CHAPTER 6

The Facade Pattern

Overview

In this chapter

I will start the study of design patterns with a pattern that you have probably implemented in the past but may not have had a name for: the Facade pattern.

In this chapter,

- I explain what the Facade pattern is and where it is used.
- I present the key features of the pattern.
- I present some variations on the Facade pattern.
- I relate the Facade pattern to the CAD/CAM problem.

Introducing the Facade Pattern

Intent: a unified, high-level interface

According to Gang of Four, the intent of the Facade pattern is to:

> "Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use." '

Basically, this is saying that we need a new way to interact with a system that is easier than the current way, or we need to use the system in a particular way (such as using a 3-D drawing program in a 2-D way). We can build such a method of interaction because we only need to use a subset of the system in question.

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, Mass.: Addison-Wesley, 1995, p. 185.

Learning the Facade Pattern

A motivating example: learn how to use our complex system!

Once, I worked as a contractor for a large engineering and manufacturing company. My first day on the job, the technical lead of the project was not in. Now, this client did not want to pay me by the hour and not have anything for me to do. They wanted me to be doing something, even if it was not useful! Haven't you had days like this?

So, one of the project members found something for me to do. She said, "You are going to have to learn the CAD/CAM system we use some time, so you might as well start now. Start with these manuals over here." Then she took me to the set of documentation. I am not making this up: there were *8 feet* of manuals for me to read . . . each page $8^1/2$ x 11 inches and in small print! This was one complex system!

Figure 6-1 Eight feet of manuals = one complex system!

I want to be insulated from this

Now, if you and I and say another four or five people were on a project that needed to use this system, not all of us would have to learn the entire thing. Rather than waste everyone's time, we would probably draw straws, and the *loser* would have to write routines that the rest of us would use to interface with the system.

This person would determine how I and others on our team were going to use the system and what API would be best for our particular needs. She would then create a new class or classes that had the interface we required. Then, I and the rest of the programming community could use this new interface without having to learn the entire complicated system (see Figure 6-2).

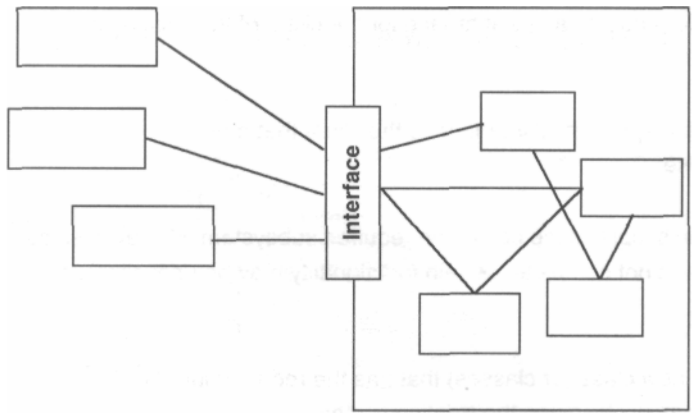

Figure 6-2 Insulating clients from the subsystem.

Works with subsets

Now, this approach only works when using a subset of the system's capabilities or when interacting with it in a particular way. If everything in the system needs to be used, it is unlikely that I can come up with a simpler interface (unless the original designers did a poor job).

This is the Facade pattern

This is the Facade pattern. It enables us to use a complex system more easily, either to use just a subset of the system or use the system in a particular way. We have a complicated system of which we need to use only a part. We end up with a simpler, easier-to-use system or one that is customized to our needs.

Most of the work still needs to be done by the underlying system. The Facade provides a collection of easier-to-understand methods. These methods use the underlying system to implement the newly defined functions.

The Facade Pattern: Key Features

Intent	You want to simplify how to use an existing system. You need to define your own interface.
Problem	You need to use only a subset of a complex system. Or you need to interact with the system in a particular way.
Solution	The Facade presents a new interface for the client of the existing system to use.
Participants and Collaborators	It presents a specialized interface to the client that makes it easier to use.
Consequences	The Facade simplifies the use of the required subsystem. However, since the Facade is not complete, certain functionality may be unavailable to the client.
Implementation	• Define a new class (or classes) that has the required interface. • Have this new class use the existing system.
GoF Reference	Pages 185-193.

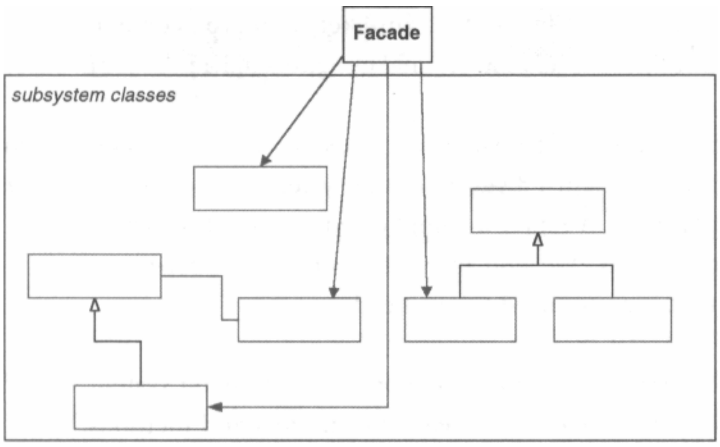

Figure 6-3 Standard, simplified view of the Facade pattern.

Field Notes: The Facade Pattern

Variations on Facade: reduce the number of objects a client must work with

Facades can be used not only to create a simpler interface in terms of method calls, but also to reduce the number of objects that a client object must deal with. For example, suppose I have a Client object that must deal with Databases, Models, and Elements. The Client must first open the Database and get a Model. Then it queries the Model to get an Element. Finally, it asks the Element for information. It might be a lot easier to create a Database-Facade that could be queried by the Client (see Figure 6-4).

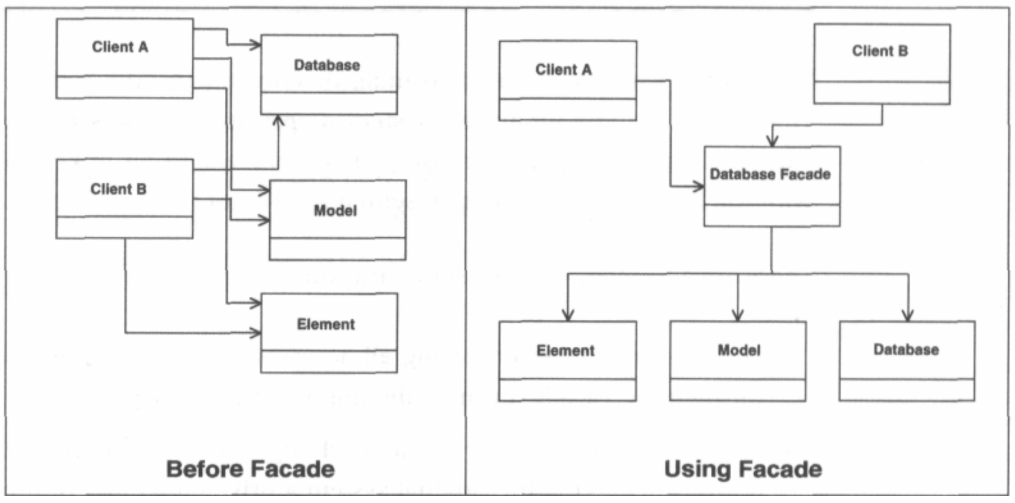

Figure 6-4 Facade reduces the number of objects for the client.

Variations on Facade: supplement existing functions with new routines

Suppose that in addition to using functions that are in the system, I also need to provide some new functionality. In this case, I am going beyond a simple subset of the system.

In this case, the methods I write for the Facade class may be supplemented by new routines for the new functionality. This is still the Facade pattern, but expanded with new functionality.

The Facade pattern sets the general approach; it got me started. The Facade part of the pattern is the fact that I am creating a new interface for the client to use instead of the existing system's interface. I can do this because the Client object does not need to use all of the functions in my original system.

Patterns set a general approach.

A pattern just sets the general approach. Whether or not to add new functions depends upon the situation at hand. Patterns are blueprints to get you started; they are not carved in stone.

Variations on Facade: an "encapsulating" layer

The Facade can also be used to hide, or encapsulate, the system. The Facade could contain the system as private members of the Facade class. In this case, the original system would be linked in with the Facade class, but not presented to users of the Facade class.

There are a number of reasons to encapsulate the system:

- *Track system usage*—By forcing all accesses to the system to go through the Facade, I can easily monitor system usage.
- *Swap out systems*—I may need to change out systems in the future. By making the original system a private member of the Facade class, I can switch out the system for a new one with minimal effort. There may still be a significant amount of effort required, but at least I will only have to change the code in one place (the Facade class).

Relating the Facade Pattern to the CAD/CAM Problem

Encapsulate the V1 system

Think of the example above. The Facade pattern could be useful to help V1Slots, VlHoles, etc., use the V1System. I will do just that in the solution in Chapter 12, "Solving the CAD/CAM Problem with Patterns."

Summary

In this chapter

The Facade pattern is so named because it puts up a new front (a facade) in front of the original system.

The Facade pattern applies when

- You do not need to use all of the functionality of a complex sys tem and can create a new class that contains all of the rules for accessing that system. If this is a subset of the original system, as it usually is, the API that you create in new class should be much simpler than the original system's API.
- You want to encapsulate or hide the original system.
- You want to use the functionality of the original system and want to add some new functionality as well.
- The cost of writing this new class is less than the cost of every body learning how to use the original system or is less than you would spend on maintenance in the future.

CHAPTER 7

The Adapter Pattern

Overview

In this chapter

I will continue our study of design patterns with the Adapter pattern. The Adapter pattern is a very common pattern, and, as you will see, it is used with many other patterns.

In this chapter,

- I explain what the Adapter pattern is, where it is used, and how it is implemented.
- I present the key features of the pattern.
- I use the pattern to illustrate polymorphism.
- I illustrate how the UML can be used at different levels of detail.
- I present some observations on the Adapter pattern from my own practice, including a comparison of the Adapter pattern and the Facade pattern.
- I relate the Adapter pattern to the CAD/CAM problem.

Note: I will only show Java code examples in the main body of this chapter. The equivalent C++ code examples can be found at the end of this chapter.

Introducing the Adapter Pattern

Intent: create a new interface

According to the Gang of Four, the intent of the Adapter pattern is to

> Convert the interface of a class into another interface that the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.[1]

Basically, this is saying that we need a way to create a new interface for an object that does the right stuff but has the wrong interface.

Learning the Adapter Pattern

A motivating example: free the client object from knowing details

The easiest way to understand the intent of the Adapter pattern is to look at an example of where it is useful. Let's say I have been given the following requirements:

- Create classes for points, lines, and squares that have the behavior "display."
- The client objects should not have to know whether they actu ally have a point, a line, or a square. They just want to know that they have one of these shapes.

In other words, I want to encompass these specific shapes in a higher-level concept that I will call a "displayable shape."

Now, as I work through this simple example, try to imagine other situations that you have run into that are similar, such as

- You have wanted to use a subroutine or a method that some one else has written because it performs some function that you need.

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, Mass.: Addison-Wesley, 1995, p. 185.

- You cannot incorporate the routine directly into your program.
- The interface or the way of calling the code is not exactly equivalent to the way that its related objects need to use it.

In other words, although the system will *have* points, lines, and squares, I want the client objects to *think* I have only *shapes*.

. . . so that it can treat details in a common way

- This allows the client objects to deal with all these objects in the same way—freed from having to pay attention to their differences.
- It also enables me to add different kinds of shapes in the future without having to change the clients (see Figure 7-1).

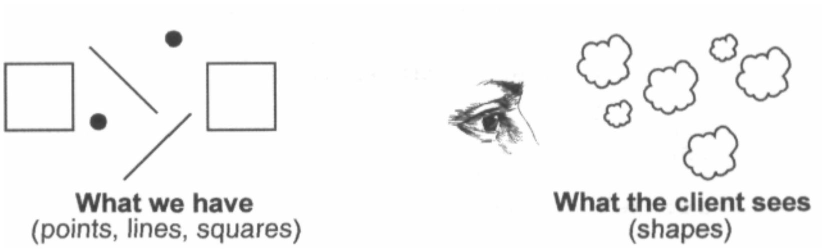

Figure 7-1 The objects we have . . . should all look just like "shapes."

I will make use of polymorphism; that is, I will have different objects in my system, but I want the clients of these objects to interact with them in a common way.

How to do this: use derived classes polymorphically

In this case, the client object will simply tell a point, line, or square to do something such as display itself or undisplay itself. Each point, line, and square is then responsible for knowing the way to carry out the behavior that is appropriate to its type.

To accomplish this, I will create a Shape class and then derive from it the classes that represent points, lines, and squares (see Figure 7-2).

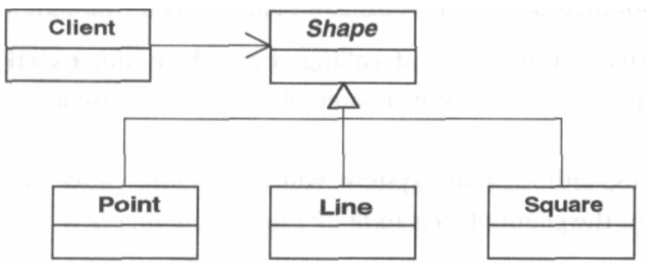

Figure 7-2 Points, Lines, and Squares are types of Shape.[2]

How to do this: define the interface and then implement in derived classes

First, I must specify the particular behavior that Shapes are going to provide. To accomplish this, I define an interface in the Shape class for the behavior and then implement the behavior appropriately in each of the derived classes.

The behaviors that a Shape needs to have are:

- Set a shape's location.
- Get a Shape's location.
- Display a Shape.
- Fill a Shape.
- Set the color of a Shape.
- Undisplay a Shape.

I show these in Figure 7-3.

Now, add a new shape

Suppose I am now asked to implement a circle, a new kind of Shape (remember, requirements always change!). To do this, I will want to create a new class—Circle—that implements the shape "circle" and derive it from the Shape class so that I can still get polymorphic behavior.

2. This and all other class diagrams in this book use the Unified Modeling Language (UML) notation. See Chapter 2, "The UML—The Unified Modeling Language," for a description of UML notation.

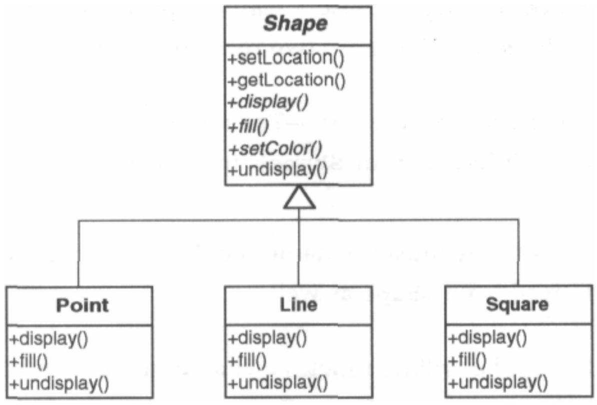

Figure 7-3 Points, Lines, **and** Squares **showing methods.**

Now, I am faced with the task of having to write the *display, fill* and *undisplay* methods for Circle. That could be a pain.

. . . but use behavior from outside

Fortunately, as I scout around for an alternative (as a good coder always should), I discover that Jill down the hall has already written a class she called xxcircle that deals with circles already (see Figure 7-4). Unfortunately, she didn't ask me what she should name the methods (and I did not ask her!). She named the methods

- *displayIt*
- *fillIt*
- *undisplayIt*

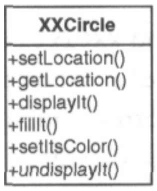

Figure 7-4 Jill's XXCircle class.

I cannot use xxcircle *directly*

I cannot use XXCircle directly because I want to preserve polymorphic behavior with Shape. There are two reasons for this:

- *I have different names and parameter lists*—The method names and parameter lists are different from Shape's method names and parameter lists.
- *I cannot derive it*—Not only must the names be the same, but the class must be derived from Shape as well.

It is unlikely that Jill will be willing to let me change the names of her methods or derive XXCircle from Shape. To do so would require her to modify all of the other objects that are currently using XXCircle. Plus, I would still be concerned about creating unanticipated side effects when I modify someone else's code.

I have what I want almost within reach, but I cannot use it and I do not want to rewrite it. What can I do?

Rather than change it, I adapt it

I can make a new class that *does* derive from shape and therefore implements shape's interface but avoids rewriting the circle implementation in XXCircle (see Figure 7-5).

- Class Circle derives from Shape.
- Circle contains XXCircle.
- Circle passes requests made to the Circle object on through to the XXCircle object.

How to implement

The diamond at the end of the line between Circle and xxcircle in Figure 7-5 indicates that Circle contains an xxcircle. When a Circle object is instantiated, it must instantiate a corresponding XXCircle object. Anything the Circle object is told to do will get passed on to the xxcircle object. If this is done consistently, and if the xxcircle object has the complete functionality the circle object needs (I will discuss shortly what happens if this is not the case), the Circle object will be able to manifest its behavior by letting the XXCircle object do the job.

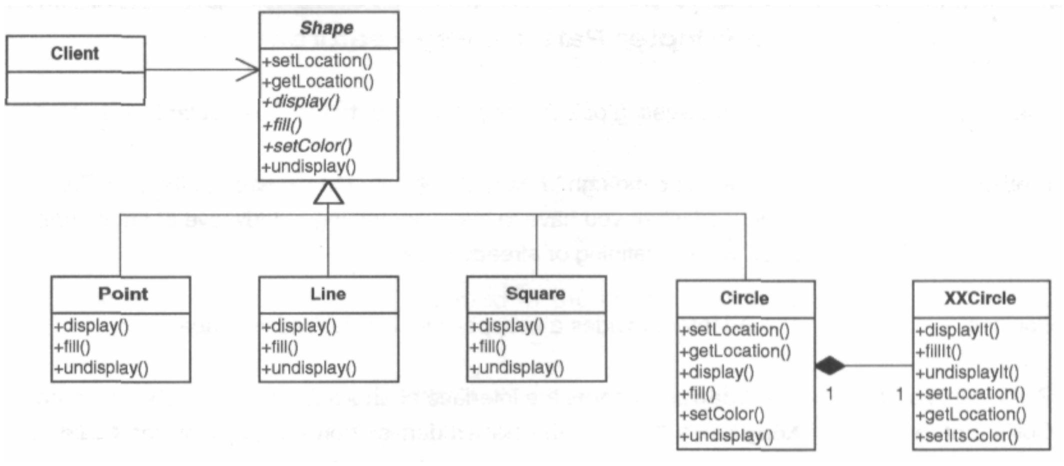

Figure 7-5 The Adapter pattern: Circle "wraps" XXCircle.

An example of wrapping is shown in Example 7-1.

Example 7-1 Java Code Fragments: Implementing the Adapter Pattern

```
class Circle extends Shape

   { private XXCircle pxc;

   public Circle () { pxc= new
     XXCircle();
   }

   void public display() {
   pxc.displayIt(); } }
```

What we accomplished

Using the Adapter pattern allowed me to continue using polymorphism with Shape. In other words, the client objects of Shape do not know what types of shapes are actually present. This is also an example of our new thinking of encapsulation as well—the class Shape encapsulates the specific shapes present. The Adapter pattern is most commonly used to allow for polymorphism. As we shall see in later chapters, it is often used to allow for polymorphism required by other design patterns.

The Adapter Pattern: Key Features

Intent	Match an existing object beyond your control to a particular interface.
Problem	A system has the right data and behavior but the wrong interface. Typically used when you have to make something a derivative of an abstract class we are defining or already have.
Solution	The Adapter provides a wrapper with the desired interface.
Participants and Collaborators	The Adapter adapts the interface of an Adaptee to match that of the Adapter's Target (the class it derives from). This allows the Client to use the Adaptee as if it were a type of Target.
Consequences	The Adapter pattern allows for preexisting objects to fit into new class structures without being limited by their interfaces.
Implementation	Contain the existing class in another class. Have the containing class match the required interface and call the methods of the contained class.
GoF Reference	Pages 139-150.

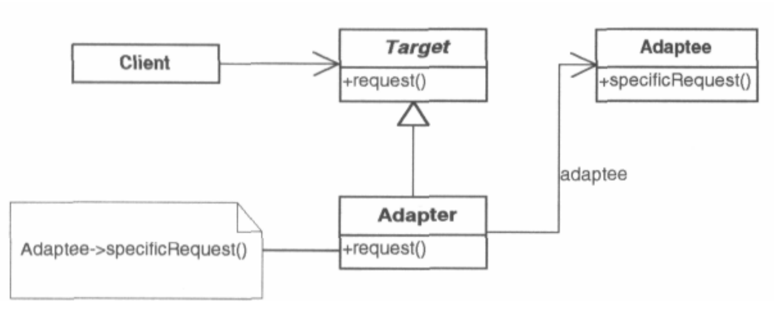

Figure 7-6 Standard, simplified view of the Adapter pattern.

Field Notes: The Adapter Pattern

You can do more than wrapping

Often, I will be in a situation similar to the one above, but the object being adapted does not do all the things I need.

In this case, I can still use the Adapter pattern, but it is not such a perfect fit. In this case,

- Those functions that are implemented in the existing class can be adapted.
- Those functions that are not present can be implemented in the wrapping object.

This does not give me quite the same benefit, but at least I do not have to implement all of the required functionality.

Adapter frees you from worrying about existing interfaces

The Adapter pattern frees me from worrying about the interfaces of existing classes when I am doing a design. If I have a class that does what I need, at least conceptually, then I know that I can always use the Adapter pattern to give it the correct interface.

This will become more important as you learn a few more patterns. Many patterns require certain classes to derive from the same class. If there are preexisting classes, the Adapter pattern can be used to adapt it to the appropriate abstract class (as Circle adapted XXCircle to Shape).

Variations: Object Adapter, Class Adapter

There are actually two types of Adapter patterns:

- *Object Adapter pattern*—The Adapter pattern I have been using is called an Object Adapter because it relies on one object (the adapting object) containing another (the adapted object).
- *Class Adapter pattern*—Another way to implement the Adapter pattern is with multiple inheritance. In this case, it is called a Class Adapter pattern.

The decision of which Adapter pattern to use is based on the different forces at work in the problem domain. At a conceptual level, I may ignore the distinction; however, when it comes time to imple ment it, I need to consider more of the forces involved.[3]

Comparing the Adapter with the Facade

In my classes on design patterns, someone almost always states that it sounds as if both the Adapter pattern and the Facade pattern are the same. In both cases there is a preexisting class (or classes) that does have the interface that is needed. In both cases, I create a new object that has the desired interface (see Figure 7-7).

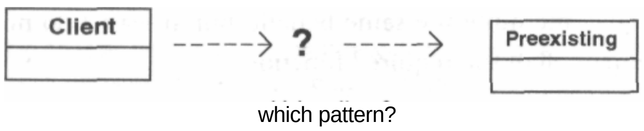

Figure 7-7 A Client object using another, preexisting object with the wrong interface.

Both are wrappers

Wrappers and object wrappers are terms that you hear a lot about. It is popular to think about wrapping legacy systems with objects to make them easier to use.

At this high view, the Facade and the Adapter patterns do seem similar. They are both wrappers. But they are different kinds of wrappers. You need to understand the differences, which can be subtle. Finding and understanding these more subtle differences gives insight into a pattern's properties. Let's look at some different forces involved with these patterns (see Table 7-1).

3. For help in deciding between Object Adapter and Class Adapter, see pages 142-144 in the Gang of Four book.

Table 7-1 Comparing the Facade Pattern with the Adapter Pattern

	Facade	Adapter
Are there preexisting classes?	Yes	Yes
Is there an interface we must design to?	No	Yes
Does an object need to behave polymorphically?	No	Probably
Is a simpler interface needed?	Yes	No

Table 7-1 tells us the following:

- In both the Facade and Adapter pattern I have preexisting classes.
- In the Facade, however, I do not have an interface I must design to, as I do in the Adapter pattern.
- I am not interested in polymorphic behavior in the Facade, while in the Adapter, I probably am. (There are times when we just need to design to a particular API and therefore must use an Adapter. In this case, polymorphism may not be an issue—that's why I say "probably").
- In the case of the Facade pattern, the motivation is to simplify the interface. With the Adapter, while simpler is better, I am trying to design to an existing interface and cannot simplify things even if a simpler interface were otherwise possible.

Not all differences are part of the pattern

Sometimes people draw the conclusion that another difference between the Facade and the Adapter pattern is that the Facade hides multiple classes behind it while the Adapter only hides one. While this is often true, it is not part of the pattern. It is possible that a Facade could be used in front of a very complex object while an Adapter wrapped several small objects that between them implemented the desired function.

Bottom line: A Facade *simplifies* an interface while an Adapter *converts* the interface into a preexisting interface.

Relating the Adapter Pattern to the CAD/CAM Problem

Adapter lets me communicate with OOGFeature

In the CAD/CAM problem (Chapter 3, "A Problem That Cries Out for Flexible Code"), the features in the V2 model will be represented by OOGFeature objects. Unfortunately, these objects do not have the correct interface (from my perspective) because I did not design them. I cannot make them derive from the Feature classes, yet, when I use the V2 system, they would do our job perfectly.

In this case, the option of writing new classes to implement this function is not even present—I must communicate with the OOGFeature objects. The easiest way to do this is with the Adapter pattern.

Summary

In this chapter

The Adapter pattern is a very useful pattern that converts the interface of a class (or classes) into another interface, which we need the class to have. It is implemented by creating a new class with the desired interface and then wrapping the original class methods to effectively contain the adapted object.

Supplement: C++ Code Example

Example 7-2 C++ **Code Fragments:** Implementing **the Adapter Pattern**

```
class Circle : public Shape {

  private:
    XXCircle *pxc;

}
Circle::Circle () {

   pxc= new XXCircle;
}
void Circle::display ()
   { pxc->displayIt();
}
```