Design Patterns

by Christopher G. Lasater

Chapter 1: Why Pattern?

Overview

Design patterns are basically design tools to improve existing code. Like a carpenter who uses a nail gun instead of a hammer to build a house because he does not hit his thumb and can nail a house together in days instead of weeks, design patterns allow the code you write to be easier to implement, build, and maintain. They are tools to improve efficiency but more importantly allow you as the developer to improve your overall design skills as well as the quality of your projects, and give you a wider scope of skill sets. They allow you to see new answers to common and specialized problems. They define a common programming model, which can translate across to other developers also familiar with patterns. They standardize common programming tasks into recognizable forms, giving your projects better cohesiveness. In general, they help to make you a better designer.

Common Aspects of Object-Oriented Languages

Design pattern models are based and depend highly on the aspects of object-oriented languages. That is to say that patterns-based programming doesn't make much sense outside such languages. Aspects of object-oriented languages like encapsulation, polymorphism, abstraction, and inheritance all extend their properties into patterns-based coding. You might say that patterns methodology is an extension of object-oriented methodology. To better understand where patterns fit into the object-oriented world, we need to examine some of these properties.

Encapsulation is one of the most important aspects of object-oriented languages. The rule of encapsulation is one of keeping things private or masked inside the domain of an object, package, namespace, class, or interface and allowing only expected access to pieces of functionality. We use the rule of encapsulation in almost every aspect of OOP (object-oriented programming). This rule allows us to build patterns like facades, proxies, bridges, and adapters. It allows us to hide with an interface or class structure some functionality that we do not wish to be publicly or globally known. It allows us to define scope inside our programs, and helps us to define and group modules of logic. It provides ways to allow objects to communicate without that communication getting either too complex or too entangled. It provides rules of engagement between different code bases, and helps us decide what functionality can be known and what needs to be hidden.

Think of encapsulation like your mortgage company. You send off your mortgage payment every month and get a statement back showing your loan data. How your payment is applied and handled inside the mortgage company's accounting department is unknown to you, but you can see evidence of it in your statements and the slowly reducing loan debt against your house. The accounting processes behind how this process works is invisible to you. It exists behind the business facade of the mortgage company. The company has processes and rules based on laws and business practices to guarantee that your money is safely deposited and applied to your loan. You know it works; you just don't know or even care *how* it works, as long as it works as expected and returns the expected results.

Working with encapsulation within your programs is very similar to the example we just read. Let's say we want to build a class that applies your mortgage payment to your loan. We obviously want to allow limited access to this class, since it handles money and sensitive information, so allowing every process around it to access every method inside it would not be a good idea. So instead we choose two methods to make public and mark the access of the remaining methods private. One of the public methods is a method to apply the payments, including the money and account number as input parameters and outputting the loan amount left after the payment is applied. The other is a method to return amortized data surrounding the payment plan.

```
class CustomerPayment
{
    public double PostPayment(int loanId, double payment)
    {
      .....performs post of payment to customer account
    }

    public ArrayList GetAmortizedSchedule(int loanId)
    {
      ...returns an amortization schedule in array
    }
}
```

Notice that the two methods in the code above are visible as public. We can assume that the `CustomerPayment` class has many other methods and code to help perform some of the functions of its two public methods, but since we cannot access them outside the class code they are in effect invisible to any other classes in the code domain. This gives us

proper encapsulation of the class methods, allowing only the required methods to be accessed. Thus, the method of encapsulation for this class is to allow only the two methods, `PostPayment()` and `GetAmortizedSchedule()`, to be accessed outside the class.

Polymorphism is another important aspect of object-oriented programming. The rule of polymorphism states that classes can be altered according to their state, in effect making them a different object based on values of attributes, type, or function. We use polymorphism in almost every coding situation, especially patterns. This rule gives us the power to use abstraction and inheritance, allowing inherited members to change according to how they implement their base class. It also allows us to change a class's purpose and function based on its current state. Polymorphism helps interfaces change their implementation class type simply by allowing several different classes to use the same interface. Polymorphic declarations of specific implementations of classes with a common base or interface are extremely common in object-oriented code.

To better understand polymorphism we can think of an example of using an interface and a group of classes that implement the interface to perform different functionality for different implementations.

Note An *interface* is a type of code that is not a class, but acts together with different classes to define a common link, which would not be possible otherwise. It is simply a protocol in object-oriented languages that exists between classes and objects to provide an agreed upon linkage.

Let's take a look at an example that illustrates the role polymorphism plays in a relationship between classes without a common base class and an interface designed to allow some common definitions between these classes.

We start out with some common `if...then...else` code as we might see in any language, either scripting or object. Our mission is to use the aspect of polymorphism to make this code more flexible.

```
if(IsAntiqueAuto)
    AntiqueAuto auto = new AntiqueAuto();
    int cylinders = auto.Cylinders();
    int numDoors = auto.NumberDoors();
    int year = auto.Year();
    string make = auto.Make();
    string model = auto.Model();
}
```

The first thing you need to answer is why do we want to change this code? The answer might be one of portability: You wish to have many car types and pass the logic of creating them via the class itself, rather than via a logical Boolean statement. Or you might wish to make sure all auto classes have the same methods so your code accessing the object always knows what to expect.

In the code example below we can see an interface, `IAuto`, and below it some classes that we have modified to implement this interface. We can assume that each class also implements the interface's methods, each functioning in its own way, returning values according to the logic specific to the implemented methods on each class.

```
public interface IAuto
{
    int Cylinders();
    int NumberDoors();
    int Year();
    string Make();
    string Model();
}
class AntiqueAuto : IAuto....

class SportsCar : IAuto....

class Sedan : IAuto.....

class SUV : IAuto.....
```

To understand how polymorphism acts in the interface-class relationship, let's look at some code where different class types are created via the `IAuto` interface. First, we see how one of the classes that implement the interface methods can use the interface to define each class's methods as independent logic. Let's look at an example of one of the classes we saw in the previous code example. Notice that each of the methods that are present in the interface are represented in the `AntiqueAuto` implementation and return values specific to its type of auto. This is mandated by the compiler, to ensure

all methods in the interface are implemented in the class to determine common functionality between this class and others that implement the `IAuto` interface. This defines one aspect of polymorphism, in that by changing its underlying class type, the interface can allow different functionality from each class implementation.

```
class AntiqueAuto : IAuto
{
    public int Cylinders()
    {
      return 4;
    }
    public int NumberDoors()
    {
      return 3;
    }
    public int Year()
    {
      return 1905;
    }
    public string Make()
    {
      return "Ford";
    }
    public string Model()
    {
      return "Model T";
    }
}
```

Now, when looking at another class that implements the same interface, we see it has a completely different implementation of each of the interface methods:

```
//another implementation of IAuto
class SportsCar : IAuto
{
    public int Cylinders()
    {
      return 8;
    }
    public int NumberDoors()
    {
      return 2;
    }
    public int Year()
    {
      return 2005;
    }
    public string Make()
    {
      return "Porsche";
    }
    public string Model()
    {
      return "Boxter";
    }
}
```

Next, we instantiate the `AntiqueAuto` class as an instance of the `IAuto` interface and call each of the interface methods. Each method returns a value from the methods implemented on the `AntiqueAuto` class.

```
IAuto auto = new AntiqueAuto();
int cylinders = auto.Cylinders();
int numDoors = auto.NumberDoors();
int year = auto.Year();
string make = auto.Make();
string model = auto.Model();
```

If we changed the implemented class to `SportsCar` or another auto type, the methods would return different values. This is how polymorphism comes into play in class relationships. By changing the class type for a common interface or abstraction, we can change the functionality and scope of the code without having to code `if... then...else`

statements to accomplish the same thing.

```
IAuto auto = new SportsCar();
int cylinders = auto.Cylinders();
int numDoors = auto.NumberDoors();
int year = auto.Year();
string make = auto.Make();
string model = auto.Model();
```

Inheritance and abstraction are also very important features of object-oriented languages. They provide a way to make polymorphic representations of objects and object relationships that can be managed at run time or compile time.

Inheritance is the ability of one object to be derived by creating a new class instance from a parent or base class and overloading the constructor(s), methods, and attributes of that parent object and implementing them in the instance. In Java this is known as *subclassing*. Inheritance is important because many times an object contains some base functionality that another object also needs and, instead of maintaining the same logic in two objects, they can share and even override or change this functionality by using a base or parent class. If this occurs, then the base or parent object should be defined in such a way that several common derived objects can use the same common functionality from the parent. The parent should only contain functionality common to *all* its children.

Abstraction is the actual method in which we use inheritance. *Abstraction* is the ability to abstract into a base class some common functionality or design that is common to several *implementation* or instanced classes. The difference — in this book and most code descriptions — between implementation and abstract classes is that abstractions of classes cannot be instanced, while implementations can. Abstraction and inheritance are both aspects of polymorphism, and the reverse is true as well.

> **Note** Actually there is also a pattern that illustrates this basic relationship. See the "Template Pattern" section in Chapter 3.

Another important aspect of object-oriented languages is how they deal with collections of objects. The equals implementation for objects is an important aspect of dealing with objects inside a collection. Languages like C#, VB.NET, and Java all use this method to help index and compare objects in collections. Let's talk about this briefly.

When a hash table or other collection object indexes and compares an object, it uses the `GetHashCode()` method to help in this indexing and comparison. This method can be overridden to capture a more accurate sampling of the intrinsic properties or state of the object. In other words, the `GetHashCode()` method can return an integer representation of the concatenated state of the properties within an object. If not overridden, then this relationship is less exact. This is important when making comparisons between objects in collection classes like iterators or generic collection objects like hash tables. You need to make accurate representations of the internal state of objects so the correct object can be compared or indexed in a collection.

There are general rules to guarantee that each object gets a unique hashing algorithm:

- Objects that compare as equal must return the same hashed value.
- `GetHashCode()` must return the same value every time, unless the internal value or state is modified.
- The hashed value is not like a GUID (global unique identifier) in that it is not globally unique, but only unique if the hashed algorithm and the object's value are not the same as any other object in the scope of the executing code.
- The default implementation of `GetHashCode()` in objects that contain state variables or values is not guaranteed to be unique. That is why if uniqueness is desired, then a proper algorithm needs to be implemented in the overridden method on a particular class.
- To provide a complete representation of state, the values of each variable that represents the object's state need to be part of the hashing algorithm.

To illustrate the proper way to implement the `GetHashCode()` method, take a look at this example:

```
public override int GetHashCode()
{
    return _name.GetHashCode() ^ _address.GetHashCode();
}
```

We see that the method has been overridden, and two instance variables have been concatenated with the ^ symbol and

returned as their sum. Another way to do this is with the + sign, which returns the same result:

```
public override int GetHashCode()
{
    return _name.GetHashCode() + _address.GetHashCode();
}
```

Taking all the variables that may change the state of the object and returning their concatenated hashed values guarantees that each object will have unique values based on state. This allows objects used as keys in collections like hash tables to act in the proper manner.

The `Equals(object obj)` method is the required method for *bitwise* comparisons of value objects. It is especially useful in sorting collections or when a comparison operation is desired in a collection. Not all primitive or object types can have bitwise equality, and so those are compared by value, as in the case of decimal 2.2000 and 2.2, which have the same value but different binary equality.

The proper operational sequence usually starts with a null check, then a class type comparison, and then a comparison of all the value types (or reference types) that influence the state of the class:

```
public override bool Equals(object obj)
{
    if(obj != null && obj is Component)
        return _name.Equals(((Component)obj).Name) &&
        _address.Equals(((Component)obj).Address);
    else
        return false;
}
```

There are some basic rules when testing the equals implementation for proper return values:

- `obj1.Equals(obj1) = true` — an object always equals itself.
- `obj1.Equals(obj2) = obj2.Equals(obj1)` — equals implementations across different class instances always return true on both classes if equal.
- `obj1.Equals(obj2) && obj2.Equals(obj3) && obj3.Equals(obj1) = true` — if object 1 is equal to object 2 and object 2 is equal to object 3, then object 3 must be equal to object 1.
- All calls to `Equals()` return the same value unless the class's state or internal value is modified.
- `Equals(null)` always returns false.

Patterns Cross-Reference Each Other

One interesting thing about design patterns is that they can complement or redefine each other. As you read the examples and sections in this book, you will notice that I include a paragraph in each section that refers to other patterns and provides some comparisons between these patterns. This is because, like aspects of object-oriented programming that work in a cohesive manner, design patterns often use and reference each other to accomplish common goals. You may also find when using design patterns other new patterns spring to life by morphing two of the known patterns. Don't get too excited though! It is a sure bet most of the morphed patterns have already been invented in some form, so yours might not be the first implementation. This is true of code in general. It has been said that there are no new inventions, just new combinations, and the same could be said of code.

Refactoring Legacy Code and Improving New Code

As you read through this book, you will see many instances in the problems sections of each chapter that start out with code mannerisms that may be familiar to you. Boolean logical code seems to be where people who are just learning patterns seem to have a common reference. Almost everyone who writes code starts out with `if...then...else` code.

If you have ever spent hours trying to write a complex set of algorithms and ended up with spaghetti code, you have probably realized the shortcomings of writing solely in Boolean logic. I wrote this book keeping in mind that people who might have an interest in patterns probably would not have a primer to interpret and decide where a pattern might fit into their code. So in each example of the 23 patterns I start out with simple Boolean logic code (or `if...then...else`) if I can, and I then demonstrate a *refactoring* effort toward that particular pattern. I do this both to demonstrate how the pattern fits into real-world code and to give you a lesson on how to perform upgrades to your code through refactoring.

Reflection and OOP

For those not familiar with how reflection works on classes, think of the IntelliSense drop-downs in your IDE and how they display classes, methods, parameters, and attributes. IntelliSense uses reflection on the class types to provide this information. Basically, *reflection* is a way to look at your class types through a run-time API and determine their location, structure, and type. Using reflection in code can be a useful tool. If you want a looser coupling of code bases, reflection can be a key component to providing this functionality. Creating classes without defining their type until run time is useful in many ways. If you did not have reflection methods in your language API, doing this would be hard, if not impossible. Reflection can be used in place of compiled code relationships to make your code more dynamic. There are a few examples in this book that use reflection. I make use of reflection to extend some of the pattern examples where this is appropriate. I do this to provide some coding solutions that are not usually part of patterns texts, but are good examples of implementations of real-world solutions.