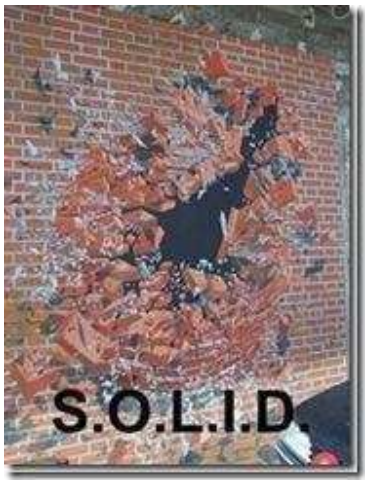
The open closed principle by Gabriel Schenker



In the previous two posts I discussed the **S** of S.O.L.I.D. which is the Single Responsibility Principle (SRP) and the D of S.O.L.I.D. which corresponds to the Dependency Inversion Principle (DI). This time I want to continue this series with the second letter of S.O.L.I.D., namely the O which represents the **Open Close Principle** (OCP).

Introduction

In object-oriented programming the open/closed principle states:

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

That is, such an entity can allow its behavior to be altered without altering its source code.

The first person who mentioned this principle was Bertrand Meyer. He used inheritance to solve the apparent dilemma of the principle. Once completed, the implementation of a class should only be modified to correct errors, but new or changed features would require a different class to be created.

In contrast to Meyer's definition there is a second way to adhere to the principle. It's called the polymorphic open/close principle and refers to the use of abstract interfaces. This implementation can be changed and multiple implementations can be created and polymorphically substituted for each other. A class is open for extension when it does not depend directly on concrete implementations. Instead it depends on abstract base classes or interfaces and remains agnostic about how the dependencies are implemented at runtime.

OCP is about arranging encapsulation in such a way that it's effective, yet open enough to be extensible. This is a compromise, i.e. "expose only the moving parts that need to change, hide everything else"

There are several ways to extend a class:

1. inheritance
2. composition
3. proxy implementation (special case for composition)

Sealed classes

Is a sealed class contradicting the OCP? What one needs to consider when facing a sealed class is whether one can extend its behavior in any other way than inheritance? If one injects all dependencies, which are extendable, they essentially become interfaces, allowing a sealed class to be open for extension. One can plug in new behavior by swapping out collaborators/dependencies.

What about inheritance?

The ability to subclass provides an additional way to accomplish that extension by not putting the abstraction in codified interfaces but in overridable behavior.

Basically you can think of it a bit like this, given a class that has virtual methods, if you put all of those into an interface and depended upon that, you'd have an equivalent openness to extension but through another mechanism; a **Template method** in the first case and a **delegation** in the second.

Since inheritance gives a much stronger coupling than delegation, the puritanical view is to delegate when you can and inherit when you must. That often leads to composable systems and overall realizes more opportunities for reuse. Inheritance based extension is somewhat easier to grasp and more common since it's what is usually thought.

Samples

OCP by Composition

As stated above the OCP can be followed when a class does not depend on the concrete implementation of dependencies but rather on their abstraction. As a simple sample consider an authentication service that references a logging service to log

who is trying to be authenticated and whether the authentications has succeeded or not.

```
public class AuthenticationService
{
   private ILogger logger = new TextFileLogger();

   public ILogger Logger { set{ logger = value; }}

   public bool Authenticate(string userName, string password)
   {
      logger.Debug("Authentication '{0}'", userName);
      // try to authenticate the user
   }
}
```

Since the authentication service depends on an (abstract) interface, **ILogger** of the logging service,

```
public interface ILogger
{
   void Debug(string message, params object[] args);
   // other methods omitted for brevity
}
```

not on a concrete implementation, the behavior of the component can be altered without changing the code of the authentication service. Instead of logging to a text file, which might be the default, we can implement another logging service that logs to the event log or to the database. The new logger service has to implement the interface **ILogger**. At runtime we can inject a different implementation of the logger service into the authentication service, e.g.

```
var authService = new AuthenticationService();
authService.Logger = new DatabaseLogger();
```

There are some good examples publicly available of how a project can adhere to the OCP by using composition. One of my favorites is the OSS project Fluent NHibernate. As an example, the auto-mapping can be modified and/or enhanced without changing the source code of the project. Let's have a look at the usage of the AutoPersistenceModel class for illustration.

```
var model = new AutoPersistenceModel();
model.WithConvention(convention =>
    {
        convention.GetTableName = type => "tbl_" + type.Name;
        convention.GetPrimaryKeyName = type => type.Name + "Id";
        convention.GetVersionColumnName = type => "Version";
    }
  );
```

Without changing the source code of the **AutoPersistenceModel** class we can change the behavior of the auto mapping process significantly. In this case we have (with the aid of some lambda expression magic –> see this post) changed some of the conventions used when auto-mapping the entities to database tables. We have declared that the name of the database tables should always be the same as the name of the corresponding entity and that the primary key of each table should have the name of the corresponding entity with a postfix "Id". Finally, the version column of each table should be named "Version".

This modification of the (runtime) behavior is possible since the **AutoPersistenceModel** class depends on abstractions – in this case lambda expressions – and not on specific implementations. The signature of the **WithConvention** method is as follows

```
public AutoPersistenceModel WithConvention(Action<Conventions> conventionAction)
```

OCP by Inheritance

Let's assume we want to implement a little paint application which can draw different shapes in a window. At first we start with a single kind of graphical shape, namely lines. A line might me defined as follows

```
public class Line
{
  public void Draw(ICanvas canvas)
  { /* draw a line on the canvas */ }
}
```

It has a draw method which expects a canvas as parameter.

Now our paint application might contain a **Painter** class which is responsible for managing all line objects and which contains a method **DrawAll** which draws all lines on a canvas.

```
public class Painter
{
  private IEnumerable<Line> lines;

  public void DrawAll()
  {
    ICanvas canvas = GetCanvas();
    foreach (var line in lines)
    {
      line.Draw(canvas);
    }
  }
```

```
    /* other code omitted for brevity */
}
```

This application has been in use for a while. Now all of the sudden the user does not only want to paint lines but also rectangles. A naive approach would now be to first implement a new class **Rectangle** similar to the line class which also has a **Draw** method.

```
public class Rectangle
{
    public void Draw(ICanvas canvas)
    { /* draw a line on the canvas */ }
}
```

Next, modify the **Painter** class to account for the fact that we now also have to manage and paint rectangles.

```
public class Painter
{
    private IEnumerable<Line> lines;
    private IEnumerable<Rectangle> rectangles;

    public void DrawAll()
    {
        ICanvas canvas = GetCanvas();
        foreach (var line in lines)
        {
            line.Draw(canvas);
        }
        foreach (var rectangle in rectangles)
        {
            rectangle.Draw(canvas);
        }
    }
}
```

One can easily see that the **Painter** class is certainly not adhering to the open/closed principle. To be able to manage and paint the rectangles we have to change its source code. As such, the **Painter** class was not *closed for modifications*.

Now, we can easily fix this problem by using inheritance. We just define a base class **Shape**, from which all other concrete shapes (e.g. lines and rectangles) inherit. The **Painter** class then only deals with shapes. Let's first define the (abstract) **Shape** class

```
public abstract class Shape
{
    public abstract void Draw(ICanvas canvas);
}
```

All concrete shapes have to inherit from this class. Thus, we have to modify the **Line** and the **Rectangle** class like this (note the override keyword on the draw method)

```
public class Line : Shape
{
   public override void Draw(ICanvas canvas)
   { /* draw a line on the canvas */ }
}

public class Rectangle : Shape
{
   public override void Draw(ICanvas canvas)
   { /* draw a line on the canvas */ }
}
```

Finally we modify the **Painter** so it only references shapes and not lines or rectangles

```
public class Painter
{
   private IEnumerable<Shape> shapes;

   public void DrawAll()
   {
      ICanvas canvas = GetCanvas();
      foreach (var shape in shapes)
      {
         shape.Draw(canvas);
      }
   }
}
```

If ever the user wants to extend the paint application and have other shapes like ellipses or Bezier curves, the **Painter** class (and especially the **DrawAll** method) does not have to be changed any more. Still, the **Painter** can draw ellipses or Bezier curves since these new shapes will have to inherit from the (abstract) base class **Shape**. The **Painter** class is now *closed for modification* but *open for extension* because we can add other kinds of shapes.

Conclusion

I'd like to see OCP done in conjunction with "prefer composition over inheritance" and as such, I prefer classes that have no virtual methods and are possibly sealed, but depend on abstractions for their extension. I consider this to be the "ideal" way to do OCP, as you've enforced the "C" and provided the "O" simultaneously. Of course, please don't construe this as meaning that there is no way to use inheritance properly or that inheritance somehow violates OCP, it doesn't. Due of the way most of us learned OOP, we tend to think of inheritance first, when people talk about "extension of an object".