CHAPTER 9

The Bridge Pattern

Overview

In this chapter

I will continue our study of design patterns with the Bridge pattern. The Bridge pattern is quite a bit more complex than the other patterns you just learned; it is also much more useful.

In this chapter,

- I derive the Bridge pattern by working through an example. I will go into great detail to help you learn this pattern.
- I present the key features of the pattern.
- I present some observations on the Bridge pattern from my own practice.

Introducing the Bridge Pattern

Intent: decouple abstraction from implementation

According to the Gang of Four, the intent of the Bridge pattern is to "De-couple an abstraction from its implementation so that the two can vary independently."[1]

This is hard to understand

I remember exactly what my first thoughts were when I read this:

Huh?

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, Mass.: Addison-Wesley, 1995, p. 151.

And then,

> *How come I understand every word in this sentence but I have no idea what it means?!*

I knew that

- *De-couple* means to have things behave independently from each other or at least explicitly state what the relationship is, and
- *Abstraction* is how different things are related to each other conceptually.

And I thought that *implementations* were the way to build the abstractions; but I was confused about how I was supposed to separate abstractions from the specific ways that implemented them.

It turns out that much of my confusion was due to misunderstanding what implementations meant. *Implementations* here means the objects that the abstract class and its derivations use to implement themselves with (not the derivations of the abstract class, which are called concrete classes). But to be honest, even if I had understood it properly, I am not sure how much it would have helped. The concept expressed in this sentence is just hard to understand at first.

If you are also confused about the Bridge pattern at this point, that is okay. If you understand the stated intent, then you are that much ahead.

It is a challenging pattern to learn because it is so powerful

The Bridge pattern is one of the toughest patterns to understand in part because it is so powerful and applies to so many situations. Also, it goes against a common tendency to handle special cases with inheritance. However, it is also an excellent example of following two of the mandates of the design pattern community: "find what varies and encapsulate it" and "favor object composition over class inheritance" (as you will see).

Learning the Bridge Pattern: An Example

Learn why *it exists, then derive the pattern*

To learn the thinking behind the Bridge pattern and what it is trying to do, I will work through an example from scratch. Starting with requirements, I will derive the pattern and then see how to apply it.

Perhaps this example will seem basic. But look at the concepts discussed in this example and then try to think of situations that you have encountered that are similar, having

- Variations in abstractions of a concept, and
- Variations in how these concepts are implemented.

You will see that this example has many similarities to the CAD/CAM problem discussed earlier. But rather than give you all the requirements up front, I am going to give them a little at a time, just as they were given to me. You can't always see the variations at the beginning of the problem.

Bottom line: During requirements definition, explore for variations early and often!

Start with a simple problem: drawing shapes

Suppose I have been given the task of writing a program that will draw rectangles with either of two drawing programs. I have been told that when I instantiate a rectangle, I will know whether I should use drawing program 1 (`DP1`) or drawing program 2 (`DP2`).

The rectangles are defined as two pairs of points, as represented in Figure 9-1. The differences between the drawing programs are summarized in Table 9-1.

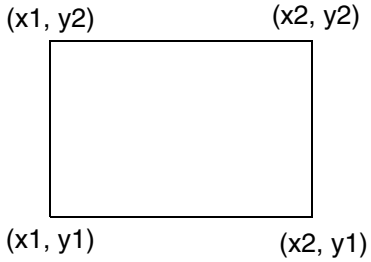

Figure 9-1 Positioning the rectangle.

Table 9-1 Different Drawing Programs

	DP1	DP2
Used to draw a line	`draw_a_line( x1, y1, x2, y2)`	`drawline( x1, x2, y1, y2)`
Used to draw a circle	`draw_a_circle( x, y, r)`	`drawcircle( x, y, r)`

Proper use of inheritance

My customer told me that the collection (the client of the rectangles) does not want to worry about what type of drawing program it should use. It occurs to me that since the rectangles are told what drawing program to use when instantiated, I can have two different kinds of rectangle objects: one that uses **`DP1`** and one that uses **`DP2`**. Each would have a draw method but would implement it differently. I show this in Figure 9-2.

A note on the implementation

By having an abstract class **`Rectangle`**, I take advantage of the fact that the only difference between the different types of **`Rectangle`**s are how they implement the *`drawLine`* method. The **`V1Rectangle`** is implemented by having a reference to a **`DP1`** object and using that object's *`draw_a_line`* method. The **`V2Rectangle`** is implemented by having a reference to a **`DP2`** object and using that object's *`drawline`* method. However, by instantiating the right type of **`Rectangle`**, I no longer have to worry about this difference.

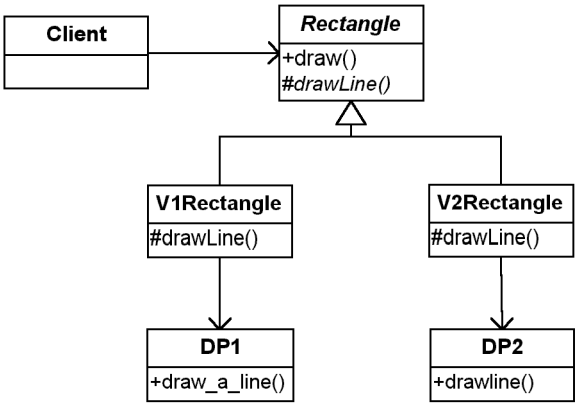

Figure 9-2 Design for rectangles and drawing programs (DP1 and DP2).

Example 9-1 Java Code Fragments

```
class Rectangle {
  public void draw () {
     drawLine(_x1,_y1,_x2,_y1);
     drawLine(_x2,_y1,_x2,_y2);
     drawLine(_x2,_y2,_x1,_y2);
     drawLine(_x1,_y2,_x1,_y1);
   }
  abstract protected void
    drawLine ( double x1, double y1,
               double x2, double y2);
}

class V1Rectangle extends Rectangle {
  drawLine( double x1, double y1,
            double x2, double y2) {
    DP1.draw_a_line( x1,y1,x2,y2);
  }
}
class V2Rectangle extends Rectangle {
  drawLine( double x1, double y1,
            double x2, double y2) {
    // arguments are different in DP2
    // and must be rearranged
    DP2.drawline( x1,x2,y1,y2);
  }
}
```

But, though requirements always change

Now, suppose that after completing this code, one of the *inevitable three* (death, taxes, and changing requirements) comes my way. I am asked to support another kind of shape—this time, a circle. However, I am also given the mandate that the collection object does not want to know the difference between **`Rectangle`**s and **`Circle`**s.

. . . I can still have a simple implementation

It occurs to me that I can simply extend the approach I've already started by adding another level to my class hierarchy. I only need to add a new class, called **`Shape`**, from which I will derive the **`Rectangle`** and **`Circle`** classes. This way, the **`Client`** object can just refer to **`Shape`** objects without worrying about what kind of **`Shape`** it has been given.

Designing with inheritance

As a beginning object-oriented analyst, it might seem natural to implement these requirements using only inheritance. For example, I could start out with something like Figure 9-2, and then, for each kind of **`Shape`**, implement the shape with each drawing program, deriving a version of **`DP1`** and a version of **`DP2`** for **`Rectangle`** and deriving a version of **`DP1`** and a version of **`DP2`** one for **`Circle`**. I would end up with Figure 9-3.

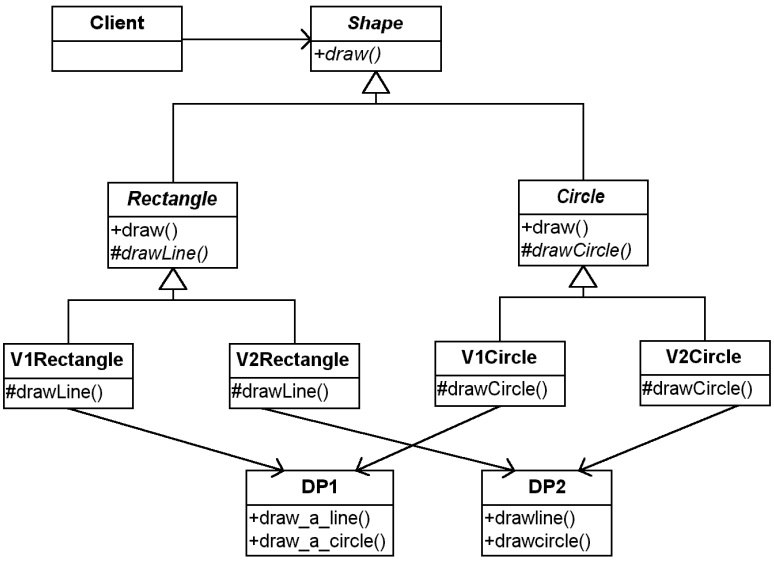

Figure 9-3 A straightforward approach: implementing two shapes and two drawing programs.

I implement the **`Circle`** class the same way that I implemented the **`Rectangle`** class. However, this time, I implement *`draw`* by using *`drawCircle`* instead of *`drawLine`*.

Example 9-2 Java Code Fragments

```
abstract class Shape {
  abstract public void draw ();
}
abstract class Rectangle extends Shape {
  public void draw () {
    drawLine(_x1,_y1,_x2,_y1);
    drawLine(_x2,_y1,_x2,_y2);
    drawLine(_x2,_y2,_x1,_y2);
    drawLine(_x1,_y2,_x1,_y1);
  }
  abstract protected void
    drawLine(
      double x1, double y1,
      double x2, double y2);
}
class V1Rectangle extends Rectangle {
  protected void drawLine (
    double x1, double y1,
    double x2, double y2) {
      DP1.draw_a_line( x1,y1,x2,y2);
  }
}
class V2Rectangle extends Rectangle {
  protected void drawLine (
    double x1, double x2,
    double y1, double y2) {
    DP2.drawline( x1,x2,y1,y2);
  }
}
abstract class Circle {
  public void draw () {
    drawCircle( x,y,r);
  }
  abstract protected void
    drawCircle (
      double x, double y, double r);
}
```

(continued)

Example 9-2 Java Code Fragments *(continued)*

```
class V1Circle extends Circle {
  protected void drawCircle() {
    DP1.draw_a_circle( x,y,r);
  }
}
class V2Circle extends Circle {
  protected void drawCircle() {
    DP2.drawcircle( x,y,r);
  }
}
```

Understanding the design

To understand this design, let's walk through an example. Consider what the *draw* method of a **V1Rectangle** does.

- **Rectangle**'s *draw* method is the same as before (calling *drawLine* four times as needed).
- *drawLine* is implemented by calling **DP1**'s *draw_a_line*.

In action, this looks like Figure 9-4.

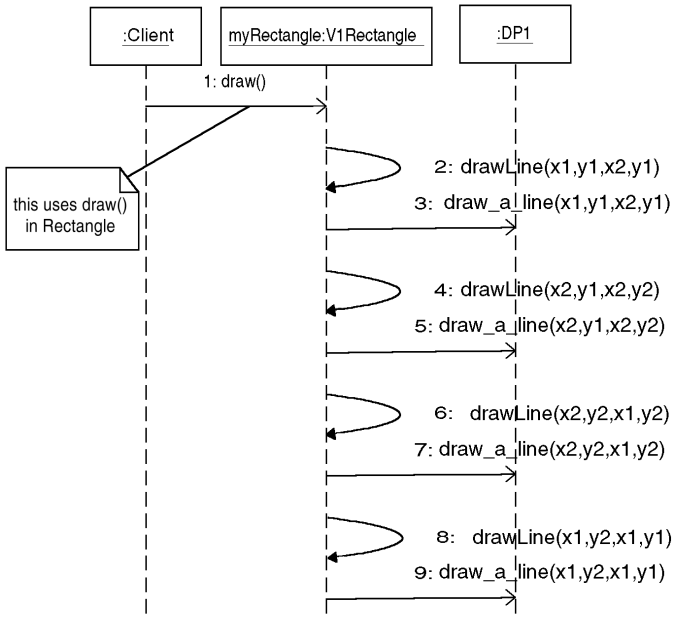

Figure 9-4 Sequence Diagram when have a V1Rectangle.

Reading a Sequence Diagram.

As I discussed in Chapter 2, "The UML—The Unified Modeling Language," the diagram in Figure 9-4 is a special kind of interaction diagram called a *Sequence Diagram*. It is a common diagram in the UML. Its purpose is to show the interaction of objects in the system.

- Each box at the top represents an object. It may be named or not.
- If an object has a name, it is given to the left of the colon.
- The class to which the object belongs is shown to the right of the colon. Thus, the middle object is named **`myRectangle`** and is an instance of **`V1Rectangle`**.

You read the diagram from the top down. Each numbered statement is a message sent from one object to either itself or to another object.

- The sequence starts out with the unnamed **`Client`** object calling the *`draw`* method of **`myRectangle`**.
- This method calls its own *`drawLine`* method four times (shown in steps 2, 4, 6, and 8). Note the arrow pointing back to the **`myRectangle`** in the timeline.
- *`drawLine`* calls **`DP1`**'s *`draw_a_line`*. This is shown in steps 3, 5, 7 and 9.

Even though the Class Diagram makes it look like there are many objects, in reality, I am only dealing with three objects (see Figure 9-5):

- The client using the rectangle
- The **`V1Rectangle`** object
- The **`DP1`** drawing program

When the client object sends a message to the **`V1Rectangle`** object (called **`myRectangle`**) to perform *`draw`*, it calls **`Rectangle`**'s *`draw`* method resulting in steps 2 through 9.

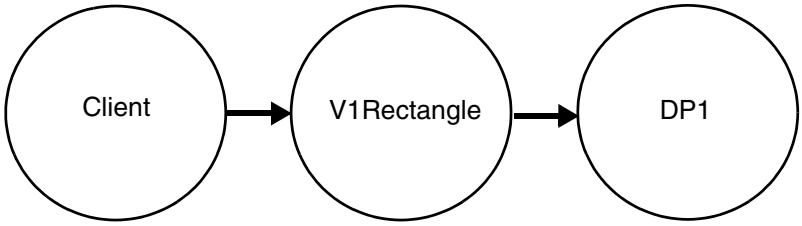

Figure 9-5 The objects present.

This solution suffers from combinatorial explosion

Unfortunately, this approach introduces new problems. Look at Figure 9-3 and pay attention to the third row of classes. Consider the following:

- The classes in this row represent the four specific types of **`Shape`**s that I have.
- What happens if I get another drawing program, that is, another variation in implementation? I will have *six* different kinds of **`Shape`**s (two **`Shape`** concepts times three drawing programs).
- Imagine what happens if I then get another type of **`Shape`**, another variation in concept. I will have *nine* different types of **`Shape`**s (three **`Shape`** concepts times three drawing programs).

. . . because of tight coupling

The class explosion problem arises because in this solution, the abstraction (the kinds of **`Shape`**s) and the implementation (the drawing programs) are tightly coupled. Each type of shape must know what type of drawing program it is using. I need a way to separate the variations in abstraction from the variations in implementation so that the number of classes only grows linearly (see Figure 9-6).

This is exactly the intent of the Bridge pattern: [to] de-couple an abstraction from its implementation so that the two can vary independently.[2]

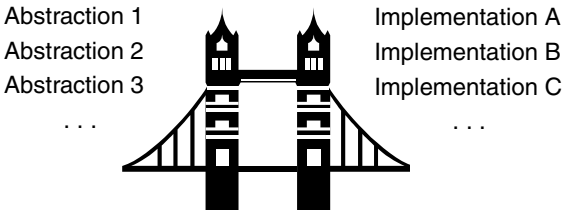

Figure 9-6 The Bridge pattern separates variations in abstraction and implementation.

There are several other problems. Our poor approach to design gave us this mess!

Before showing a solution and deriving the Bridge pattern, I want to mention a few other problems (beyond the combinatorial explosion).

Looking at Figure 9-3, ask yourself what else is poor about this design.

- Does there appear to be redundancy?
- Would you say things have high cohesion or low cohesion?
- Are things tightly or loosely coupled?
- Would you want to have to maintain this code?

The overuse of inheritance.

As a beginning object-oriented analyst, I had a tendency to solve the kind of problem I have seen here by using special cases, taking advantage of inheritance. I loved the idea of inheritance because it seemed new and powerful. I used it whenever I could. This seems to be normal for many beginning analysts, but it is naive: given this new "hammer," everything seems like a nail.

2. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, Mass.: Addison-Wesley, 1995, p. 151.

Unfortunately, many approaches to teaching object-oriented design focus on data abstraction—making designs overly based on the "*is*-ness" of the objects. As I became an experienced object-oriented designer, I was still stuck in the paradigm of designing based on inheritance—that is, looking at the characteristics of my classes based on their "*is*-ness." Characteristics of objects should be based on their responsibilities, not on what they might contain or be. Objects, of course, may be responsible for giving information about themselves; for example, a customer object may need to be able to tell you its name. Think about objects in terms of their responsibilities, not in terms of their structure.

Experienced object-oriented analysts have learned to use inheritance selectively to realize its power. Using design patterns will help you move along this learning curve more quickly. It involves a transition from using a different specialization for each variation (inheritance) to moving these variations into used or owned objects (composition).

An alternative approach

When I first looked at these problems, I thought that part of the difficulty might have been that I simply was using the wrong kind of inheritance hierarchy. Therefore, I tried the alternate hierarchy shown in Figure 9-7.

Not really a lot better, just bad in a different way

I still have the same four classes representing all of my possible combinations. However, by first deriving versions for the different drawing programs, I eliminated the redundancy between the **`DP1`** and **`DP2`** packages.

Unfortunately, I am unable to eliminate the redundancy between the two types of **`Rectangle`**s and the two types of **`Circle`**s, each pair of which has the same *`draw`* method.

In any event, the class explosion that was present before is still present here.

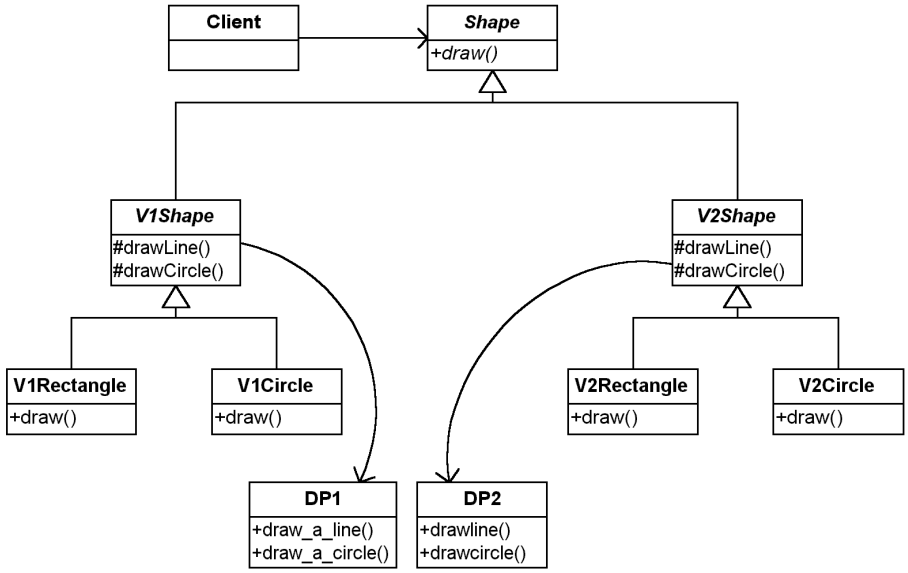

Figure 9-7 An alternative implementation.

The sequence diagram for this solution is shown in Figure 9-8.

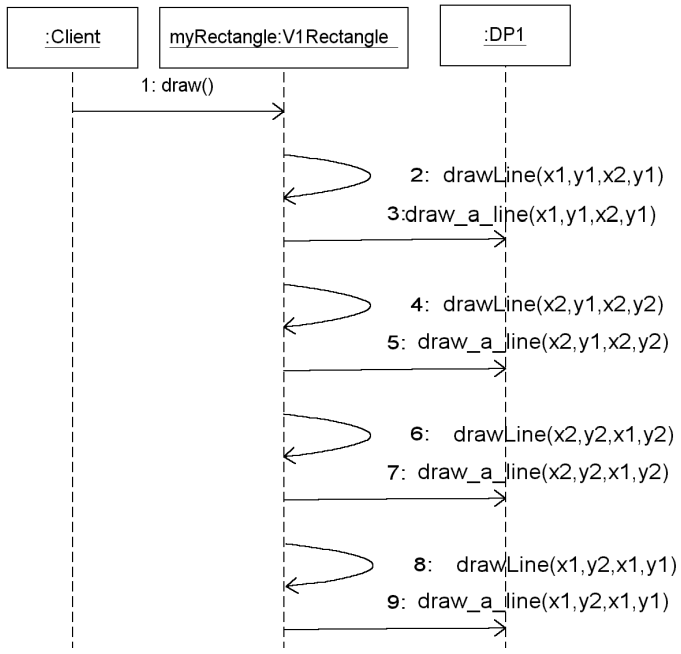

Figure 9-8 Sequence Diagram for new approach.

It still has scaling problems

While this may be an improvement over the original solution, it still has a problem with scaling. It also still has some of the original cohesion and coupling problems.

Bottom line: I do not want to have to maintain this version either! There must be a better way.

> **Look for alternatives in initial design.**
>
> Although my alternative design here was not significantly better than my original design, it is worth pointing out that finding alternatives to an original design is a good practice. Too many developers take what they first come up with and go with that. I am not endorsing an in-depth study of all possible alternatives (another way of getting "paralysis by analysis"). However, stepping back and looking at how we can overcome the design deficiencies in our original design is a great practice. In fact, it was just this stepping back, a refusal to move forward with a known, poor design, that led me to understanding the powerful methods of using design patterns that this entire book is about.

An Observation About Using Design Patterns

A new way to look at design patterns

When people begin to look at design patterns, they often focus on the solutions the patterns offer. This seems reasonable because they are advertised as providing good solutions to the problems at hand.

However, this is starting at the wrong end. When you learn patterns by focusing on the solutions they present, it makes it hard to determine the situations in which a pattern applies. This only tells us *what to do* but not *when to use it* or *why to do it*.

I find it much more useful to focus on the context of the pattern—the problem it is trying to solve. This lets me know the *when* and the *why*. It is more consistent with the philosophy of Alexander's patterns: "Each pattern describes a problem which occurs over and over again in the environment, and then describes the core of the solution to that problem . . ."[3]

What I have done here is a case in point. What is the problem being solved by the Bridge pattern?

> *The Bridge pattern is useful when you have an abstraction that has different implementations. It allows the abstraction and the implementation to vary independently of each other.*

The characteristics of the problem fit this nicely. I can know that I ought to be using the Bridge pattern even though I do not know yet how to implement it. Allowing for the abstraction to vary independently from the implementation would mean I could add new abstractions without changing my implementations and vice versa.

The current solution does not allow for this independent variation. I can see that it would be better if I could create an implementation that would allow for this.

The bottom line

It is very important to realize that, without even knowing how to implement the Bridge pattern, you can determine that it would be useful in this situation. You will find that this is generally true of design patterns. That is, you can identify when to apply them to your problem domain before knowing exactly how to implement them.

3. Alexander, C., Ishikawa, S., Silverstein, M., *A Pattern Language: Towns/Buildings/Construction*, New York: Oxford University Press, 1977, p. x.

Learning the Bridge Pattern: Deriving It

Deriving a solution

Now that you have been through the problem, we are in a position to derive the Bridge pattern together. Doing the work to derive the pattern will help you to understand more deeply what this complex and powerful pattern does.

Let's apply some of the basic strategies for good object-oriented design and see how they help to develop a solution that is very much like the Bridge pattern. To do this, I will be using the work of Jim Coplien[4] on commonality and variability analysis.

Design patterns are solutions that occur again and again.

Design patterns are solutions that have recurred in several problems and have therefore proven themselves over time to be good solutions. The approach I am taking in this book is to *derive* the pattern in order to teach it so that you can understand its characteristics.

In this case, I know the pattern I want to derive—the Bridge pattern—because I was shown it by the Gang of Four and have seen how it works in my own problem domains. It is important to note that patterns are not really derived. By definition, they must be recurring—having been demonstrated in at least three independent cases—to be considered patterns. What I mean by "derive" is that we will go through a design process where you create the pattern as if you did not know it. This is to illustrate some key principles and useful strategies.

First, use commonality/ variability analysis

Coplien's work on commonality/variability analysis tells us how to find variations in the problem domain and identify what is common across the domain. Identify where things vary (commonality analysis) and then identify how they vary (variability analysis).

4. Coplein, J., *Multi-Paradigm Design for C++*. Reading, Mass.: Addison-Wesley, 1998.

Commonality

According to Coplien, "Commonality analysis is the search for common elements that helps us understand how family members are the same."[5] Thus, the process of finding out how things are common defines the family in which these elements belong (and hence, where things vary).

Variability

Variability analysis reveals how family members vary. Variability only makes sense within a given commonality.

> Commonality analysis seeks structure that is unlikely to change over time, while variability analysis captures structure that is likely to change. Variability analysis makes sense only in terms of the context defined by the associated commonality analysis . . . From an architectural perspective, commonality analysis gives the architecture its longevity; variability analysis drives its fitness for use.[6]

In other words, if variations are the specific concrete cases in the domain, commonality defines the concepts in the domain that tie them together. The common concepts will be represented by abstract classes. The variations found by variability analysis will be implemented by the concrete classes (that is, classes derived from the abstract class with specific implementations).

A new paradigm for finding objects

It is almost axiomatic with object-oriented design methods that the designer is supposed to look in the problem domain, identify the nouns present, and create objects representing them. Then, the designer finds the verbs relating to those nouns (that is, their actions) and implement them by adding methods to the objects. This process of focusing on nouns and verbs typically leads to larger class hierarchies than we might want. I suggest that using commonality/variability analysis as a primary tool in creating objects is a better approach than looking at just nouns and verbs (actually, I believe this is a restatement of Jim Coplien's work).

---

5. ibid, p. 63.
6. ibid, pp. 60, 64.

Strategies to handle variations

There are two basic strategies to follow in creating designs to deal with the variations:

- Find what varies and encapsulate it.
- Favor composition over inheritance.

In the past, developers often relied on extensive inheritance trees to coordinate these variations. However, the second strategy says to try composition when possible. The intent of this is to be able to contain the variations in independent classes, thereby allowing for future variations without affecting the code. One way to do this is to have each variation contained in its own abstract class and then see how the abstract classes relate to each other.

Reviewing encapsulation.

Most object-oriented developers learned that "encapsulation" is data-hiding. Unfortunately, this is a very limiting definition. True, encapsulation does hide data, but it can be used in many other ways. If you look back at Figure 7-2, you will see encapsulation operates at many levels. Of course, it works at hiding data for each of the particular **`Shape`**s. However, notice that the **`Client`** object is not aware of the particular kinds of shapes. That is, the **`Client`** object has no idea that the **`Shape`**s it is dealing with are **`Rectangle`**s and **`Circle`**s. Thus, the concrete classes that **`Client`** deals with are hidden (or encapsulated) from **`Client`**. This is the kind of encapsulation that the Gang of Four is talking about when they say, "find what varies and encapsulate it". They are finding what varies, and encapsulating it "behind" an abstract class (see Chapter 8, "Expanding Our Horizons").

Try it: identify what is varying

Follow this process for the rectangle drawing problem.

First, identify what it is that is varying. In this case, it is different types of **`Shape`**s and different types of drawing programs. The common concepts are therefore shapes and drawing programs. I represent this in Figure 9-9 (note that the class names are shown in italics because the classes are abstract).

Figure 9-9 What is varying.

At this point, I mean for **`Shape`** to encapsulate the concept of the types of shapes that I have. Shapes are responsible for knowing how to draw themselves. **`Drawing`** objects, on the other hand, are responsible for drawing lines and circles. I represent these responsibilities by defining methods in the classes.

Try it: represent the variations

The next step is to represent the specific variations that are present. For **`Shape`**, I have rectangles and circles. For drawing programs, I will have a program that is based on **`DP1`** (**`V1Drawing`**) and one based on **`DP2`** (**`V2Drawing`**), respectively. I show this in Figure 9-10.

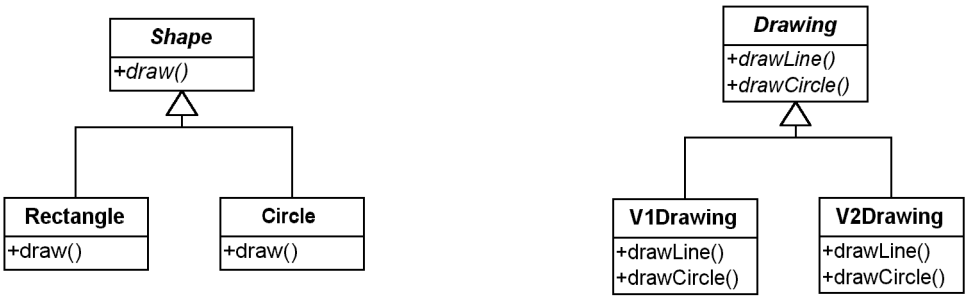

Figure 9-10 Represent the variations.

At this point, the diagram is simply notional. I know that **V1Drawing** will use **DP1** and **V2Drawing** will use **DP2** but I have not said *how*. I have simply captured the concepts of the problem domain (shapes and drawing programs) and have shown the variations that are present.

Tying the classes together: who uses whom?

Given these two sets of classes, I need to ask how they will relate to one another. I do not want to come up with a new set of classes based on an inheritance tree because I know what happens if I do that (look at Figures 9-3 and 9-7 to refresh your memory). Instead, I want to see if I can relate these classes by having one use the other (that is, follow the mandate to favor composition over inheritance). The question is, which class uses the other?

Consider these two possibilities: either **Shape** uses the **Drawing** programs or the **Drawing** programs use **Shape**.

Consider the latter case first. If drawing programs could draw shapes directly, then they would have to know some things about shapes in general: what they are, what they look like. But this violates a fundamental principle of objects: an object should only be responsible for itself.

It also violates encapsulation. **Drawing** objects would have to know specific information about the **Shape**s (that is, the kind of **Shape**) in order to draw them. The objects are not really responsible for their own behaviors.

Now, consider the first case. What if I have **Shape**s use **Drawing** objects to draw themselves? **Shape**s wouldn't need to know what type of **Drawing** object it used since I could have **Shape**s refer to the **Drawing** class. **Shape**s also would be responsible for controlling the drawing.

This looks better to me. Figure 9-11 shows this solution.

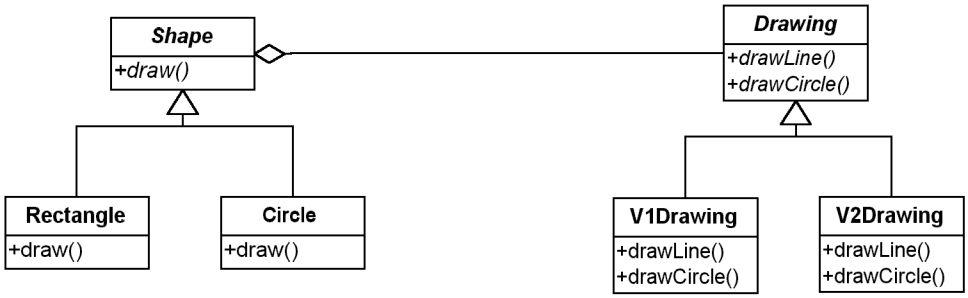

Figure 9-11 Tie the classes together.

Expanding the design

In this design, **`Shape`** uses **`Drawing`** to manifest its behavior. I left out the details of **`V1Drawing`** using the **`DP1`** program and **`V2Drawing`** using the **`DP2`** program. In Figure 9-12, I add this as well as the protected methods *`drawLine`* and *`drawCircle`* (in **`Shape`**), which calls **`Drawing`**'s *`drawLine`*, and *`drawCircle`*, respectively.

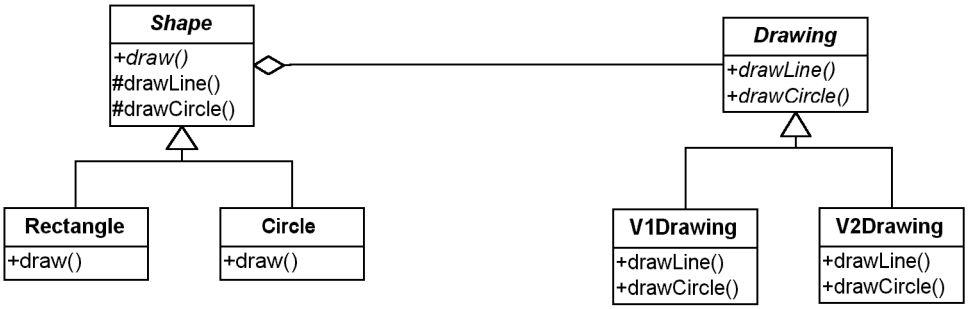

Figure 9-12 Expanding the design.

One rule, one place.

A very important implementation strategy to follow is to have only one place where you implement a rule. In other words, if you have a rule how to do things, only implement that once. This typically results in code with a greater number of smaller methods. The extra cost is minimal, but it eliminates duplication and often prevents many future problems. Duplication is bad not only because of the extra work in typing things multiple times, but because of the likelihood of something changing in the future and then forgetting to change it in all of the required places.

While the *draw* method or **Rectangle** could directly call the *drawLine* method of whatever **Drawing** object the **Shape** has, I can improve the code by continuing to follow the one rule, one place strategy and have a *drawLine* method in **Shape** that calls the *drawLine* method of its **Drawing** object.

I am not a purist (at least not in most things), but if there is one place where I think it is important to always follow a rule, it is here. In the example below, I have a *drawLine* method in **Shape** because that describes my rule of drawing a line with **Drawing**. I do the same with *drawCircle* for circles. By following this strategy, I prepare myself for other derived objects that might need to draw lines and circles.

Where did the one rule, one place strategy come from? While many have documented it, it has been in the folklore of object-oriented designers for a long time. It represents a best practice of designers. Most recently, Kent Beck called this the "once and only once rule."*

* Beck, K., *Extreme Programming Explained: Embrace Change*, Reading, Mass.: Addison Wesley, 2000, pp. 108–109.

He defines it as part of his constraints:

- The system (code and tests together) must communicate everything you want to communicate.
- The system must contain no duplicate code. (1 and 2 together constitute the Once and Only Once rule).

The pattern llustrated

Figure 9-13 illustrates the separation of the **`Shape`** abstraction from the **`Drawing`** implementation.

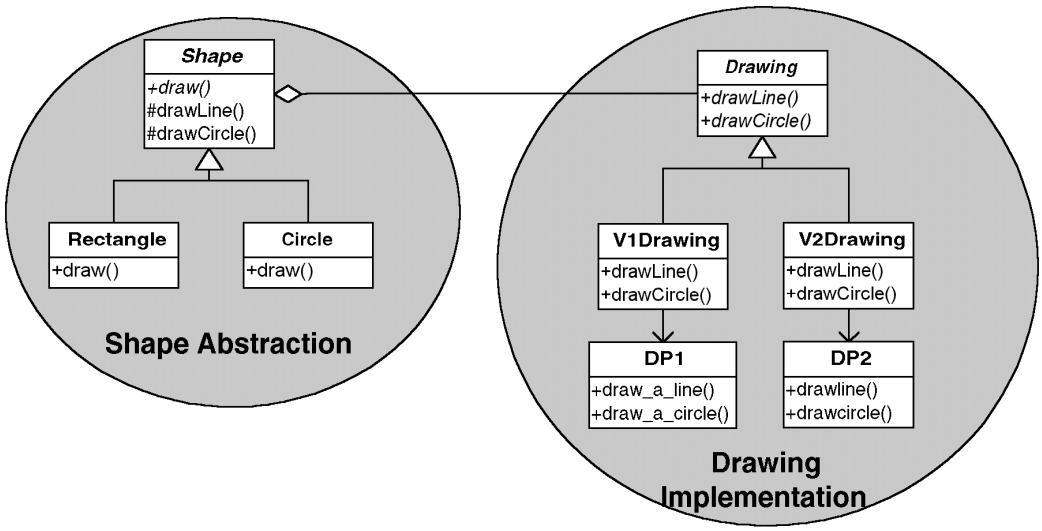

Figure 9-13 Class diagram illustrating separation of abstraction and implementation.

Relating this to the inheritance-based design

From a method point of view, this looks fairly similar to the inheritance-based implementation (such as shown in Figure 9-3). The biggest difference is that the methods are now located in different objects.

I said at the beginning of this chapter that my confusion over the Bridge pattern was due to my misunderstanding of the term "implementation." I thought that implementation referred to how I implemented a particular abstraction.

The Bridge pattern let me see that viewing the implementation as something outside of my objects, something that is *used by* the objects, gives me much greater freedom by hiding the variations in implementation from my calling program. By designing my objects this way, I also noticed how I was containing variations in separate class hierarchies. The hierarchy on the left side of Figure 9-13 contains the variations in my abstractions. The hierarchy on the right side of Figure 9-13 contains the variations in how I will implement those abstractions. This is consistent with the new paradigm for creating objects (using commonality/variability analysis) that I mentioned earlier.

From an object perspective

It is easiest to visualize this when you remember that there are only three objects to deal with at any one time, even though there are several classes (see Figure 9-14).

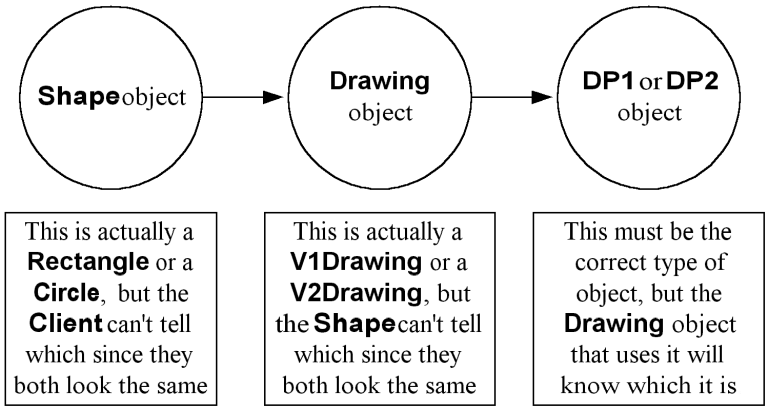

Figure 9-14 There are only three objects at a time.

A reasonably complete code example is shown in Example 9-3 for Java and in the Examples beginning on page 157 for C++.

Code examples

Example 9-3 Java Code Fragments

```
class Client {
  public static void main
    (String argv[]) {
      Shape r1, r2;
      Drawing dp;

      dp= new V1Drawing();
      r1= new Rectangle(dp,1,1,2,2);

      dp= new V2Drawing ();
      r2= new Circle(dp,2,2,3);

      r1.draw();
      r2.draw();
   }
}

abstract class Shape {
  abstract public draw() ;
  private Drawing _dp;

  Shape (Drawing dp) {
    _dp= dp;
  }
  public void drawLine (
    double x1,double y1,
    double x2,double y2) {
      _dp.drawLine(x1,y1,x2,y2);
  }

  public void drawCircle (
    double x,double y,double r) {
      _dp.drawCircle(x,y,r);
  }
}

abstract class Drawing {
  abstract public void drawLine (
    double x1, double y1,
    double x2, double y2);
```

(continued)

Example 9-3 Java Code Fragments *(continued)*

```
  abstract public void drawCircle (
    double x,double y,double r);
}

class V1Drawing extends Drawing {
  public void drawLine (
    double x1,double y1,
    double x2,double y2) {
    DP1.draw_a_line(x1,y1,x2,y2);
  }
  public void drawCircle (
    double x,double y,double r) {
    DP1.draw_a_circle(x,y,r);
  }
}

class V2Drawing extends Drawing {
  public void drawLine (
    double x1,double y1,
    double x2,double y2) {
    // arguments are different in DP2
    // and must be rearranged
    DP2.drawline(x1,x2,y1,y2);
  }
  public void drawCircle (
    double x, double y,double r) {
    DP2.drawcircle(x,y,r);
  }
}

class Rectangle extends Shape {
  public Rectangle (
    Drawing dp,
    double x1,double y1,
    double x2,double y2) {
      super( dp) ;
      _x1= x1; _x2= x2 ;
      _y1= y1; _y2= y2;
  }

  public void draw () {
   drawLine(_x1,_y1,_x2,_y1);
   drawLine(_x2,_y1,_x2,_y2);
```

(continued)

Example 9-3 Java Code Fragments *(continued)*

```
   drawLine(_x2,_y2,_x1,_y2);
   drawLine(_x1,_y2,_x1,_y1);
  }
}

class Circle extends Shape {
  public Circle (
    Drawing dp,
    double x,double y,double r) {
      super( dp) ;
      _x= x; _y= y; _r= r ;
  }

  public void draw () {
   drawCircle(_x,_y,_r);
  }
}

// We've been given the implementations for DP1 and DP2

class DP1 {
  static public void draw_a_line (
    double x1,double y1,
    double x2,double y2) {
      // implementation
  }
  static public void draw_a_circle(
    double x,double y,double r) {
      // implementation
  }
}

class DP2 {
  static public void drawline (
    double x1,double x2,
    double y1,double y2) {
      // implementation
  }
  static public void drawcircle (
    double x,double y,double r) {
    // implementation
   }
}
```

The Bridge Pattern in Retrospect

The essence of the pattern

Now that you've seen how the Bridge pattern works, it is worth looking at it from a more conceptual point of view. As shown in Figure 9-13, the pattern has an abstraction part (with its derivations) and an implementation part. When designing with the Bridge pattern, it is useful to keep these two parts in mind. The implementation's interface should be designed considering the different derivations of the abstract class that it will have to support. Note that a designer shouldn't necessarily put in an interface that will implement all possible derivations of the abstract class (yet another possible route to paralysis by analysis). Only those derivations that actually are being built need be supported. Time and time again, the authors have seen that the mere consideration of flexibility at this point often greatly improves a design.

Note: In C++, the Bridge pattern's implementation must be implemented with an abstract class defining the public interface. In Java, either an abstract class or an interface can be used. The choice depends upon whether implementations share common traits that abstract classes can take advantage of. See Peter Coad's *Java Design*, discussed on page 316 of the Bibliography, for more on this.

Field Notes: Using the Bridge Pattern

The Bridge pattern often incorporates the Adapter pattern

Note that the solution presented in Figures 9-12 and 9-13 integrates the Adapter pattern with the Bridge pattern. I do this because I was given the drawing programs that I must use. These drawing programs have preexisting interfaces with which I must work. I must use the Adapter to adapt them so that they can be handled in the same way.

While it is very common to see the Adapter pattern incorporated into the Bridge pattern, the Adapter pattern is not part of the Bridge pattern.

The Bridge Pattern: Key Features

Intent	Decouple a set of implementations from the set of objects using them.
Problem	The derivations of an abstract class must use multiple implementations without causing an explosion in the number of classes.
Solution	Define an interface for all implementations to use and have the derivations of the abstract class use that.
Participants and Collaborators	The **`Abstraction`** defines the interface for the objects being implemented. The **`Implementor`** defines the interface for the specific implementation classes. Classes derived from the **`Abstraction`** use classes derived from the **`Implementor`** without knowing which particular **`ConcreteImplementor`** is in use.
Consequences	The decoupling of the implementations from the objects that use them increases extensibility. Client objects are not aware of implementation issues.
Implementation	• Encapsulate the implementations in an abstract class. • Contain a handle to it in the base class of the abstraction being implemented. *Note:* In Java, you can use interfaces instead of an abstract class for the implementation.
GoF Reference	Pages 151–162.

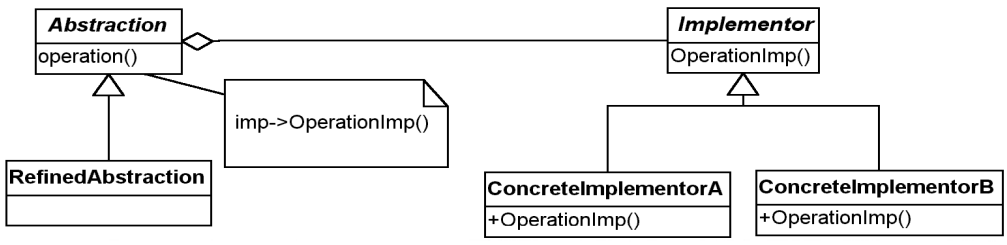

Figure 9-15 Standard, simplified view of the Bridge pattern.

Compound design patterns

When two or more patterns are tightly integrated (like my Bridge and Adapter), the result is called a composite design pattern.[7,8] It is now possible to talk about patterns of patterns!

Instantiating the objects of the Bridge pattern

Another thing to notice is that the objects representing the abstraction (the **`Shape`**s) were given their implementation while being instantiated. This is not an inherent part of the pattern, but it is very common.

Now that you understand the Bridge pattern, it is worth reviewing the Gang of Four's Implementation section in their description of the pattern. They discuss different issues relating to how the abstraction creates and/or uses the implementation.

An advantage of Java over C++ in the Bridge pattern

Sometimes when using the Bridge pattern, I will share the implementation objects across several abstraction objects.

- In Java, this is no problem; when all the abstraction objects go away, the garbage collector will realize that the implementation objects are no longer needed and will clean them up.
- In C++, I must somehow manage the implementation objects. There are many ways to do this; keeping a reference counter or even using the Singleton pattern are possibilities. It is nice, however, not to have to consider this effort. This illustrates another advantage of automatic garbage collection.

---

7. Compound design patterns used to be called composite design patterns, but are now called compound design patterns to avoid confusion with the composite pattern.
8. For more information, refer to Riehle, D., "Composite Design Patterns," In, *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Application*s (OOPSLA '97), New York: ACM Press, 1997, pp. 218–228. Also refer to "Composite Design Patterns (They Aren't What You Think)," *C++ Report*, June 1998.

The Bridge pattern solution is good, but not always perfect

While the solution I developed with the Bridge pattern is far superior to the original solution, it is not perfect. One way of measuring the quality of a design is to see how well it handles variation. Handling a new implementation is very easy with a Bridge pattern in place. The programmer simply needs to define a new concrete implementation class and implement it. Nothing else changes.

However, things may not go so smoothly if I get a new concrete example of the abstraction. I may get a new kind of **`Shape`** that can be implemented with the implementations already in the design. However, I may also get a new kind of **`Shape`** that requires a new drawing function. For example, I may have to implement an ellipse. The current **`Drawing`** class does not have the proper method to do ellipses. In this case, I have to modify the implementations. However, even if this occurs, I at least have a well-defined process for making these changes (that is, modify the interface of the **`Drawing`** class or interface, and modify each **`Drawing`** derivative accordingly)—this localizes the impact of the change and lowers the risk of an unwanted side effect.

Bottom line: Patterns do not always give perfect solutions. However, because patterns represent the collective experience of many designers over the years, they are often better than the solutions you or I might come up with on our own.

Follow one rule, one place to help with refactoring

In the real world, I do not always start out with multiple implementations. Sometimes, I know that new ones are *possible*, but they show up unexpectedly. One approach is to prepare for multiple implementations by always using abstractions. You get a very generic application.

But I do not recommend this approach. It leads to an unnecessary increase in the number of classes you have. It is important to write code in such a way that when multiple implementations do occur (which they often will), it is not difficult to modify the code to

incorporate the Bridge pattern. Modifying code to improve its structure without adding function is called *refactoring*. As defined by Martin Fowler, "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."[9]

In designing code, I was always attending to the possibility of refactoring by following the one rule, one place mandate. The `drawLine` method was a good example of this. Although the place the code was actually implemented varied, moving it around was fairly easy.

Refactoring.

Refactoring is commonly used in object-oriented design. However, it is not strictly an OO thing . . . It is modifying code to improve its structure without adding function.

A useful way to look at the bridge pattern

While deriving the pattern, I took the two variations present (shapes and drawing programs) and encapsulated each in their own abstract class. That is, the variations of shapes are encapsulated in the **`Shape`** class, the variations of drawing programs are encapsulated in the **`Drawing`** class.

Stepping back and looking at these two polymorphic structures, I should ask myself, "What do these abstract classes represent?" For the shapes, it is pretty evident that the class represents different kinds of shapes. The **`Drawing`** abstract class represents how I will implement the **`Shape`**s. Thus, even in the case where I described how new requirements for the **`Drawing`** class may arise (say, if I need to implement ellipses) there is a clear relationship between the classes.

9. Fowler, M., *Refactoring: Improving the Design of Existing Code*, Reading, Mass.: Addison-Wesley, 2000, p. xvi.

Summary

In this chapter

In learning the Bridge pattern, I looked at a problem where there were two variations in the problem domain—shapes and drawing programs. In the problem domain, each of these varied. The challenge came in trying to implement a solution based on all of the special cases that existed. The initial solution, which naively used inheritance too much, resulted in a redundant design that had tight coupling and low cohesion, and was thus difficult to maintain.

You learned the Bridge pattern by following the basic strategies for dealing with variation:

- Find what varies and encapsulate it.
- Favor composition over inheritance.

Finding what varies is always a good step in learning about the problem domain. In the drawing program example, I had one set of variations using another set of variations. This indicates that the Bridge pattern will probably be useful.

In general, you should identify which patterns to use by matching them with the characteristics and behaviors in the problem domain. By understanding the *whys* and *whats* of the patterns in your repertoire, you can be more effective in picking the ones that will help you. You can select patterns to use before deciding how the pattern's implementation will be done.

By using the Bridge pattern, the design and implementation are more robust and better able to handle changes in the future.

Summary of object-oriented principles used in the Bridge pattern

While I focused on the pattern during the chapter, it is worth pointing out several object-oriented principles that are used in the Bridge pattern.

Concept	Discussion
Objects are responsible for themselves	I had different kinds of **`Shape`**s, but all drew themselves (via the *`draw`* method). The **`Drawing`** classes were responsible for drawing elements of objects.
Abstract class	I used abstract classes to represent the concepts. I actually had rectangles and circles in the problem domain. The concept "**`Shape`**" is something that lives strictly in our head, a device to bind the two concepts together; therefore, I represent it in the **`Shape`** class as an *abstract class*. **`Shape`** will never get instantiated because it never exists in the problem domain (only **`Rectangle`**s and **`Circle`**s do). The same thing is true with drawing programs.
Encapsulation via an abstract class	I have two examples of encapsulation through the use of an abstract class in this problem. • A client dealing with the Bridge pattern will have only a derivation of **`Shape`** visible to it. However, the client will not know what type of **`Shape`** it has (it will be just a **`Shape`** to the client). Thus, I have encapsulated this information. The advantage of this is if a new type of **`Shape`** is needed in the future, it does not affect the client object. • The **`Drawing`** class hides the different drawing derivations from the **`Shape`**s. In practice, the abstraction may know which implementation it uses because it might instantiate it. See page 155 of the Gang of Four book for an explanation as to why this might be a good thing to do. However, even when that occurs, this knowledge of implementations is limited to the abstraction's constructor and is easily changed.
One rule, one place	The abstract class often has the methods that actually use the implementation objects. The derivations of the abstract class call these methods. This allows for easier modification if needed, and allows for a good starting point even before implementing the entire pattern.

Supplement: C++ Code Examples

Example 9-4 C++ Code Fragments: Rectangles Only

```
void Rectangle::draw () {
   drawLine(_x1,_y1,_x2,_y1);
   drawLine(_x2,_y1,_x2,_y2);
   drawLine(_x2,_y2,_x1,_y2);
   drawLine(_x1,_y2,_x1,_y1);
}

void V1Rectangle::drawLine
   (double x1, double y1,
    double x2, double y2) {
    DP1.draw_a_line(x1,y1,x2,y2);
}

void V2Rectangle::drawLine
   (double x1, double y1,
    double x2, double y2) {
    DP2.drawline(x1,x2,y1,y2);
}
```

Example 9-5 C++ Code Fragments: Rectangles and Circles without Bridge

```
class Shape {
  public: void draw ()=0;
}
class Rectangle : Shape {
  public:
    void draw();
  protected:
    void drawLine(
        double x1,y1, x2,y2)=0;
}
void Rectangle::draw () {
  drawLine(_x1,_y1,_x2,_y1);
  drawLine(_x2,_y1,_x2,_y2);
  drawLine(_x2,_y2,_x1,_y2);
  drawLine(_x1,_y2,_x1,_y1);
}
```

(continued)

Example 9-5 C++ Code Fragments: Rectangles and Circles without Bridge *(continued)*

```
// V1Rectangle and V2Rectangle both derive from
// Rectangle header files not shown
void V1Rectangle::drawLine (
  double x1,y1, x2,y2) {
  DP1.draw_a_line(x1,y1,x2,y2);
}
void V2Rectangle::drawLine (
   double x1,y1, x2,y2) {
   DP2.drawline(x1,x2,y1,y2);
   }
}

class Circle : Shape {
  public:
    void draw() ;
  protected:
    void drawCircle(
      double x, y, z) ;
}
void Circle::draw () {
  drawCircle();

}

// V1Circle and V2Circle both derive from Circle
// header files not shown
void V1Circle::drawCircle (
  DP1.draw_a_circle(x, y, r);
}

void V2Circle::drawCircle (
  DP2.drawcircle(x, y, r);
}
```

Example 9-6 C++ Code Fragments: The Bridge Implemented

```
void main (String argv[]) {
  Shape *s1;
  Shape *s2;
  Drawing *dp1, *dp2;

  dp1= new V1Drawing;
  s1=new Rectangle(dp,1,1,2,2);

  dp2= new V2Drawing;
  s2= new Circle(dp,2,2,4);

  s1->draw();
  s2->draw();

  delete s1; delete s2;
  delete dp1; delete dp2;
}

// NOTE: Memory management not tested.
// Includes not shown.

class Shape {
  public: draw()=0;
  private: Drawing *_dp;
}
Shape::Shape (Drawing *dp) {
  _dp= dp;
}
void Shape::drawLine(
  double x1, double y1,
  double x2, double y2)
    _dp->drawLine(x1,y1,x2,y2);
}

Rectangle::Rectangle (Drawing *dp,
  double x1, y1, x2, y2) :
  Shape( dp) {
  _x1= x1; _x2= x2;
  _y1= y1; _y2= y2;
}
```

(continued)

Example 9-6 C++ Code Fragments: The Bridge Implemented *(continued)*

```
void Rectangle::draw () {
  drawLine(_x1,_y1,_x2,_y1);
  drawLine(_x2,_y1,_x2,_y2);
  drawLine(_x2,_y2,_x1,_y2);
  drawLine(_x1,_y2,_x1,_y1);
}
class Circle {
  public: Circle (
      Drawing *dp,
      double x, double y, double r);
};

Circle::Circle (
    Drawing *dp,
    double x, double y,
    double r) : Shape(dp) {
    _x= x;
    _y= y;
    _r= r;
}

Circle::draw () {
    drawCircle( _x, _y, _r);
}

class Drawing {
  public: virtual void drawLine (
      double x1, double y1,
      double x2, double y2)=0;
};

class V1Drawing :
  public Drawing {
    public: void drawLine (
        double x1, double y1,
        double x2, double y2);
      void drawCircle(
        double x, double y, double r);
};

void V1Drawing::drawLine (
  double x1, double y1,
  double x2, double y2) {
  DP1.draw_a_line(x1,y1,x2,y2);
}
```

(continued)

Example 9-6 C++ Code Fragments: The Bridge Implemented *(continued)*

```
void V1Drawing::drawCircle (
   double x1, double y, double r) {
     DP1.draw_a_circle (x,y,r);
}
class V2Drawing : public
   Drawing {
   public:
     void drawLine (
       double x1, double y1,
       double x2, double y2);
     void drawCircle(
       double x, double y, double r);
};
void V2Drawing::drawLine (
  double x1, double y1,
  double x2, double y2) {
    DP2.drawline(x1,x2,y1,y2);
}

void V2Drawing::drawCircle (
  double x, double y, double r) {
    DP2.drawcircle(x, y, r);
}

// We have been given the implementations for
// DP1 and DP2

class DP1 {
  public:
    static void draw_a_line (
      double x1, double y1,
      double x2, double y2);
    static void draw_a_circle (
      double x, double y, double r);
};

class DP2 {
  public:
    static void drawline (
      double x1, double x2,
      double y1, double y2);
    static void drawcircle (
      double x, double y, double r);
};
```