CHAPTER 10

The Bridge Pattern

Overview

This chapter continues the discussion of design patterns with the Bridge pattern. The Bridge pattern is quite a bit more complex than the other patterns you have learned. It is also much more useful.

This chapter

• Provides an example to help you derive the Bridge pattern. I go into great detail to help you learn this pattern.
• Presents the key features of the pattern.
• Presents some observations on the Bridge pattern from my own practice.

Introducing the Bridge Pattern

According to the Gang of Four, the intent of the Bridge pattern is to “decouple an abstraction from its implementation so that the two can vary independently.”

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

And then: How come I understand every word in this sentence, but I have no idea what it means?

I knew that

- Decouple means to have things behave independently from each other or at least explicitly state what the relationship is.
- 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 (not the derivations of the abstract class, which are called concrete classes). 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, 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. It also 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 aggregation over class inheritance” (as you will see).
Learning the Bridge Pattern: An Example

To help you understand 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.
- Variations in how these concepts are implemented.

You will see that this example has many similarities to the CAD/CAM problem discussed earlier. Rather than give you all the requirements up front, however, 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!

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 10-1. The differences between the drawing programs are summarized in Table 10-1.
Our analysis specifies that we don't want the code that draws the rectangles to worry about what type of drawing program it should use. It occurs to me that because 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. Figure 10-2 shows this.

Table 10-1 Different Drawing Programs

<table>
<thead>
<tr>
<th></th>
<th>DP1</th>
<th>DP2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used to draw a line</td>
<td><code>draw_a_line(x1, y1, x2, y2)</code></td>
<td><code>drawline(x1, x2, y1, y2)</code></td>
</tr>
<tr>
<td>Used to draw a circle</td>
<td><code>draw_a_circle(x, y, r)</code></td>
<td><code>drawcircle(x, y, r)</code></td>
</tr>
</tbody>
</table>

Proper use of inheritance

Figure 10-2 Design for rectangles and drawing programs (DP1 and DP2).
By having an abstract class Rectangle, I take advantage of the fact that the only difference between the different types of Rectangles 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.

Example 10-1 Java Code Fragments

```java
abstract public class Rectangle {
    private double _x1, _y1, _x2, _y2;
    public Rectangle (double x1, double y1, double x2, double y2) {
        x1= x1; _y1= y1; _x2= x2; _y2= y2;
    }
    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);
}
```

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 Rectangles and Circles.

It occurs to me that I can just 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.
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 10-2, 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 10-3.

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 10-2 Java Code Fragments

```java
abstract class Shape {
    abstract public void draw();
}
```
// the only change to Rectangle is
abstract class Rectangle extends Shape {
    //
    // V1Rectangle and V2Rectangle don't change

    abstract public class Circle extends Shape {
        protected double _x, _y, _r;
        public Circle (double x, double y, double r) {
            _x= x; _y= y; _r= r;
        }
        public void draw() {
            drawCircle();
        }
        abstract protected void drawCircle();
    }
    public class V1Circle extends Circle {
        public V1Circle (double x, double y, double r) {
            super(x,y,r);
        }
        protected void drawCircle () {
            DP1.draw_a_circle( _x, _y, _r);
        }
    }
    public class V2Circle extends Circle {
        public V2Circle(double x, double y, double r) {
            super( x, y, r);
        }
        protected void drawCircle () {
            DP2.drawCircle( _x, _y, _r);
        }
    }
}

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 DPI's draw_a_line.

In action, this looks like Figure 10-4.
Even though the class diagram makes it look as if there are many objects, in reality I am only dealing with three objects (see Figure 10-5):

- The **Client** using the rectangle
- The **V1Rectangle** object
- The **DPI** 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.
Unfortunately, this approach introduces new problems. Look back at Figure 10-3 and pay attention to the third row of classes. Consider the following:

---

This solution suffers from combinatorial explosion.
• The classes in this row represent the four specific types of Shapes that I have.

• What happens if I get another drawing program—that is, another variation in implementation? I will have six different kinds of Shapes (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 Shapes (three Shape concepts times three drawing programs).

The class explosion problem arises because in this solution the abstraction (the kinds of Shapes) 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 10-6).

This is exactly the intent of the Bridge pattern: “[t]o decouple an abstraction from its implementation so that the two can vary independently.”

\[ \text{Abstraction 1} \quad \text{Abstraction 2} \quad \text{Abstraction 3} \quad \ldots \]
\[ \text{Implementation A} \quad \text{Implementation B} \quad \text{Implementation C} \quad \ldots \]

\textbf{Figure 10-6} The Bridge pattern separates variations in abstraction and implementation.

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

Looking at Figure 10-3, ask yourself what else is poor about this design.

\[ 2. \text{ Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software, Boston: Addison-Wesley, 1995, p. 151.} \]
• Does there appear to be redundancy?
• Would you say things have strong cohesion or weak 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. Unfortunately, many approaches to teaching object-oriented design focus on the approach of handling variation through specialization, deriving new classes from existing classes. With an overfocus on the “is-ness” of objects, it has programmers create objects in monolithic hierarchies that work reasonably well at first but become more difficult to maintain as time goes on (as discussed in Chapter 9, “The Strategy Pattern”).

As I became an experienced object-oriented designer, I was still stuck in this paradigm of designing based on inheritance—that is, looking at the characteristics of my classes based on their “is-ness” without regard for how complex the structures were becoming.

Thinking with design patterns eventually led me out of this mess. I learned to think about objects in terms of their responsibilities rather than 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 (aggregation).
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 alternative hierarchy shown in Figure 10-7.

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 \textit{DP1} and \textit{DP2} classes.

Unfortunately, I am unable to eliminate the redundancy between the two types of \textit{Rectangles} and the two types of \textit{Circles}, each pair of which has the same \textit{draw} method.

In any event, the class explosion that was present before is still present here.
The sequence diagram for this solution is shown in Figure 10-8.

![Sequence Diagram](image)

**Figure 10-8 Sequence diagram for new approach.**

Although 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.
An Observation About Using Design Patterns

When people begin to look at design patterns, they often focus on the solutions the patterns offer. This seems reasonable because design patterns are advertised as providing good solutions to the problems at hand. However, this is starting at the wrong end. Before trying to apply a solution to a problem, you should understand the problem. Taking an approach that looks for where you can apply patterns tells you 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 our environment, and then describes the core of the solution to that problem....”

What I have done so far in this chapter 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.

**Learning the Bridge Pattern: Deriving It**

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 Coplein\textsuperscript{4} on commonality and variability analysis.

\begin{center}
\textbf{Design Patterns Are Solutions That Occur Again and Again}
\end{center}

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 \textit{derive} the pattern 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. It also demonstrates that it is at least equally important to know these principles as it is to know patterns because the principles always apply, whereas the patterns are found only in certain circumstances.

\textit{A new paradigm for finding objects}

It is almost axiomatic with object-oriented design methods that the designer is supposed to look at 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 implements them by adding methods to the objects. This process of focusing on nouns and verbs typically leads to larger class hierarchies than we would like. I suggest that using commonality and 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 Coplein’s work.)

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

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

In the past, developers often relied on extensive inheritance trees to coordinate these variations. However, the second strategy says to try aggregation 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 shapes. 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 shapes it is dealing with are rectangles and circles. 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 6, “Expanding Our Horizons”).

Follow this process for the rectangle-drawing problem.
First identify what it is that is varying. In this case, it is different types of shapes and different types of drawing programs. The common concepts are therefore shapes and drawing programs. I represent this in Figure 10-9. (Note that the class names are shown in italics because the classes are abstract.)

![Figure 10-9 What is varying?](image)

At this point, I intend **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 10-10.

![Figure 10-10 Represent the variations.](image)
At this point, the diagram is simply notional. I know that \texttt{V1Drawing} will use \texttt{DP1} and \texttt{V2Drawing} will use \texttt{DP2}, but I have not said \textit{how}. I have simply captured the concepts of the problem domain (shapes and drawing programs) and have shown the variations present.

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 10-3 and 10-7 to refresh your memory.) Instead, I want to see whether I can relate these classes by having one use the other (that is, follow the mandate to favor aggregation over inheritance). The question is, which class uses the other?

Consider these two possibilities: either \texttt{Shape} uses the \texttt{Drawing} programs or the \texttt{Drawing} programs use \texttt{Shape}.

Consider the latter case first. If drawing programs could draw shapes directly, 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. \texttt{Drawing} objects would have to know specific information about \texttt{Shapes} (that is, the kind of \texttt{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 \texttt{Shapes} use \texttt{Drawing} objects to draw themselves? \texttt{Shapes} wouldn’t need to know what type of \texttt{Drawing} object they used because I could have \texttt{Shapes} refer to the \texttt{Drawing} class. \texttt{Shapes} also would be responsible for controlling the drawing.

This looks better to me. Figure 10-11 shows this solution.
In this design, \textit{Shape} uses \textit{Drawing} to manifest its behavior. I left out the details of \textit{V1Drawing} using the \textit{DP1} program and \textit{V2Drawing} using the \textit{DP2} program. In Figure 10-12, I add this as well as the protected methods \textit{drawLine} and \textit{drawCircle} (in \textit{Shape}), which calls \textit{Drawing}'s \textit{drawLine} and \textit{drawCircle} respectively.

Figure 10-13 illustrates the separation of the \textit{Shape} abstraction from the \textit{Drawing} implementation.
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.

Although 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 following example, 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.

Figure 10-13 Class diagram illustrating separation of abstraction and implementation.
From a method point of view, this looks fairly similar to the inheritance-based implementation (such as shown in Figure 10-3). The biggest difference is that the methods are now located in different classes.

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 10-13 contains the variations in my abstractions. The hierarchy on the right side of Figure 10-13 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.

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 10-14).
Example 10-3 shows a reasonably complete Java code example.

**Example 10-3 Java Code Fragments**

```java
public class Client {
    static public void main () {
        Shape myShapes[];
        Factory myFactory= new Factory;

        // get rectangles from some other source
        myShapes= myFactory.getShapes();
        for (int i= 0; i < myShapes.length; i++) {
            myShapes[i].draw();
        }
    }
}

abstract public class Shape {
    protected Drawing myDrawing;
    abstract public void draw();

    Shape (Drawing drawing) {
        myDrawing= drawing;
    }

    protected void drawLine (double x1,double y1, double x2,double y2) {
        myDrawing.drawLine(x1,y1,x2,y2);
    }

    protected void drawCircle (double x,double y,double r) {
        myDrawing.drawCircle(x,y,r);
    }
}

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

    public void draw() {
        drawLine( _x1, _y1, _x2, _y1);
        drawLine( _x2, _y1, _x2, _y2);
        drawLine( _x2, _y2, _x1, _y2);
    }
protected void drawLine(double x1, double y1, double x2, double y2) {
    myDrawing.drawLine(x1, y1, x2, y2);
}

public class Circle extends Shape {
    private double _x, _y, _r;
    public Circle (Drawing dp, double x, double y, double r) {
        super(dp);
        _x= x; _y= y; _r= r;
    }
    public void draw() {
        myDrawing.drawCircle(_x, _y, _r);
    }
}

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

public 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);
    }
}

public 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) {
    }
}
The Bridge Pattern in Retrospect

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 10-13, 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 define an interface that will account for all conceivable 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 C# and Java, either an abstract class or an interface can be used. The choice depends on whether implementations share common traits that abstract classes can take advantage of.5

Field Notes: Using the Bridge Pattern

Print drivers are perhaps the class example of the Bridge. They are also the easiest to see when to apply the Bridge. The real power of the Bridge pattern, in my mind, however, is that it helps me see

when to abstract out the implementations that are present in my problem domain. In other words, sometimes I’ll have an entity X that uses a system S and an entity Y that uses a system T. I may think that X always and only comes with S and Y always and only comes with T and link them (couple) them together. The Bridge reminds me that I may be better off abstracting out the differences between S and T (the implementations of X and Y) and allowing X and Y to use either S and T. In other words, the Bridge is most useful when I might not have decoupled my abstraction from my implementation unless I considered whether the Bridge pattern applied.

The Bridge pattern often incorporates the Adapter pattern. Note that the solution presented in Figures 10-12 and 10-13 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.

Although it is very common to see the Adapter pattern used with the Bridge pattern, the Adapter pattern is not part 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 compound design pattern. 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 Shapes) were given their implementation while being instantiated. This is not an inherent part of the pattern, but it is very common.

---

6. Compound design patterns used to be called composite design patterns, but are now called compound design patterns to avoid confusion with the Composite 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
Abstraction defines the interface for the objects being implemented. Implementor defines the interface for the specific implementation classes. Classes derived from Abstraction use classes derived from Implementor without knowing which particular ConcreateImplementor 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.

![Figure 10-15 Generic structure of the Bridge pattern.]

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

- In C# and 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.

Although 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 just 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, at least I have a well-defined process for making changes: Modify the interface of the Drawing class or interface and then modify each Drawing derivative accordingly. This process localizes the impact of the change and lowers the risk of an unwanted side effect.

I am left with a fine solution, even if it is not “perfect.” And the Bridge pattern has given me a handle on the problem if I need to
consider a more general implementation. Design patterns help me think more abstractly and more generally about my solutions. Whether I want to implement this more general solution is up to me; the pattern doesn’t mandate that.

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

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.”

When 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.

While deriving the pattern, I took the two variations present (shapes and drawing programs) and encapsulated each in its 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 Shapes. The pattern is about the relationship between these different abstractions. Thus, 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 that tells me how to implement it.

Summary

While reviewing 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 when trying to implement a solution based on all the special cases that existed. The initial solution, which naively used inheritance too much, resulted in a redundant design that had tight coupling and weak 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 aggregation 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 consider 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 consider before deciding how the pattern’s implementation will be done.

**Consideration vs. Use**

I used the word *consider* rather than *use* in the prior paragraph deliberately. Actually, you should “use” patterns by “considering” the issues they imply and the body of knowledge about them. Unfortunately, when people hear the word *use* regarding a pattern, they tend to think about “using the implementation” of the pattern. The word *consider* helps people realize that they need to use the pattern as a guideline, a list of considerations.

If you use⁹ the Bridge pattern, the design and implementation are more robust and better able to handle changes in the future.

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

---

⁹. Of course, now I mean use as in “using the pattern to consider the issues, forces, etc., present.”
## Concept Discussion

<table>
<thead>
<tr>
<th>Concept</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects are responsible for themselves</td>
<td>I had different kinds of Shapes, but all drew themselves (via the <code>draw</code> method). The Drawing classes were responsible for drawing elements of objects.</td>
</tr>
<tr>
<td>Abstract class</td>
<td>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 Rectangles and Circles do). The same thing is true with drawing programs.</td>
</tr>
<tr>
<td>Encapsulation via an abstract class</td>
<td>I have two examples of encapsulation through the use of an abstract class in this problem:</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• The Drawing class hides the different drawing derivations from the Shapes. 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.</td>
</tr>
<tr>
<td>One rule, one place</td>
<td>The abstract class often has the methods that actually use the implementation objects. The subclasses 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.</td>
</tr>
<tr>
<td>Testability</td>
<td>Imagine writing tests for the shapes and drawing programs with both our original solution and our later solution. For example, suppose you have N shapes and M implementations. The first solution would require N+M tests. The second solution only requires M+N tests: First test the M implementations, and then test the N shapes with arbitrarily chosen implementations (because all of the shapes work with all of the implementors in exactly the same way).</td>
</tr>
</tbody>
</table>
Review Questions

Observations

1. Define *decouple* and *abstraction*.

2. How is *implementation* defined in the context of the Bridge pattern?

3. What are the basic elements of a sequence diagram?

4. What is Alexander’s view of how to use patterns? Does he advocate starting with the solution first or the problem to be solved first?

5. What does commonality analysis seek to identify? What does variability analysis seek to identify?

6. What is the basic problem being solved by the Bridge pattern?

7. Define the “one rule, one place” strategy.

8. What are the consequences of the Bridge pattern?

Interpretations

1. The Gang of Four says that the intent of the Bridge pattern is to “decouple an abstraction from its implementation so that the two can vary independently.”
   - What does this mean?
   - Give an example.

2. Why can tight coupling lead to an explosion in the number of classes?
Opinions and Applications

1. “Look at objects in terms of their responsibilities rather than their behaviors.” How does this affect your view of the use of inheritance in an object-oriented system?

2. Why do you think the Gang of Four call this pattern “Bridge”? Is it an appropriate name for what it is doing? Why or why not?
Business-Driven Software Development (BDSD) is Net Objectives’ proprietary integration of Lean-Thinking with Agile methods across the business, management and development teams to maximize the value delivered from a software development organization. BDSD has built a reputation and track record of delivering higher quality products faster and with lower cost than other methods.

BDSD goes beyond the first generation of Agile methods such as Scrum and XP by viewing the entire value stream of development. Lean-Thinking enables product portfolio management, release planning and critical metrics to create a top-down vision while still promoting a bottom-up implementation.

BDSD integrates business, management and teams. Popular Agile methods, such as Scrum, tend to isolate teams from the business side and seem to have forgotten management’s role altogether. These are critical aspects of all successful organizations. In BDSD:

- **Business** provides the vision and direction; properly selecting, sizing and prioritizing those products and enhancements that will maximize your investment.
- **Teams** self-organize and do the work; consistently delivering value quickly while reducing the risk of developing what is not needed.
- **Management** bridges the two; providing the right environment for successful development by creating an organizational structure that removes impediments to the production of value. This increases productivity, lowers cost and improves quality.

**Become a Lean-Agile Enterprise**

All levels of your organization will experience impacts and require change management. We help prepare executive, mid-management and the front-line with the competencies required to successfully change the culture to a Lean-Agile enterprise.

**Prioritization is only half the problem.** Learn how to both prioritize and size your initiatives to enable your teams to implement them quickly.

**Learn to come from business need not just system capability.** There is a disconnect between the business side and development side in many organizations. Learn how BDSD can bridge this gap by providing the practices for managing the flow of work.

**Why Net Objectives**

While many organizations are having success with Agile methods, many more are not. Much of this is due to organizations either starting in the wrong place (e.g., the team when that is not the main problem) or using the wrong method (e.g., Scrum, just because it is popular). Net Objectives is experienced in all of the Agile team methods (Scrum, XP, Kanban, Scrumban) and integrates business, management and teams. This lets us help you select the right method for you.

**Assessments**

See where you are, where you want to go, and how to get there.

**Business and Management Training**

Lean Software Development
Product Portfolio Management
Enterprise Release Planning

**Productive Lean-Agile Team Training**

Team training in Kanban, Scrum
Technical Training in ATDD, TDD, Design Patterns

**Roles Training**

Lean-Agile Project Manager
Product Owner

For more information about Business Driven Software Development Services
Contact Mike Shalloway at Net Objectives: mike.shalloway@netobjectives.com / 404-593-8375
NetObjectives.com / 1-888-LEAN-244 (888-532-6244)