3

ADAPTER

AN OBJECT IS A **client** if it needs to call your code. In some cases, client code will be written after your code exists and the developer can mold the client to use the interfaces of the objects that you provide. In other cases, clients may be developed independently of your code. For example, a rocket-simulation program might be designed to use rocket information that you supply, but such a simulation will have its own definition of how a rocket should behave. In such circumstances, you may find that an existing class performs the services that a client needs but has different method names. In this situation, you can apply the ADAPTER pattern.

The intent of ADAPTER is to provide the interface that a client expects while using the services of a class with a different interface.

Adapting to an Interface

When you need to adapt your code, you may find that the client developer planned well for such circumstances. This is evident when the developer provides an interface that defines the services that the client code needs, as the example in Figure 3.1 shows. A client class makes calls to a `requiredMethod()` method that is declared in an interface. You may have found an existing class with a method with a name such as `usefulMethod()` that can fulfill the client's needs. You can adapt the existing class to meet the client's needs by writing a class that extends `ExistingClass`, implements `RequiredInterface`, and overrides `requiredMethod()` so that it delegates its requests to `usefulMethod()`.

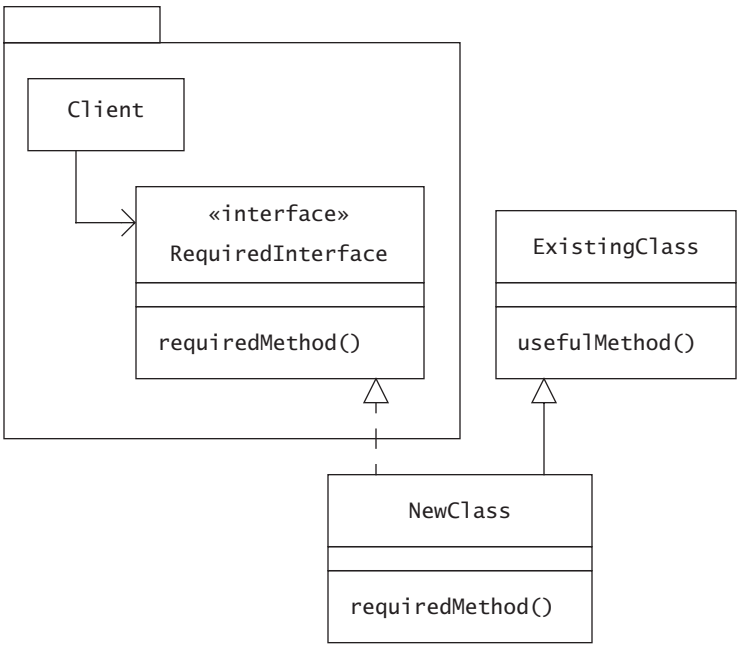

FIGURE 3.1 When a developer of client code thoughtfully defines the client's needs, you may be able to fulfill the interface by adapting existing code.

The `NewClass` class in Figure 3.1 is an example of ADAPTER. An instance of this class is an instance of `RequiredInterface`. In other words, the `NewClass` class meets the needs of the client.

For a more concrete example, suppose that you are working with a package that simulates the flight and timing of rockets such as those you manufacture at Oozinoz. The simulation package includes an event simulator that explores the effects of launching several rockets, along with an interface that specifies a rocket's behavior. Figure 3.2 shows this package.

Suppose that at Oozinoz, you have a `PhysicalRocket` class that you want to plug into the simulation. This class has methods that supply, approximately, the behavior that the simulator needs. In this situation, you can apply ADAPTER, creating a subclass of `PhysicalRocket` that implements the `RocketSim` interface. Figure 3.3 partially shows this design.

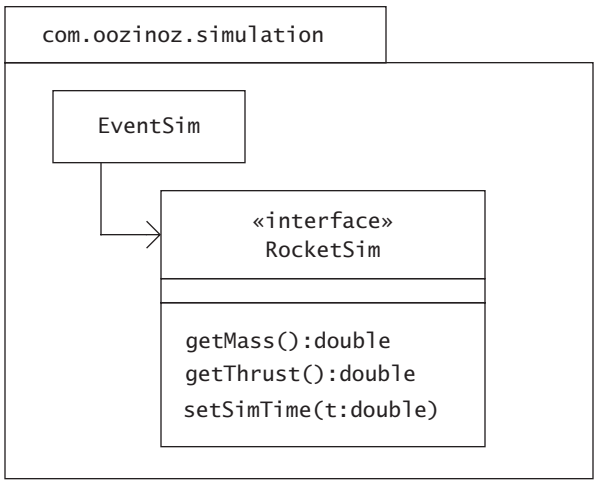

FIGURE 3.2 The `Simulation` package clearly defines its requirements for simulating the flight of a rocket.

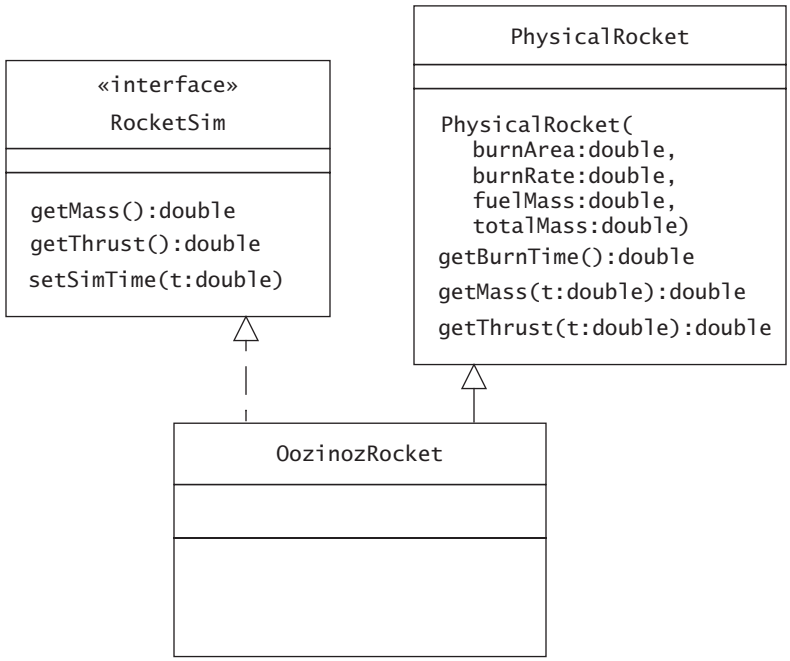

FIGURE 3.3 When completed, this diagram will show the design of a class that adapts the `Rocket` class to meet the needs of the `RocketSim` interface.

The PhysicalRocket class has the information that the simulator needs, but its methods do not exactly match those that the simulation declares in the RocketSim interface. Most of the differences occur because the simulator keeps an internal clock and occasionally updates simulated objects by calling a setSimTime() method. To adapt the PhysicalRocket class to meet the simulator's needs, an OozinozRocket object can maintain a time instance variable that it can pass to the methods of the PhysicalRocket class as needed.

CHALLENGE 3.1

Complete the class diagram in Figure 3.3 to show the design of an OozinozRocket class that lets a PhysicalRocket object participate in a simulation as a RocketSim object. Assume that you can't alter either RocketSim or PhysicalRocket.

A solution appears on page 348.

The code for PhysicalRocket is somewhat complex, as it embodies the physics that Oozinoz uses to model a rocket. However, that is exactly the logic that we want to reuse. The OozinozRocket adapter class simply translates calls to use its superclass's methods. The code for this new subclass will look something like this:

```
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() {
        // Challenge!
    }
```

```
    public double getThrust() {
        // Challenge!
    }

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

CHALLENGE 3.2

Complete the code for the `OozinozRocket` class, including methods `getMass()` and `getThrust()`.

A solution appears on page 349.

When a client defines its expectations in an interface, you can apply ADAPTER by supplying a class that implements that interface and that subclasses an existing class. You may also be able to apply ADAPTER even if no interface exists to define a client's expectations. In this situation, you must use an "object adapter."

Class and Object Adapters

The designs in Figures 3.1 and 3.3 are *class adapters* that adapt through subclassing. In a class adapter design, the new adapter class implements the desired interface and subclasses an existing class. This approach will not always work, notably when the set of methods that you need to adapt is not specified in an interface. In such a case, you can create an **object adapter**—an adapter that uses delegation rather than subclassing. Figure 3.4 shows this design. (Compare this to the earlier diagrams.)

The `NewClass` class in Figure 3.4 is an example of ADAPTER. An instance of this class is an instance of the `RequiredClass` class. In other words, the `NewClass` class meets the needs of the client. The `NewClass` class can adapt the `ExistingClass` class to meet the client's needs by using an instance of `ExistingClass`.

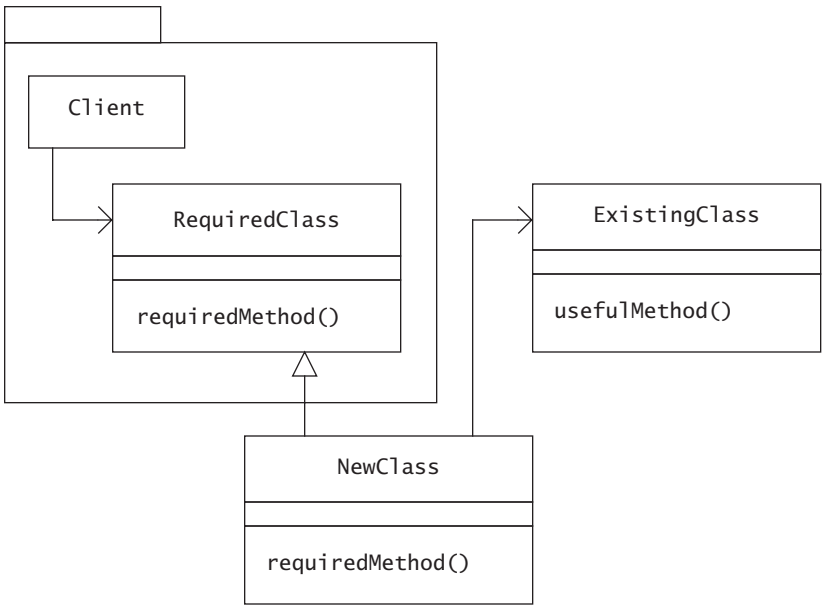

FIGURE 3.4 You can create an object adapter by subclassing the class that you need, fulfilling the required methods by relying on an object of an existing class.

For a more concrete example, suppose that the simulation package worked directly with a `Skyrocket` class, without specifying an interface to define the behaviors that the simulation needs. Figure 3.5 shows this class.

The `Skyrocket` class uses a fairly primitive model of the physics of a rocket. For example, the class assumes that the rocket is entirely consumed as its fuel burns. Suppose that you want to apply the more sophisticated physical model that the Oozinoz `PhysicalRocket` class uses. To adapt the logic in the `PhysicalRocket` class to the needs of the simulation, you can create an `OozinozSkyrocket` class as an object adapter that subclasses `Skyrocket` and that uses a `PhysicalRocket` object, as Figure 3.6 shows.

As an object adapter, the `OozinozSkyrocket` class subclasses from `Skyrocket`, not `PhysicalRocket`. This will allow an `OozinozSkyrocket` object to serve as a substitute wherever the simulation client needs a `Skyrocket` object. The `Skyrocket` class supports subclassing by making its `simTime` variable `protected`.

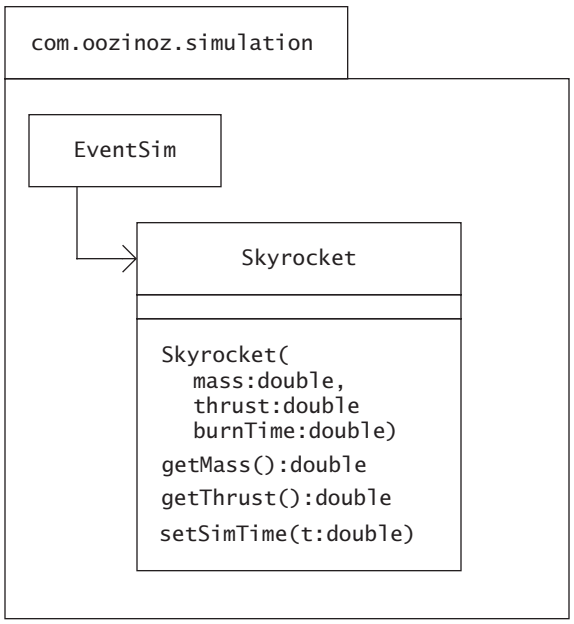

FIGURE 3.5 In this alternative design, the `com.oozinoz.simulation` package does not specify the interface it needs for modeling a rocket.

CHALLENGE 3.3

Complete the class diagram in Figure 3.6 to show a design that allows `OozinozRocket` objects to serve as `Skyrocket` objects.

A solution appears on page 350.

The code for the `OozinozSkyrocket` class might be as follows:

```
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);
    }
}
```

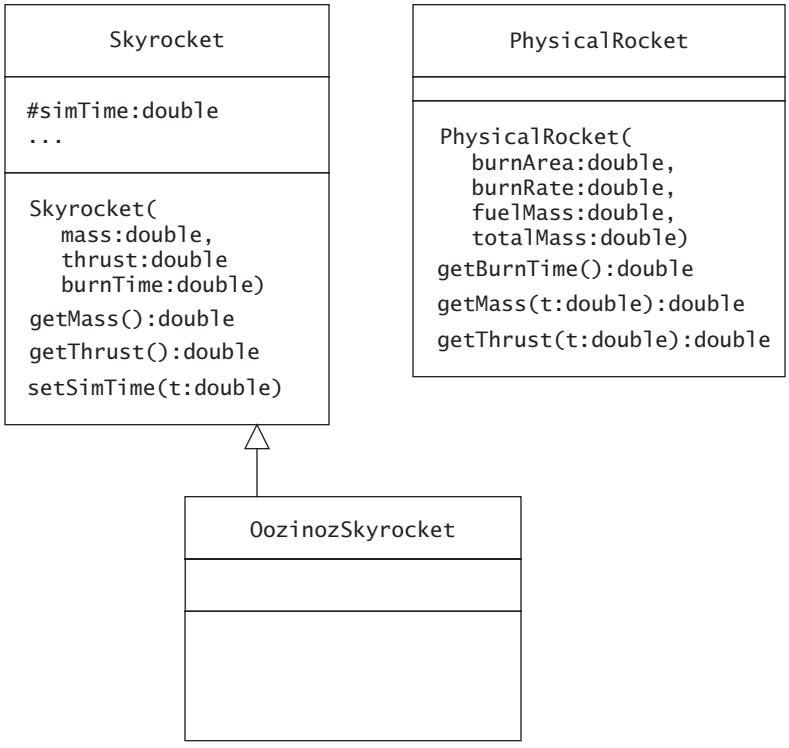

FIGURE 3.6 When completed, this diagram will show an object adapter design that uses information from an existing class to meet the needs that a client has of a `Skyrocket` object.

The `OozinozSkyrocket` class lets you supply an `OozinozSkyrocket` object anywhere that the simulation package requires a `Skyrocket` object. In general, object adapters partially overcome the problem of adapting an object to an interface that is not expressly defined.

CHALLENGE 3.4

Name one reason why the object adapter design that `OozinozSkyrocket` class uses may be more fragile than a class adapter approach.

Solutions appear on page 350.

The object adapter for the `Skyrocket` class is a riskier design than the class adapter that implements the `RocketSim` interface. But we should not complain too much. At least no methods were marked `final`, which would have prevented us from overriding them.

Adapting Data for a JTable

A common example of an object adapter comes when you want to display data in a table. Swing provides the `JTable` widget to display tables. Obviously, the designers of this widget didn't know what data you would want to display. Rather than hard-code some data structure into the widget, they provided an interface, `TableModel`. `JTable` does its work in terms of this interface. You then provide an adapter that makes your data conform to the `TableModel`. See Figure 3.7.

Many of the methods in `TableModel` suggest the possibility of a default implementation. Fortunately, the **Java Development Kit** (JDK) supplies an abstract class that provides default implementations of all but the most domain-specific methods in `TableModel`. Figure 3.8 shows this class.

Suppose that you want to show a few rockets in a table, using a Swing user interface. As Figure 3.9 shows, you can create a `RocketTable-Model` class that adapts an array of rockets to the interface that `TableModel` expects.

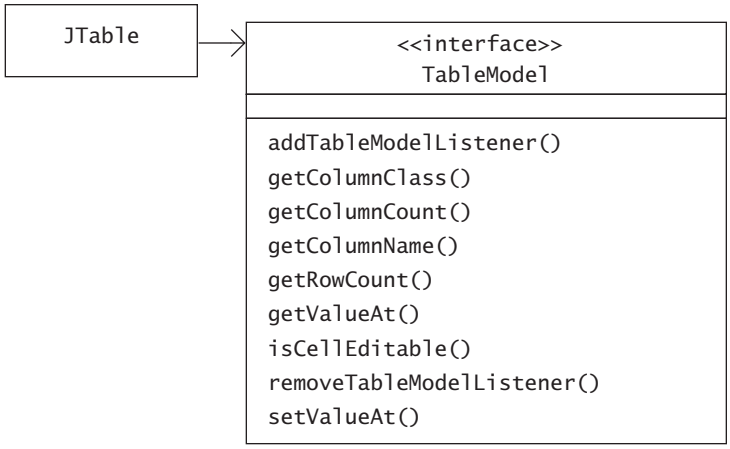

FIGURE 3.7 The `JTable` class is a Swing component that displays data from an implementation of `TableModel` into a GUI table.

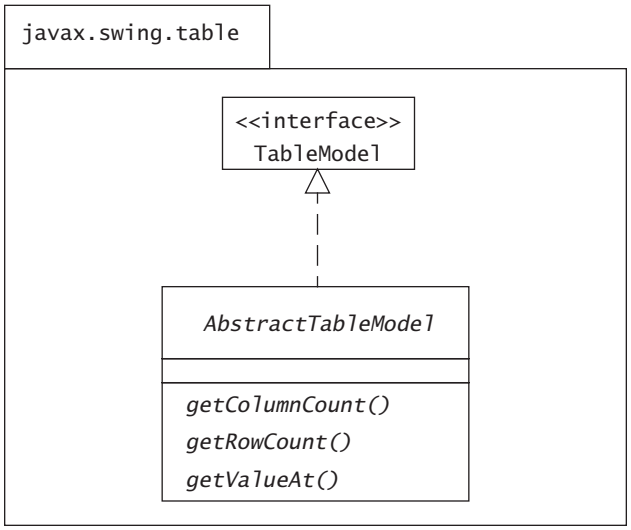

FIGURE 3.8 The `AbstractTableModel` class provides defaults for all but a few of the methods in `TableModel`.

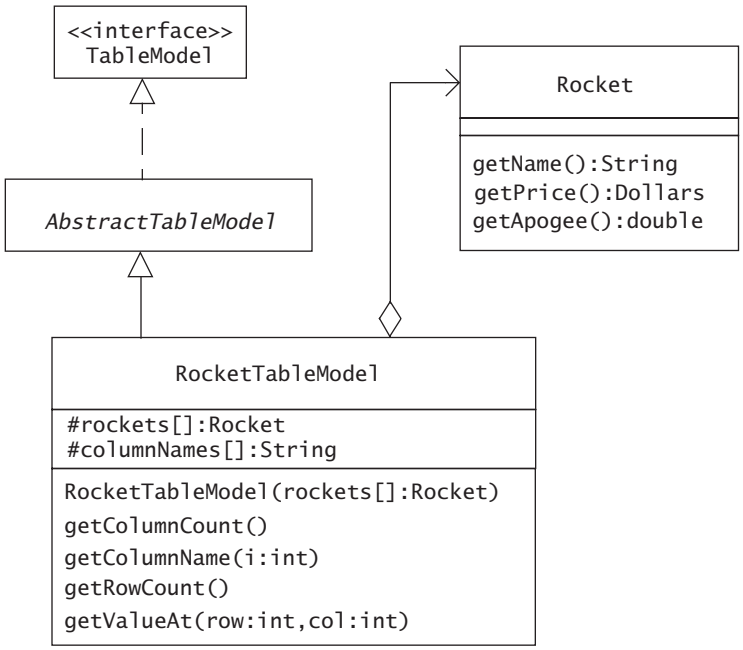

FIGURE 3.9 The `RocketTableModel` class adapts the `TableModel` interface to the `Rocket` class from the Oozinoz domain.

The `RocketTableModel` class has to subclass `AbstractTableModel` because `AbstractTableModel` is a class, not an interface. Whenever the interface you are adapting to is supported with an abstract class that you want to use, you must use an object adapter. In this instance, a second reason you do not want to use a class adapter is that a `RocketTableModel` is not a kind or a subtype of `Rocket`. When an adapter class must draw on information from more than one object, you will usually implement it as an object adapter.

Note the difference: A class adapter subclasses an existing class and implements a target interface; an object adapter subclasses a target class and delegates to an existing class.

Once you create the `RocketTableModel` class, you can easily display information about rockets in a Swing `JTable` object, as Figure 3.10 shows.

FIGURE 3.10 An instance of `JTable` fueled with rocket data

```
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() {
        // Challenge!
    }

    public String getColumnName(int i) {
        // Challenge!
    }

    public int getRowCount() {
        // Challenge!
    }

    public Object getValueAt(int row, int col) {
        // Challenge!
    }
}
```

CHALLENGE 3.5

Fill in the code for the `RocketTableModel` methods that adapt an array of `Rocket` objects to serve as a `TableModel`.

A solution appears on page 352.

To launch the display that Figure 3.10 shows, you can create a couple of example rocket objects, place them in an array, construct an instance of `RocketTableModel` from the array, and use Swing classes to display the table. The `ShowRocketTable` class provides an example:

```
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);
    }
}
```

With fewer than 20 statements of its own, the `ShowRocketTable` class sits above thousands of statements that collaborate to produce a table component within a **graphical user interface** (GUI) framework. The `JTable` class can handle nearly every aspect of displaying a table but can't know in advance what data you will want to present. To let you supply the data it needs, the `JTable` class sets you up to apply ADAPTER. To use `JTable`, you implement the `TableModel` interface that `JTable` expects, along with a class that provides the data you want to display.

Identifying Adapters

In Chapter 2, you explained the value of the `WindowAdapter` class. The `MouseAdapter` class, as Figure 3.11 shows, is another example of a class that stubs out the methods required by an interface.

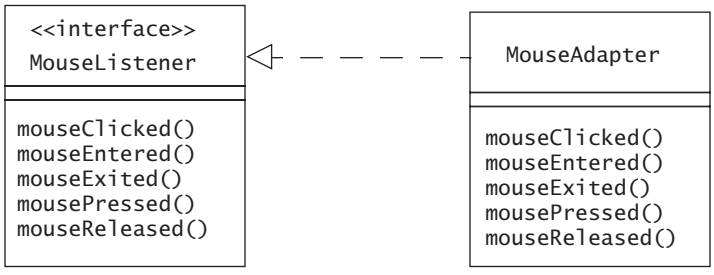

FIGURE 3.11 The `MouseAdapter` class stubs out the requirements of the `MouseListener` interface.

CHALLENGE 3.6

Are you applying the ADAPTER pattern when you use the `MouseAdapter` class? Explain how (or why not).

A solution appears on page 353.

Summary

The ADAPTER pattern lets you use an existing class to meet a client class's needs. When a client specifies its requirements in an interface, you can usually create a new class that implements the interface and subclasses an existing class. This approach creates a **class adapter** that translates a client's calls into calls to the existing class's methods.

When a client does not specify the interface it requires, you may still be able to apply ADAPTER, creating a new client subclass that uses an instance of the existing class. This approach creates an *object adapter* that forwards a client's calls to an instance of the existing class. This approach can be dangerous, especially if you don't (or perhaps can't) override all the methods that the client might call.

The `JTable` component in Swing is a good example of a class whose developers applied the ADAPTER pattern. A `JTable` component sets itself up as a client that needs table information as defined by the `TableModel` interface. This makes it easy for you to write an adapter that feeds the table data from domain objects, such as instances of the `Rocket` class.

To use `JTable`, you frequently write an object adapter that delegates calls to instances of an existing class. Two aspects of `JTable` make it less likely that you'll use a class adapter. First, you will usually create your table adapter as a subclass of `AbstractTableModel`, so you can't also subclass your existing class. Second, the `JTable` class requires a collection of objects, and an object adapter is best suited to adapt information from more than one object.

When you design your own systems, consider the power and flexibility that you and other developers can derive from an architecture that uses ADAPTER to advantage.