CHAPTER 8

Expanding Our Horizons

Overview

In this chapter

In previous chapters, I discussed three fundamental concepts of object-oriented design: objects, encapsulation, and abstract classes. How a designer views these concepts is important. The traditional ways are simply too limiting. In this chapter I step back and reflect on topics discussed earlier in the book. My intent is to describe a new way of seeing object-oriented design, which comes from the perspective that design patterns create.

In this chapter,

- I compare and contrast the traditional way of looking at objects—as a bundle of data and methods—with the new way—as things with responsibilities.
- I compare and contrast the traditional way of looking at encapsulation—as hiding data—with the new way—as the ability to hide anything. Especially important is to see that encapsulation can be used to contain variation in behavior.
- I compare and contrast the traditional way of using inheritance—for specialization and reuse—with the new way—as a method of classifying objects.
- The new viewpoints allow for containing variation of behaviors in objects.
- I show how the conceptual, specification, and implementation perspectives relate to an abstract class and its derived classes.

Acknowledgment

Perhaps this new perspective is not all that original. I believe that this perspective is one that many developers of the design patterns held when they developed what ended up being called a pattern. Certainly, it is a perspective that is consistent with the writings of Christopher Alexander, Jim Coplien, and the Gang of Four.

While it may not be original, it has also not been expressed in quite the way I do in this chapter and in this book. I have had to distill this way of looking at patterns from the way design patterns behave and how they have been described by others.

When I call it a new perspective, what I mean is that it is most likely a new way for most developers to view object orientation. It was certainly new to me when I was learning design patterns for the first time.

Objects: the Traditional View and the New View

The traditional view: data with methods

The traditional view of objects is that they are data with methods. One of my teachers called them "smart data." It is just a step up from a database. This view comes from looking at objects from an implementation perspective.

The new view: things with responsibilities

While this definition is accurate, as explained in Chapter 1, "The Object-Oriented Paradigm," it is based on the implementation perspective. A more useful definition is one based on the conceptual perspective—an object is an entity that has responsibilities. These responsibilities give the object its behavior. Sometimes, I also think of an object as an entity that has specific behavior.

This is a better definition because it helps to focus on what the objects are supposed to *do*, not simply on how to implement them. This enables me to build the software in two steps:

1. Make a preliminary design without worrying about all of the details involved.
2. Implement the design achieved.

Ultimately, this perspective allows for better object selection and definition (in a sense, the main point of design anyway). Object definition is more flexible; by focusing on what an object does, inheritance allows us to use different, specific behaviors when needed. A focus on implementation may achieve this, but flexibility typically comes at a higher price.

It is easier to think in terms of responsibilities because that helps to define the object's public interface. If an object has a responsibility, there must be some way to ask it to perform its responsibility. However, it does not imply anything about what is *inside* the object. The information for which the object is responsible may not even be inside the object itself.

For example, suppose I have a **`Shape`** object and its responsibilities are

- To know where it is located
- To be able to draw itself on a display
- To be able to remove itself from a display

These responsibilities imply that a particular set of method calls must exist:

- `getLocation( ... )`
- `drawShape( ... )`
- `unDrawShape( ... )`

There is no implication about what is inside of **`Shape`**. I only care that **`Shape`** is responsible for its own behaviors. It may have

attributes inside it or it may have methods that calculate or even refer to other objects. Thus, **`Shape`** might contain attributes about its location or it might refer to another database object to get its location. This gives you the flexibility you need to meet your modeling objectives.

Interestingly, you will find that focusing on motivation rather than on implementation is a recurring theme in design patterns.

Look at objects this way. Make it your basic viewpoint for objects. If you do, you will have superior designs.

Encapsulation: the Traditional View and the New View

My object-oriented umbrella

In my classes on pattern-oriented design, I often ask my students, "Who has heard encapsulation defined as 'data hiding'?" Almost everyone raises his or her hand.

Then I proceed to tell a story about my umbrella. Keep in mind that I live in Seattle, which has a reputation for being wetter than it is, but is still a pretty wet place in the fall, winter, and spring. Here, umbrellas and hooded coats are personal friends!

Let me tell you about my great umbrella. It is large enough to get into! In fact, three or four other people can get in it with me. While we are in it, staying out of the rain, I can move it from one place to another. It has a stereo system to keep me entertained while I stay dry. Amazingly enough, it can also condition the air to make it warmer or colder. It is one cool umbrella.

My umbrella is convenient. It sits there waiting for me. It has wheels on it so that I do not have to carry it around. I don't even have to push it because it can propel itself. Sometimes, I will open the top of my umbrella to let in the sun. (Why I am using my umbrella when it is sunny outside is beyond me!)

In Seattle, there are hundreds of thousands of these umbrellas in all kinds of colors.

Most people call them *cars*.

But I think of mine as an umbrella because an umbrella is something you use to keep out of the rain. Many times, while I am waiting outside for someone to meet me, I sit in my "umbrella" to stay dry!

Definitions can be limitations

Of course, a car isn't really an umbrella. Yes, you can use it to say out of the rain, but that is too limited a view of a car. In the same way, encapsulation isn't just for hiding data. That is too limited a view of encapsulation. To think of it that way constrains my mind when I design.

How to think about encapsulation

Encapsulation should be thought of as "any kind of hiding." In other words, it *can* hide data. But it can also hide implementations, derived classes, or any number of things. Consider the diagram shown in Figure 8-1. You might recollect this diagram from Chapter 7, "The Adapter Pattern."

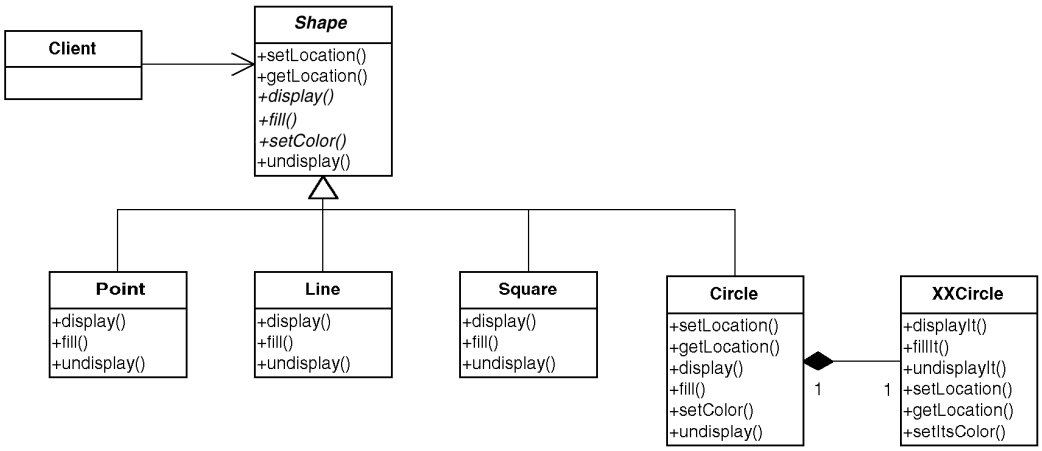

Figure 8-1 Adapting `XXCircle` with `Circle`.

Multiple levels of encapsulation

Figure 8-1 shows many kinds of encapsulation:

- *Encapsulation of data*—The data in **`Point`**, **`Line`**, **`Square`**, and **`Circle`** are hidden from everything else.
- *Encapsulation of methods*—For example, **`Circle`**'s `setLocation`.
- *Encapsulation of subclasses*—Clients of **`Shape`** do not see **`Point`**s, **`Line`**s, **`Square`**s, or **`Circle`**s.
- *Encapsulation of other objects*—Nothing but **`Circle`** is aware of **`xxCircle`**.

One type of encapsulation is thus achieved when there is an abstract class that behaves polymorphically without the client of the abstract class knowing what kind of derived class actually is present. Furthermore, adapting interfaces hides what is behind the adapting object.

The advantage of this new definition

The advantage of looking at encapsulation this way is that it gives me a better way to split up (decompose) my programs. The encapsulating layers become the interfaces I design to. By encapsulating different kinds of **`Shape`**s, I can add new ones without changing any of the client programs using them. By encapsulating **`XXCircle`** behind **`Circle`**, I can change this implementation in the future if I choose to or need to.

Inheritance as a concept versus inheritance for reuse

When the object-oriented paradigm was first presented, reuse of classes was touted as being one of its big benefits. This reuse was usually achieved by creating classes and then deriving new classes based on these base classes. Hence the term *specialized* classes for those subclasses that were derived from other classes (which were called *generalized* classes).

I am not arguing with the accuracy of this, rather I am proposing what I believe to be a more powerful way of using inheritance. In the example above, I can do my design based on different special

types of **Shape**s (that is, **Point**s, **Line**s, **Square**s and **Circle**s). However, this will probably not have me hide these special cases when I think about using **Shape**s—I will probably take advantage of the knowledge of these concrete classes.

If, however, I think about **Shape**s as a way of classifying **Point**s, **Line**s, **Square**s and **Circle**s, I can more easily think about them as a whole. This will make it more likely I will design to an interface (**Shape**s). It also means if I get a new **Shape**, I will be less likely to have designed myself into a difficult maintenance position (because no client object knows what kind of **Shape** it is dealing with anyway).

Find What Is Varying and Encapsulate It

Using inheritance this way in design patterns

In *Design Patterns: Elements of Reusable Object-Oriented Software,* the Gang of Four suggests the following:

> *Consider what should be variable in your design.* This approach is the opposite of focusing on the cause of redesign. Instead of considering what might *force* a change to a design, consider what you want to be *able* to change without redesign. The focus here is on *encapsulating the concept that varies,* a theme of many design patterns.[1]

Or, as I like to rephrase it, "Find what varies and encapsulate it."

These statements seem odd if you only think about encapsulation as data-hiding. They are much more sensible when you think of encapsulation as hiding classes using abstract classes. Using composition of a reference to an abstract class hides the variations.

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, Mass.: Addison-Wesley, 1995, p. 29.

In effect, many design patterns use encapsulation to create layers between objects—enabling the designer to change things on different sides of the layers without adversely affecting the other side. This promotes loose-coupling between the sides.

This way of thinking is very important in the Bridge pattern, which will be discussed in Chapter 9, "The Bridge Pattern." However, before proceeding, I want to show a bias in design that many developers have.

Containing variation in data versus containing variation in behavior

Suppose I am working on a project that models different characteristics of animals. My requirements are the following:

- Each type of animal can have a different number of legs.
  - Animal objects must be able to remember and retrieve this information.
- Each type of animal can have a different type of movement.
  - Animal objects must be able to return how long it will take to move from one place to another given a specified type of terrain.

A typical approach of handling the variation in the number of legs would be to have a data member containing this value and having methods to set and get it. However, one typically takes a different approach to handling variation in behavior.

Suppose there are two different methods for moving: walking and flying. These requirements need two different pieces of code: one to handle walking and one to handle flying; a simple variable won't work. Given that I have two different methods, I seem to be faced with a choice of approach:

- Having a data member that tells me what type of movement my object has.

- Having two different types of **`Animal`**s (both derived from the base **`Animal`** class)—one for walking and one for flying.

Unfortunately, both of these approaches have problems:

- *Tight coupling*—The first approach (using a flag with presumably a switch based on it) may lead to tight coupling if the flag starts implying other differences. In any event, the code will likely be rather messy.
- *Too many details*—The second approach requires that I also manage the subtype of **`Animal`**. And I cannot handle **`Animal`**s that can both walk and fly.

Handling variation in behavior with objects

A third possibility exists: have the **`Animal`** class contain an object that has the appropriate movement behavior. I show this in Figure 8-2.

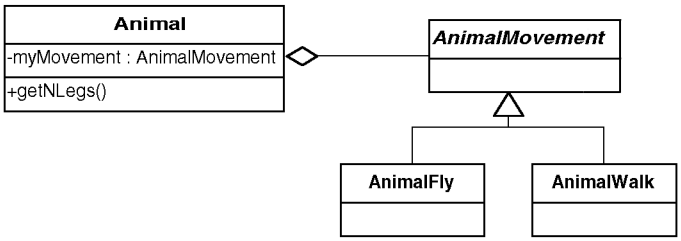

Figure 8-2 `Animal` containing `AnimalMovement` object.

Overkill?

This may seem like overkill at first. However, it's nothing more than an **`Animal`** containing an object that contains the movement behavior of the **`Animal`**. This is very analogous to having a member containing the number of legs—in which case an intrinsic type object is containing the number of legs. I suspect these appear more different in concept than they really are, because Figures 8-2 and 8-3 appear to be different.

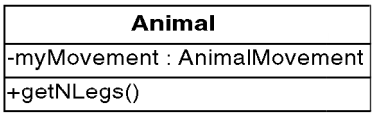

Figure 8-3 Showing containment as a member.

Comparing the two

Many developers tend to think that one object containing another object is inherently different from an object having a mere data member. But data members that appear not to be objects (integers and doubles, for example) really are. In object-oriented programming, *everything* is an object, even these intrinsic data types, whose behavior is arithmetic.

Using objects to contain variation in attributes and using objects to contain variation in behavior are very similar; this can be most easily shown through an example. Let's say I am writing a point-of-sale system. In this system, there is a sales receipt. On this sales receipt there is a total. I could start out by making this total be a type **`double`**. However, if I am dealing with an international application, I quickly realize I need to handle monetary conversions, and so forth. I might therefore make a **`Money`** class that contains an amount and a currency. Total can now be of type **`Money`**.

Using the **`Money`** class this way appears to be using an object just to contain more data. However, when I need to convert **`Money`** from one currency to the next, it is the **`Money`** object itself that should do the conversion, because objects should be responsible for themselves. At first it may appear that this conversion can be done by simply having another data member that specifies what the conversion factor is.

However, it may be more complicated than this. For example, perhaps I need to be able to convert currency based on past dates. In that case, if I add behaviors to the **`Money`** or **`Currency`** classes I am essentially adding different behaviors to the **`SalesReceipt`** as well,

based upon which **`Money`** objects (and therefore which **`Currency`** objects) it contains.

I will demonstrate this strategy of using contained objects to perform required behavior in the next few design patterns.

Commonality/Variability and Abstract Classes

Object-oriented design captures all three perspectives

Consider Figure 8-4. It shows the relationship between

- Commonality/variability analysis
- The conceptual, specification, and implementation perspectives
- An abstract class, its interface, and its derived classes

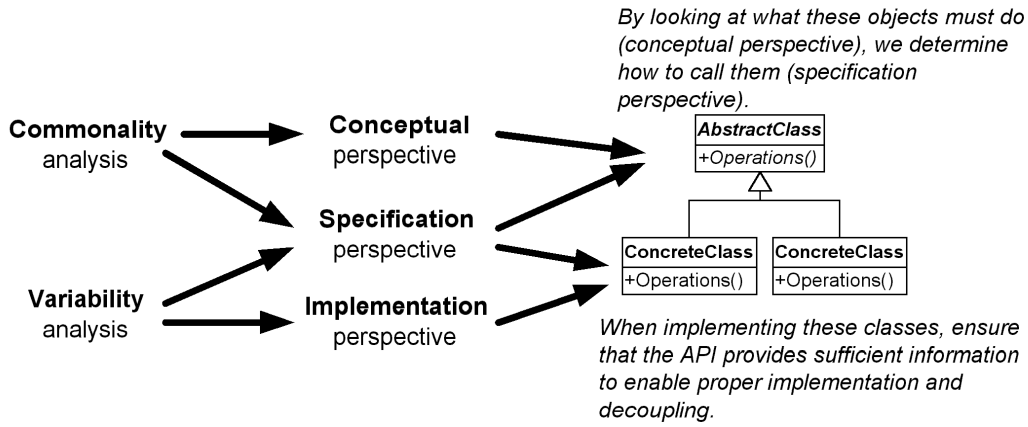

Figure 8-4 The relationship between commonality/variability analysis, perspectives, and abstract classes.

As you can see in Figure 8-4, commonality analysis relates to the conceptual view of the problem domain and variability analysis relates to the implementation, that is, to specific cases.

Now, specification gives a better understanding of abstract classes

The specification perspective lies in the middle. Both commonality and variability are involved in this perspective. The specification describes how to communicate with a set of objects that are conceptually similar. Each of these objects represents a variation of the common concept. This specification becomes an abstract class or an interface at the implementation level.

In the new perspective of object-oriented design, I can now say the following:

Mapping with Abstract Classes	Discussion
Abstract class → the central binding concept	An abstract class represents the core concept that binds together all of the derivatives of the class. This core concept is what defines the commonality.
Commonality → which abstract classes to use	The commonalities define the abstract classes I need to use.
Variations → derivation of an abstract class	The variations identified *within* that commonality become derivations of the abstract classes.
Specification → interface for abstract class	The interface for these classes corresponds to the specification level.

This simplifies the design process of the classes into a two-step procedure:

When Defining . . .	You Must Ask Yourself. . .
An abstract class (commonality)	What *interface* is needed to handle all of the *responsibilities* of this class?
Derived classes	Given this particular implementation (this variation), how can I implement it with the given specification?

The relationship between the specification perspective and the conceptual perspective is this: *It identifies the interface I need to use to handle all of the cases of this concept (that is, the commonality).*

The relationship between the specification perspective and the implementation perspective is this: *Given this specification, how can I implement this particular case (this variation)?*

Summary

In this chapter

The traditional way of thinking about objects, encapsulation, and inheritance is very limiting. Encapsulation exists for so much more than simply hiding data. By expanding the definition to include any kind of hiding, I can use encapsulation to create layers between objects—enabling me to change things on one side of a layer without adversely affecting the other side.

Inheritance is better used as a method of consistently dealing with different concrete classes that are conceptually the same rather than as a means of specialization.

The concept of using objects to hold variations in behavior is not unlike the practice of using data members to hold variations in data. Both allow for the encapsulation (and therefore extension) of the data/behavior being contained.