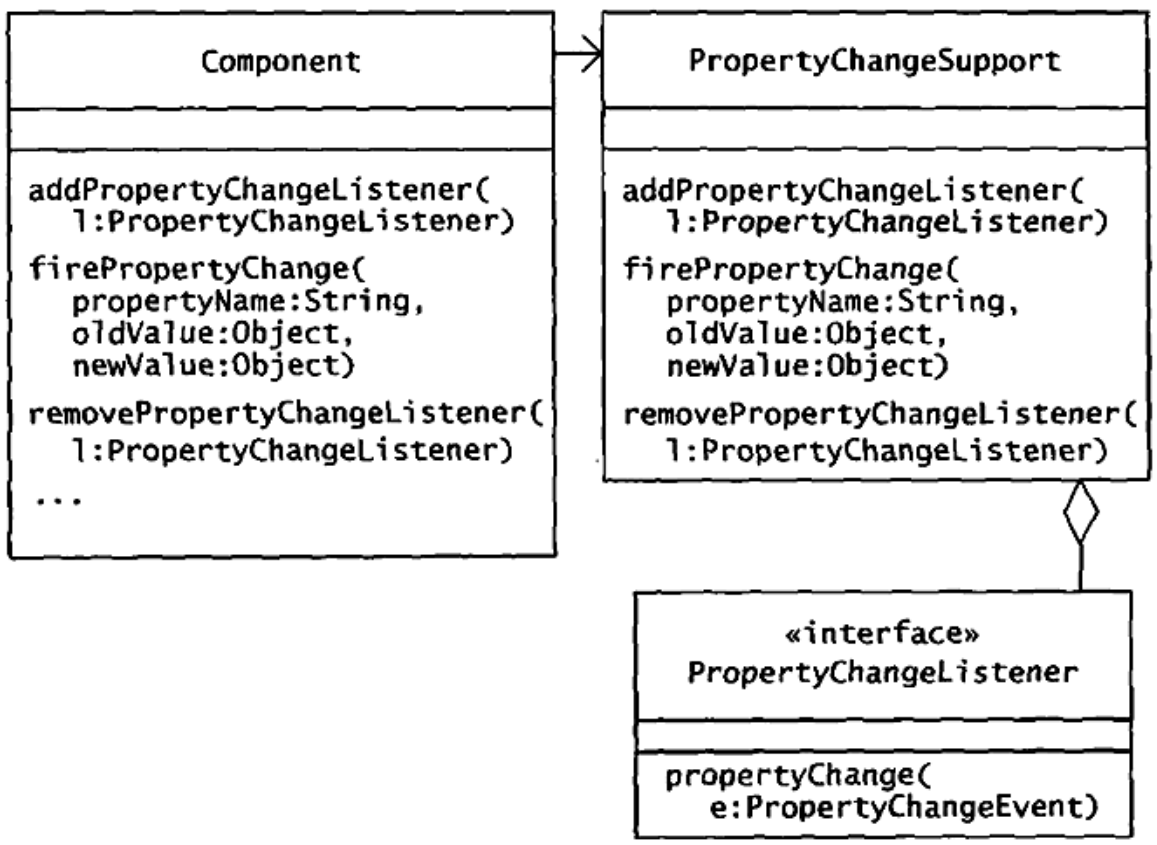

图 9.6　一个 Component 对象维护一个 PropertyChangeSupport 对象，一个 PropertyChangeSupport 对象维护一组侦听器集合

Component 类复制了 PropertyChangeSupport 类的部分接口。Component 类的每一个方法都会将消息调用转发给 PropertyChangeSupport 类的实例。

> **挑战 9.7**
>
> 完成图 9.7 中的类图，该类图展示了 Tpeak 使用 PropertyChangeSupport 对象来管理侦听器。
>
> 答案参见第 316 页

使用 Observer 也好，使用 PropertyChangeSupport 或其他类也罢，构建观察者模式的要点是要在对象间建立起一个一对多的关系。当一个对象状态发生改变时，所有依赖它的对象都会被通知，并做出相应的更新。这有助于缩小职责范围，使得维护观察者和被观察对象变得轻而易举。

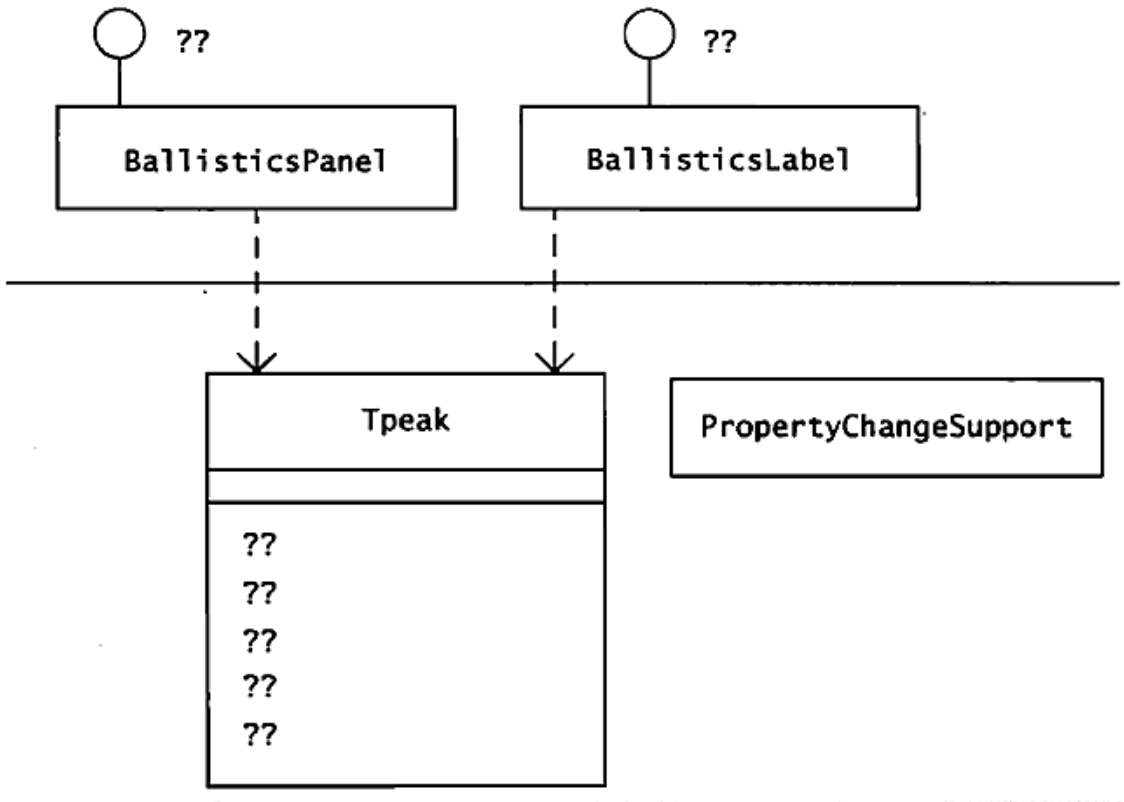

图 9.7　Tpeak 业务对象可以将影响侦听者的调用委托给 PropertyChangeSupport 对象

小结

观察者模式经常出现在 GUI 应用程序中，它是 Java GUI 类库的基础模式。有了这些组件，当需要简单地将事件通知给相关对象时，就无须再改变或者继承自组件类了。对规模较小的应用程序而言，通常的做法是注册一个单独的对象，它将负责接收 GUI 中的所有事件。这一做法本身没有问题，但你必须知道它违背了观察者模式职责分离的意图。对于大型的 GUI 程序，建议使用 MVC 模式：让每一个相关对象都注册自己的事件，而不是让一个中间对象负责注册所有的事件。MVC 还使你能够创建相互独立的松耦合的层级系统，系统的每一层都可以运行在不同的机器上。