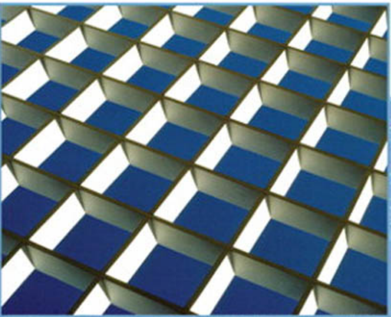
DESIGN PATTERNS EXPLAINED

A New Perspective on Object-Oriented Design

SECOND EDITION



ALAN SHALLOWAY

JAMES R. TROTT

NET OBJECTIVES

SOFTWARE PATTERNS SERIES

Praise for *Design Patterns Explained, Second Edition:*

The explanation of fundamental object-oriented concepts throughout is exceptional. I have struggled to teach similar concepts to beginners in my classes and I definitely plan to borrow some of the authors' approaches (and recommend the book, of course)!

—Clif Nock

Well-written, thought-provoking, and very enlightening. A must-read for anyone interested in design patterns and object-oriented development.

—James Huddleston

- It also enables me to add different kinds of shapes in the future without having to change the clients (see Figure 7-1).

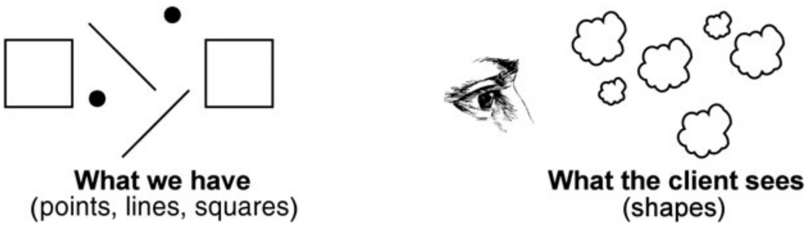


Figure 7-1 The objects we have ...should all look just like "shapes."

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).

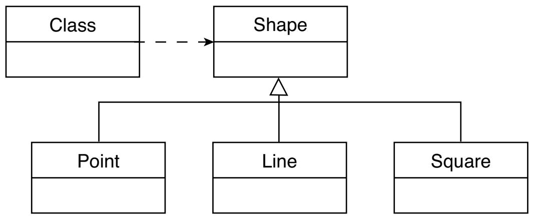


Figure 7-2 Points, Lines, and Squares are types of Shape.

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

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

The behaviors that a **Shape** needs to have are as follows:

- 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.

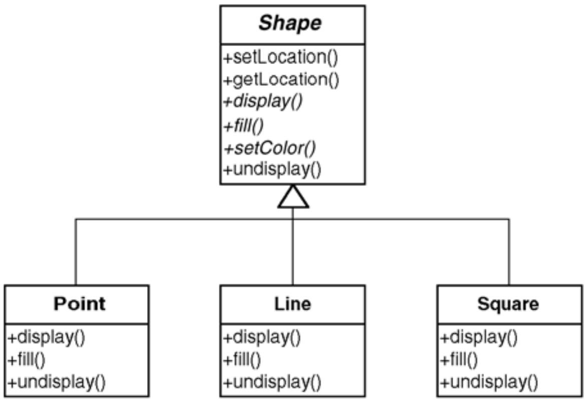


Figure 7-3 Points, Lines, and Squares showing methods.

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.

…but use behavior from outside

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

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 already handles circles (see Figure 7-4). Unfortunately, she didn't ask me what she should name the methods. She named the methods as follows:

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

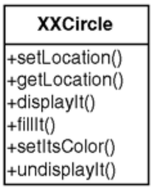


Figure 7-4 Jill's XXCircle class.

*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 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 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 to the **XXCircle** object. If this is done consistently, and if the **XXCircle** object has the complete functionality the **Circle** object needs (I discuss soon what happens if this is not the case), the **Circle**

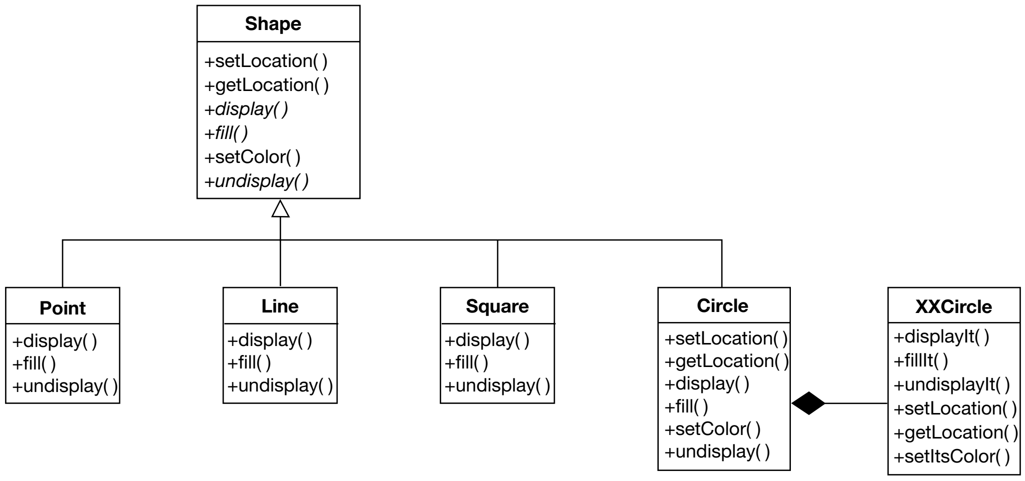


Figure 7-5 The Adapter pattern: Circle "wraps" XXCircle.

object will be able to manifest its behavior by letting the **XXCircle** object do the job.

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 myXXCircle;
   ...
   public Circle () {
     myXXCircle= new XXCircle();
   }

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

What we accomplished

Using the Adapter pattern enabled 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 about encapsulation as well—the class **Shape** encapsulates the specific shapes present. The Adapter pattern is most commonly used to allow for polymorphism. As you shall see in later chapters, it is often used to allow for polymorphism required by other design patterns.

Field Notes: The Adapter Pattern

You can do more than wrapping

Often I am in a situation similar to the one just described, 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 such as case

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.
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.

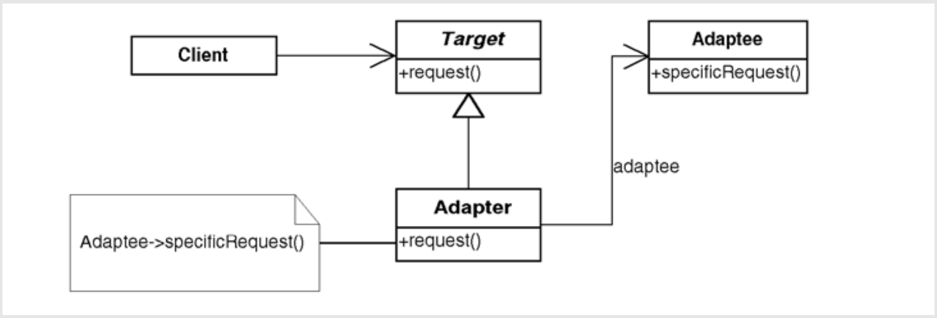


Figure 7-6 Generic structure of the Adapter pattern.

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

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