Learn the Lingo: Design Patterns

An overview of Design Patterns by JeremyBytes.com

Overview

You probably use design patterns in your software development without even realizing it. It's important to learn the lingo so that we have a shared vocabulary as software developers. This way, when someone says he is using the Observer or the Factory Method pattern, you have a good idea of the concepts in play. Implementations will vary – as we'll see, this is an important facet of design patterns.

We'll take a look at what design patterns are, get an introduction to the Gang of Four, and see common implementations that we already use. Then we'll see five of these patterns in action in common code. From here, you can jump off into your own investigation of design patterns.

What is a Design Pattern?

The most common definition of design patterns comes from Christopher Alexander: "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, in such as way that you can use this solution a million times over, without ever doing it the same way twice."

The interesting thing is that Alexander was referring to architecture – buildings and structures. The Gang of Four takes this same concept and applies it to the world of software.

The Gang of Four

The Gang of Four (also referred to as GoF) are the authors of one of the best-known books on software patterns – *Design Patterns: Elements of Reusable Object Oriented Software* – Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. This book was originally published in 1994, but is still relevant today. (My copy is the 36th printing from July 2008.)

The GoF describes a set of 23 patterns – definitely not an exhaustive collection (nor is that their goal), but it is a good starting point.

A small warning about the GoF: this book is not for the faint of heart. It is quite technical and not particularly friendly toward new developers or those just getting started with design patterns. A good introductory book is *Head First Design Patterns* by Eric Freeman and Elisabeth Freeman. This book is extremely approachable and covers 12 of the GoF patterns. The one caveat is that the examples are in Java. But since Java shares the C syntax, it's close enough to C# that it's easy to follow along.

The Anatomy of a Pattern

Pattern descriptions are broken down into four parts (most pattern descriptions contain additional sections – but these four are considered to be essential).

Pattern Name

The pattern name is the shared vocabulary that we can use. The pattern authors strive to make these names relevant and recognizable.

Problem

This is the problem space that the pattern is set to solve. As noted in Alexander's definition, this is a common problem that occurs over and over in the environment.

Solution

The solution is the description of the elements that make up the design – including relationships, responsibilities and collaborations between those elements. This is just a description, not a concrete implementation. It is up to the developer to determine the implementation that is appropriate to the specific problem.

Consequences

Each pattern has both benefits and costs. Everything has trade-offs, and it is important that we understand the consequences of our decisions. A design pattern is simply a tool in the toolbox. No tool is always the right tool, nor should we always try to use the same pattern in every situation.

We won't be going into the details of each of these sections here. We'll be taking a more informal approach. But you should be aware of these parts for when you go on to look through pattern catalogs.

The GoF Patterns

As mentioned earlier, the GoF describes a set of 23 patterns. These patterns are listed below:

Abstract Factory	Chain of Responsibility
Builder	Command
Factory Method	Interpreter
Prototype	Iterator
Singleton	Mediator
Adapter	Memento
Bridge	Observer
Composite	State
Decorator	Strategy
Façade	Template Method
Flyweight	Visitor
Proxy	

Why Should We Care?

So, now that we've see what patterns are, the next question is why should we care? There are a number of reasons to learn the standard design patterns and to also extend beyond those to other pattern sets.

Well-Described Solutions

First, design patterns are well-described solutions. Someone else has already thought through the problem space and has come up with a solution. There are few times that we want to reinvent the wheel. It's almost always best to start with someone else's working solution. We can always make tweaks to that solution for our specific application.

Shared Vocabulary

Next, design patterns give us a shared vocabulary in our field. As developers, a common language helps us to communicate concepts with each other. And when we say that we are using the Observer pattern, other developers will know exactly what we mean.

Concise Language

Along the same lines as shared vocabulary, design patterns give us a very concise language that we can use. Instead of saying, "I have a class that publishes an event and then notifies a set of subscribers when that event fires," we can simply say, "I'm using an Observer." Generally, when communicating at the design level, the concept is more important than the specific implementation.

Stay in Design Longer

Because design patterns are abstract and do not deal with specific implementations, they allow us to stay in design mode longer. Rather than thinking about specific lines of code or classes that we are going to implement, we can think about the patterns and how they fit together. Because of this, we can fine-tune our design before writing a single line of code.

Encourage Other Developers

Finally, if we regularly use design patterns in our communication with other developers, it will encourage those who are not familiar with design patterns to investigate them and learn them. This will expand the use of the shared vocabulary to the next generation, and we can continue to share those advantages we've seen above with other developers.

Patterns You Already Use

As an introduction to design patterns, we'll take a look at several patterns that you probably already use: Observer, Proxy, Chain of Responsibility, Adapter, and Iterator. For each of these, we'll look at the GoF description, a real-world example of the pattern, and a common C# implementation in a sample program. These five patterns are just a sampling; you probably are using many of the other GoF patterns as well.

Sample Code

We'll be using 5 small sample applications in a single solution. You can download the source code here: http://www.jeremybytes.com/Downloads.aspx. Each project is named for the pattern it demonstrates.

Observer

GoF Description

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."

Real World Example

The Observer is best thought of as a publish/subscribe model. Twitter is a good example of this. When you decide to follow someone on Twitter, you become an Observer of that other user (the Subject). Then when that user tweets, all of that user's followers will be notified – including you.

You're Already Using This

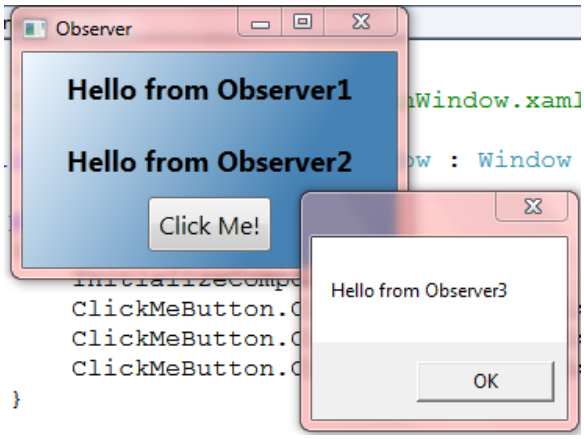
If you have used an event handler in .NET, then you have used the Observer pattern. Let's take a look at some sample code.

The "Observer" project contains a simple form with 2 text blocks and a button. Let's hook up some event handlers to the button's `Click` event. Here's the code-behind from MainWindow.xaml.cs:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        ClickMeButton.Click += Observer1;
        ClickMeButton.Click += Observer2;
        ClickMeButton.Click += Observer3;
    }

    void Observer1(object sender, RoutedEventArgs e)
    {
        TextBlock1.Text = "Hello from Observer1";
    }

    void Observer2(object sender, RoutedEventArgs e)
    {
        TextBlock2.Text = "Hello from Observer2";
    }

    void Observer3(object sender, RoutedEventArgs e)
    {
        MessageBox.Show("Hello from Observer3");
    }
}
```

In this code we are setting up 3 Observers. Each one is saying to the button, "Please tell me when you get clicked" – subscribing to that event. This is recorded by the Subject (the button) and is registered as a dependency. And when the "object changes state" (from the GoF description above), each event handler is called.

Here's our output:



Whenever the button is clicked, all three methods fire. This populates the 2 text blocks and a pop-up message.

With Event Handlers, the Subject gets passed as the `sender` parameter. This allows the Observer to inspect the state of the subject. As an example, if we had a check box instead of a button, when the Click event fired, we could inspect the `sender` to see if the check box was checked or not. In addition, the same method can be hooked up to multiple subjects (e.g. a single handler that is hooked up to multiple buttons). The handler can then use the subject (`sender`) to determine which object fired the event.

As with all design patterns, there are considerations when deciding whether to implement the Observer and how to implement it. A major consideration is that the Observer object has no idea when it will be notified or how often it will be notified. In the case of our button's `Click` event, the button may be clicked repeatedly in quick succession, or not at all. If we were doing more significant processing in our Observer methods, we would need to allow for those situations.

If you want to look into the Observer pattern a little further, the .NET 4.0 framework provides 2 interfaces to do just that: IObservable<T> and IObserver<T>. Check the samples in the Visual Studio help for more information on using these interfaces.

Proxy

GoF Description

"Provide a surrogate or placeholder for another object to control access to it."

Real World Example

Someone with power of attorney (such as an agent) is a proxy. When you interact with the agent, you interact as if you were communicating directly with the person or entity (the subject) that the agent represents. If you create a contract, there is no difference whether the contract is signed by the agent or the original subject. From your perspective, you are contracting directly with the subject and the act of working with the agent is transparent (at least from a legal standpoint). The agent acts as a proxy for that subject.

You're Already Using This

If you have used a Web Service or WCF service in .NET, then you have used the Proxy pattern. Let's take a look at some sample code.

The "Proxy" project contains a simple form with a list box and a button. In addition, we have added a service reference to the `PersonService` in the "WCFService" project. The `IPersonService` interface and `Person` class are defined in the "WCFService" as follows:

```
[ServiceContract]
public interface IPersonService
{
    [OperationContract]
    List<Person> GetPeople();
}

public class Person
{
    public string FirstName;
    public string LastName;
    public DateTime StartDate;
    public int Rating;
}
```

And the `PersonService` is implemented here:

```
public class PersonService : IPersonService
{
    public List<Person> GetPeople()
    {
        var p = new List<Person>()
        {
            new Person() { FirstName="John", LastName="Koenig",
                StartDate = DateTime.Parse("10/17/1975"), Rating=6 },
            // Additional entries removed to save space
        };
        return p;
    }
}
```

So we can see that this is the class that is actually going to be doing our work; this is our subject. But there's a problem, it's located across a network and exposed as a SOAP service. If we were to call this from our code manually, we would need to construct a network channel, generate an XML document

formatted for the service, wait for a response, and then parse the resulting XML into an object that we can use in our code.

But instead of doing all of that, we use a Proxy. In this case, when we set up the service reference (by choosing to "Add Service Reference" to our project and then running through the wizard), Visual Studio created a Proxy class that we can use.

Back in our "Proxy" project, here's how we use the proxy in the MainWindow.xaml.cs file.

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    private void ClickMeButton_Click(object sender, RoutedEventArgs e)
    {
        var proxy = new PersonServiceClient();
        PersonListBox.ItemsSource = proxy.GetPeople();
    }
}
```

We create a new `PersonServiceClient` and assign it to a variable called `proxy`. At this point, we can treat `proxy` as if we were dealing with the `PersonService` class directly. As you can see in the second line of code, we call the `GetPeople()` method of the proxy object just as we would if we it were a `PersonService` object..

The Proxy object allows us to interact with the class as if it were part of our project. But in reality, it is generating an SOAP packet, making a call across the network, letting that call do the actual work, and then turning the result back into an object in our local code. This is much easier than having to code all of this manually.

Let's look a little further into the code that Visual Studio generated for us. Right-click on "`PersonServiceClient`" and choose "Go to definition". This will take you to a "References.cs" file. Here's part of the class:

```
public partial class PersonServiceClient :
    System.ServiceModel.ClientBase<Proxy.PersonService.IPersonService>,
    Proxy.PersonService.IPersonService {

    public PersonServiceClient() {
    }
    ...

    public Proxy.PersonService.Person[] GetPeople() {
        return base.Channel.GetPeople();
    }
}
```

You can see that the `PersonServiceClient` references the `Proxy` object created by Visual Studio. The good part of this is that you do not normally see this code (nor do you need to).

The proxy provides an interface identical to that of the subject. This can be substituted transparently for the real object in our code. In the case of our WCF Service, Visual Studio uses the services WSDL to generate the proxy object with the correct interface.

So, just like our example with the power of attorney above, we can interact with the proxy in our local project as if it were the actual entity that is performing the work. It is completely transparent to us. The underlying infrastructure takes care of the rest.

Chain of Responsibility

GoF Description

"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it."

Real World Example

Let's look at everyone's favorite pastime: calling tech support. If you have a problem with your internet connection, you call the tech support line and get Level 1 support. This is generally someone who simply reads off a list of things to try ("Did you try turning it off and on again?"). If Level 1 support can provide a fix, then the issue is closed. (And even though it's frustrating to go through this step, I have to imagine that it must work for the majority of callers.) If Level 1 support cannot resolve the issue, then it is escalated to Level 2 (usually someone with some technical knowledge). If Level 2 cannot resolve the issue, then it is escalated to Level 3 (such as engineering staff).

Each of these Levels is part of a Chain of Responsibility. If the support person can handle the issue, then he/she does and the issue is closed. Otherwise, the issue is passed to the next person in the chain.

You're Already Using This

If you have done exception handling in .NET, then you have used the Chain of Responsibility pattern. Let's take a look at some sample code.

The "ChainOfResponsibility" project contains a text block, a combo box, and a button. The button's `Click` event calls a method, and that method calls a second method. This final method throws an exception. Here's the code:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
```

```
        private void ClickMeButton_Click(object sender, RoutedEventArgs e)
        {
            Method1();
        }

        private void Method1()
        {
            Method2();
        }

        private void Method2()
        {
            switch (ExceptionBox.Text)
            {
                case "AccessViolationException":
                    throw new AccessViolationException();
                case "NullReferenceException":
                    throw new NullReferenceException();
                case "ArgumentException":
                    throw new ArgumentException();
                case "Exception":
                    throw new Exception();
            }
        }
    }
```

If we run the application right now, we get the default exception behavior: it simply reports the error and then shuts down. We'll add some `try` / `catch` blocks to handle the exceptions. First, in `Method2()`:

```
        private void Method2()
        {
            try
            {
                switch (ExceptionBox.Text)
                {
                    case "AccessViolationException":
                        throw new AccessViolationException();
                    case "NullReferenceException":
                        throw new NullReferenceException();
                    case "ArgumentException":
                        throw new ArgumentException();
                    case "Exception":
                        throw new Exception();
                }
            }
            catch (AccessViolationException ex)
            {
                TextBlock1.Text =
                    "Caught AccessViolationException in Method 2";
            }
        }
```

This `catch` block is the first link in the Chain of Resonsibility. If the `catch` block can handle the exception (i.e. if it is an `AccessViolationException`), then the `TextBlock` gets updated and the

application continues. If it cannot handle the exception, it gets pass to the next link of the chain. Let's fill in those other links:

```
private void ClickMeButton_Click(object sender, RoutedEventArgs e)
{
    try
    {
        Method1();
    }
    catch (ArgumentException ex)
    {
        TextBlock1.Text =
            "Caught ArgumentException in Click Event";
    }
}

private void Method1()
{
    try
    {
        Method2();
    }
    catch (NullReferenceException ex)
    {
        TextBlock1.Text =
            "Caught NullReferenceException in Method 1";
    }
}
```

With in place, the Chain of Responsibility works as follows: First, the catch block in `Method2()` tries to handle the exception, if it cannot, then it gets passed to the catch block in `Method1()`. If `Method1()` cannot handle the exception, it gets passed to the button's `Click` event handler.

One of the consequences of the Chain of Responsibility is that it's possible for the request to drop off the end of the chain. In our sample, if you choose "`Exception`", it does not get handled by `Method2()`, so it gets passed to `Method1()`. It does not get handled by `Method1()`, so it gets passed to the button's `Click` handler. It does not get handled by the button's `Click` handler, so it drops off the end of the chain, and the application reports the exception and then shuts down.

(As a side note, this demonstrates why it's a best practice to include a default exception handler at the application level. But that's a discussion for another time.)

Adapter

GoF Description

"Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."

Real World Example

Adapters convert something you can't use into something you can use. In the computer world, we deal with a lot of physical adapters. We have SD card adapters that let us use a miniSD card in a standard SD card slot. On desktop computers, we often have adapters that change the keyboard connection from USB to PS/2 (or vice-versa). We also have a number of USB connection types (whether full-size, mini-USB or micro-USB) along with adapters to physically connect the different types. These are all adapters that let us connect 2 otherwise-incompatible interfaces.

You're Already Using This

If you have used data binding to display non-string fields to users in a .NET application, then you have used the Adapter pattern. Let's take a look at some sample code.

The "Adapter" project contains a set of text boxes for input and a set of text blocks for output. (We've pre-populated the text boxes just so that we don't have to type in the values when we run our application.) The code-behind (MainWindow.xaml.cs) consists of the following:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void ClickMeButton_Click(object sender, RoutedEventArgs e)
    {

    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime StartDate { get; set; }
    public int Rating { get; set; }
}
```

Here we have an empty `Click` event handler for the button as well as a description of the Person class. Now, let's add some code to the event handler. This will take the values from the text boxes, populate the fields in the `Person` object, then put the `Person` object fields into the output text blocks on the screen.

```
    private void ClickMeButton_Click(object sender, RoutedEventArgs e)
    {
        Person newPerson = new Person();
        newPerson.FirstName = FirstNameInput.Text;
        newPerson.LastName = LastNameInput.Text;
        newPerson.StartDate = DateTime.Parse(StartDateInput.Text);
        newPerson.Rating = Int32.Parse(RatingInput.Text);
```

```
            FirstNameOutput.Text = newPerson.FirstName;
            LastNameOutput.Text = newPerson.LastName;
            StartDateOutput.Text =
                newPerson.StartDate.ToString("MM/dd/yyyy");
            RatingOutput.Text = newPerson.Rating.ToString();
        }
```

So, why is an Adapter needed? Let's start here:

```
            newPerson.StartDate = DateTime.Parse(StartDateInput.Text);
```

The problem is that our input is coming to us as a `string`, but we're trying to assign this to a `DateTime` property – our "incompatible interfaces." The `DateTime.Parse()` method does a conversion in this case – so that our `string` and `DateTime` can work together. The same thing is happening with the `Rating` property (with `int` and a `string`).

We are doing the opposite when we populate the output text blocks. The `ToString()` methods of the `StartDate` and `Rating` properties allow us to populate a `string` value in the text blocks.

This normally happens for us behind the scenes with data binding. The data binding engine works as the Adapter that resolves the incompatibility between the strings (the user-displayable values) and the underlying data (in the `Person` object). If we need more control over this process, we can use value converters in the XAML world (if you want more information on value converters, you can check out "Introduction to Data Templates and Value Converters in Silverlight" at http://www.jeremybytes.com/Demos.aspx.)

As long as we're talking about data binding, let's go back to the Observer pattern. Data binding sets up notification so that when our underlying object changes, it notifies the UI elements that are data-bound to it so they can update with the latest values. The reverse is also true. Often you will see multiple patterns at work in complex infrastructures. "Thinking in patterns" allows us to combine simple ideas into complex structures.

Iterator

GoF Description

"Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation."

Real World Example

An Iterator allows you to ask for the next item in a sequence. Your television remote control does this. The Channel Up button goes to the next channel. Internally, you don't care how the channels are stored or whether the television decides to skip channels with no signal. You can just keep clicking "next" until you get to the end of the list. Note: all examples break down at some point. Most iterators stop when they get to the end of the sequence; the television will start over at the beginning.

You're Already Using This

If you have used a `foreach` loop in .NET, then you have used the Iterator pattern. Let's take a look at some sample code.

In the "Iterator" project, we have a simple application with a `ListBox` and a `Button`. In addition, there is a separate `People` class (this is the same as the class we used in the WCFService above). We'll add some code to get a list of `Person` objects and then load them into the `ListBox`:

```
private void ClickMeButton_Click(object sender, RoutedEventArgs e)
{
    List<Person> people = People.GetPeople();

    var enumerator = people.GetEnumerator();
    while (enumerator.MoveNext())
    {
        PersonListBox.Items.Add(enumerator.Current);
    }
}
```

Our Iterator is available in the `List<Person>` variable (`people`). More specifically, `List<T>` implements `IEnumerable<T>`, which describes an Iterator implementation. Our sample codes gets the enumerator for our people object, and then calls `MoveNext()` to iterate through the collection. (`MoveNext()` will return false if it is already on the last item in the collection). The enumerator also has a `Current` property which points to the current item. With this, we can iterate through our list and add each item to the list box UI control.

In the real world, you're probably not doing it this way. This will look more familiar:

```
private void ClickMeButton_Click(object sender, RoutedEventArgs e)
{
    List<Person> people = People.GetPeople();
    foreach (var person in people)
    {
        PersonListBox.Items.Add(person);
    }
}
```

The `foreach` construct in C# will work with any class that implements `IEnumerable` or `IEnumerable<T>`. Behind the scenes, the code will get the enumerator (Iterator) and use it to traverse the collection.

As a side note, the GoF description has the interator as an object separate from the actual collection. This varies from the .NET implementation of having the iterator functions within the `List<T>` itself. However, they note that ReadStream (from SmallTalk) is an object that contains the iterator methods; and there's nothing wrong with that implementation. The .NET StreamReader classes are much the same. As with all implementations, there are pros and cons to having the objects combined or separated, and you need to take that into consideration if implementing your own Iterator.

A Million Implementations

These are examples of implementations of just a few of the GoF patterns. As a reminder, patterns and implementations are completely independent; from our definition: "…describes the core of the solution to that problem, in such as way that you can use this solution a million times over, without ever doing it the same way twice." Design Patterns are simply descriptions of solutions, not descriptions of implementations (although most pattern books have sample implementations as well).

One question you might ask is whether these patterns are still relevant. After all, the examples that we've looked at show how these design patterns are implemented in .NET – in the Event Handler subsystem, the Proxy and the Service Reference wizard, the Exception Handling subsystem, ToString() and Parse() methods, and the IEnumerable<T> interface. But we can better understand how to use these constructs in .NET if we understand the patterns. This allows us to recognize the problem space and pull out an appropriate design pattern from our toolbox.

The GoF patterns were published in 1994, back when people were writing their own libraries of utilities and helper objects (in fact, the GoF book has a few sample objects in Appendix C "Foundation Classes"). 15 years later the basic problems haven't changed; but we are able to benefit from the .NET designers' decisions to give us some easy-to-use implementations of several of these patterns. And always realize that we can create our own implementations if the .NET-provided ones don't fit our particular problem.

Learn the Lingo

Design Patterns are a common language that we can use as developers. Since we've taken a look at some specific patterns above, you should be comfortable when another developer says that he/she is using an Observer or a Proxy. In addition, you can be clearer in communicating with others when you say you are using an Iterator or a Chain of Responsibility. With a little more study, you can include Factory, Singleton, Decorator, Facade, and Command to your vocabulary as well. And as you use these pattern names in describing your code, you may inspire developers who don't know patterns to start looking into them.

Wrap Up

Design Patterns are well described and well tested solutions in the programming world. But remember that these are just tools in the toolbox. No tool is appropriate for every situation, so study the benefits and costs when considering whether a specific pattern is appropriate for your problem.

The GoF patterns are a good starting point, there are innumerable other pattern catalogs. (A few pattern books are listed on my website: http://www.jeremybytes.com/Bookshelf.aspx#DP.) Some of the more common patterns that are popular today include MVVM (Model-View-ViewModel), IoC (Inversion of Control), DI (Dependency Injection), and MVC (Model-View-Controller). But don't use a pattern just because it is popular; understand it first. Then go out and create some brilliant solutions.

Happy coding!