Chapter 6

Fitting Round Pegs into Square Holes with the Adapter and Facade Patterns

In This Chapter

- Using the Adapter pattern
- Creating adapters
- Adapting `Ace` objects as `Acme` objects
- Handling adapter issues
- Using the Facade pattern

Sometimes, objects just don't fit together as they should. A class may have changed, or an object turns out to be just too difficult to work with. This chapter comes to the rescue by covering two design patterns: the Adapter pattern and the Facade pattern. The Adapter design pattern lets you adapt what an object or class has to offer so that another object or class can make use of it. The Facade design pattern is similar, in that it changes the look of an object, but the goal here is a little different: You use this design pattern to *simplify* the exposed methods of an object or class, making it easier to work with that object or class.

The Adapter Scenario

"Alright," says the MegaGigaCo team leader, entering the room, "hold everything. Management has decreed that we switch our back-end framework to the one sold by the CEO's nephew's company."

"Hmm," says a programmer, "that could be a problem. Our online user interface takes customer orders using software from the Ace company and packages them in objects of the `Ace` class. What type of objects can we pass to the new back end?"

"Only the new `Acme` objects," the team leader says, "not `Ace` objects."

"Uh oh," everyone says. "There go our jobs."

You can see the problem. Currently, the `Ace` objects that are passed to the back end fit right in, as shown in Figure 6-1.

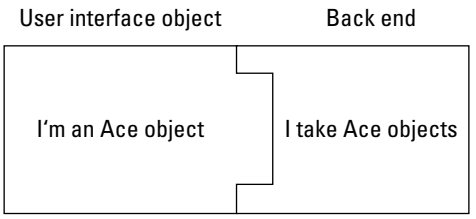

Figure 6-1: These two objects fit together well.

But when the back end is switched to take `Acme` objects (instead of `Ace` objects), the current `Ace` objects created by the user interface won't fit. That scenario looks like Figure 6-2.

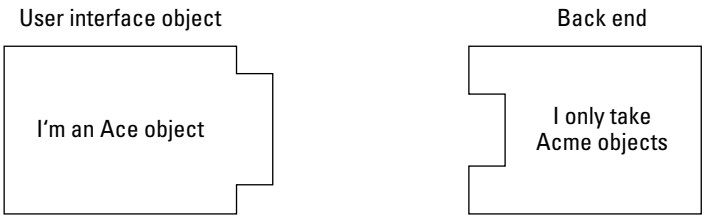

Figure 6-2: The user interface no longer works with the back end.

"I have the solution," you say. Everyone turns to you and you say, "Of course, as a consultant, I'll have to charge a whopper fee on this."

"Anything," the team leader says. "The mortgage folks don't understand about missing payments if I lose my job."

"You need to use the Adapter pattern," you explain. "The Adapter pattern lets you adapt what an object or class exposes to what another object or class expects." You draw the solution on the whiteboard like that shown in Figure 6-3.

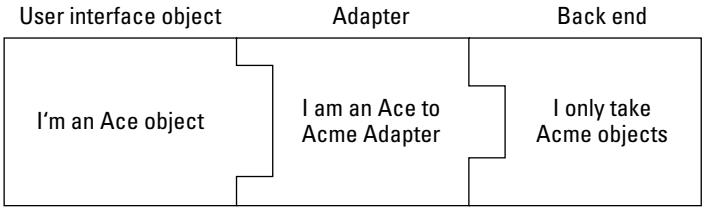

Figure 6-3: The adapter fits the pieces together.

"Ah," says the development team. "We are beginning to understand."

"Fine," you say, "pay me some money."

Fixing Connection Problems with Adapters

The Adapter design pattern lets you fix the interface between objects and classes without having to modify the objects or classes directly. When you're working with store-bought applications, you often can't get inside to alter what one application produces to make it more palatable to another application.

This is particularly important in online development. As more and more companies start going for larger-scale enterprise solutions, they're ditching the smaller corporations' software in favor of soup-to-nuts solutions from the big boys like IBM. And that's a shame because the issue is almost always one of compatibility — the smaller corporation's software can't talk to one or two other components in the whole system. But turning to an expensive solution isn't always necessary. Usually, the problems can be fixed with a small adapter. In other words, letting the big boys win at your expense could be avoided with just a little effort here.

How the Adapter pattern works is best seen in an example. Currently, the MegaGigaCo user interface, which I discuss in previous sections of this chapter, packages user data in objects of the `Ace` class. This class handles customer names with these two methods:

- `setName`
- `getName`

But, as you know, MegaGigaCo is switching to Acme software for the back end, which has to be able to handle customer orders in a different way. The problem is that the Acme back end expects customer orders to be packaged in `Acme` objects. And `Acme` objects use four methods, not two, to handle the customer's name. They are:

- `setFirstName`
- `setLastName`
- `getFirstName`
- `getLastName`

So you need an adapter to make sure that the Acme back end can handle `Ace` objects. This adapter calls the two methods supported by the `Ace` object and extends that into a set of four methods that `Acme` objects usually offer, as shown in Figure 6-4.

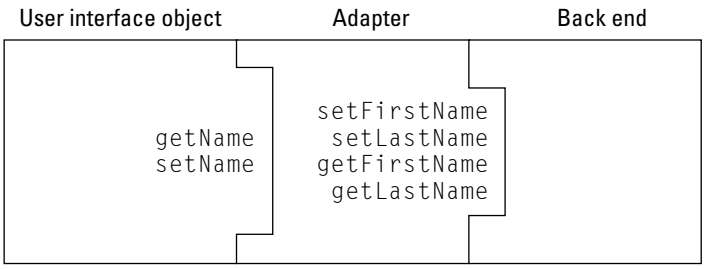

Figure 6-4: The Ace and Acme adapter.

That's the idea and what the Adapter design pattern is all about.

The Gang of Four (GoF) book (*Design Patterns: Elements of Reusable Object-Oriented Software,* 1995, Pearson Education, Inc. Publishing as Pearson Addison Wesley) says the Adapter pattern lets you "Convert the interface of a class into another interface the client expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."

Although the official definition of the Adapter pattern talks about classes, this pattern actually has two variations: one for objects and one for classes. I look at both in this chapter.

You use the Adapter design pattern when you're trying to fit a square peg into a round hole. If what a class or object exposes isn't what you need to end up with, you can add an adapter — much like an electrical outlet adapter for international travel — to give you what you need.

This design pattern is particularly good when you're working with legacy code that can't be changed, while the software that interacts with that code does change.

Now to get down to actually putting the Adapter pattern to work.

Creating Ace objects

Before the CEO's nephew ruined everything for your department, `Ace` objects handled customer names with just two methods: `setName` and `getName` — here's an interface which specifies those two methods:

```
public interface AceInterface
{
  public void setName(String n);
  public String getName();
}
```

The `Ace` objects that came out of the user interface were objects of the `AceClass`, which implemented this interface:

```
public class AceClass implements AceInterface
{
    .
    .
    .
}
```

The two methods, `setName` and `getName`, were simplicity itself to add.

```
public class AceClass implements AceInterface
{
  String name;

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

  public String getName()
  {
    return name;
  }
}
```

That's all you needed when the user interface produced `Ace` objects and the back end consumed `Ace` objects. But now the company is switching to an Acme back end, which consumes `Acme` objects. (Thanks again to that nephew!)

Creating Acme objects

`Acme` objects must handle customer names with four methods: `setFirstName`, `setLastName`, `getFirstName`, and `getLastName`. Here's an interface, `AcmeInterface`, which lists these methods:

```
public interface AcmeInterface
{
  public void setFirstName(String f);
  public void setLastName(String l);
  public String getFirstName();
  public String getLastName();
}
```

`Acme` objects are based on the `Acme` class, which implements the `AcmeInterface`.

```
public class AcmeClass implements AcmeInterface
{
    .
    .
    .
}
```

Here are the four methods this class exposes:

```
public class AcmeClass implements AcmeInterface
{
  String firstName;
  String lastName;

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

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

  public String getFirstName()
  {
```

```
    return firstName;
  }

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

At this point, you've got the `Ace` objects produced by the user interface and the `Acme` objects consumed by the back end. Now you've got to create an adapter that lets you plug `Ace` objects into the Acme back end.

Creating an Ace-to-Acme object adapter

You want to create an adapter to let software that expects an `Acme` object to actually work with an `Ace` object, so you should create an object adapter. Object adapters work by *composition* (see Chapter 2 for more on composition) — the adapter stores the object it's adapting inside itself.

Continuing with the example in this chapter, I name the adapter `AceToAcme Adapter`, and because it has to look like an `Acme` object, it implements the `AcmeInterface` interface.

```
public class AceToAcmeAdapter implements AcmeInterface
{
    .
    .
    .
}
```

This adapter uses object composition to hold the object it's supposed to be adapting, an `AceClass` object. You can pass that object to the adapter's constructor, which will store the `Ace` object.

```
public class AceToAcmeAdapter implements AcmeInterface
{
  AceClass aceObject;

  public AceToAcmeAdapter(AceClass a)
  {
    aceObject = a;
  }
  .
  .
  .
}
```

The difference between `Ace` and `Acme` objects is that `Ace` objects store the customer's name as a single string, while `Acme` objects store the first name and last name separately. To adapt between `Ace` and `Acme` objects, I split the name stored in the `Ace` object passed to the constructor into first and last names. You can recover the customer name from the stored `Ace` object using its `getName` method.

```
public class AceToAcmeAdapter implements AcmeInterface
{
  AceClass aceObject;
  String firstName;
  String lastName;

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

Now you've got the customer's first and last names. To mimic an `Acme` object, you have to implement the `Acme` methods `setFirstName`, `setLastName`, `getFirstName`, and `getLastName`, returning or setting the customer's first and last names as needed. Here's what those methods look like:

```
public class AceToAcmeAdapter implements AcmeInterface
{
  AceClass aceObject;
  String firstName;
  String lastName;

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

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

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

  public String getFirstName()
```

```
  {
    return firstName;
  }

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

Excellent — you've got your adapter. Is it going to work?

Testing the adapter

Throughout this section, you have been adapting `Ace` objects so they look like `Acme` objects. Now it's time to see if the Adapter pattern is working the way you want it to. You can test this with the `TestAdapter.java` test harness, which starts by creating an `Ace` object that contains the customer name Cary Grant.

```
public class TestAdapter
{
  public static void main(String args[])
  {
    AceClass aceObject = new AceClass();

    aceObject.setName("Cary Grant");
      .
      .
      .
}
```

Then you pass this `Ace` object to an `AceToAcmeAdapter` object.

```
public class TestAdapter
{
  public static void main(String args[])
  {
    AceClass aceObject = new AceClass();

    aceObject.setName("Cary Grant");

    AceToAcmeAdapter adapter = new AceToAcmeAdapter(aceObject);
      .
      .
      .
}
```

And you're good to go — you can use the `Acme` methods like `getFirstName` and `getLastName` with no problem.

```
public class TestAdapter
{
  public static void main(String args[])
  {
    AceClass aceObject = new AceClass();

    aceObject.setName("Cary Grant");

    AceToAcmeAdapter adapter = new AceToAcmeAdapter(aceObject);

    System.out.println("Customer's first name: " +
      adapter.getFirstName());
    System.out.println("Customer's last name: " +
      adapter.getLastName());
  }
}
```

Running this code gives you:

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

Just what you'd expect if you were using a bona fide `Acme` object; the calling code need never know it's not dealing with an `Acme` object.

That's how object adapters work. An adapter uses composition to store the object it's supposed to adapt, and when the adapter's methods are called, it translates those calls into something the adapted object can understand and passes the calls on to the adapted object. The code that calls the adapter never needs to know that it's not dealing with the kind of object it thinks it is, but an adapted object instead.

Using object composition to wrap the adapted object is good object-oriented design, as discussed in Chapter 2. And note that if you subclass the adapted object, the adapter wrapper will be able to handle the subclassed objects with minimal changes.

Inheriting class adapters

There's another kind of adapter besides object adapters — class adapters. You explain to the company programmers: "While object adapters use composition to store the object they're adapting, class adapters are designed to use multiple inheritance to merge the adapted class and the class you're adapting it to."

"There's a flaw here," say the company programmers, "if you're working with Java."

"And that is?" you ask.

"Java doesn't support multiple inheritance," they say.

"Right you are," you say. "Which means you can't create true class adapters in Java."

The GoF book uses languages like C++ and Smalltalk when discussing class adapters, but not Java because Java doesn't support multiple inheritance. If it did, you could inherit from both the adapted class and the target class you want to mimic in an adapter class, as you can see in Figure 6-5.

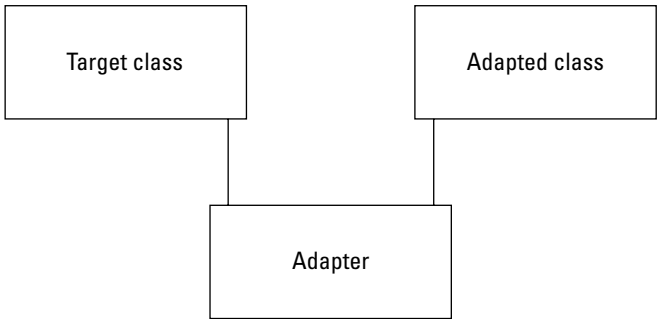

Figure 6-5: Inheriting from both the target and adapted class.

Here's an example using single inheritance in Java, which is as close as you can get to creating class adapters. The user interface team comes to you and says, "We like Java AWT check boxes, and we're not so fond of Swing check boxes."

"Ever thought of entering the 21st century?" you ask.

Ignoring the cheap jab, the UI (user interface) team says, "The problem is that the rest of the user interface uses Swing — and we want to not only stick with AWT check boxes, but also make them look like Swing check boxes to any Swing code that needs to use them."

"That's going to be expensive," you say.

"Do we need to put in a special request to Sun Microsystems?" they ask.

"No, I'll do it. But it's going to cost you plenty."

You write the user interface code for Swing check boxes and determine if a check box is checked using the `isSelected` method. But AWT check boxes don't support `isSelected`; the AWT method is `getState`. So you need an adapter to wrap an SWT check box and adapt the `getState` to `isSelected` instead, as shown in Figure 6-6.

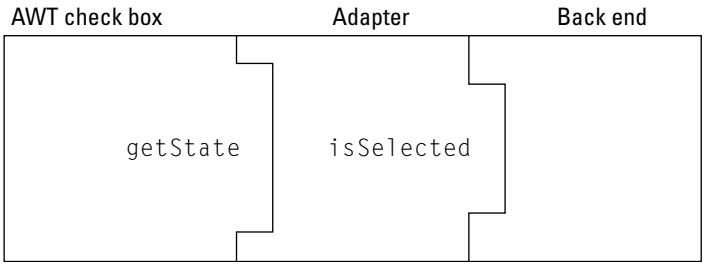

Figure 6-6: Using an adapter between AWT and SWT.

The class adapter, `CheckboxAdapter`, is going to inherit from the AWT `Checkbox` class.

```
import java.awt.*;

public class CheckboxAdapter extends Checkbox
{
    .
    .
    .
}
```

The public constructor passes control back to the AWT `Checkbox` constructor this way:

```
import java.awt.*;

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

To implement the `isSelected` method of the `CheckboxAdapter` class, you just pass on the data you get back from the AWT `getState` method.

```
import java.awt.*;

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

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

You can use the adapted check boxes in Swing UI code; here's how that works in an example, `Checkboxes.java`, which builds a `JFrame` object that implements the `ItemListener` interface to catch check box events:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Checkboxes extends JFrame implements ItemListener
{
}
```

The `main` method creates a new `Checkboxes` object and displays it.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Checkboxes extends JFrame implements ItemListener
{
        .
        .
        .
    public static void main(String args[])
    {
        final Checkboxes f = new Checkboxes();

        f.setBounds(100, 100, 400, 300);
        f.setVisible(true);
        f.setDefaultCloseOperation(DISPOSE_ON_CLOSE);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```

The `Checkboxes` constructor creates four `CheckboxAdapter` objects and adds them to the content pane.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Checkboxes extends JFrame implements ItemListener
{
    CheckboxAdapter checks[];
    JTextField text;

    public Checkboxes()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        checks = new CheckboxAdapter[4];

        for(int loopIndex = 0; loopIndex
          <= checks.length - 1; loopIndex++){
            checks[loopIndex] = new CheckboxAdapter("Check " +
              loopIndex);
            checks[loopIndex].addItemListener(this);
            contentPane.add(checks[loopIndex]);
        }

        text = new JTextField(30);

        contentPane.add(text);
    }
        .
        .
        .
    public static void main(String args[])
    {
        final Checkboxes c = new Checkboxes();

        c.setBounds(100, 100, 400, 300);
        c.setVisible(true);
        c.setDefaultCloseOperation(DISPOSE_ON_CLOSE);

        c.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }

}
```

You can handle the `CheckboxAdapter` objects as you would standard Swing check boxes when it comes to the `isSelected` method. When there's a check box event, the `itemChanged` method will be called, and you can check which of the check boxes is/are checked using `isSelected` in that method — the selected check boxes will be displayed in the text control.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Checkboxes extends JFrame implements ItemListener
{
    CheckboxAdapter checks[];
    JTextField text;

    public Checkboxes()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        checks = new CheckboxAdapter[4];

        for(int loopIndex = 0; loopIndex
          <= checks.length - 1; loopIndex++){
            checks[loopIndex] = new CheckboxAdapter("Check " +
              loopIndex);
            checks[loopIndex].addItemListener(this);
            contentPane.add(checks[loopIndex]);
        }

        text = new JTextField(30);

        contentPane.add(text);
    }

    public void itemStateChanged(ItemEvent e)
    {
        String outString = new String("Selected: ");

        for(int loopIndex = 0; loopIndex
          <= checks.length - 1; loopIndex++){
            if(checks[loopIndex].isSelected()) {
                outString += " checkbox " + loopIndex;
            }
        }
        text.setText(outString);
    }

    public static void main(String args[])
```

```
    {
        final Checkboxes f = new Checkboxes();

        f.setBounds(100, 100, 400, 300);
        f.setVisible(true);
        f.setDefaultCloseOperation(DISPOSE_ON_CLOSE);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }

}
```

So object adapters rely on object composition, while class adapters rely on inheritance. Object adapters can be more flexible because they can work with not just the objects they've been designed to adapt but also subclassed objects of the adapted objects. But with class adapters, you have to modify the class adapter to do the same thing. To be more flexible, in general, don't forget the design principle that says you should favor composition over inheritance.

One final note on adapters — besides adapting the behavior of a class or object, adapters can also *improve* that behavior by adding their own methods. For example, an adapted object that reports temperatures in Fahrenheit might be improved if its adapter also adds a method that reports temperatures in Centigrade.

Drawbacks of adapters? Not many — mostly that there's an additional layer of code added, and so to maintain. But if rewriting legacy code isn't an option, adapters provide a good option.

Simplifying Life with Facades

Similar to the Adapter pattern is the Facade design pattern. These two patterns work in much the same way, but they have different purposes. The Adapter pattern adapts code to work with other code. But the Facade pattern gives you a wrapper that makes the original code easier to deal with.

For example, say that someone's designed a printer and shows it to you proudly. "How do I make it print?" you ask.