

第3章

适配器（Adapter）模式

客户端（Client）就是需要调用我们代码的对象。通常，在代码已经存在的情况下编写客户端代码，开发人员可以采取模拟客户端的方式调用我们提供的接口对象。然而，客户端代码也可能与你的代码单独进行开发。例如，设计的火箭仿真程序会使用你所提供的火箭信息，但是对于火箭应该拥有怎样的行为，仿真器也会拥有自己的定义。在这样的情况下，会发现现有的类虽然提供了客户端需要的服务，却被定义为不同的方法名。这时，我们就需要运用适配器（Adapter）模式。

适配器模式的意图在于，使用不同接口的类所提供的服务为客户端提供它所期望的接口。

接口适配

当我们需要适配现有代码时，可能会发现客户端开发人员已经事先考虑到这种情形。开发人员为客户端使用的服务提供了接口，如图 3.1 所示。**RequiredInterface** 接口声明了 **Client** 类所要调用的 **requiredMethod()** 方法。在 **ExistingClass** 类中，则定义了 **usefulMethod()** 方法，它是 **Client** 类需要的实现。若要对 **ExistingClass** 类进行适配，满足客户端对象的需要，就可以编写一个继承自 **ExistingClass**，并同时实现 **RequiredInterface** 接口的类，通过重写 **requiredMethod()** 方法将客户端的请求委派给 **usefulMethod()** 方法。

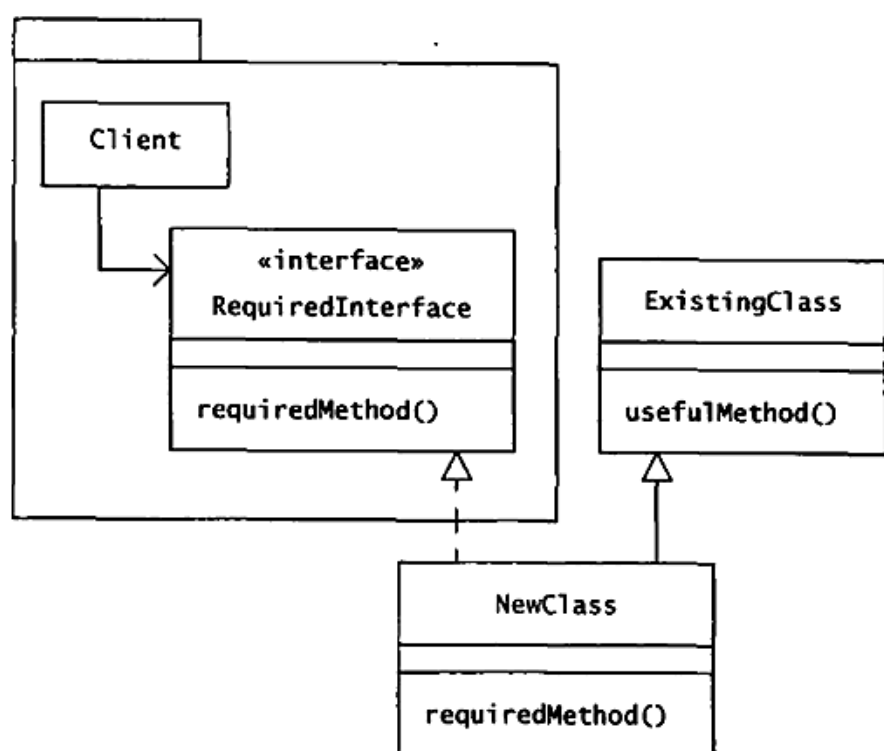


图 3.1 当客户端代码的开发人员在考虑如何满足客户端需求时，可以考虑通过适配现有代码来实现接口

图 3.1 中的 `NewClass` 类就是适配器模式的一个例子。该类的实例同时也是 `RequiredInterface` 的实例。换言之，`NewClass` 类满足了客户端的需求。

为了更具体地说明，假定我们为 Oozinoz 公司开发仿真火箭的飞行与实时控制程序。在仿真功能包中，包含了一个事件仿真器，它能够探测到多个火箭启动时产生的影响。这是一个指定了火箭行为的接口，图 3.2 展示了该功能包。

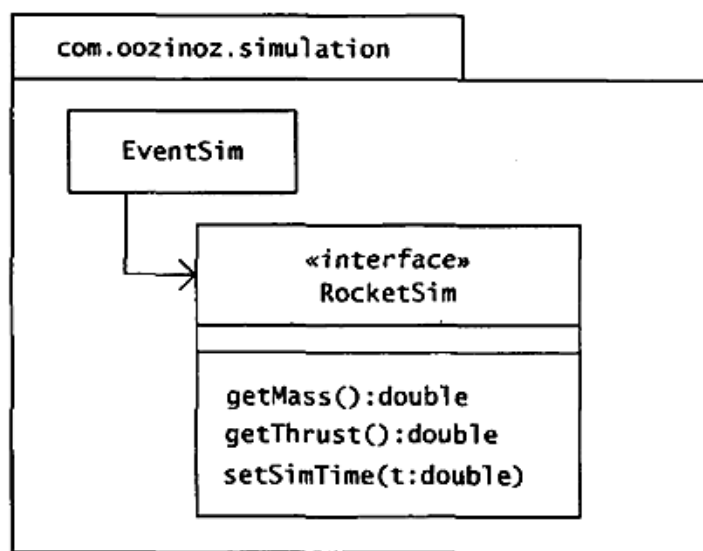


图 3.2 仿真功能包清晰地定义了火箭仿真飞行的需求

假设在 Oozinoz 公司, 需要将 `PhysicalRocket` 类放到仿真功能中。该类提供的方法类似仿真器需要的功能行为。此时, 就可以运用适配器模式, 创建 `PhysicalRocket` 的子类, 并同时实现 `RocketSim` 接口。图 3.3 展示了这一设计的部分内容。

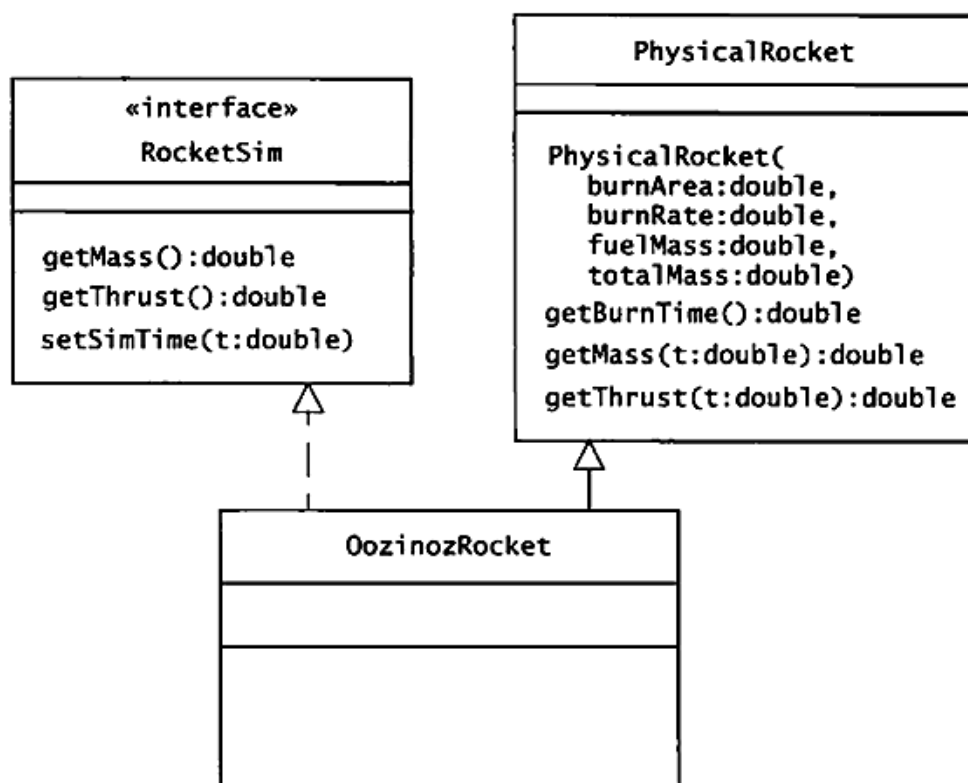


图 3.3 本图展示了设计完成后, 类的设计对 `Rocket` 类进行适配, 以满足 `RocketSim` 接口的需要

`PhysicalRocket` 类拥有仿真器需要的信息, 但它的方法并不完全匹配 `RocketSim` 接口中声明的仿真功能。主要的差异在于仿真器保留了一个内部时钟, 它不时地会调用 `setSimTime()` 方法更新仿真对象。若要适配 `PhysicalRocket` 类以满足仿真器的需求, `OozinozRocket` 对象可以维持一个实例变量, 用来传递 `PhysicalRocket` 类需要的方法。

挑战 3.1

完成图 3.3 中的类图, 展现 `OozinozRocket` 类的设计, 它将让 `PhysicalRocket` 对象作为 `RocketSim` 对象参与到仿真行为中。假定你无法修改 `RocketSim` 与 `PhysicalRocket` 类。

答案参见第 298 页

`PhysicalRocket` 类的代码稍显复杂, 因为它包含了 Oozinoz 用来模拟火箭行为的物理逻辑。

辑。然而，这正是我们希望重用的部分。OozinozRocket 适配器类只是简单地将调用转为使用超类拥有的方法。这个新的子类的代码如下所示：

```
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() {
        // 挑战!
    }

    public double getThrust() {
        // 挑战!
    }

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

挑战 3.2

完成包括 `getMass()` 与 `getThrust()` 方法的 `OozinozRocket` 类的代码。

答案参见第 299 页

当客户端在接口中定义了它所期待的行为时，就可以运用适配器模式，提供一个实现该接口的类，并同时令其成为现有类的接口。倘若没有定义客户端期待的接口，也可以运用适配器模式，但必须使用“对象适配器”。

类与对象适配器

图 3.1 与图 3.3 的设计属于类的适配器，通过子类进行适配。在类的适配器中，新的适配类实现了需要的接口，并继承自现有的类。当你需要适配的一组方法并非被定义在接口中时，这种方式就不奏效了。此时就可以创建一个对象适配器，它使用了委派而非继承。图 3.4 展现了这样的设计（可以对比之前的类图）。

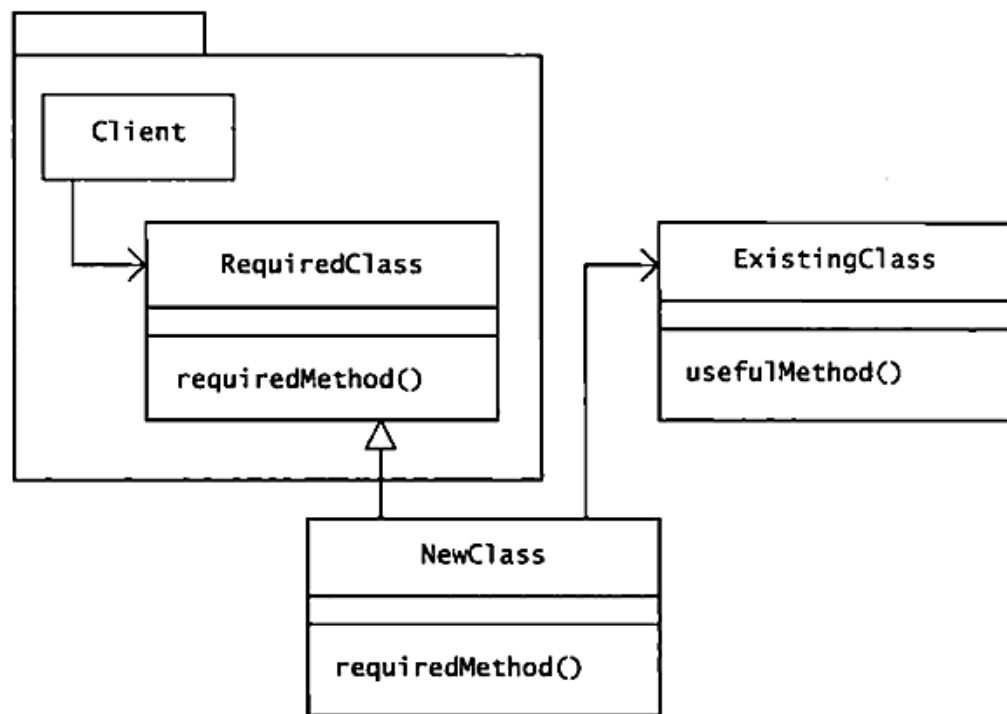


图 3.4 通过继承你所需要的类，可以创建一个对象适配器，利用现有类的实例对象，满足所需方法

图 3.4 中的 `NewClass` 类是适配器的一个例子。该类的实例同时也是 `RequiredClass` 类的实例。换言之，`NewClass` 类满足了客户端的需要。`NewClass` 类通过使用 `ExistingClass` 实例对象，可以将 `ExistingClass` 类适配为符合客户端的需要。

一个更为具体的例子是仿真程序包，它直接与 `skyrocket` 类协作，而没有指定接口去定义仿真系统需要的行为。图 3.5 展示了该类的设计。

`skyrocket` 类使用了火箭的基本模型。例如，类假设火箭要在燃料烧尽之后才会坠毁。假设你希望添加一些更为复杂的物理模型，而该模型由 Oozinoz 系统中的 `PhysicalRocket` 类使用。为了适配 `PhysicalRocket` 类以满足仿真系统，需要创建 `OozinozSkyrocket` 类作为对象适配器，它继承自 `skyrocket`，同时使用了 `PhysicalRocket` 对象，如图 3.6 所示。

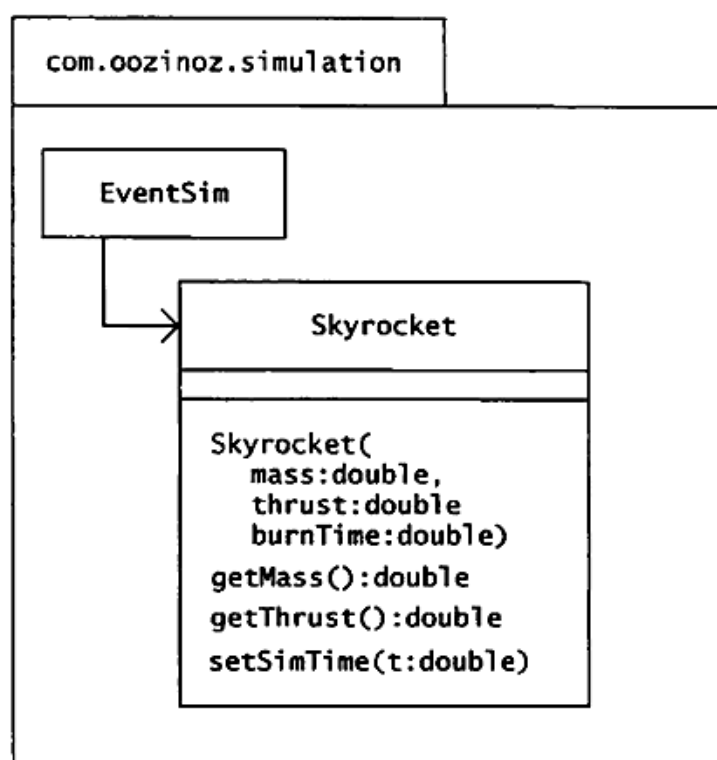


图 3.5 在这个替代设计中，com.oozinoz.simulation 包并没有指定对火箭进行建模所需的接口

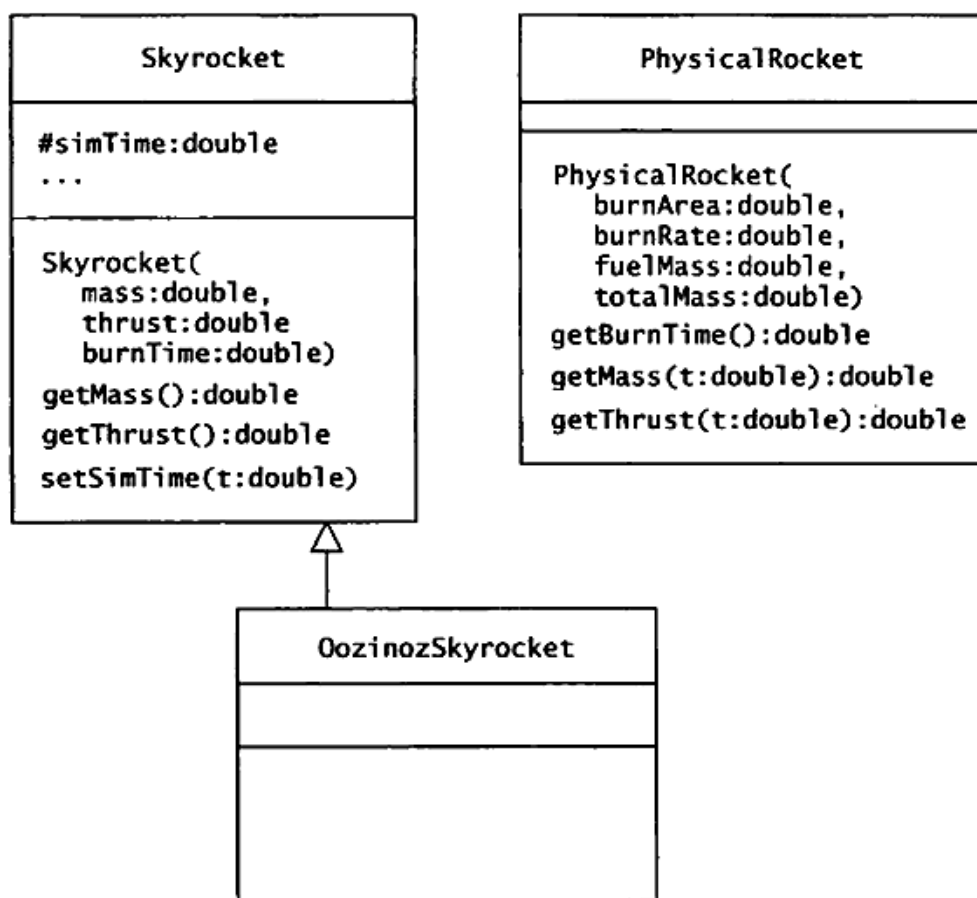


图 3.6 完成该类图使其能够体现对象适配器的设计，将现有的类转换，以满足拥有 Skyrocket 对象的客户端需求

作为一个对象适配器，OozinozSkyrocket 类继承自 Skyrocket，而非 PhysicalRocket。当仿真程序客户端需要 Skyrocket 对象时，可以令 OozinozSkyrocket 对象代替它。通过让 simTime 变量成为受保护的，Skyrocket 类就能够支持它的子类化。

挑战 3.3

完成图 3.6 所示的类图，使得 OozinozRocket 对象支持 Skyrocket 对象。

答案参见第 300 页

OozinozSkyrocket 类的代码如下所示：

```
package com.oozinoz.firework;
import com.oozinoz.simulation.*;

public class OozinozSkyrocket extends Skyrocket {
    private PhysicalRocket rocket;
    public OozinozSkyrocket(PhysicalRocket r) {
        super(
            r.getMass(0),
            r.getThrust(0),
            r.getBurnTime());
        rocket = r;
    }

    public double getMass() {
        return rocket.getMass(simTime);
    }

    public double getThrust() {
        return rocket.getThrust(simTime);
    }
}
```

OozinozSkyrocket 类可以为需要 Skyrocket 对象的仿真程序包提供 OozinozSkyrocket 类型的对象。总体而言，对象适配器在一定程度上解决了这一问题，即将对象适配为没有明确定义的接口。

与实现 RocketSim 接口相比，运用了对象适配器的 Skyrocket 类存在更大的风险。但我

们却不应该吹毛求疵，因为它仅仅是没有将方法标记为 `final`，使得我们不能防止子类去重写它们。

挑战 3.4

分析为何 `OozinozSkyrocket` 类使用的对象适配器设计要比类的适配器方式更加脆弱。

答案参见第 301 页

JTable 对数据的适配

在表中显示数据时，通常会运用对象适配器。Swing 提供了 `JTable` 控件用以显示表。显然，该控件的设计者并不知道你所要显示的数据。它并没有硬将数据接口塞到控件中，而是定义了 `TableModel` 接口。`JTable` 的实现使用了该接口。然后，你可以提供一个适配器，将数据转换为 `TableModel`，如图 3.7 所示。

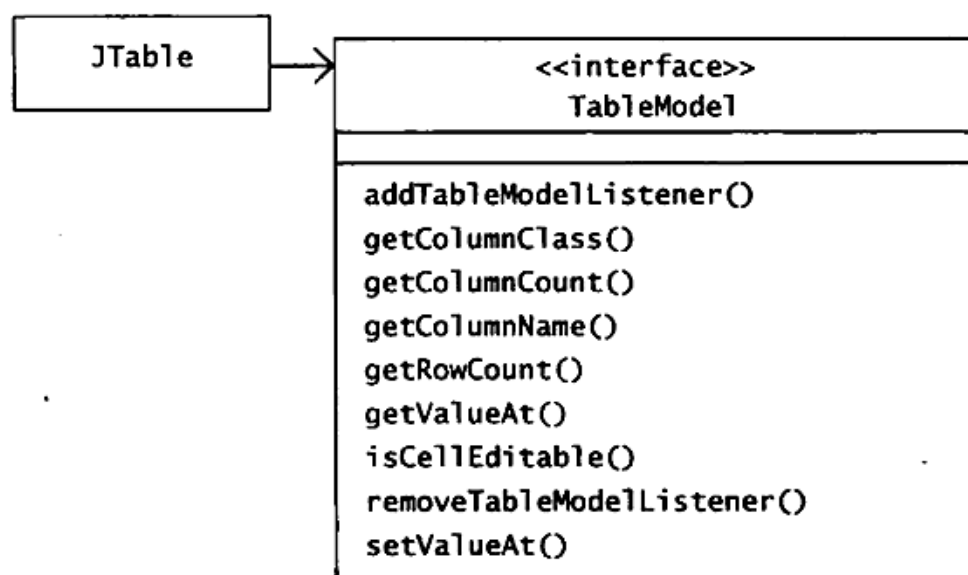


图 3.7 Swing 组件中的 `JTable` 类可以将实现了 `TableModel` 的数据显示到图形界面的表中

`TableModel` 定义了许多给出默认实现的方法。幸运的是，JDK 提供了抽象类的机制，它可以为 `TableModel` 中与特定领域逻辑有关的方法提供默认实现。图 3.8 展现了该类的设计。

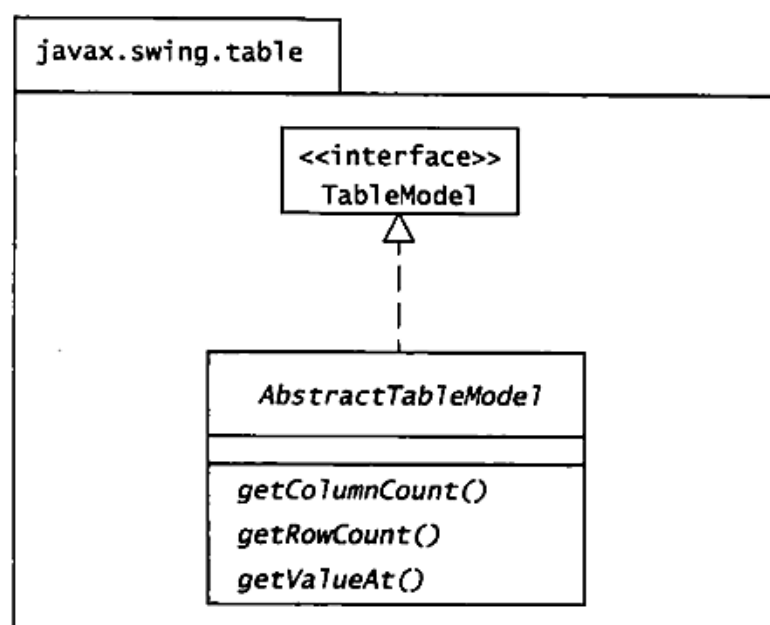


图 3.8 AbstractTableModel 类提供了定义在 TableModel 接口中大多数方法的实现

假设需要使用 Swing 的用户界面在表中显示几个火箭。如图 3.9 所示，可以创建 `RocketTableModel` 类将这组火箭适配为 `TableModel` 所期待的接口。

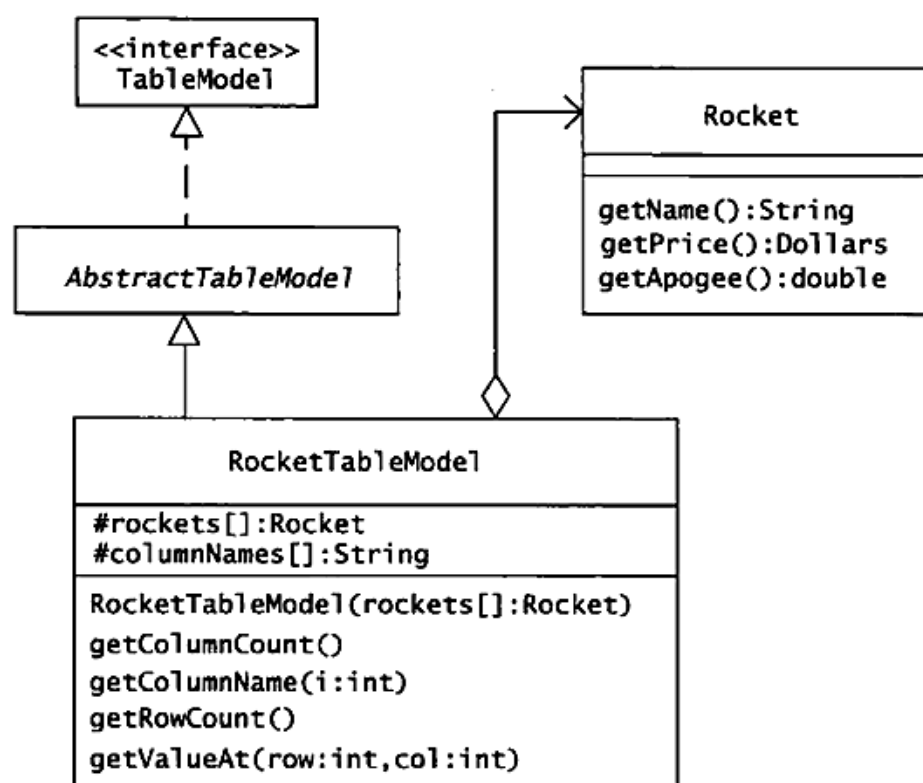


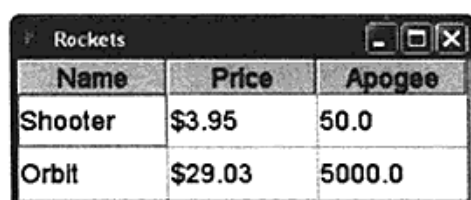
图 3.9 RocketTableModel 类将 TableModel 接口适配为 Oozinoz 领域对象 Rocket 类

`RocketTableModel` 类必须继承自 `AbstractTableModel`，因为后者是类而不是接口。无

论何时，只要我们需要使用的抽象类对要适配的接口提供支持，就必须使用对象适配器方式^{译注1}。不能使用类适配器的第二个原因是 `RocketTableModel` 并非 `Rocket` 的子类型。当适配类必须从多个对象处获得相关信息时，通常就应该使用对象的适配器。

注意区分：类的适配器继承自现有的类，同时实现目标接口；对象适配器继承自目标类，同时引用现有的类。

一旦创建了 `RocketTableModel` 类，就很容易在 Swing 的 `JTable` 对象中显示相关信息，如图 3.10 所示。



Name	Price	Apogee
Shooter	\$3.95	50.0
Orbit	\$29.03	5000.0

图 3.10 填充了火箭数据信息的 `JTable` 实例

```
package app.adapter;  
import javax.swing.table.*;  
import com.oozinoz.firework.Rocket;  
  
public class RocketTableModel extends AbstractTableModel {  
    protected Rocket[] rockets;  
    protected String[] columnNames =  
        new String[] { "Name", "Price", "Apogee" };  
  
    public RocketTableModel(Rocket[] rockets) {  
        this.rockets = rockets;  
    }  
  
    public int getColumnCount() {  
        // 挑战!  
    }  
}
```

译注1：这里的假设前提是适配器对象必须要使用抽象类，因为抽象类提供了适配器对象需要的部分实现，而该抽象类又实现了客户端期待的接口，这就要求适配器对象必须继承抽象类。而同时，适配器对象还需要重用第三方对象（即目标对象）。重用的方式只能是继承或组合方式。由于 Java 是单继承语言，在已经继承了抽象类的情况下，无法再使用类的继承方式，因此只能将目标对象（这里即 `Rocket` 对象）以组合的方式传给适配器对象。

```
public String getColumnName(int i) {  
    // 挑战!  
}  
  
public int getRowCount() {  
    // 挑战!  
}  
  
public Object getValueAt(int row, int col) {  
    // 挑战!  
}  
}
```

挑战 3.5

完成 `RocketTableModel` 方法中的代码,使其能够将 `Rocket` 对象数组适配为 `TableModel`。

答案参见第 301 页

要实现图 3.10 所示的显示结果,可以创建一组 `rocket` 对象,并将其放到数组中,然后再根据该数组创建 `RocketTableModel` 实例,并使用 `Swing` 的类显示表的信息。`ShowRocketTable` 类给出了这一例子的实现:

```
package app.adapter;  
  
import java.awt.Component;  
import java.awt.Font;  
  
import javax.swing.*;  
  
import com.oozinoz.firework.Rocket;  
import com.oozinoz.utility.Dollars;  
  
public class ShowRocketTable {  
    public static void main(String[] args) {  
        setFonts();  
        JTable table = new JTable(getRocketTable());  
        table.setRowHeight(36);  
    }  
}
```

```
        JScrollPane pane = new JScrollPane(table);
        pane.setPreferredSize(
            new java.awt.Dimension(300, 100));
        display(pane, " Rockets");
    }

    public static void display(Component c, String title) {
        JFrame frame = new JFrame(title);
        frame.getContentPane().add(c);
        frame.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }

    private static RocketTableModel getRocketTable() {
        Rocket r1 = new Rocket(
            "Shooter", 1.0, new Dollars(3.95), 50.0, 4.5);
        Rocket r2 = new Rocket(
            "Orbit", 2.0, new Dollars(29.03), 5000, 3.2);
        return new RocketTableModel(new Rocket[] { r1, r2 });
    }

    private static void setFonts() {
        Font font = new Font("Dialog", Font.PLAIN, 18);
        UIManager.put("Table.font", font);
        UIManager.put("TableHeader.font", font);
    }
}
```

只需要不到 20 行代码, `ShowRocketTable` 就实现了在图形化用户界面框架中生成表组件的功能。如果不使用适配器模式, 可能需要上千行代码。`JTable` 类几乎可以处理显示表数据的各种功能, 但它却无法事先知道你所要显示的数据。若要提供它所需要的数据, 就应该求助于适配器模式。为了使用 `JTable`, 可以实现 `JTable` 所期待的 `TableModel` 接口, 提供希望显示的数据。

识别适配器

在第 2 章, 我们已经分析了 `WindowAdapter` 类的价值所在。图 3.11 所示的 `MouseAdapter`

类，则是另一个范例，它可以为接口所需的方法提供桩实现。

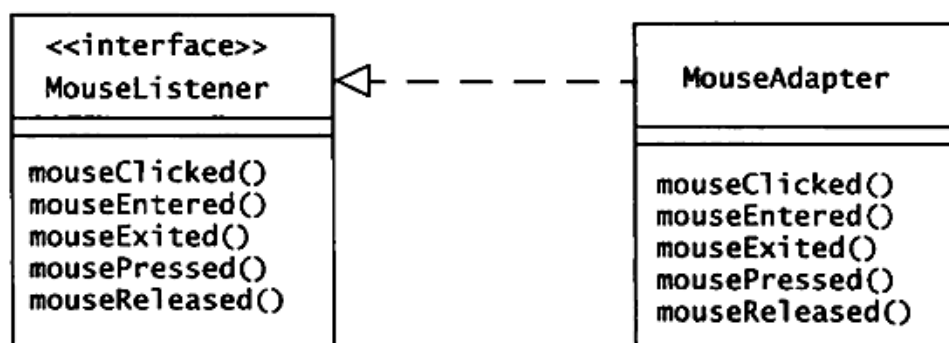


图 3.11 `MouseAdapter` 类可以为 `MouseListener` 接口提供需要的桩实现

挑战 3.6

在使用 `MouseAdapter` 类时，你是否运用了适配器模式？请阐释其原理（如果没有运用，请给出理由）。

答案参见第 302 页

小结

适配器模式使我们可以重用现有的类，以满足客户端的需要。当客户端通过接口表达其需求时，通常可以创建一个实现了该接口的新类，同时使该类继承自现有类。这种方式即类的适配器，它能够将客户端的调用转换为对现有类方法的调用。

当客户端没有指定它所需要的接口时，你就可以使用适配器模式。可能需要创建一个新的客户端子类，它将使用现有类的实例。这种方式通过创建一个对象适配器，将客户端的调用指向现有类的实例。如果我们不需要（或许不能）重写客户端可能调用的方法时，这种方式可能存在一定的危险性。

Swing 中的 `JTable` 组件是运用适配器模式的一个绝佳范例。通过定义 `TableModel` 接口，`JTable` 组件将客户端需要的表信息存储到自身对象中。通过编写一个适配器对象，轻易就可以让一个领域对象满足表数据的需求，例如 `Rocket` 类。

若要使用 `JTable`，需要定义一个对象适配器，可以将调用委派给现有类的实例对象。对于 `JTable`，有两个原因不能使用类的适配器。首先，需要创建一个表的适配器作为 `AbstractTableModel` 的子类，这样就无法再继承自现有类了。其次，`JTable` 需要一组对象，而对象适配器更适合需要将多个对象的信息进行适配的情形。

在设计自己的系统时，需要充分考虑功能的强大与灵活性。这样你和其他程序员都可以从适配器模式架构的使用中获益。