How I explained Design Patterns to my wife: Part 1

Introduction

Me and my wife had some interesting conversations on Object Oriented Design principles. After publishing the conversation on CodeProject, I got some good responses from the community and that really inspired me. So, I am happy to share our next conversation that took place on Object Oriented Design Patterns. Here it is.

What is a Design Pattern?

Shubho: I guess you already have some basic idea about Object Oriented Design principles. We had a nice talk on the OOD principles (SOLID principles), and I hope you didn't mind that I published our conversation in a CodeProject article. You can find it here: How I explained OOD to my wife.

Design Patterns are nothing but applications of those principles in some specific and common situations, and standardizing some of those. Let's try to understand what Design Patterns are by using some examples.

Farhana: Sure, I love examples.

Shubho: Let's talk about our car. It's an object, though a complex one, which consists of thousands of other objects such as the engine, wheels, steering, seats, body, and thousands of different parts and machinery.

Different parts of a car.

While building this car, the manufacturer gathered and assembled all the different smaller parts that are subsystems of the car. These different smaller parts are also some complex objects, and some other manufacturers had to build and assemble those too. But, while building the car, the car company doesn't really bother too much about how those objects were built and assembled (well, as long as they are sure about the quality of these smaller

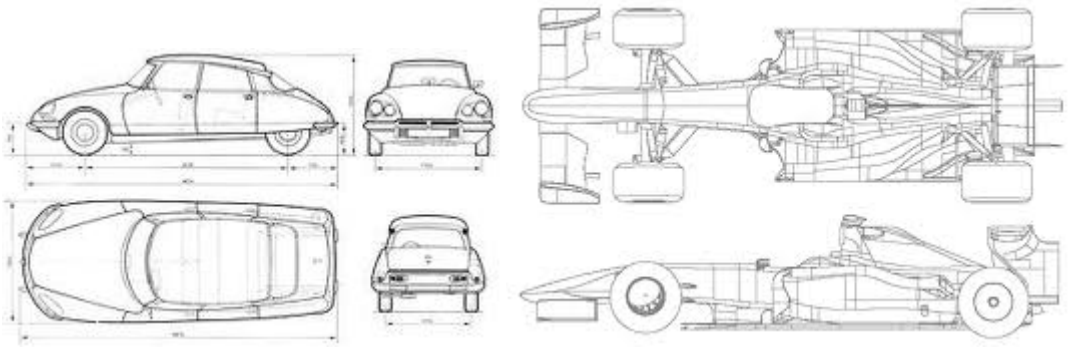
objects/equipments). Rather, the car manufacturer cares about how to assemble those different objects into different combinations and produce different models of cars.

Different models of cars, produced by assembling different parts and following different designs.

Farhana: The car manufacturer company must have some designs or blue prints for each different model of car which they follow, right?

Shubho: Definitely, and, these designs are well-thought designs, and they've put a good amount of time and effort to sketch those designs. Once the designs are finalized, producing a car is just a matter of following the designs.

Farhana: Hm.. it's good to have some good designs upfront and following those allows to produce different products in a quick amount of time, and each time the manufacturer has to build a product for a specific model, they don't have to develop a design from scratch or re-invent the wheel, they just follow the designs.



Different design plans for producing different models of products (cars).

Shubho: You got the point. Now, we are software manufacturers and we build different kinds of software programs with different components or functionality based upon the requirements. While building such different software systems, we often have to develop code for some situations that are common in many different software systems, right?

Farhana: Yes. And often, we face common design problems while developing different software applications.

Shubho: We try to develop our software applications in an Object Oriented manner and try to apply OOD principles for achieving code that is manageable, reusable, and expandable. Wouldn't it be nice whenever we see such design problems, we have a pool of some carefully made and well tested designs of objects for solving those?

Farhana: Yes, that would save us time and would also allow us to build better software systems and manage them later.

Shubho: Perfect. The good news is, you don't have to really develop that pool of object designs from scratch. People already have gone through similar design problems for years, and they already have identified some good design solutions which have been standardized already. We call these Design Patterns.

We must thank the Gang of Four (GoF) for identifying the 23 basic Design Patterns in their book **Design Patterns: Elements of Reusable Object-Oriented Software**. In case you are wondering who formed this famous gang, they are Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. There are many Object Oriented Design Patterns, but these 23 patterns are generally considered the foundation for all other Design Patterns.

Farhana: Can I create a new pattern? Is that possible?

Shubho: Yes darling, why not? Design Patterns are not something invented or newly created by scientists. They are just discovered. That means, for each kind of common problem scenario, there must be some good design solutions there. If we are able to identify an object oriented design that could solve a new design related problem, that would be a new Design Pattern defined by us. Who knows? If we discover some a Design Pattern, someday people may call us Gang of Two.. Ha ha.

Fahana: :)

How will we learn Design Patterns?

Shubho: As I have always believed, examples are the greatest way of learning. In our learning approach, we won't discuss the theories first and implement later. I think this is a BAD approach. Design Patterns were not invented based on theories. Rather, the problem situations occurred first and based upon the requirement and context, some design solutions were evolved, and later some of them were standardized as patterns. So, for each design pattern we discuss, we will try to understand and analyze some real life example problems, and then we will try to formulate a design in a step by step process and end up with a design that will match with some patterns; Design Patterns were discovered in this same process. What do you think?

Farhana: I think this approach makes more sense to me. If I can end up with Design Patterns by analyzing problems and formulating solutions, I won't have to memorize design diagrams and definitions. Please proceed using your approach.

A basic design problem and its solution

Shubho: Let's consider the following scenario:

Our room has some electric equipments (lights, fans etc). The equipments are arranged in a way where they could be controlled by switches. At any time, you can replace or

troubleshoot an electrical equipment without touching the other things. For example, you can replace a light with another without replacing or changing the switch. Also, you can replace a switch or troubleshoot it without touching or changing the corresponding light or fan; you can even connect the light with the fan's switch and connect the fan with the light's switch, without touching the switches.

Electrical equipments: A fan and a light.

Two different switches for fan and light, one is normal and the other is fancy.

Farhana: Yes, but that's natural, right?

Shubho: Yes, that's very natural, and that's how the arrangement should be. When different things are connected together, they should be connected in a way where change or replacement of one system doesn't affect another, or even if there is any effect, it stays minimal. This allows you to manage your system easily and at low cost. Just imagine if changing the light in your room requires you to change the switch also. Would you care to purchase and set up such a system in your house?

Farhana: Definitely no.

Shubho: Now, let's think how the lights or fans are connected with the switches so that changing one doesn't have any impact on the other. What do you think?

Farhana: The wire, of course!

Shubho: Perfect. It's the wire and the electrical arrangement that connect the lights/fans with the switches. We can generalize it as a bridge between the different systems that can

get connected through it. The basic idea is, things shouldn't be directly connected with one another. Rather, they should be connected though some bridges or interfaces. That's what we call "loose coupling" in software world.

Farhana: I see. I got the idea.

Shubho: Now, let's try to understand some key issues in the light/fan and switch analogy, and try to understand how they are designed and connected.

Farhana: OK, let me try.

We have switches in our example. There may be some specific kinds of switches like normal switches, fancy ones, but, in general, they are switches. And, each switch can be turned on and off.

So, we will have a base `Switch` class as follows:

Collapse | Copy Code

```
public class Switch
{
  public void On()
  {
    //Switch has an on button
  }
  public void Off()
  {
    //Switch has an off button
  }
}
```

And, as we may have some specific kinds of switches, for example a fancy switch, a normal switch etc., we will also have `FancySwitch` and `NormalSwitch` classes extending the `Switch` class:

Collapse | Copy Code

```
public class NormalSwitch : Switch
{
}

public class FancySwitch : Switch
{
}
```

These two specific switch classes may have their own specific features and behaviours, but for now, let's keep them simple.

Shubho: Cool. Now, what about fan and light?

Farhana: Let me try. I learned from the Open Closed principles from Object Oriented Design principles that we should try to do abstractions whenever possible, right?

Shubho: Right.

Farhana: Unlike switches, fan and light are two different things. For switches, we were able to use a base `Switch`class, but as fan and light are two different things, instead of defining a base class, an interface might be more appropriate. In general, they are all electrical equipments. So, we can define an interface, say,`IElectricalEquipment`, for abstracting fans and lights, right?

Shubho: Right.

Farhana: OK, each electrical equipment has some common functionality. They could all be turned on or off. So the interface may be as follows:

Collapse | Copy Code

```
public interface IElectricalEquipment
{
    void PowerOn(); //Each electrical equipment can be turned on
    void PowerOff(); //Each electrical equipment can be turned off
}
```

Shubho: Great. You are getting good at abstracting things. Now, we need a bridge. In real world, the wires are the bridges. But, in our object design, a switch knows how to turn on or off an electrical equipment, and the electrical equipment somehow needs to be connected with the switches, As we don't have any wire here, the only way to let the electrical equipment be connected with the switch is encapsulation.

Farhana: Yes, but switches don't know the fans or lights directly. A switch actually knows about an electrical equipment `IElectricalEquipment` that it can turn on or off. So, that means, an `ISwitch` should have an`IElectricalEquipment` instance, right?

Shubho: Right. Here, the encapsulated instance, which is an abstraction of fan or light (`IElectricalEquipment`) is the bridge. So, let's modify the `Switch` class to encapsulate an electrical equipment:

Collapse | Copy Code

```
public class Switch
{
  public IElectricalEquipment equipment
  {
    get;
    set;
  }
  public void On()
  {
    //Switch has an on button
  }
```

```
  public void Off()
  {
    //Switch has an off button
  }
}
```

Farhana: Understood. Let me try to define the actual electrical equipments, the fan and the light. As I see, these are electrical equipments in general, so these would simply implement the `IElectricalEquipment` interface.

Following is the `Fan` class:

Collapse | Copy Code

```
public class Fan : IElectricalEquipment
{
  public void PowerOn()
  {
    Console.WriteLine("Fan is on");
  }
  public void PowerOff()
  {
    Console.WriteLine("Fan is off");
  }
}
```

And, the `Fan` class would be as follows:

Collapse | Copy Code

```
public class Light : IElectricalEquipment
{
  public void PowerOn()
  {
    Console.WriteLine("Light is on");
  }
  public void PowerOff()
  {
    Console.WriteLine("Light is off");
  }
}
```

Shubho: Great. Now, let's make switches work. The switches should have the ability inside them to turn on and turn off the electrical equipment (it is connected to) when the switch is turned on and off.

These are the key issues:

- When the *On* button is pressed on the switch, the electrical equipment connected to it should be turned on.
- When the *Off* button is pressed on the switch, the electrical equipment connected to it should be turned off.

Basically, following is what we want to achieve:

Collapse | Copy Code

```
static void Main(string[] args)
{
  //We have some electrical equipments, say Fan, Light etc.
  //So, lets create them first.

  IElectricalEquipment fan = new Fan();
  IElectricalEquipment light = new Light();

  //We also have some switches. Lets create them too.

  Switch fancySwitch = new FancySwitch();
  Switch normalSwitch = new NormalSwitch();

  //Lets connect the Fan to the fancy switch

  fancySwitch.equipment = fan;

  //As the switch now has an equipment (Fan),
  //so switching on or off should
  //turn on or off the electrical equipment

  fancySwitch.On(); //It should turn on the Fan.

  //so, inside the On() method of Switch,
  //we must turn on the electrical equipment.

  //It should turn off the Fan. So, inside the On() method of
  fancySwitch.Off();
  //Switch, we must turn off the electrical equipment

  //Now, lets plug the light to the fancy switch

  fancySwitch.equipment = light;
  fancySwitch.On(); //It should turn on the Light now
  fancySwitch.Off(); //It should be turn off the Light now
}
```

Farhana: I got it. So, the `On()` method of the actual switches should internally call the `TurnOn()` method of the electrical equipment, and the `Off()` should call the `TurnOff()` method on the equipment. So, the `Switch` class should be as follows:

Collapse | Copy Code

```
public class Switch
{
  public void On()
  {
    Console.WriteLine("Switch on the equipment");
    equipment.PowerOn();
  }
  public void Off()
  {
    Console.WriteLine("Switch off the equipment");
    equipment.PowerOff();
```

```
  }
}
```

Shubho: Great work. Now, this certainly allows you to plug a fan from one switch to another. But you see, the opposite should also work. That means, you can change the switch of a fan or light without touching the fan or light. For example, you can easily change the switch of the light from `FancySwitch` to `NormalSwitch` as follows:

Collapse | Copy Code

```
normalSwitch .equipment = light;
normalSwitch.On(); //It should turn on the Light now
normalSwitch.Off(); //It should be turn off the Light now
```

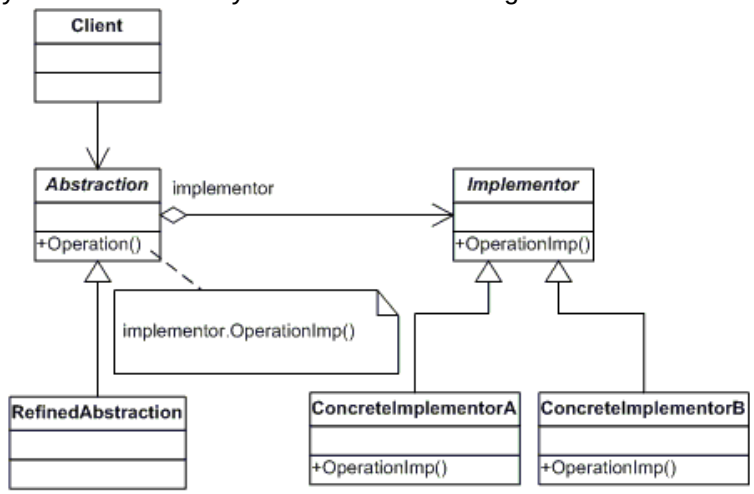
So, you see, you can vary both the switches and the electrical equipments without any effect on the other, and connecting an abstraction of the electrical equipment with a switch (via encapsulation) is letting you do that. This design looks elegant and good. The Gang of Four has named this a pattern: The Bridge Pattern.

Farhana: Cool. I think I've understood the idea. Basically, two systems shouldn't be connected or dependent on another directly. Rather, they should be connected or dependent via abstraction (as the Dependency Inversion principle and the Open-Closed principle say) so that they are loosely coupled, and thus we are able to change our implementation when required without much effect on the other part of the system.

Shubho: You got it perfect darling. Let's see how the Bridge Pattern is defined:

"Decouple an abstraction from its implementation so that the two can vary independently"

You will see that our design perfectly matches the definition. If you have a class designer (in Visual Studio, you can do that, and other modern IDEs should also support this feature), you will see that you have a class diagram similar to the following:



Class diagram of Bridge pattern.

Here, **Abstraction** is the base `Switch` class. **RefinedAbstraction** is the specific switch classes (`FancySwitch`,`NormalSwitch` etc.). **Implementor** is the `IElectricalEquipment` interface. **ConcreteImplementorA** and**ConcreteImplementorB** are the `Fan` and `Light` classes.

Farhana: Let me ask you a question, just curious. There are many other patterns as you said, why did you start with the Bridge pattern? Any important reason?

Shubho: A very good question. Yes, I started with the Bridge pattern and not any other pattern (unlike many others) because of a reason. I believe the Bridge pattern is the base of all Object Oriented Design Patterns. You see:

- It teaches how to think abstract, which is the key concept of all Object Oriented Design Patterns.
- It implements the basic OOD principles.
- It is easy to understand.
- If this pattern is understood correctly, learning other Design Patterns becomes easy.

Farhana: Do you think I have understood it correctly?

Shubho: I think you have understood it perfectly darling.

Farhana: So, what's next?

Shubho: By understanding the Bridge pattern, we have just started to understand the concepts of Design Patterns. In our next conversation, we would learn other Design Patterns, and I hope you won't get bored learning them.

Farhana: I won't. Believe me.

Ref: http://www.codeproject.com/Articles/98598/How-I-explained-Design-Patterns-to-my-wife-Part-1