CHAPTER 14

The Strategy Pattern

Overview

In this chapter

This chapter introduces a new case study, which comes from the area of *e-tailing* (electronic retailing over the Internet). It also begins a solution using the Strategy pattern. The solution to this case study will continue to evolve through Chapter 20, "The Analysis Matrix."

In this chapter,

- I describe an approach to handling new requirements.
- I introduce the new case study.
- I describe the Strategy pattern and show how it handles a new requirement in the case study.
- I describe the key features of the Strategy pattern.

An Approach to Handling New Requirements

Disaster often comes in the long run from suboptimal decisions in the short run

Many times in life and many times in software applications, you have to make choices about the general approach to performing a task or solving a problem. Most of us have learned that taking the easiest route in the short run can lead to serious complications in the long run. For example, none of us would ignore oil changes for our car beyond a certain point. True, I may not change the oil every 3,000 miles, but I also do not wait until 30,000 miles before changing the oil (if I did so, there would be no need to change the oil any more: the car would not work!). Or consider desktop filing—the technique many of us have of using the tops of our desks as a filing cabinet. It works well in the short run, but in the long run, it

becomes tough to find things as the piles grow. Disaster often comes in the long run from suboptimal decisions made in the short run.

This is true in software as well: we focus on immediate concerns and ignore the longer term

Unfortunately, when it comes to software development, many people have not learned these lessons yet. Many projects are only concerned with handling immediate, pressing needs, without concern for future maintenance. There are several reasons projects tend to ignore long-term issues like ease of maintenance or ability to change. Common excuses include

- "We really can't figure out how the new requirements are going to change."
- "If we try to see how things will change, we'll stay in analysis forever."
- "If we try to write our software so it can add new functionality, we'll stay in design forever."
- "We don't have the time or budget to do so."

The choices seem to be

- Overanalyze or overdesign—I like to call this "paralysis by analysis," or
- Just jump in, write the code without concern for long-term issues, and then get on another project before this short sightedness causes too many problems. I like to call this "abandon (by) ship (date)!"

Since management is under pressure to deliver and not to maintain, maybe these results are not surprising. However, with a moment's reflection, it becomes apparent that there is an underlying belief system that prevents many software developers from seeing other alternatives—the belief that designing for change is more costly than designing without considering change.

But this is not necessarily the case. Indeed, the opposite is often true: When you step back to consider how your system may change over time, a better design usually becomes apparent to you—in virtually the same amount of time that would be required to do a "standard" get-it-done-now design.

Designing for change

The approach I use in the following this case study considers how systems may change. However, it is important to note that I will be anticipating that changes will occur and look to see *where* they will occur. I will not be trying to anticipate the exact nature of the change. This approach is based on the principles described in the Gang of Four book:

- *"Program to an interface, not an implementation."*[1]
- *"Favor object composition over class inheritance."*[2]
- *"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."[3]

What I suggest is that when faced with modifying code to handle a new requirement, you should at least consider following these strategies. If following these strategies will not cost significantly more to design and implement, then use them. You can expect a long-term benefit from doing so, with only a modest short-term cost (if any).

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, Mass.: Addison-Wesley, 1995, p. 18.
2. ibid, p. 20.
3. ibid, p. 29.

I am not proposing to follow these strategies blindly, however. I can test the value of an alternative design by examining how well it conforms to the good principles of object-oriented design. This is essentially the same approach I used in deriving the Bridge pattern in Chapter 9, "The Bridge Pattern." In that chapter, I measured the quality of alternative designs by seeing which one followed object-oriented principles the best.

Initial Requirements of the Case Study

A motivating example: an e-tail system

Suppose I am writing an e-tail system that supports sales in the United States. The general architecture has a controller object that handles sales requests. It identifies when a sales order is being requested and hands the request off to a SalesOrder object to process the order.

The system looks something like Figure 14-1.

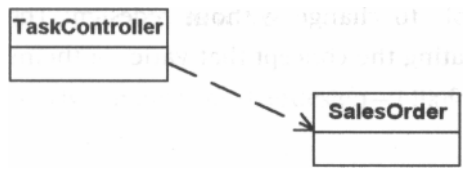

Figure 14-1 Sales order architecture for an e-tail system.

Some features of the system

The functions of SalesOrder include

- Allow for filling out the order with a GUI.
- Handle tax calculations.
- Process the order. Print a sales receipt.

Some of these functions are likely to be implemented with the help of other objects. For example, SalesOrder would not necessarily print itself; rather, it serves as a holder for information about sales

orders. A particular SalesOrder object could call a SalesTicket object that prints the SalesOrder.

Handling New Requirements

New requirement for taxation rules

In the process of writing this application, suppose I receive a new requirement to change the way I have to handle taxes. Now, I have to be able to handle taxes on orders from customers outside the United States. At a minimum, I will need to add new rules for computing these taxes.

One approach: reuse the SalesOrder *object*

How can I handle these new rules? I could attempt to reuse the existing SalesOrder object, processing this new situation like a new kind of sales order, only with a different set of taxation rules. For example, for Canadian sales, I could derive a new class called CanadianSalesOrder from SalesOrder that would override the tax rules. I show this solution in Figure 14-2.

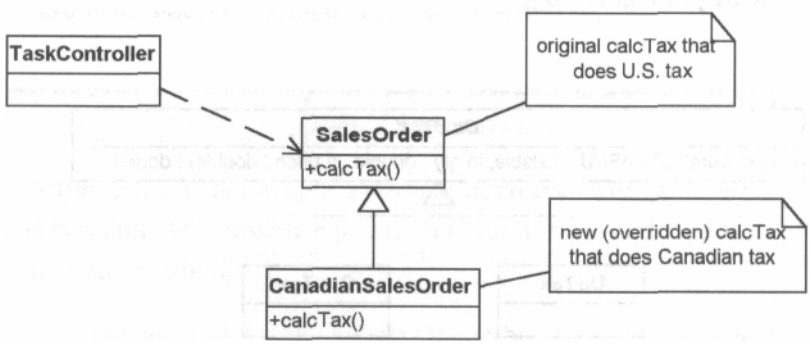

Figure 14-2 Sales order architecture for an e-commerce system.

… but this violates a fundamental rule of design patterns

Now, design patterns repeatedly demonstrate a fundamental rule of design patterns: "Favor object composition over class inheritance."[4] The solution in Figure 14-2 does just the opposite! In other words, I

4. ibid, p. 20.

have handled the variation in tax rules by using inheritance to derive a new class with the new rule.

Take a different approach

How could I approach this differently? Following the rules I stated above: attempt to "consider what should be variable in your design" . . . and "encapsulate the concept that varies."[5]

Following this two-step approach, I should do the following:

1. Find what varies and encapsulate it in a class of its own.
2. Contain this class in another class.

Step 1: Find what varies and encapsulate it

In this example, I have already identified that the tax rules are varying. To encapsulate this would mean creating an abstract class that defines how to accomplish the task conceptually, and then derive concrete classes for each of the variations. In other words, I could create a CalcTax object that defines the interface to accomplish this task. I could then derive the specific versions needed. I show this, in Figure 14-3.

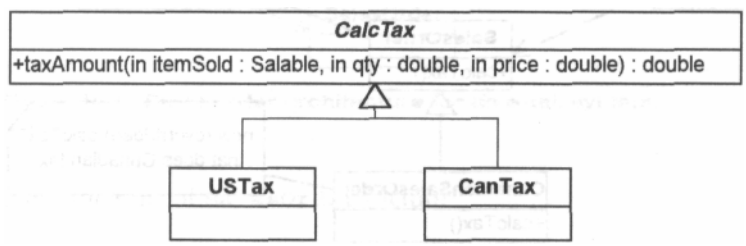

Figure 14-3 Encapsulating tax rules.

5. ibid, p. 29.

Continuing on, I now use composition instead of inheritance. This means, instead of making different versions of sales orders (using inheritance), I will contain the variation with composition. That is, I will have one SalesOrder class and have it contain the CalcTax class to handle the variation. I show this in Figure 14-4.

Step 2: Favor composition

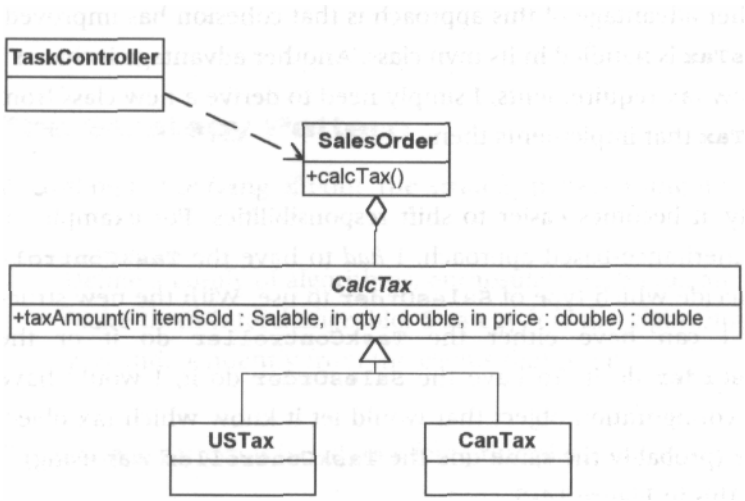

Figure 14-4 Favoring composition over inheritance.

UML Diagrams

In the UML, it is possible to define parameters in the methods. This is done by showing a parameter and its type in the parenthesis of the method.

Thus, in Figure 14-4, the *taxAmount* method has three parameters:

- itemSold of type Salable
- *qty* of type double
- price of type double

All of these are inputs denoted by the "in." The *taxAmount* method also returns a double.

How this works

I have defined a fairly generic interface for the CalcTax object. Presumably, I would have a Saleable class that defines saleable items (and how they are taxed). The SalesOrder object would give that to the CalcTax object, along with the quantity and price. This would be all the information the CalcTax object would need.

Improves cohesion, aids flexibility

Another advantage of this approach is that cohesion has improved. SalesTax is handled in its own class. Another advantage is that as I get new tax requirements, I simply need to derive a new class from CalcTax that implements them.

Easier to shift responsibilities

Finally, it becomes easier to shift responsibilities. For example, in the inheritance-based approach, I *had* to have the TaskController decide which type of SalesOrder to use. With the new structure, I can have either the TaskController do it or the SalesOrder do it. To have the SalesOrder do it, I would have some configuration object that would let it know which tax object to use (probably the same one the TaskController was using). I show this in Figure 14-5.

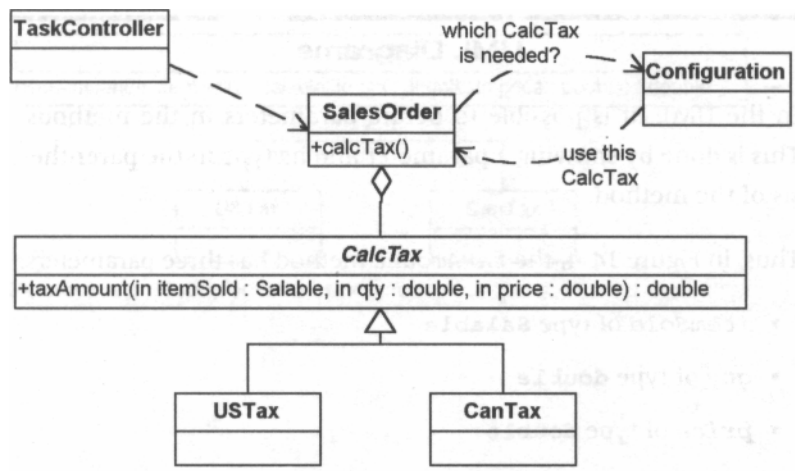

Figure 14-5 The SalesOrder object using Configuration to tell it which CalcTax to use.

The Strategy pattern

This approach allows the business rule to vary independently from the SalesOrder object that uses it. Note how this works well for current variations I have as well as any future ones that might come along. Essentially, this use of encapsulating an algorithm in an abstract class (CalcTax) and using one of them at a time interchangeably is the Strategy pattern.

The Strategy Pattern

The intent, according to the Gang of Four

According to the Gang of Four, the Strategy pattern's intent is to

> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.[6]

The motivations of the Strategy pattern

The Strategy pattern is based on a few principles:

- Objects have responsibilities.
- Different, specific implementations of these responsibilities are manifested through the use of polymorphism.
- There is a need to manage several different implementations of what is, conceptually, the same algorithm.
- It is a good design practice to separate behaviors that occur in the problem domain from each other—that is, to decouple them. This allows me to change the class responsible for one behavior without adversely affecting another.

6. ibid, p. 315.

The Strategy Pattern: Key Features

Intent	Allows you to use different business rules or algorithms depending upon the context in which they occur.
Problem	The selection of an algorithm that needs to be applied depends upon the client making the request or the data being acted upon. If you simply have a rule in place that does not change, you do not need a Strategy pattern.
Solution	Separates the selection of algorithm from the implementation of the algorithm. Allows for the selection to be made based upon context.
Participants and Collaborators	• The strategy specifies how the different algorithms are used. • The concreteStrategies implement these different algorithms. • The Context uses the specific ConcreteStrategy with a reference of type Strategy. The strategy and Context interact to implement the chosen algorithm (sometimes the strategy must query the Context). The Context forwards requests from its Client to the Strategy.
Consequences	• The Strategy pattern defines a family of algorithms. • Switches and/or conditionals can be eliminated. • You must invoke all algorithms in the same way (they must all have the same interface). The interaction between the ConcreteStrategies and the Context may require the addition of *getstate* type methods to the Context.
Implementation	Have the class that uses the algorithm (the Context) contain an abstract class (the stragegy) that has an abstract method specifying how to call the algorithm. Each derived class implements the algorithm as needed. *Note:* this method wouldn't be abstract if you wanted to have some default behavior. *Note:* In the prototypical Strategy pattern, the responsibility for selecting the particular implementation to use is done by the Client object and is given to the context of the Strategy pattern.

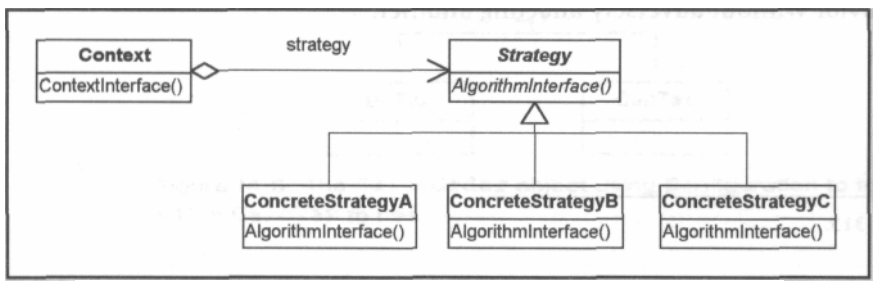

Figure 14-6 Standard, simplified view of the Strategy pattern.

Field Notes: Using the Strategy Pattern

The limits proves the pattern

I had been using the e-tail example in my pattern classes when someone asked, "Are you aware that in England people over a certain age don't get taxed on food?" I wasn't aware of this and the interface for the CalcTax object did not handle this case. I could handle this in at least one of three ways:

1. Pass the age of the Customer into the CalcTax object and use it if needed.
2. Be more general by passing in the Customer object itself and querying it if needed.
3. Be more general still by passing a reference to the SalesOrder object (that is, this) and letting the CalcTax object query it.

While it is true I have to modify the SalesOrder and CalcTax classes to handle this case, it is clear how to do this. I am not likely to introduce a problem because of this.

Encapsulating business rules

Technically, the Strategy pattern is about encapsulating algorithms. However, in practice, I have found that it can be used for encapsulating virtually any kind of rule. In general, when I am doing analysis and I hear about applying different business rules at different times, I consider the possibility of a Strategy pattern handling this variation for me.

Coupling between context and strategies

The Strategy pattern requires that the algorithms (business rules) being encapsulated now lie outside of the class that is using them (the Context). This means that the information needed by the strategies must either be passed in or obtained in some other manner.

Ways of eliminating class explosions with the Strategy pattern

The only serious drawback I have found with the Strategy pattern is the number of additional classes I have to create. While well worth the cost, there are a few things I have done to minimize this when I have control of all of the strategies. In this situation, if I am using

C++, I might have the abstract strategy header file contain all of the header files for the concrete strategies. I also have the abstract strategy cpp file contain the code for the concrete strategies. If I am using Java, I use inner classes in the abstract strategy class to contain all of the concrete strategies. I do not do this if I do not have control over all of the strategies; that is, if other programmers need to implement their own algorithms.

Summary

In this chapter

The Strategy pattern is a way to define a family of algorithms. Conceptually, all of these algorithms do the same things. They just have different implementations.

I showed an example that used a family of tax calculation algorithms. In an international e-tail system, there might be different tax algorithms to use for different countries. Strategy would allow me to encapsulate these rules in one abstract class and have a family of concrete derivations.

By deriving all the different ways of performing the algorithm from an abstract class, the main module (SalesOrder in the example above) does not need to worry about which of many possibilities is actually in use. This allows for new variations but also creates the need to manage these variations—a challenge I will discuss in Chapter 20, 'The Analysis Matrix."