Pablo's SOLID Software Development

The Los Techies crew is proud to compile a number of blog posts focused on a particular subject in addition to their regular blogging. Pablo's Topic of the Month for the month of March 2008 was on Bob Martin's S.O.L.I.D. design principles. We tried to cover all of them by the end of the month and duplicated some over time.

Table of Contents

PABLO'S SOLID SOFTWARE DEVELOPMENT 1

WHAT IS S.O.L.I.D.? 4

SRP: Single Responsibility Principle 4

OCP: Open Closed Principle 4

LSP: Liskov Substitution Principle 4

ISP: Interface Segregation Principle 4

DIP: Dependency Inversion Principle 4

SRP: SINGLE RESPONSIBILITY PRINCIPLE 5

SINGLE RESPONSIBILITY PRINCIPLE BY SEAN CHAMBERS 6

SINGLE RESPONSIBILITY PRINCIPLE BY JASON MERIDTH 8

REAL SWISS DON'T NEED SRP, DO THEY? BY GABRIEL SCHENKER 12

Introduction 12

Swiss people think differently... 13

Why does the rest of the world consider SRP to be important? 14

Refactoring step by step 16

Step 1: Defining a model 257

Step 2: Extracting the loading(parsing) into a repositoryl 18

Step 3: Introducing a presenter and extracting logic 25

Step 4: Implementing the Repository 24

The sample code 25

Summary 25

OCP: OPEN CLOSED PRINCIPLE 27

OPEN CLOSED PRINCIPLE BY JOE OCAMPO 28

A case study in OCP ignorance 28

Where'd we go wrong? 30

Solutions 31

OCP REVISITED IN RUBY BY JOE OCAMPO 36

But wait there is more! 40

THE OPEN CLOSED PRINCIPLE BY GABRIEL SCHENKER 42

Introduction 42

Sealed classes 43

Samples 46

Conclusion 50

LSP: LISKOV SUBSTITUTION PRINCIPLE 51

LISKOV SUBSTITUTION PRINCIPLE BY CHAD MYERS 52

A case study in LSP ignorance 52

Where'd we go wrong? 51

Solutions 51

ISP: INTERFACE SEGREGATION PRINCIPLE 53

INTERFACE SEGREGATION PRINCIPLE BY RAY HOUSTON 54

DIP: DEPENDENCY INVERSION PRINCIPLE 60

DEPENDENCY INVERSION PRINCIPLE BY JIMMY BOGARD 61

Spider webs and bad design 61

Too many responsibilities 63

Towards a better design 63

Factoring out the dependencies 63

Filling out the implementations 64

Isolating the ugly stuff 66

THE DEPENDENCY INVERSION PRINCIPLE BY GABRIEL SCHENKER 70

What is Bad Design? 70

An immobile design 70

The Dependency Inversion Principle 72

Why call it dependency inversion? 71

Summary 72

TOPICS RELATED TO S.O.L.I.D. 73

SINGLE-RESPONSIBILITY VERSUS NEEDLESS COMPLEXITY BY RAY HOUSTON 74

DIP: CREATING AND WORKING WITH A CLOUD OF OBJECTS BY DERICK BAILEY 76

CONCLUSIONS AND BENEFITS OF S.O.L.I.D. 803

LOW COUPLING 803

HIGH COHESION 803

ENCAPSULATION 803

COPYRIGHT AND CONTACT INFORMATION 814

What is S.O.L.I.D.?

S.O.L.I.D. is a collection of best-practice, object-oriented design principles which can be applied to your design, allowing you to accomplish various desirable goals such as loose-coupling, higher maintainability, intuitive location of interesting code, etc. S.O.L.I.D. is an acronym for the following principles (which are, themselves acronyms -- confused yet?).

These principles were pioneered and first collected into a written work by Robert 'Uncle Bob' Martin. You can find more details here:
http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Each of the various letters in the S.O.L.I.D. acronym is yet another acronym

SRP: Single Responsibility Principle

THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE.

OCP: Open Closed Principle

SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION BUT CLOSED FOR MODIFICATION.

LSP: Liskov Substitution Principle

FUNCTIONS THAT USE ... REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES WITHOUT KNOWING IT.

ISP: Interface Segregation Principle

CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE

DIP: Dependency Inversion Principle

A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS

B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS

SRP: Single Responsibility Principle

THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE.
http://www.objectmentor.com/resources/articles/srp.pdf

WENGER
SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

Single Responsibility Principle by Sean Chambers

After Chad and Ray, I followed suit as well and am doing Pablo's Topic of the month post on the Single Responsibility Principle or SRP for short.

In SRP a reason to change is defined as a responsibility, therefore SRP states, "An object should have only one reason to change". If an object has more than one reason to change then it has more than one responsibility and is in violation of SRP. An object should have one and only one reason to change.

Let's look at an example. In the example below I have a BankAccount class that has a couple of methods:

```
public abstract class BankAccount
{
    double Balance { get; }
    void Deposit(double amount) {}
    void Withdraw(double amount) {}
    void AddInterest(double amount) {}
    void Transfer(double amount, IBankAccount toAccount) {}
}
```

Let's say that we use this BankAccount class for a person's Checking and Savings account. That would cause this class to have more than two reasons to change. This is because Checking accounts do not have interest added to them and only Savings accounts have interest added to them on a monthly basis or however the bank calculates it.

Some people may say that the class would even have 3 reasons to change because of the Deposit/Withdraw methods as well but I think you can definitely get a little crazy with SRP. That being said, I believe it just depends on the context.

So, let's refactor this to be more SRP friendly.

```
public abstract class BankAccount
{
    double Balance { get; }
    void Deposit(double amount);
    void Withdraw(double amount);
    void Transfer(double amount, IBankAccount toAccount);
}

public class CheckingAccount : BankAccount
{
}
```

```

public class SavingsAccount : BankAccount
{
    public void AddInterest(double amount);
}
```

So what we have done is simply create an abstract class out of BankAccount and then created a concrete CheckingAccount and SavingsAccount class so that we can isolate the methods that are causing more than one reason to change.

When you actually think about it, every single class in the .Net Framework is violating SRP all of the time. The GetHashCode() and ToString() methods are causing more than one reason to change, although you could say that these methods are exempt because they exist in the framework itself and out of our reach for change.

I'm sure you can come up with a lot more instances where you have violated SRP, and even instances where it just depends on the context. As stated on Object Mentor: "The SRP is one of the simplest of the principle, and one of the hardest to get right".

Here is a link to the SRP pdf on Object Mentor for more information.

Single Responsibility Principle by Jason Meridth

This post is about the first letter in Uncle Bob's SOLID acronym, Single Responsibility Principle, and a continuation of The Los Techies Pablo's Topic of the Month - March: SOLID Principles. Sean has already posted on this, but I'd like to "contribute".

Note about the SOLID acronym and this blog "storm"*:*
This "principle" is more or less common sense, as are most of the other items in the SOLID acronym. I like the idea of this series because I personally have interviewed with companies who would ask about possible code scenarios and I respond with one of these principles or one of the GOF patterns and they look back at me with a blank stare. I know these are just labels, but if they can reduce the miscommunication possibilities and start standardizing our industry, I'm all for it. I know some of the new ideas and labels out there are still being hammered out (i.e., like the BDD discussions as of late), but that is part of the process and what has to happen in such a young industry like ours.

Single-Responsibility Principle (SRP):

A class should have only one reason to change.

A good anti-example is the Active Record pattern. This pattern is in contradiction of SRP. A domain entity handles persistence of its information. (Note: There is nothing wrong with using Active Record; I've recently used it on a quick demo site and it worked perfectly) Normally, you would have a controller method/action pass a "hydrated" entity to a method of a repository instance.

Like my favorite quote says:

Talk is cheap, show me the code ~ Linus Torvalds

Let's look at some .NET code.

Anti-SRP (Active Record)

Imagine you have a User entity that has a username and password property. I'm using the Castle Active Record libraries for this example.

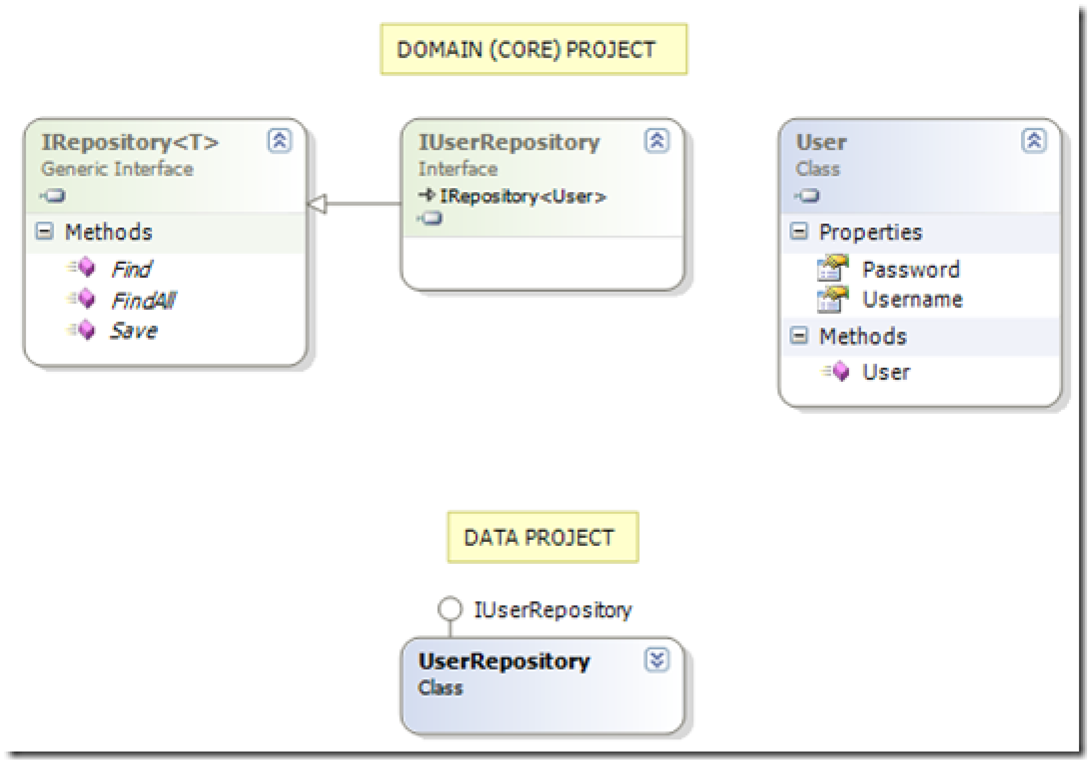
```
using System;
using Castle.ActiveRecord;

namespace ActiveRecordSample
{
    [ActiveRecord]
    public class User : ActiveRecordBase<User>
    {
        private int id;
        private string username;
        private string password;

        public User()
        {
        }

        public User(string username, string password)
        {
            this.username = username;
            this.password = password;
        }

        [PrimaryKey]
        public int Id
        {
            get { return id; }
            set { id = value; }
        }

        [Property]
        public string Username
        {
            get { return username; }
            set { username = value; }
        }

        [Property]
        public string Password
        {
            get { return password; }
            set { password = value; }
        }
    }
}
```

As you can see, you use attributes to dictate how your properties map to columns in your database table. Your entity name usually matches your table name, when using the ActiveRecord attribute with no explicit table name (i.e., [ActiveRecord("UserTableName")].

To save the user you would take an instantiated user and call user.Save(); This would cause an update to fire if the user instance had identity (aka an Id) and insert if it did not.

Translation to SRP

What I would normally do is have architecture like the following:



The UserRepository would be used by a web controller (I use monorail for my web projects), being passed a User instance, and Save(user) would be called.

```
using System.Collections.Generic;
using Castle.MonoRail.Framework;
using SrpPost.Core;
using SrpPost.Data;

namespace SrpPost.Web.Controllers
{
    [Layout("default"), Rescue("generalerror")]
    public class UserController : SmartDispatcherController
    {
        private readonly IUserRepository userRepository;

        public UserController(IUserRepository userRepository)
        {
            this.userRepository = userRepository;
        }

        public void Index()
        {
            RenderView("userlist");
        }

        public void Save([DataBind("user", Validate = true)] User user)
        {
            userRepository.Save(user);
            Flash["LoginError"] = "User saved successfully.";
            RenderView("userlist");
        }
    }
}
```

So, what it boils down to is that the user class now knows nothing on how it is persisted to the database.

SRP is one of the hardest principles to enforce because there is always room for refactoring out one class to multiple; each class has one responsibility. It is personal preference because class explosion does cause some people to become code zealots. One of my other favorite quotes lately is:

Always code as if the guy maintaining your code would be a violent psychopath and he knows where you live.

Enjoy!

Real Swiss don't need SRP, do they? by Gabriel Schenker

Introduction

You may ask yourself why I publish another article about the ***single responsibility principle (SRP)****. We already had some very good post about this principle at Los Techies, e.g. here, here and here to mention just a few. Well, the reason is that I consider this principle* ***one of the most helpful ones to achieve higher quality of code and better design*** *when applied consistently. And I want to approach this topic from a different standpoint than is usually done by other authors...*

What does **SRP** mean? The theoretical explanation (you might have read or heard many times already) is

"There is one and only one reason to change a class."

What does this mean? This means that we should start to think small. Each complex problem cannot easily be solved as a whole. It is much easier to first divide the problem in smaller sub-problems. Each sub-problem has reduced complexity compared to the overall problem and can then be tackled separately. There is a proverb whose origin is from the Romans which says: "Divide et imperat", translated to English this means: "Divide and reign". It was not possible for the Roman emperor to reign the whole empire alone. No, he divided the empire into **independent** regions and appointed a king or sovereign to each region. At the end, the Roman emperor had not much more to do than orchestrate those kings or sovereigns that reported to him.

Now what does this mean for me as a developer? Each developer has to solve problems. Very often these problems are rather complex. Often the boundary conditions defining the problem even change. Now we should start thinking in terms of *"divide et imperat".* Let's find the sub-problems in the domain we are working in. Keep on dividing each sub-problem into sub-sub-problems until you reach the point where such a "mini-problem" has just one single task left. Let's then solve each of those "mini-problem" in its own class. Since each class only has one single task to fulfill, there is (as a consequence) only one reason left to change this class. We only have to change this class if the corresponding task changes.

Instead of tasks, one often talks of **responsibility**. *Each class should have a single responsibility.* The responsibility is to just accomplish the assigned task.

I want to say it again: Applying the single responsibility principle leads to higher quality of code and to better design! Why? Because the code is

- more readable, that is easier to understand
- less error prone
- more robust
- better testable
- better maintainable and extendable

Swiss people think differently...

But wait a moment. Swiss people are not descendants of the Romans...

A real Swiss does not need to follow the **single responsibility principle**. That's something for others but not for us. As a representative sample I want to present you one of our most successful products: the **Swiss army knife**.

One can clearly see that this product has not just one single responsibility. There are several responsibilities assembled in a single unit. We have 2 knifes, one can opener, one bottle opener, an awl and a corkscrew. This unit is very handy and fits well into the pocket of every **real** Swiss man.

Some people prefer to even pack more functionality into this unit as you can see in this second picture at right.

Well, this one is even better! But now I have to admit, that not every pocket is big enough for **this** tool. Thus only the **strongest** Swiss men get one.

Another tool comes to my mind when remembering the time I passed in the Swiss army. Our helmet is also considered to be a multipurpose tool. We primarily use it to protect our heads from injuries but it has as well served me many times as a pillow. It is even considered as an anti-hand-grenade tool. We were told that if a hand grenade is thrown at us and we have no time or possibility to throw it away then we should just put our helmet over it and burden it with our body. To be honest, I've never tried it...

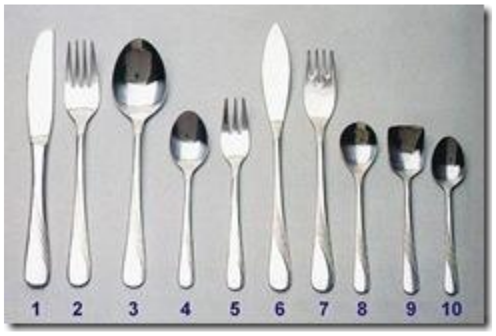
Hey, wait a moment. I can give you another sample where we clearly show to the rest of the world that the SRP is not for us. It's our famous Swiss cows. They are not only good for providing us milk and eventually meet; no, they are also very good soccer players! Currently we have 3 of them in our national soccer team.

Not to forget the famous Milka cow! Here a cow is used as an advertising medium. This is a very important responsibility by its own.

Well, I could possibly continue to give you good samples of Swiss products that are really successful without respecting the SRP.

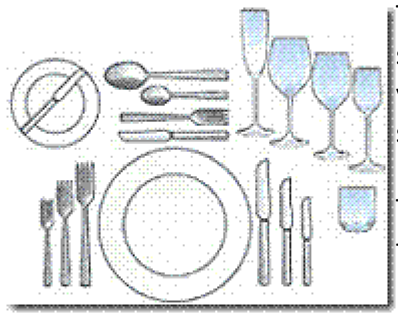
Why does the rest of the world consider SRP to be important?

Imagine one of the items of a Swiss army knife gets bent.
It would possibly render the whole item useless or at least harm its functionality. I will have to throw away the whole knife. What a waste!



Or imagine that I am really happy with my Swiss army knife but just one element does not totally fit my needs. I would like to replace just this element with another one, which is better suited to my needs. I cannot do it! It's just not possible without (negatively) affecting all the other elements of the knife.

Imagine having a nice dinner with your wife or husband in a first class restaurant. You certainly have had lots of knives, forks and spoons, each element serving for a single purpose. There are knives to cut steaks or pizzas or knives to eat fish and so on. Each item is optimized for its specific task. If one of these items gets broken or if it doesn't fulfill its duty any more then it can be replaced without affecting the other items.



The same can be said for the glasses and dishes. It doesn't make sense to have only one single glass for all kinds of beverages, if you like wine then you know what I mean. Red wine tastes significantly better in bigger glasses than white wine.

The same can be said about the dishes. It just doesn't make sense to serve soup in the same dish as a nice T-bone steak with baked potato is served.

Let's start coding!

Too many developers still don't respect the SRP. I consider this one of the primary reasons why an application gets unmanageable over time. More and more, the code base resembles a plate of Spaghetti. Consider the following sample.

```
public partial class Form1 : Form
{
  public Form1()
  {
    InitializeComponent();
  }

  private void btnBrowse_Click(object sender, EventArgs e)
  {
    openFileDialog1.Filter = "XML Document (*.xml)|*.xml|All Files (*.*)|*.*";
    var result = openFileDialog1.ShowDialog();
    if (result == DialogResult.OK)
    {
      txtFileName.Text = openFileDialog1.FileName;
      btnLoad.Enabled = true;
    }
  }

  private void btnLoad_Click(object sender, EventArgs e)
  {
    listView1.Items.Clear();
    var fileName = txtFileName.Text;
    using (var fs = new FileStream(fileName, FileMode.Open))
    {
      var reader = XmlReader.Create(fs);
      while (reader.Read())
      {
        if(reader.Name != "product") continue;
        var id = reader.GetAttribute("id");
        var name = reader.GetAttribute("name");
        var unitPrice = reader.GetAttribute("unitPrice");
        var discontinued = reader.GetAttribute("discontinued");
        var item = new ListViewItem(
           new string[]{id, name, unitPrice, discontinued});
```

```
            listView1.Items.Add(item);
        }
      }
    }
  }
```

A sample XML document could be

```
<?xml version="1.0" encoding="utf-8" ?>
<products>
  <product id="1" name="IPod Nano" unitPrice="129.55" discontinued="false"/>
  <product id="2" name="IPod Touch" unitPrice="259.10" discontinued="false"/>
  <product id="3" name="IPod" unitPrice="78.95" discontinued="true"/>
</products>
```

Such and example of code can be found **all** the time in **any** type of company. Such code is not only produced by part time developers but also by a lot of developers considering themselves as being professional developers. I would consider this not to be an exception but rather the norm.

The story behind this code is:

The user can select an XML document which contains a list of products from the file system. This XML document is then loaded and display on screen.

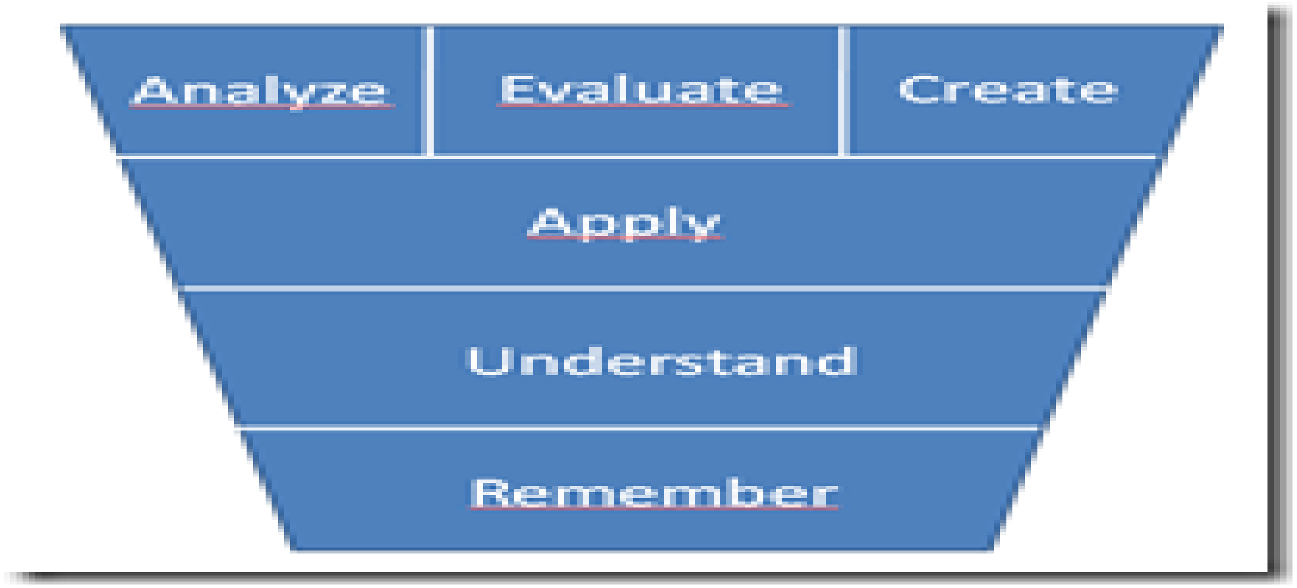
I have kept the above code sample as short as possible. The structure of the XML is very simple, the product has very few attributes and there is no error handling. In reality the above code would be much longer and convoluted.

Now let's analyze which and how many responsibilities the above code has:

Refactoring step by step

Step 1: Defining a model

One of the first concepts one can find is an implicit model. Since we are importing product data from an XML document it makes sense to introduce a **Product** entity as our model.

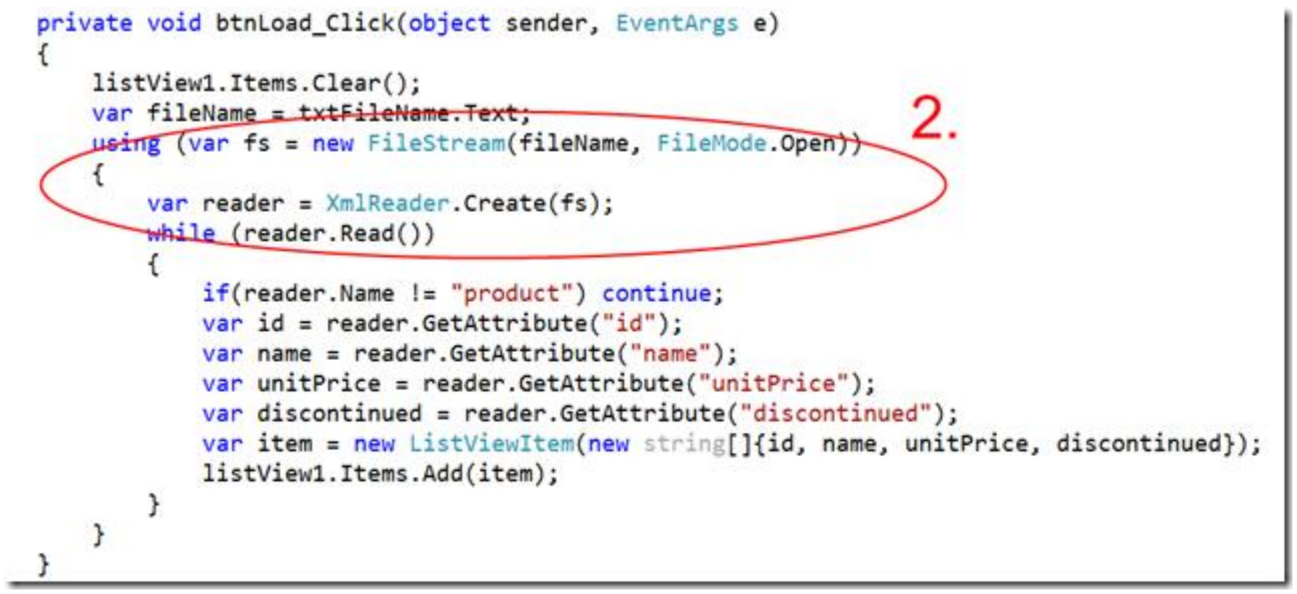


By carefully analyzing the above code snippet and the given XML document we can define the following model

```
public class Product
{
   public int Id { get; set; }
   public string Name { get; set; }
   public decimal UnitPrice { get; set; }
   public bool Discontinued { get; set; }
}
```

Step 2: Extracting the loading (and parsing) into a repository

We also can recognize a distinct concern of loading a list of products from a data source. In this case the data source is a XML document



```
private void btnLoad_Click(object sender, EventArgs e)
{
    listView1.Items.Clear();
    var fileName = txtFileName.Text;
    using (var fs = new FileStream(fileName, FileMode.Open))
    {
        var reader = XmlReader.Create(fs);
        while (reader.Read())
        {
            if(reader.Name != "product") continue;
            var id = reader.GetAttribute("id");
            var name = reader.GetAttribute("name");
            var unitPrice = reader.GetAttribute("unitPrice");
            var discontinued = reader.GetAttribute("discontinued");
            var item = new ListViewItem(new string[]{id, name, unitPrice, discontinued});
            listView1.Items.Add(item);
        }
    }
}
```

Wouldn't it make sense to have a specific component concerned with loading a list of products from a data source and return a list of instances of type Product? Something like this:

```
public interface IProductRepository
{
    IEnumerable<Product> GetByFileName(string fileName);
}
```

The repository is responsible to load, parse and map the XML document and return a list of Product items to the caller.

First Refactoring

If we now refactor the sample with the assumption of having a product entity and a product repository, the code might look like this:

```
private IProductRepository repository;

private void btnLoad_Click(object sender, EventArgs e)
{
    listView1.Items.Clear();
    var fileName = txtFileName.Text;
    var products = repository.GetByFileName(fileName);
    foreach (Product product in products)
    {
        var item = new ListViewItem(new[]
                         {
                            product.Id.ToString(),
                            product.Name,
                        product.UnitPrice.ToString(),
                        product.Discontinued.ToString()
                      });
    listView1.Items.Add(item);
  }
}
```

Step 3: Introducing a presenter and extracting logic not related to presentation from the view

Although our code already looks much more polished than before, we should still not be happy. A view (and here the form is a view) should only contain logic that is strictly related to presenting data and delegating requests triggered by the user to a *controller* or *presenter*. Thus we introduce a pattern which separates the concerns of a) visualization, b) orchestration and c) (data-) model. A pattern that perfectly fits our needs is the Model-View-Presenter pattern (MVP). The presenter is the component that orchestrates the interactions between model, view and (external) services. In this pattern the presenter is in command.

Let's analyze what the responsibility of the view should be:

- delegate the user's request to choose an XML document to the presenter
- delegate the user's request to load the data from the selected XML document to the presenter
- provide the name of the selected XML document to the presenter
- accept a file name (of a selected XML document) from the presenter
- display a given list of products provided by the presenter (in a ListView control)

Second refactoring

Assuming that we have such a **ProductPresenter** class, we can then refactor the view (or form-) code like this:

```
public partial class Form1 : Form
{
  public Form1()
  {
    InitializeComponent();
  }

  private ProductPresenter presenter;

  private void btnBrowse_Click(object sender, EventArgs e)
  {
    presenter.BrowseForFileName();
  }

  private void btnLoad_Click(object sender, EventArgs e)
  {
    presenter.GetProducts();
    }

    public void ShowProducts(IEnumerable<Product> products)
    {
        listView1.Items.Clear();
        foreach (Product product in products)
        {
            var item = new ListViewItem(new[]
                                {
                                    product.Id.ToString(),
                                    product.Name,
                                    product.UnitPrice.ToString(),
                                    product.Discontinued.ToString()
                                });
            listView1.Items.Add(item);
        }
    }

    public string GetFileName()
    {
        return txtFileName.Text;
    }

    public void SetFileName(string fileName)
    {
        txtFileName.Text = fileName;
        btnLoad.Enabled = true;
    }
}
```

Note that the above code now contains only display related code or code that delegates a user request to the presenter. So far, we have achieved a good separation of concerns.

Now we have to implement the presenter. As said before, the presenter is responsible to orchestrate the collaboration of model, view and external/additional services. Thus, the presenter should not contain any business logic. The presenter should be slim and slick! He delegates all work to other components.

Avoid implementing a fat presenter which is considered to be an anti-pattern.

We have identified so far (see code above) that the presenter needs at least the following two methods:

```
public class ProductPresenter
{
    public void BrowseForFileName()
    { ... }

    public IEnumerable<Product> GetProducts()
    { ... }
}
```

The presenter accesses its view (that is in this case the form), via an interface:

```
public interface IProductView
{
    void Initialize(ProductPresenter presenter);
    string GetFileName();
    void ShowProducts(IEnumerable<Product> products);
    void SetFileName(string fileName);
}
```

Which has to be implemented by the form, that is:

```
public partial class Form1 : Form, IProductView
{ ... }
```

Let's have a look at the implementation of the presenter:

```
public class ProductPresenter
{
    private readonly IOpenFileDialog openFileDialog;
    private readonly IProductRepository repository;
    private readonly IProductView view;

    public ProductPresenter()
    {
        view = new Form1();
        view.Initialize(this);
        repository = new ProductRepository();
        openFileDialog = new OpenFileDialogWrapper();
    }

    public void BrowseForFileName()
    {
        openFileDialog.Filter = "XML Document (*.xml)|*.xml|All Files (*.*)|*.*";
        var result = openFileDialog.ShowDialog();
        if (result == DialogResult.OK)
            view.SetFileName(openFileDialog.FileName);
    }

    public void GetProducts()
    {
        var products = repository.GetByFileName(view.GetFileName());
        view.ShowProducts(products);
    }
}
```

It's obvious from the above code that the presenter does not do much more than orchestrate. It interacts with the view and the repository as well as with an open file dialog service (this is just a wrapper around the OpenFileDialog class of the .NET framework).

Please note the second line in the constructor. The presenter calls the Initialize() method of the view and passes itself as a reference. As such the view gets knowledge of its responsible presenter. Remember that the presenter is in command of the model-view-presenter triad!

Starting the application

How can the application be started? After the refactoring, we do not give the command to the view/form but to the presenter. Thus we might have something like this:

```
static class Program
{
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);

        var presenter = new ProductPresenter();
        Application.Run((Form) presenter.View);
    }
}
```

In the Main() method we instantiate a product presenter, which in turn internally creates an instance of its dedicated view (which is a form in our case). We then use the presenter's view and pass it to the Run() method of the application object.

We have to add a property View to the presenter to complete its implementation

```
public IProductView View
{
  get { return view; }
}
```

Step 4: Implementing the repository

We still have to implement the repository. There is just one method we need to implement:

```
public class ProductRepository : IProductRepository
{
    public IEnumerable<Product> GetByFileName(string fileName)
```

```
  {
    var products = new List<Product>();
    using (var fs = new FileStream(fileName, FileMode.Open))
    {
      var reader = XmlReader.Create(fs);
      while (reader.Read())
      {
        if (reader.Name != "product") continue;
        var product = new Product();
        product.Id = int.Parse(reader.GetAttribute("id"));
        product.Name = reader.GetAttribute("name");
        product.UnitPrice = decimal.Parse(reader.GetAttribute("unitPrice"));
        product.Discontinued = bool.Parse(reader.GetAttribute("discontinued"));
        products.Add(product);
      }
    }
    return products;
  }
}
```

Now I have every piece needed to make the application run.

Refactoring again

But wait a moment! There are still at least 3 concerns handled by the repository class. One is the retrieval of the data, another one is the looping over the nodes of the XML document and the third one is the mapping of a XML node to a product. So let's refactor again:

```
public class ProductRepository : IProductRepository
{
  private readonly IFileLoader loader;
  private readonly IProductMapper mapper;

  public ProductRepository()
  {
    loader = new FileLoader();
    mapper = new ProductMapper();
  }

  public IEnumerable<Product> GetByFileName(string fileName)
  {
    var products = new List<Product>();
    using (Stream input = loader.Load(fileName))
    {
      var reader = XmlReader.Create(input);
      while (reader.Read())
      {
        if (reader.Name != "product") continue;
        var product = mapper.Map(reader);
        products.Add(product);
      }
    }
    return products;
  }
}
```

Now I have a file loader which is responsible for loading the XML document and returning it as a stream to me. And I also have a mapper, which is responsible to map a single XML node to a product. As a result, the code of the repository has become very simple and manageable.

So let's have a look at the implementation of the mapper component:

```
public interface IProductMapper
{
    Product Map(XmlReader reader);
}

public class ProductMapper : IProductMapper
{
    public Product Map(XmlReader reader)
    {
        if (reader == null)
            throw new ArgumentNullException("XML reader used when mapping cannot be null.");
        if (reader.Name != "product")
            throw new InvalidOperationException("XML reader is not on a product fragment.");

        var product = new Product();
        product.Id = int.Parse(reader.GetAttribute("id"));
        product.Name = reader.GetAttribute("name");
        product.UnitPrice = decimal.Parse(reader.GetAttribute("unitPrice"));
        product.Discontinued = bool.Parse(reader.GetAttribute("discontinued"));
        return product;
    }
}
```

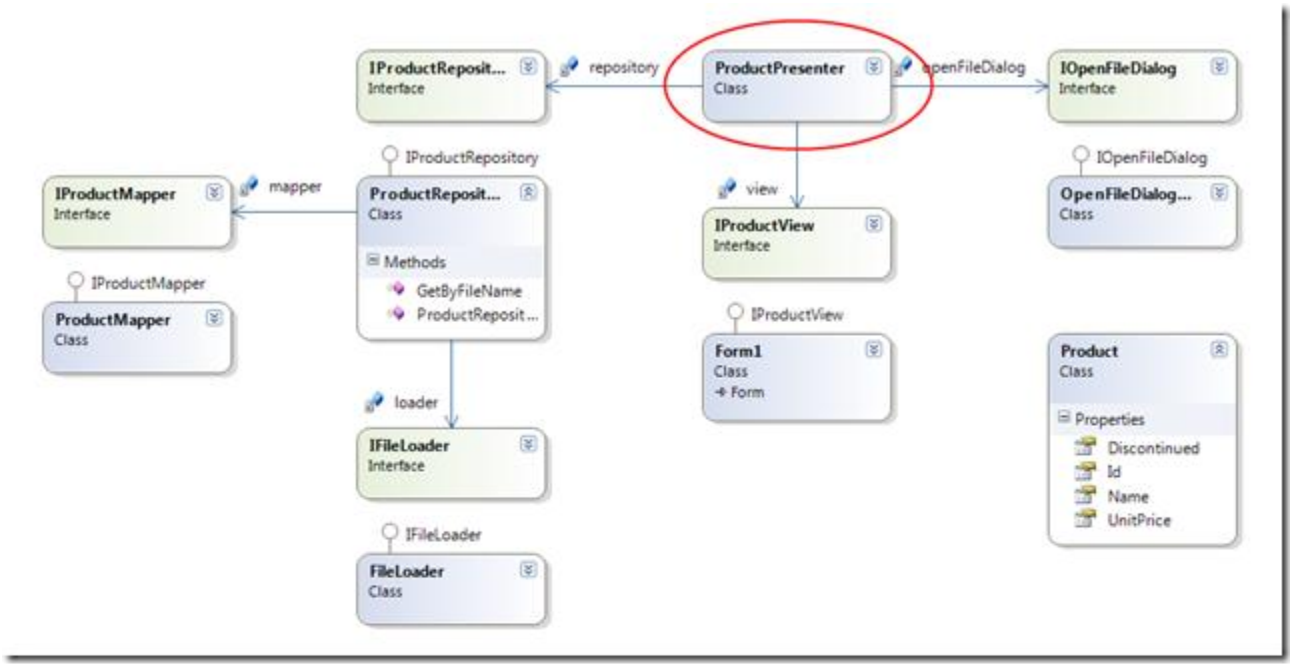
The mapper code is very straight forward. I have even introduced some basic error handling. Note, that I could still go farther with SRP and introduce an XML attribute parser (helper-) class if I want to go to the max... but let's just stop here for the moment!

The implementation of the (file-) loader is also very simple

```
public interface IFileLoader
{
    Stream Load(string fileName);
}

public class FileLoader : IFileLoader
{
    public Stream Load(string fileName)
    {
        return new FileStream(fileName, FileMode.Open);
    }
}
```

Class diagram of the fully refactored sample

The image below shows the class diagram of the fully refactored sample. There are many components involved in this little sample. But each component is very simple and has just one single responsibility.



The sample code

You can find the code of the original and the fully refactored sample, here. Just use a SVN client like TortoiseSVN to download the code.

Summary

You might be overwhelmed by the sheer amount of classes (and code) introduced by the refactoring. For this simple sample, it is certainly overhead, not worth the investment. But don't forget that **real** applications are more complex than this simple example. The more complex and the bigger an application becomes the more important SRP becomes. With the aid of the

SRP, a complex problem can be reduced to many small sub-problems which are easy to solve in isolation. Just remember the sample of the Roman empire I gave you in the introduction of this post.

When respecting and applying the SRP in my daily work, I have always attained huge benefits. My code is more robust, more stable, better understandable and maintainable. At the end, I get a better and much cleaner design.

OCP: Open Closed Principle

SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION BUT CLOSED FOR MODIFICATION.
http://www.objectmentor.com/resources/articles/ocp.pdf

Open Closed Principle by Joe Ocampo

The open closed principle is one of the oldest principles of Object Oriented Design. I won't bore you with the history since you can find countless articles out on the net. But if you want a really comprehensive read, please checkout Robert Martin's excellent write up on the subject.

The open closed principle can be summoned up in the following statement.

The open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification";[1] that is, such an entity can allow its behavior to be modified without altering its source code.

Sounds easy enough but many developers seem to miss the mark on actually implementing this simple extensible approach. I don't think it is a matter of skill set, as much as I feel that they have never been taught how to approach applying OCP to class design.

A case study in OCP ignorance

Scenario: We need a way to filter products based off the color of the product.

All entities in a software development ecosystem behave a certain way that is dependent upon a governed context. In the scenario above, you realize that you are going to need a Filter class that accepts a color and then filters all the products that adhere to that color.

The filter class' responsibility is to filter products (its job), based off the action of filtering by color (its behavior). So your goal is to write a class that will always be able to filter products. (Work with me on this I am trying to get you into a mindset because that is all OCP truly is at its heart.)

To make this easier I like to tell developers to fill in the following template.

The {class} is responsible for {its job} by {action/behavior}

The ***ProductFilter*** is responsible for ***filtering products*** by ***color***

Now, let's write our simple class to do this:

```
public class ProductFilter
{
   public IEnumerable<Product> ByColor(IList<Product> products, ProductColor productColor)
   {
      foreach (var product in products)
      {
         if (product.Color == productColor)
            yield return product;
      }
   }
}
```

As you can see this pretty much does the job of filtering a product by color. Pretty simple, but imagine if you had the following typical conversation with one of your users.

User: "We need to also be able to filter by size."

Developer: "Just size alone or color and size? "

User: "Umm probably both."

Developer: "Great!"

So let's use our OCP scenario template again.

The ***ProductFilter*** is responsible for ***filtering products*** by ***color***

The ***ProductFilter*** is responsible for ***filtering products*** by ***size***

The ***ProductFilter*** is responsible for ***filtering products*** by ***color and size***

Now the code:

```
public class ProductFilter
{
   public IEnumerable<Product> ByColor(IList<Product> products, ProductColor productColor)
   {
      foreach (var product in products)
      {
         if (product.Color == productColor)
            yield return product;
      }
   }

   public IEnumerable<Product> ByColorAndSize(IList<Product> products,
                                    ProductColor productColor,
                                    ProductSize productSize)
   {
      foreach (var product in products)
      {
         if ((product.Color == productColor) &&
            (product.Size == productSize))
            yield return product;
      }
   }

   public IEnumerable<Product> BySize(IList<Product> products,
                              ProductSize productSize)
   {
      foreach (var product in products)
      {
         if ((product.Size == productSize))
            yield return product;
      }
   }
}
```

This is great but this implementation is violating OCP.

Where'd we go wrong?

Let's revisit again what Robert Martin has to say about OCP.

Robert Martin says modules that adhere to Open-Closed Principle have 2 primary attributes:

1. "Open For Extension" - It is possible to extend the behavior of the module as the requirements of the application change (i.e. change the behavior of the module).

2. "Closed For Modification" - Extending the behavior of the module does not result in the changing of the source code or binary code of the module itself.

Let's ask the following question to ensure we **ARE** violating OCP.

Every time a user asks for new criteria to filter a product, do we have to modify the ProductFilter class?
Yes! This means it is not CLOSED for modification.

Every time a user asks for new criteria to filter a product, can we extend the behavior of the ProductFilter class to support the new criteria, without opening up the class file again and modifying it?
No! This means it is not OPEN for extension.

Solutions

One of the easiest ways to implement OCP is utilize a template or strategy pattern. If we still allow the Product filter to perform its job of invoking the filtering process, we can put the implementation of filtering in another class. This is achieved by mixing in a little LSP.

Here is the template for the ProductFilterSpecification:

```
public abstract class ProductFilterSpecification
{
   public IEnumerable<Product> Filter(IList<Product> products)
   {
      return ApplyFilter(products);
   }

   protected abstract IEnumerable<Product> ApplyFilter(IList<Product> products);
}
```

Let’s go ahead and create our first criteria, which is a color specification.

```
public class ColorFilterSpecification : ProductFilterSpecification
{
   private readonly ProductColor productColor;

   public ColorFilterSpecification(ProductColor productColor)
   {
      this.productColor = productColor;
   }

   protected override IEnumerable<Product> ApplyFilter(IList<Product> products)
   {
      foreach (var product in products)
      {
         if (product.Color == productColor)
            yield return product;
      }
   }
}
```

Now all we have to do is extend the actual ProductFilter class to accept our template ProductFilterSpecification.

```
public IEnumerable<Product> By(IList<Product> products, ProductFilterSpecification filterSpecification)
{
   return filterSpecification.Filter(products);
}
```

OCP goodness!

So let’s make sure we are **NOT** violating OCP and ask the same questions we did before.

Every time a user asks for new criteria to filter a product do we have to modify the ProductFilter class?
No! Because we have marshaled the behavior of filtering to the ProductFilterSpecification. "Closed for modification"

Every time a user asks for new criteria to filter a product can we extend the behavior of the ProductFilter class to support the new criteria, without opening up the class file again and modifying it?
Yes! All we simply have to do is, pass in a new ProductFilterSpecification. "Open for extension"

Now let’s just make sure we haven’t modified too much from our intentions of the ProductFilter. All we simply have to do is validate that our ProductFilter still has the same behavior as before.

The ***ProductFilter*** is responsible for ***filtering products*** by ***color: Yes it still does that!***

The ***ProductFilter*** is responsible for ***filtering products*** by ***size: Yes it still does that!***

The ***ProductFilter*** is responsible for ***filtering products*** by ***color and size: Yes it still does that!***

If you are a good TDD/BDD practitioner you should already have all these scenarios covered in your Test Suite.

Here is the final code:

```
namespace OCP_Example.Good
{
   public class ProductFilter
   {
      [Obsolete("This method is obsolete; use method 'By' with ProductFilterSpecification")]
      public IEnumerable<Product> ByColor(IList<Product> products, ProductColor productColor)
      {
         foreach (var product in products)
         {
            if (product.Color == productColor)
               yield return product;
         }
      }

      [Obsolete("This method is obsolete; use method 'By' with ProductFilterSpecification")]
      public IEnumerable<Product> ByColorAndSize(IList<Product> products,
                                    ProductColor productColor,
                                    ProductSize productSize)
```

```
    {
        foreach (var product in products)
        {
            if ((product.Color == productColor) &&
                (product.Size == productSize))
                yield return product;
        }
    }

    [Obsolete("This method is obsolete; use method 'By' with ProductFilterSpecification")]
    public IEnumerable<Product> BySize(IList<Product> products,
                                    ProductSize productSize)
    {
        foreach (var product in products)
        {
            if ((product.Size == productSize))
                yield return product;
        }
    }

    public IEnumerable<Product> By(IList<Product> products, ProductFilterSpecification filterSpecification)
    {
        return filterSpecification.Filter(products);
    }
}

public abstract class ProductFilterSpecification
{
    public IEnumerable<Product> Filter(IList<Product> products)
    {
        return ApplyFilter(products);
    }

    protected abstract IEnumerable<Product> ApplyFilter(IList<Product> products);
}

public class ColorFilterSpecification : ProductFilterSpecification
{
    private readonly ProductColor productColor;

    public ColorFilterSpecification(ProductColor productColor)
    {
        this.productColor = productColor;
    }

    protected override IEnumerable<Product> ApplyFilter(IList<Product> products)
    {
        foreach (var product in products)
        {
            if (product.Color == productColor)
                yield return product;
        }
    }
}

public enum ProductColor
{
    Blue,
    Yellow,
    Red,
    Gold,
    Brown
}

public enum ProductSize
{
    Small, Medium, Large, ReallyBig
}

public class Product
{
    public Product(ProductColor color)
    {
        this.Color = color;
    }

    public ProductColor Color { get; set; }

    public ProductSize Size { get; set; }
}

[Context]
public class Filtering_by_color
{
    private ProductFilter filterProduct;
    private IList<Product> products;

    [SetUp]
    public void before_each_spec()
    {
        filterProduct = new ProductFilter();
        products = BuildProducts();
    }

    private IList<Product> BuildProducts()
    {
        return new List<Product>
                    {
                        new Product(ProductColor.Blue),
                        new Product(ProductColor.Yellow),
                        new Product(ProductColor.Yellow),
                        new Product(ProductColor.Red),
                        new Product(ProductColor.Blue)
                    };
    }


    [Specification]
    public void should_filter_by_the_color_given()
    {
      int foundCount = 0;
      foreach (var product in filterProduct.By(products, new ColorFilterSpecification(ProductColor.Blue)))
      {
        foundCount++;
      }

      Assert.That(foundCount, Is.EqualTo(2));
    }
  }
}
```

OCP revisited in Ruby by Joe Ocampo

I was playing with some Ruby code this weekend and thought I would show some OCP with Ruby.

For more of an in-depth discussion on OCP, please read my previous post.

Now the first thing I want to point out is that dynamic languages are naturally by default open for extension. Since the types are dynamic, there are no fixed (static) types. This enables us to have awesome extensibility. It is the closure part of the equation that really scares me more than anything else. If you really aren't careful when you are programming with dynamic languages you can quickly make a mess of things. This doesn't take away from the power of a dynamic language you just have to exercise greater care, that is all.

So let's use our OCP scenario template again and apply them to the example scenario.

- The ***ProductFilter*** is responsible for ***filtering products*** by ***color***
- The ***ProductFilter*** is responsible for ***filtering products*** by ***size***
- The ***ProductFilter*** is responsible for ***filtering products*** by ***color and size***

Let's go ahead and create the ProductFilter first.

```
class ProductFilter

  attr_reader :products

  def initialize(products)
    @products = products
  end

  def by(filter_spec)

  end
end
```

For those that have never created a class in Ruby, let me breakdown the syntax structures.

class keyword is used to define a class followed by the name of the class. It is important to note that class names must start with an uppercase letter. The uppercase letter signifies to the Ruby interpreter that this is constant; meaning that whenever the term "ProductFilter" comes up it will always reference this class structure.

attr_reader keyword used to signify a read only accessor (read only property). The property name follows the colon.

def keyword is used to declare a method block. The "initialize" method is the same as the constructor method in C#.

The **@** symbol denotes an instance variable. Notice that the "products" read accessor is never typed but is assigned in the constructor through @products reference. The instance variable @products is assigned to the products parameter variable that is passed into the constructor.

Now that we are talking about products we have to create the actual product class.

```
class Product
  attr_accessor :color
  attr_accessor :size

  def initialize(color, size)
    @color = color
    @size = size
  end
end
```

Nothing fancy here, just a class with two read/write accessors Color and Size.

If you remember from my previous post, I was using a template pattern to serve as the basis for extending the behavior of my filter class. Well I am going to do the same thing here (kind of) and define an Item_color_filter_spec class.

```
class Item_color_filter_spec
  attr_accessor :color

  def initialize(color)
    @color = color
  end

  def apply_to(items)

  end
end
```

Now, I have a class that accepts a color and has an "apply_to" method that accepts "items". I have left out the implementation code of this method on purpose.

The next thing I am going to do is create an array of products I can use against the ProductFilter class. Ruby makes this pretty painless:

```
products = [
  Product.new("Blue", "Large"),
  Product.new("Red", "Large"),
  Product.new("Blue", "Medium"),
  Product.new("Red", "Small"),
  Product.new("Blue", "Large"),
  Product.new("Yellow", "Small"),
]
```

So what's going on here? I declared a variable called "products" and assigned it to array by using the square braces "[ ]". Yup, that simple!

What you may have noticed is that I actually instantiated several new products in the array. To instantiate a class you simply use the "new" method that is part of the Object class that all objects in Ruby inherit from similar to C#.

Now that I have a collection of products, I am going to give to my product filter class:

```
product_filter = Product_Filter.new(products)
```

In the example below I am going to filter all the "Blue" products.

```
blue_products = product_filter.by(Item_color_filter_spec.new("Blue"))
```

At this point I am creating a new "Item_color_filter_spec" and giving the color "Blue" as the color to filter on. The ProductFilter class would then simply call the "apply_to" method on the "Item_color_filter_spec".

If you ran the code at this point nothing would happen because we haven't actually written our filter code yet. So to do that we are going to modify the "apply_to" method of our "Item_color_filter_spec" class with the following code:

```
def apply_to(items)
  items.select{|item| item.color == @color}
end
```

In the code above the "apply_to" method is expecting an array of items to be passed in. We then call the "select" method on the array class and pass in a filter "proc" by enclosing the statement in curly braces {} (In Ruby a "proc" is an object that holds a chunk of code but more on this later). The pipe block "||" is used to signify parameters to the proc, similar to the parameters of a method. The actual statement that is executed is after the pipe block. We are trusting that the "select" method of the array object is going to enumerate over each object (product) it contains and pass it into Proc. Once in the Proc, we simply determine if the color of product matches the instance variable "@color" of the "Item_color_filter_spec", in this case the color "blue". When the Proc evaluates to true it passes the item to an array that the select method returns.

You can possibly equate a "proc" to a lambda expression in C#.

(Normally I would have used rSpec to govern everything I am doing but I didn't want to explain BDD as well. I wanted to focus on the Ruby language in general.)

Now if you run the code, you will have three products in the "blue_products" variable. How do you know? Well let's iterate over it by using the following code.

```
blue_products.each do |product|
  puts(product.color)
end
```

Now remember, "blue_products" is an array object. The array object has an "each" method (as do other objects in Ruby) that accepts a "proc" object. The "proc" block is denoted by the "do" keyword and terminated with the "end" keyword. In our block we expecting that the array object's "each" method is going to give us a product. As the each method iterates over each product, it passes it to the "proc" and we simply tell Ruby to write it to the screen using the "puts" method. The result of this small statement block in the following:

```
Blue

Blue

Blue
```

So there you have it, 3 blue products!

I know, nothing special right? Well now let's play with some Ruby sweetness!

That sure is a lot effort to filter a product by color. Imagine if you had to keep adding different color filters. You would have to create several "color filter specs" over and over again. That is dull and most static languages have played this out! So let's use some Ruby Lambda's to accomplish the same thing as the "color filter spec"

We are going to create a filter spec that filters all products that are yellow.

Remember I told you that Ruby views all uppercase variable names as constants; well we are going to harness the power of this convention and create a constant to hold the reference to the lambda. After all, DRY principles still apply here.

```
YELLOW_COLOR_FILTER_SPEC = lambda do |products|
  products.select{|product| product.color == "Yellow"}
end
```

Nothing in the code block above should be foreign at this point, except for the "lambda" key word. What the "lambda" keyword does is tell the Ruby interpreter to assign the "Proc" to the constant "YELLOW_COLOR_FILTER_SPEC".

We have to now modify our ProductFilter class to accept a Proc object. All you have to do is add the following method to the ProductFilter class.

```
def by_proc(&filter_spec_proc)
  filter_spec_proc.call(@products)
end
```

This method has a parameter named "filter_spec_proc" but if you notice there is an ampersand preceding its declaration. This tells the Ruby interpreter to expect a Proc object to be passed in. Since we are expecting a Proc object, all we have to do, is use the "call" method and pass the products instance variable to the Proc.

Now let's wire the whole thing together.

```
yellow_products = product_filter.by_proc(&YELLOW_COLOR_FILTER_SPEC)
```

Pretty simple, about the only thing you have to remember is that you have to place the ampersand before the Proc constant when you call the "by_proc" method on the ProductFilter class.

But wait there is more!

Our ProductFilter class now has a method that accepts a Proc object. In our previous example, we passed it a constant that referenced a lambda Proc block. But since it accepts a Proc we can simply write the block in line with the method call like this.

With our new found knowledge let's filter all the "Red" products.

```
red_products = product_filter.by_proc do |products|
    products.select{|product| product.color == "Red"}
  end
```

Parentheses are optional in Ruby when you call a method.

Pretty nice huh?

Now for something really freaky, for all you static type people!

Let's say you aren't dealing with products anymore and you are dealing with Cars.

Cars have 4 wheels but they also have color don't they. Gee, it would be nice if we could filter the Cars just like we are able to filter the Products. Wait! We can, we already wrote it!

We have that “Item_color_filter_spec” class. We could use the ProductFilter class but that class already has a purpose but the “Item_color_filter_spec” is pretty agnostic.

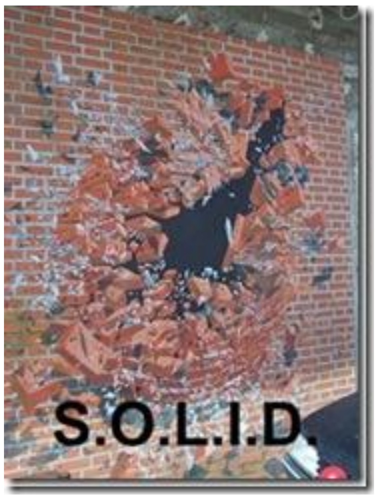
```
class Car
  attr_accessor :color
  def initialize(color)
    @color = color
  end
end

cars = [
  Car.new("Red"),
  Car.new("Blue"),
  Car.new("Red"),
  Car.new("Blue")
]

blue_filter = Item_color_filter_spec.new("Blue")
blue_cars = blue_filter.apply_to(cars)
```

Wow, talk about reuse!!!

How is this possible? Well, remember types do not exist in Ruby. There are objects but objects are duck typed when you ask them to perform an action. Meaning, if it walks like a duck or quacks like a duck, it must be a duck. The “Item_color_filter_spec” only cares that it is passed an array of objects. It is then going to iterate over the array of objects and call the “color” accessor on each object to check for equality against the instance variable that was passed in through the constructor. It doesn’t care if the array contains cars, products or both; just that whatever the object is, it has to have an accessor of "color."

I know this is a ton on information to digest all at once but I am just very passionate about the Ruby language. I see tremendous potential in its future, especially with its entrance into the .Net space, through the IronRuby project. I can easily see it over throwing the Visual Basic crowd once it becomes more main stream in the .Net community.

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".

LSP: Liskov Substitution Principle

FUNCTIONS THAT USE ... REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES WITHOUT KNOWING IT.
http://www.objectmentor.com/resources/articles/lsp.pdf

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Liskov Substitution Principle by Chad Myers

In my first (of hopefully more than one) post for The Los Techies Pablo's Topic of the Month - March: SOLID Principles effort, I'm going to talk about The Liskov Substitution Principle, as made popular by Robert 'Uncle Bob' Martin in The C++ Report.

I'm going to try as much as possible not to repeat everything that Uncle Bob said in the afore-linked PDF, you can go read the important stuff there. I'm going to try to give some real examples and relate this to the .NET world.

In case you're too lazy to read the link, let me start off with a quick summary of what LSP is: If you have a base class BASE and subclasses SUB1 and SUB2, the rest of your code should always refer to BASE and ***NOT*** SUB1 and SUB2.

A case study in LSP ignorance

The problems that LSP solves are almost always easily avoidable. There are some usual tell-tale signs that an LSP-violation is appearing in your code. Here's a scenario that walks through how an LSP-violation might occur. I'm sure we've all run into similar situations. Hopefully by walking through this, you can start getting used to spotting the trend up front and cutting it off before you paint yourself into a corner.

Let's say that somewhere in your data access code you had a nifty method through which all your DAO's/Entities passed and it did common things like setting the CreatedDate/UpdatedDate, etc.

```
public void SaveEntity(IEntity entity)
{
   DateTime saveDate = DateTime.Now;

   if( entity.IsNew )
      entity.CreatedDate = saveDate;

   entity.UpdatedDate = saveDate;

   DBConnection.Save(entity);
}
```

Clever, works like a champ. Many of you will hopefully have cringed at this code. I had a hard time writing it, but it's for illustration. There's a lot of code out there written like this. If you didn't cringe and you don't see what's wrong with that code, please continue reading. Now, the stakeholders come to you with a feature request:

Whenever a user saves a Widget, we need to generate a Widget Audit record in the database for tracking later.

You might be tempted to add it to your handy-dandy SaveEntity routine through which all entities pass:

```
public void SaveEntity(IEntity entity)
{
  WidgetEntity widget = entity as WidgetEntity;
  if( widget != null )
  {
    GenerateWidgetAuditEntry(widget);
  }

  // ...
```

Great! That also works like a champ. But a few weeks later, they come to you with a list of 6 other entities that need similar auditing features. So you plug in those 6 entities. A few weeks later, the come to you and ask you something like this:

When an Approval record is saved, we need to verify that the Approval is of the correct level. If it's not the correct level, we need to prompt the user for an excuse, otherwise they can't continue saving.

Oh boy, that's tricky. Well, now our SaveEntity looks something like this:

```
public void SaveEntity(IEntity entity)
{
  if( (entity as WidgetEntity) != null ){
    GenerateWidgetAuditEntry((WidgetEntity) entity);
  }

  if ((entity as ChocolateEntity) != null){
    GenerateChocolateAuditEntry((ChocolateEntity)entity);
  }

  // ...

  ApprovalEntity approval = entity as ApprovalEntity;
  if( approval != null && approval.Level < 2 ){
    throw new RequiresApprovalException(approval);
  }

  // ...
```

Pretty soon your small, clever SaveEntity method is 1,500 lines long and knows everything about every entity in the entire system.

Where'd we go wrong?

Well, there are several places to start. Centralizing the saving of entities isn't the greatest idea. Putting the logic for auditing whether entries need to be created or not into the SaveEntity method was definitely the wrong thing to do. Finally, due to the complexities of handling wildly differing business logic for different entities, you have a control flow problem with the approval level that requires the use of a thrown exception to break out of the flow (which is akin to a 'goto' statement in days of yore).

The concerns of auditing, setting created/updated dates, and approval levels are separate and orthogonal from each other and shouldn't be seen together, hanging around in the same method, generally making a mess of things.

More to the point of this blog post; SaveEntity violates the Liskov Substitution Principle. That is to say, SaveEntity takes an IEntity interface/base class but deals with specific sub-classes and implementations of IEntity. This violates a fundamental rule of object-oriented design (polymorphism) since SaveEntity pretends to work with any particular IEntity implementation when, in fact, it doesn't. More precisely, it doesn't treat all IEntity's exactly the same, some get more attention than others.

Why is this a problem? What if you were reusing your terribly clever SaveEntity method on another project and have dozens of IEntity implementations and the stakeholders for that project also wanted the auditing feature. Now you've got a problem.

Solutions

One fine approach to this problem of doing things at-the-moment-of-saving would be to use the Visitor Pattern as described by Matthew Cory in this post. Though, I would say in this particular example, there is a much deep-rooted and systemic design problem which revolves around the centralization of data access.

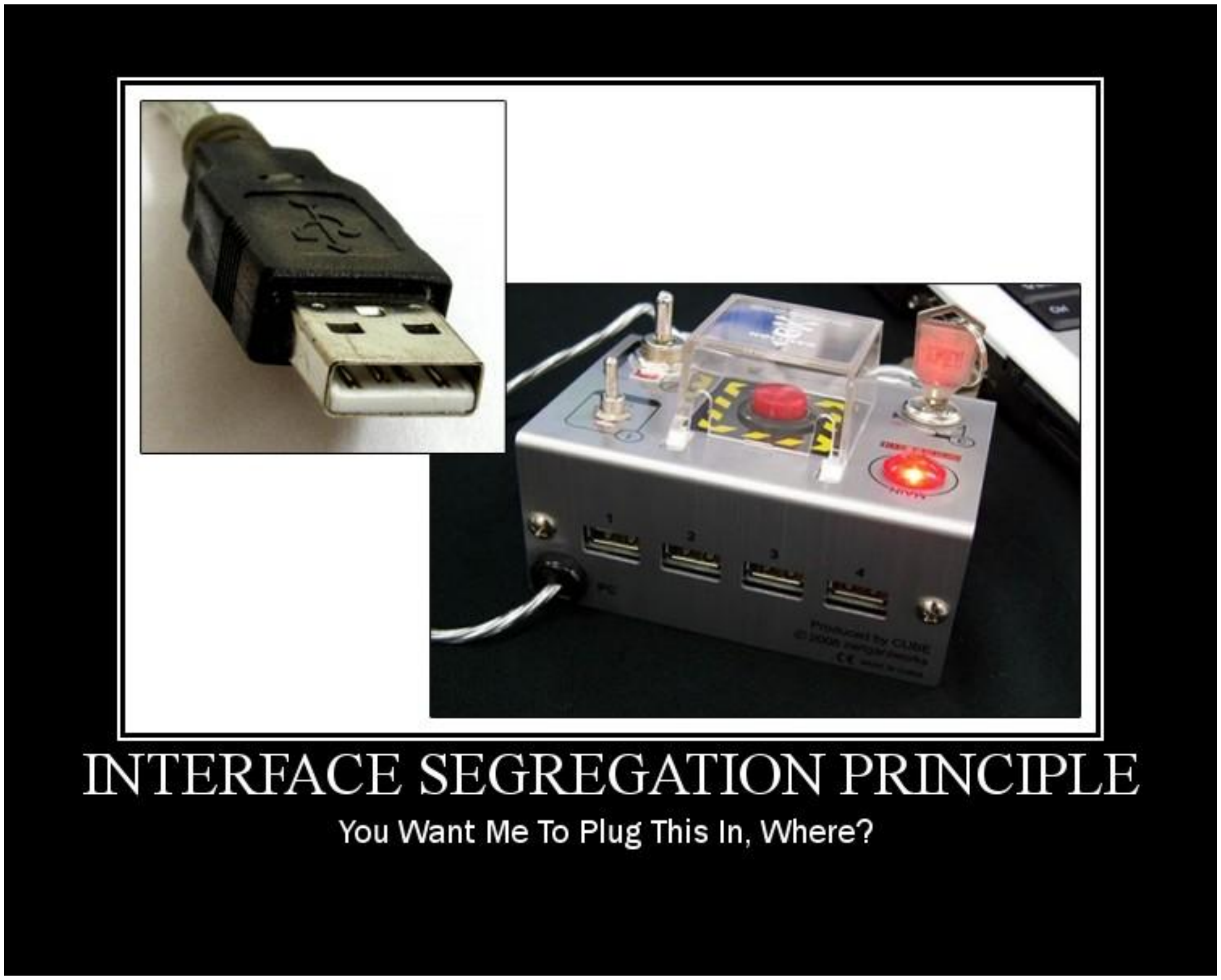
Another, in our case more preferable, way to go might be to use the repository pattern for managing data access. Rather than having "One Method to Rule them All", you could have your repositories worry about the Created/Updated date time and devise a system whereby all the repository implementations share some of the Created/Updated date entity save/setup logic.

As specific one-off problems arise (such as auditing, extra approval/verification, etc.), they can be handled in a similarly one-off manner. This is achieved by the individual entity's related repository (who knows all about that one type of entity and that's it). If you notice that several entities are doing the same sort of thing (i.e. auditing). You can, create a class and method to handle auditing in a common manner and provide the various repositories which need auditing with that functionality. Resist, if at all possible, the urge to create an 'AuditingRepositoryBase' class that provides the auditing functionality. Inevitably, one of those audit-requiring entities will have another, orthogonal concern for which you will have another *Base class and, since you can't do multiple inheritance in .NET, you are now stuck. Prefer composition of functionality over inheritance of functionality whenever possible.

If you have a rich domain model, perhaps the most elegant approach of all would be to make things like auditing a first-class feature of the domain model. Every Widget always has at least one WidgetAuditEntry associated with it and this association is managed through the domain logic itself. Likewise, the approval level would be best handled higher up in the logic chain to prevent last minute "gotchas" in the lifecycle that would require something less than elegant like an exception as a thinly veiled 'goto' bailout.

ISP: Interface Segregation Principle

CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE
http://www.objectmentor.com/resources/articles/isp.pdf


INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

Interface Segregation Principle by Ray Houston

In following suite with the The Los Techies Pablo's Topic of the Month - March: SOLID Principles, I chose to write a little about the The Interface Segregation Principle (ISP). As Chad pointed out with LSP, the ISP is also one of Robert 'Uncle Bob' Martin's S.O.L.I.D design principles.

Basically ISP tells us that clients shouldn't be forced to implement interfaces they don't use. In other words, if you have an abstract class or an interface, then the implementers should not be forced to implement parts that they don't care about.

I was having trouble thinking of a real world example for ISP but then was reminded about implementing a custom Membership Provider in ASP.NET 2.0. I had completely blocked that monstrosity out of my mind (for good reason).

The Membership Provider was a way to integrate with some of the ASP.NET's built in management of users and its associated server controls. For me, it ended up being a lot more trouble than it was worth, but it turns out to be a good example of a fat interface. In order to implement your own Membership Provider you "simply" implement the abstract class MembershipProvider like so:

```
public class CustomMembershipProvider : MembershipProvider
{
    public override string ApplicationName
    {
        get
        {
            throw new Exception("The method or operation is not implemented.");
        }
        set
        {
            throw new Exception("The method or operation is not implemented.");
        }
    }

    public override bool ChangePassword(string username, string oldPassword, string newPassword)
    {
        throw new Exception("The method or operation is not implemented.");
    }

    public override bool ChangePasswordQuestionAndAnswer(string username, string password,
        string newPasswordQuestion, string newPasswordAnswer)
    {
        throw new Exception("The method or operation is not implemented.");
    }
```

```
public override MembershipUser CreateUser(string username, string password, string email,
   string passwordQuestion, string passwordAnswer, bool isApproved, object providerUserKey,
   out MembershipCreateStatus status)
{
   throw new Exception("The method or operation is not implemented.");
}

public override bool DeleteUser(string username, bool deleteAllRelatedData)
{
   throw new Exception("The method or operation is not implemented.");
}

public override bool EnablePasswordReset
{
   get { throw new Exception("The method or operation is not implemented."); }
}

public override bool EnablePasswordRetrieval
{
   get { throw new Exception("The method or operation is not implemented."); }
}

public override MembershipUserCollection FindUsersByEmail(string emailToMatch, int pageIndex,
   int pageSize, out int totalRecords)
{
   throw new Exception("The method or operation is not implemented.");
}

public override MembershipUserCollection FindUsersByName(string usernameToMatch, int pageIndex,
   int pageSize, out int totalRecords)
{
   throw new Exception("The method or operation is not implemented.");
}

public override MembershipUserCollection GetAllUsers(int pageIndex, int pageSize, out int totalRecords)
{
   throw new Exception("The method or operation is not implemented.");
}

public override int GetNumberOfUsersOnline()
{
   throw new Exception("The method or operation is not implemented.");
}

public override string GetPassword(string username, string answer)
{
   throw new Exception("The method or operation is not implemented.");
}

public override MembershipUser GetUser(string username, bool userIsOnline)
{
   throw new Exception("The method or operation is not implemented.");
}

public override MembershipUser GetUser(object providerUserKey, bool userIsOnline)
{
   throw new Exception("The method or operation is not implemented.");
}

public override string GetUserNameByEmail(string email)
{
   throw new Exception("The method or operation is not implemented.");
}

public override int MaxInvalidPasswordAttempts
{
   get { throw new Exception("The method or operation is not implemented."); }
}

public override int MinRequiredNonAlphanumericCharacters
{
   get { throw new Exception("The method or operation is not implemented."); }
}

public override int MinRequiredPasswordLength
{
   get { throw new Exception("The method or operation is not implemented."); }
}

public override int PasswordAttemptWindow
{
   get { throw new Exception("The method or operation is not implemented."); }
}

public override MembershipPasswordFormat PasswordFormat
{
   get { throw new Exception("The method or operation is not implemented."); }
}

public override string PasswordStrengthRegularExpression
{
   get { throw new Exception("The method or operation is not implemented."); }
}

public override bool RequiresQuestionAndAnswer
{
   get { throw new Exception("The method or operation is not implemented."); }
}

public override bool RequiresUniqueEmail
{
   get { throw new Exception("The method or operation is not implemented."); }
}
```

```
    public override string ResetPassword(string username, string answer)
    {
        throw new Exception("The method or operation is not implemented.");
    }

    public override bool UnlockUser(string userName)
    {
        throw new Exception("The method or operation is not implemented.");
    }

    public override void UpdateUser(MembershipUser user)
    {
        throw new Exception("The method or operation is not implemented.");
    }

    public override bool ValidateUser(string username, string password)
    {
        throw new Exception("The method or operation is not implemented.");
    }
}
```

Holy guacamole! That's a lot of stuff. Sorry for the code puke there, but wanted you to feel a little pain as I did trying to actually implement this thing. Hopefully you didn't get tired of scrolling through that and you're still with me. ;)

It turns out that you don't have to implement the parts you don't need, but this clearly violates the Interface Segregation Principle. This interface is extremely fat and not cohesive. A better approach would have been to break it up into smaller interfaces that allow the implementers to only worry about the parts that they need. I'm not going to go into the details of splitting this up, but I think you get the idea.

Since I cannot think of another real world example, let's look at a completely bogus example. Say you have the following code:

```
public abstract class Animal
{
    public abstract void Feed();
}

public class Dog : Animal
{
    public override void Feed()
    {
        // do something
    }
}

public class Rattlesnake : Animal
{
    public override void Feed()
    {
        // do something
    }
}
```

But then you realize that you have a need for some of the animals to be treated as pets and have them groomed. You may be tempted to do

```
public abstract class Animal
{
  public abstract void Feed();
  public abstract void Groom();
}
```

which would be fine for the Dog, but it may not be fine for the Rattlesnake (although I'm sure there is some freako out there that grooms their pet rattlesnake)

```
public class Rattlesnake : Animal
{
  public override void Feed()
  {
    // do something
  }

  public override void Groom()
  {
    // ignore - I'm not grooming a freaking rattlesnake
  }
}
```

Here we have violated the ISP by polluting our Animal interface. This requires us to implement a method that doesn't make sense for the Rattlesnake object. A better choice would be to implement an IPet interface, which only Dog could implement, without affecting Rattlesnake. You might end up with something like this:

```
public interface IPet
{
  void Groom();
}

public abstract class Animal
{
  public abstract void Feed();
}

public class Dog : Animal, IPet
{
  public override void Feed()
```

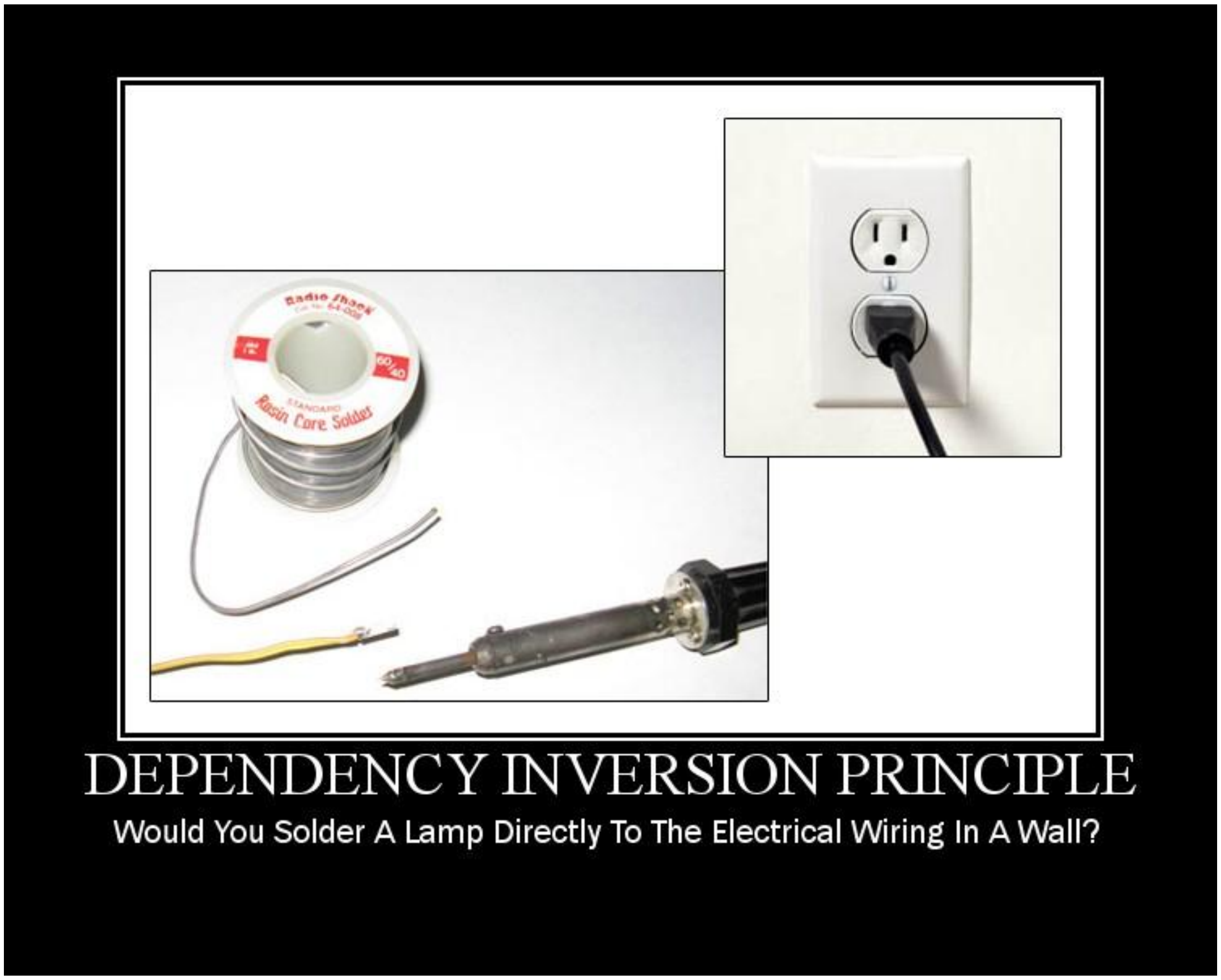
```
    {
        // do something
    }

    public void Groom()
    {
        // do something
    }
}

public class Rattlesnake : Animal
{
    public override void Feed()
    {
        // do something
    }
}
```

I think the key is if you find yourself creating interfaces that don't get fully implemented in its clients, then that's a good sign that you're violating the ISP. You can check out the link to this pdf for more complete information on the subject.

DIP: Dependency Inversion Principle

A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS

B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS
http://www.objectmentor.com/resources/articles/dip.pdf


Radio Shack
Rosin Core Solder
60/40
DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle by Jimmy Bogard

The Dependency Inversion Principle, the last of the Uncle Bob "SOLID" object-oriented design principles, can be thought of the natural progression of the Liskov Substitution Principle, the Open Closed Principle and even the Single Responsibility Principle. This post is the latest in the set of SOLID posts:

- PTOM: The Single Responsibility Principle
- PTOM: The Open Closed Principle
- PTOM: The Liskov Substitution Principle
- PTOM: The Interface Segregation Principle

The Dependency Inversion Principle, or DIP, is often used interchangeably with Dependency Injection and Inversion of Control. However, following DIP does not mean we must automatically use an IoC container like Spring.NET, Windsor or StructureMap. IoC containers are tools to assist in applications adhering to DIP, but we can follow DIP without using IoC containers.

The Dependency Inversion Principle states:

- **High level modules should not depend upon low level modules. Both should depend upon abstractions.**
- **Abstractions should not depend upon details. Details should depend upon abstractions.**

The DIP can be a little vague, as it talks about "abstractions" but doesn't describe *what* is being abstracted. It speaks of "modules", which don't have much meaning in .NET unless you consider "modules" to be assemblies. If you're looking at Domain-Driven Design, modules mean something else entirely.

The Dependency Inversion Principle along with the other SOLID principles, are meant to alleviate the problems of bad designs. The typical software I run into in existing projects has code organized into classes, but it still isn't easy to change. I usually see big balls of mud along with a crazy spider web of dependencies where you can't sneeze without breaking code on the other side of the planet.

Spider webs and bad design

Bad designs and poor code is not good because it's hard to change. Bad designs are:

- Rigid (change affects too many parts of the system)
- Fragile (every change breaks something unexpected)
- Immobile (impossible to reuse)

Some people's ideas of "bad design" would be something like seeing string concatenation instead of a StringBuilder. While this may not be the best performing choice, string concatenation isn't necessarily a bad design.

It's pretty easy to spot bad designs. These are sections of code or entire applications that you dread touching. A typical example of rigid, fragile and immobile (bad) code would be:

```
public class OrderProcessor
{
  public decimal CalculateTotal(Order order)
  {
    decimal itemTotal = order.GetItemTotal();
    decimal discountAmount = DiscountCalculator.CalculateDiscount(order);

    decimal taxAmount = 0.0M;

    if (order.Country == "US")
      taxAmount = FindTaxAmount(order);
    else if (order.Country == "UK")
      taxAmount = FindVatAmount(order);

    decimal total = itemTotal - discountAmount + taxAmount;

    return total;
  }

  private decimal FindVatAmount(Order order)
  {
    // find the UK value added tax somehow
    return 10.0M;
  }

  private decimal FindTaxAmount(Order order)
  {
    // find the US tax somehow
    return 10.0M;
  }
}
```

The OrderProcessor sets out to do something very simple: calculate the total of an Order. To do so, it needs to know the item total of the order, any discounts applied, as well as the tax amount (which depends on the Order's Country).

Too many responsibilities

To see why the DIP goes hand-in-hand with the Single Responsibility Principle, let's list out the responsibilities of the OrderProcessor:

- Knowing how to calculate the item total
- Finding the discount calculator and finding the discount
- Knowing what country codes mean
- Finding the correct taxing method for each country code
- Knowing how to calculate tax for each country (commented out for brevity's sake)
- Knowing how to combine all of the results into the correct final total

If a single class (or a single method in this case) answers too many questions (how, where, what, why etc.), it's a good indication that this class has too many responsibilities.

To move towards a good design, we need to remove the external dependencies of the class and pare it down to its core responsibility, i.e. finding the order total. Offhand, the dependencies I see are:

- DiscountCalculator
- Tax decisions

In the future, we might need to support more countries, which mean more tax services, and more responsibilities. To reduce the rigidity, fragility and immobility of this design, we need to move these dependencies outside of this class.

Towards a better design

When following the DIP, you notice that the Strategy pattern begins to show up in a lot of your designs. Strategy tends to solve the "details should depend on abstractions" part of the DIP. Factoring out the DiscountCalculator and the tax decisions, we wind up with two new interfaces:

- IDiscountCalculator
- ITaxStrategy

I'm not a huge fan of the "ITaxStrategy" name, but it will suffice until we find a better name from our model.

Factoring out the dependencies

To factor out the dependencies, first I'll create a couple of interfaces that match the existing method signatures:

```
public interface IDiscountCalculator
{
   decimal CalculateDiscount(Order order);
}

public interface ITaxStrategy
{
   decimal FindTaxAmount(Order order);
}
```

Now that I have a couple of interfaces defined, I can modify the OrderProcessor to use these interfaces instead:

```
public class OrderProcessor
{
   private readonly IDiscountCalculator _discountCalculator;
   private readonly ITaxStrategy _taxStrategy;

   public OrderProcessor(IDiscountCalculator discountCalculator,
                   ITaxStrategy taxStrategy)
   {
      _taxStrategy = taxStrategy;
      _discountCalculator = discountCalculator;
   }

   public decimal CalculateTotal(Order order)
   {
      decimal itemTotal = order.GetItemTotal();
      decimal discountAmount = _discountCalculator.CalculateDiscount(order);

      decimal taxAmount = _taxStrategy.FindTaxAmount(order);

      decimal total = itemTotal - discountAmount + taxAmount;

      return total;
   }
}
```

The CalculateTotal method looks much cleaner now, delegating the details of discounts and tax to the appropriate abstractions. Instead of the OrderProcessor depending directly on details, it depends solely on the abstracted interfaces we created earlier. The specifics of how to find the correct tax method is now encapsulated from the OrderProcessor, as is the hard dependency on a static method in the DiscountCalculator.

Filling out the implementations

Now that we have the interfaces defined, we need actual implementations for these dependencies. Looking at the DiscountCalculator, which is a static class, I find that I can't immediately change it to a non-static class. There are many other places with references to this DiscountCalculator, and since it's the real world, none of these other places have tests.

Instead, I can just use the Adapter pattern to adapt the interface I need for an IDiscountCalculator:

```
public class DiscountCalculatorAdapter : IDiscountCalculator
{
  public decimal CalculateDiscount(Order order)
  {
    return DiscountCalculator.CalculateDiscount(order);
  }
}
```

In applying the Adapter pattern, I just wrap the real DiscountCalculator in a different class. In this case, the advantage of the Adapter pattern is the existing DiscountCalculator can continue to exist, when the mechanism for calculating discounts changes, my OrderProcessor does not need to change.

For the tax strategies, I can create two implementations for each kind of tax calculation being used today:

```
public class USTaxStrategy : ITaxStrategy
{
  public decimal FindTaxAmount(Order order)
  {
  }
}

public class UKTaxStrategy : ITaxStrategy
{
  public decimal FindTaxAmount(Order order)
  {
  }
}
```

I left the implementations out, but I basically moved the methods from the OrderProcessor into these new classes. Neither of the original methods used any instance fields, so I could copy them straight over.

My OrderProcessor now has dependencies factored out, so its single responsibility is easily discerned from looking at the code. Additionally, the implementations of IDiscountCalculator and ITaxStrategy can change without affecting the OrderProcessor.

Isolating the ugly stuff

For me, the DIP is all about isolating the ugly stuff. For calculating order totals, I shouldn't be concerned about where the discounts are or how to decide what tax strategy should be used. We did increase the number of classes significantly, but this is what happens when we move away from a procedural mindset to a true object-oriented design.

I still have the complexity to solve of pushing the dependencies into the OrderProcessor. Clients of the OrderProcessor now have the burden of creating the correct dependencies and giving them to OrderProcessor. That problem is already solved with Inversion of Control (IoC) containers like Spring.NET, Windsor, StructureMap, Unity and others.

These IoC containers let me configure the "what" when injecting dependencies, so even that decision is removed from the client. If I didn't want to go with an IoC container, even a simple creation method or factory class could abstract the construction of the OrderProcessor with the correct dependencies.

By adhering to the Dependency Inversion Principle, I can create designs that are clean, with clearly defined responsibilities. With the dependencies extracted out, the implementation details of each dependency can change without affecting the original class.

That's my ultimate goal: code that is easy to change. Easier to change means a lower total cost of ownership and higher maintainability. Since we know that requirements will eventually change, it's in our best interest to promote a design that facilitates change through the Dependency Inversion Principle.

The Dependency Inversion Principle by Gabriel Schenker

In this post I want to discuss the ***D*** of the **S.O.L.I.D.** principles and patterns. The principles and patterns subsumed in S.O.L.I.D. can be seen as the cornerstones of "good" application design. In this context, ***D*** is the place holder for the *dependency inversion* principle. In a previous post, I discussed the ***S***, which is the placeholder for the *single responsibility* principle.

What is Bad Design?

Let's first discuss the meaning of bad design. Is bad design when somebody claims:

That's not the way I would have done it...

Well, sorry, but this is not a valid measure for the quality of the design! This statement is purely based on personal preferences. So let's find other, better criteria to define bad design. If a system exhibits any or all of the following three traits, then we have identified bad design

- the system is **rigid**: it's hard to change a part of the system without affecting too many other parts of the system
- the system is **fragile**: when making a change, unexpected parts of the system break
- the system or component is **immobile**: it is hard to reuse it in another application because it cannot be disentangled from the current application

An immobile design

Let's have a look at the latter of the traits of bad design mentioned above. A design is *immobile* when the desirable parts of the design are highly dependent upon other details that are not desired.

Imagine a sample where we have developed a class which contains a highly sophisticated encryption algorithm. This class takes a source file name and a target file name as inputs. The content to encrypt is then read from the source file and the encrypted content is written to the target file.

```
public class EncryptionService
{
  public void Encrypt(string sourceFileName, string targetFileName)
  {
    // Read content
    byte[] content;
    using(var fs = new FileStream(sourceFileName, FileMode.Open, FileAccess.Read))
    {
      content = new byte[fs.Length];
      fs.Read(content, 0, content.Length);
    }

    // encrypt
    byte[] encryptedContent = DoEncryption(content);

    // write encrypted content
    using(var fs = new FileStream(targetFileName, FileMode.CreateNew, FileAccess.ReadWrite))
    {
      fs.Write(encryptedContent, 0, encryptedContent.Length);
    }
  }

  private byte[] DoEncryption(byte[] content)
  {
    byte[] encryptedContent = null;
    // put here your encryption algorithm...
    return encryptedContent;
  }
}
```

Listing 1: Encryption Service depending on Details

The problem with the above class is that it is highly coupled to a certain input and output. In this case input and output are both files. You might have invested quite some time and resources to develop the encryption algorithm which is the core responsibility of this service. It's a shame that this encryption algorithm cannot be used in another context. The content to be encrypted might not be present in a file but rather in a database and the encrypted content might not be written to a file but sent to a web service.

Certainly we could make the above service more flexible and change its implementation. We can put in place a switch for the content source type and one for the destination type of the encrypted content.

```
public enum ContentSource { File, Database }
public enum ContentTarget { File, WebService }

public class EncryptionService_2
{
  public void Encrypt(ContentSource source, ContentTarget target)
  {
    // Read content
    byte[] content;
    switch (source)
    {
      case ContentSource.File:     content = GetFromFile(); break;
      case ContentSource.Database: content = GetFromDatabase(); break;
    }

    // encrypt
```

```
        byte[] encryptedContent = DoEncryption(content);

        // write encrypted content
        switch (target)
        {
            case ContentTarget.File:      WriteToFile(encryptedContent); break;
            case ContentTarget.WebService: WriteToWebService(encryptedContent); break;
        }
    }

    // rest of code omitted for brevity
}
```

Listing 2: Slightly improved Encryption Service

However this adds new interdependencies to the system. As time goes on, and more and more source and/or destination types must participate in the encryption program, the "Encrypt" method will be littered with switch/case statements and will be dependent upon many lower level modules. It will eventually become rigid and fragile.

Here comes the *dependency inversion principle* to the rescue.

The Dependency Inversion Principle

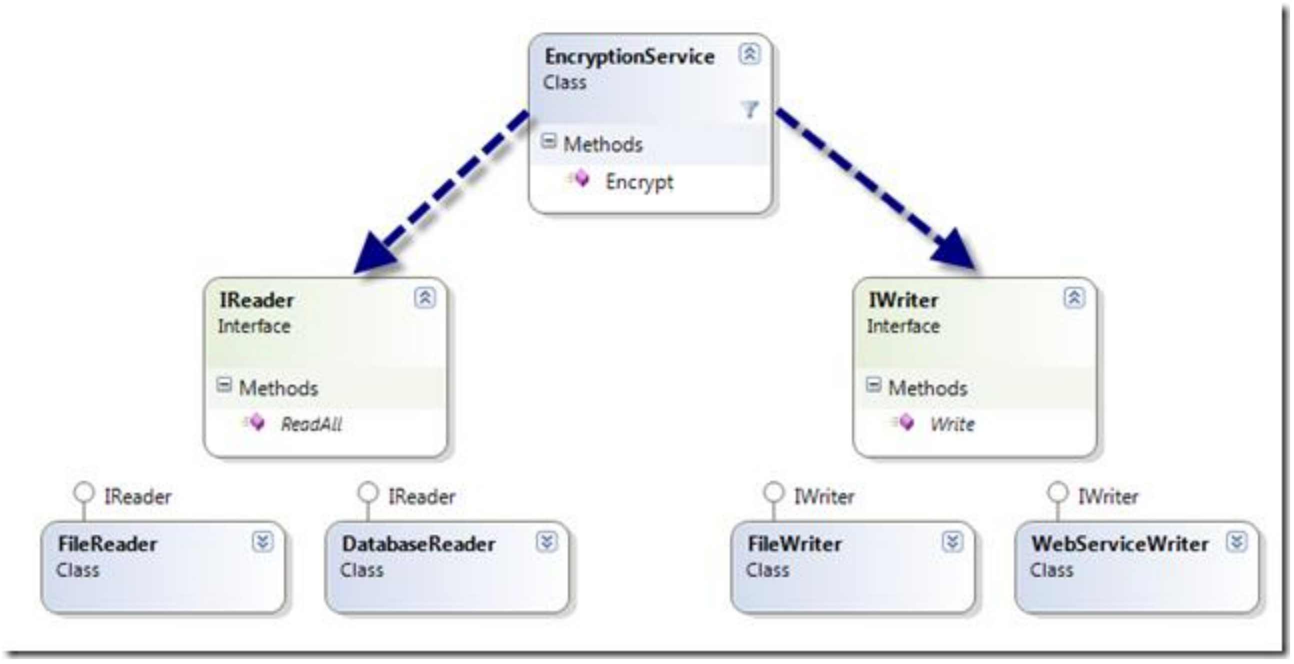
Theory: the *dependency inversion* principle states:

a) High level modules should not depend upon low level modules. Both should depend upon abstractions

b) Abstractions should not depend upon details. Details should depend upon abstractions

One way to characterize the problem above is to notice that the method containing the high level policy, i.e. the *Encrypt* method, is dependent upon the low level detailed method that it controls, i.e. *GetFromFile* and *WriteToWebService*. If we could find a way to make the *Encrypt* method independent of the details that it controls, then we could reuse it freely. We could produce other applications which used this service to encrypt content originating from any content source to any destination.

Consider the simple class diagram below.



Here we have an *EncryptionService* class which uses an abstract "Reader" class, identified by an interface *IReader* and an abstract "Writer" class, identified by an interface *IWriter*. Note that the abstraction in this case is not achieved through inheritance but through the use of interfaces. We have separated the interface from the implementation.

The Encrypt method uses the "Reader" to get the content and sends the encrypted content to the "Writer".

```
public class EncryptionService
{
  public void Encrypt(IReader reader, IWriter writer)
  {
    // Read content
    byte[] content = reader.ReadAll();

    // encrypt
    byte[] encryptedContent = DoEncryption(content);

    // write encrypted content
    writer.Write(encryptedContent);
  }

  // rest of code omitted for brevity...
}
```

Listing 3: Encryption Service only depends on Abstractions

The *Encrypt* method of the encryption service is now independent of a specific content reader or writer. The dependencies have been **inverted**; the *EncryptionService* class depends upon abstractions, and the detailed readers and writers depend upon the same abstractions.

The definition of the two interfaces used is:

```
public interface IReader
{
   byte[] ReadAll();
}

public interface IWriter
{
   void Write(byte[] content);
}
```

Listing 4: Reader and Writer Interfaces

Now we can reuse the encryption service. We can invent new kinds of "Reader" and "Writer" implementations that we can supply to the *Encrypt* method of the service. Moreover, no matter how many kinds of "Readers" and "Writers" are created, the encryption service will depend upon none of them. There will be no interdependencies to make the application fragile or rigid. And the encryption service can be used in many different contexts. The service is mobile.

Why call it dependency inversion?

The dependency structure of a well designed object oriented application is "inverted" with respect to the dependency structure that normally results from a "traditional" application which is implemented in a more procedural style. In a procedural application high level modules depend upon low level modules and abstractions depend upon details (as in listing 1 and 2).

Consider the implications of high level modules that depend upon low level modules. It is the high level modules that contain the important policy decisions and business models of an application. It is these models that contain the identity of the application. Yet, when these modules depend upon the lower level modules, changes to the lower level modules can have direct effects upon them; and can force them to change.

This predicament is absurd! It is the high level modules that ought to be forcing the low level modules to change. The high level modules should take precedence over the lower level modules. High level modules simply should not depend upon low level modules in any way.

Moreover, it is high level modules that we want to be able to reuse. When high level modules depend upon low level modules, it becomes very difficult to reuse those high level modules in different contexts. However, when the high level modules are independent of the low level modules, then the high level modules can be reused quite simply.

Summary

When implementing an application the modules and components of a higher abstraction level should never directly depend upon the (implementation) details of modules or components of a lower abstraction level. It's the high level components that make an application unique. It's the high level modules that contain most of the business value. Thus, the high level components should dictate whether low level components have to change or not and not vice versa.

When a component does not depend on lower level components directly but only through abstractions this component is *mobile* - that is, the component is reusable in many different contexts.

Furthermore, when the design is respecting the *dependency inversion principle* an application is less brittle or fragile. If some changes have to be made to a low level module there are no side effects that manifest themselves in other (possibly unexpected) parts of the system.

Topics Related to S.O.L.I.D.

The core understanding of the SOLID principles quickly lead us into many other discussions on how we can and should apply these principles in the real world. There is also a wealth of knowledge in other patterns and principles, which SOLID code enhances. The end goal of SOLID, after all, is not to be a strict, rigid set of rules that must be applied at all cost. Rather, it is intended to be a foundational set of principles upon which other principles and practices can be built. With this in mind, we have identified a few items that directly relate to our SOLID implementations.

Single-Responsibility Versus Needless Complexity by Ray Houston

At Pablo's Day of TDD, we were discussing the Single-Responsibility Principle (SRP) while working through one of the labs and a question came up about a piece of code. The code in question looked something like the following (warning: this is over simplified code to show a point):

```
public bool Login(string username, string password)
{
   var user = userRepo.GetUserByUsername(username);

   if(user == null)
      return false;

   if (loginValidator.IsValid(user, password))
      return true;

   user.FailedLoginAttempts++;

   if (user.FailedLoginAttempts >= 3)
      user.LockedOut = true;

   return false;
}
```

This was from the LoginService class and this was its only method. The question was whether or not this violates SRP. It appears to have multiple responsibilities, it is in charge of incrementing FailLoginAttempts as well as locking the user out after 3 failed attempts. I believe we answered the question with a "depends", but it bothered me that we didn't have a better answer. Personally, I wouldn't have busted this up into another class, but I didn't have a good argument to stand on.

Today I went searching through Agile Principle, Patterns, and Practices in C# looking for a better answer. In the chapter on SRP, the book gives an example of an interface of a modem that can Dial/Hang-up and Send/Receive. The former is connection management and the later is data communication. The book asks the question as to whether or not these responsibilities should be separated. The answer is

"That depends on how the application is changing."

It then gives an example to how a change might violate SRP, but then states:

"If, on the other hand, the application is not changing in ways that cause the two responsibilities to change at different times, there is no need to separate them. Indeed, separating them would smell of needless complexity."

Ah, there's the backup wisdom that I needed to validate my gut feeling. Here's one final quote from the book:

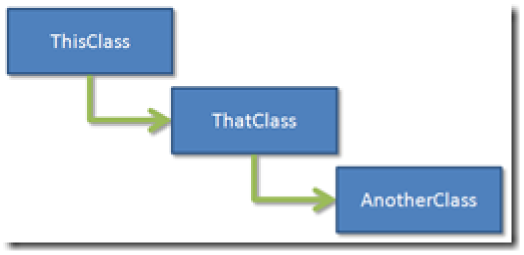
"There is a corollary here. An axis of change is an axis of change only if the changes occur. It is not wise to apply SRP (or any other principle, for that matter) if there is no symptom."

I think applying SRP is about using good judgment. You certainly don't want to wait until you have to make a change before you think about SRP. You also don't want to over do it either and end up with classes with one method, each having only a couple lines of code.

DIP: Creating and Working With A Cloud of Objects by Derick Bailey

A few months ago, I posted some thoughts and questions on the proper use of Inversion of Control (IoC) containers and the Dependency Inversion (DI) principle. Since then, I've had the opportunity to do some additional study and teaching of DI. I've had that light bulb moment for the proper use of an IoC container. I haven't talked about it or tried to present the info to my team(s) yet because I had not verified my thoughts were on the right track, until recently. I got to spend a few hours at the Dovetail office with Chad, Jeremy, Josh, etc, and had the pleasure of being able to pick their brains on some of the questions and thoughts that I've had concerning DI and IoC. In the end, Chad confirmed some of my current thoughts and helped me put them into a metaphor that I find to be very useful in understanding what Dependency Inversion really is. It is a cloud objects that can be strung together into a necessary hierarchy, at runtime.

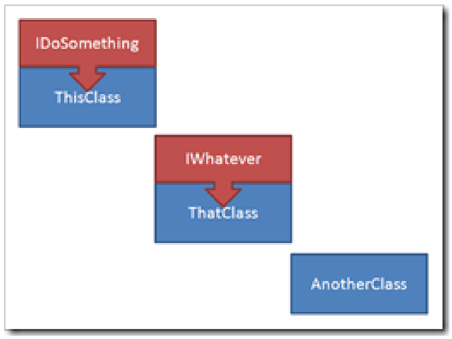
Consider a set of classes that need to be instantiated into the correct hierarchy so that we can get the functionality needed. It's really easy to have the highest level class, the one that we really want to call method on, instantiate the class for next level down. Then have that class instantiate it's next level down, and so-on. For example:



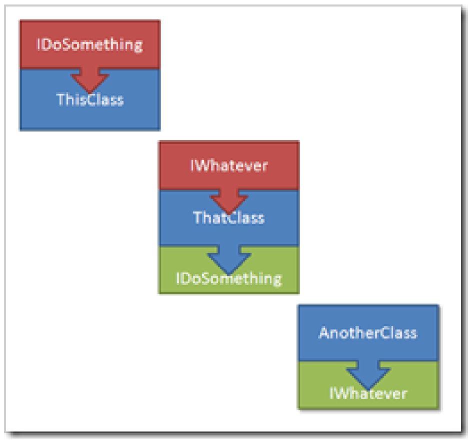
This creates the necessary hierarchy, but breaks the core object oriented principle of loose coupling. We would not be able to use ThisClass without bringing ThatClass along, and we would not be able to use ThatClass without bringing AnotherClass along.

By introducing a better abstraction for each class and putting Dependency Inversion into play, we can break the spaghetti mess apart. We then introduce the ability to use any of these individual classes without requiring the specific implementation of the dependent class.

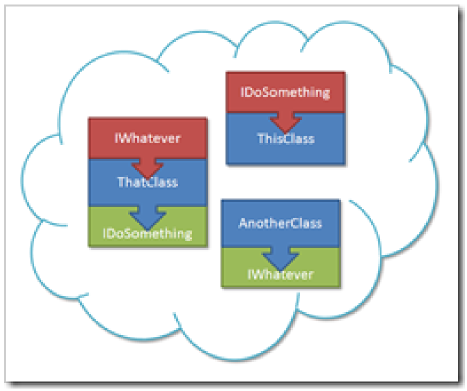
For starters, let's introduce an interface for ThisClass to depend on and an interface for ThatClass to depend on.



Now that that both of these classes can depend on an interface, instead of the explicit implementation of the child object, we need to have the expected child implement the interface in question. For example, we expect ThatClass to be used by ThisClass, so we will want ThatClass to implement the IDoSomething interface. By the same notion, we want AnotherClass to implement the IWhatever interface. This will allow us to provide AnotherClass as the dependency implementation for ThatClass. Our object model now looks like this:



What we have now is not just a set of classes that all depend on each other, but a "cloud" of objects with dependencies and interface implementations that will let us build the hierarchy we need, when we need it.

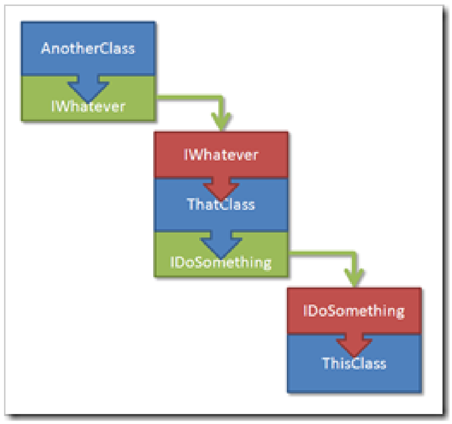


The real beauty of this is we no longer care about the implementation specifics of IDoSomething from ThisClass. ThisClass can focus entirely on doing its duty and calling into IDoSomething when needed. By passing in the dependency as an abstraction, we're able to replace the dependency implementation at any time; runtime, unit test time, etc. This also makes our system much easier to learn, understand and most importantly, easier to change.

Now that we have our cloud of implementations and abstractions in place, we can reconstruct the hierarchy in order to call into ThisClass and have it perform its operations. Here's where Dependency Inversion meets up with Inversion of Control.

- To create ThisClass, we need an implementation of IDoSomething
- ThatClass implements IDoSomething, so we'll instantiate it before ThisClass
- ThatClass needs an instance of IWhatever
- AnotherClass implements IWhatever, so we'll instantiate it before ThatClass
- Once we have AnotherClass instantiated, we can pass it into ThatClass's constructor
- Once we have ThatClass instantiated, we can pass it into ThisClass's constructor

We end up with a hierarchy of objects that is instantiated in reverse order, like this:



We have now successfully inverted our system's construction; each implementation detail is created and passed into the object that depends on it, re-creating our hierarchy from the bottom up. In the end, we have an instance of ThisClass that we can call into with the same basic hierarchy of classes that we started with. The real difference is we can change this hierarchy at any time without worrying about breaking the functionality of the system.

Once we have our Dependency Inversion and Inversion of Control in place, we can start utilizing the existing IoC frameworks to automatically create our hierarchy of objects based on the advertised dependencies (an advertised dependency is a dependency that is specified as a constructor parameter of a class). Tools like StructureMap, Spring.net, Windsor, Ninject and others, all provide auto-magic ways of creating each dependency of the object that is requested all the way up/down the hierarchy. Utilizing one of these IoC containers can greatly simplify our code base and eliminate the many object instantiations that would start to litter our code. As I said in my previous post, I know all about what not to do with IoC containers. Good IoC usage, though, is another subject for another post.

Conclusions and Benefits of S.O.L.I.D.

Low Coupling

By abstracting many of our implementation needs into various interfaces and introducing the concepts on OCP and DIP, we've created a system that has very low coupling. Many of these individual pieces can be taken out of the system with little to no spaghetti mess trailing after it. Separating the various concerns into the various object implementations has helped us ensure that we can change the system's behavior as needed, with little modification to the overall system, just update the one piece that contains the behavior in question.

High Cohesion

This really is a direct result of low coupling and SRP; we have a lot of small pieces that can be stacked together like building blocks to create something larger and more complex. Any of these individual pieces may not represent much functionality or behavior, but then, an individual piece isn't much fun to use without a bunch of other pieces. DIP has also allowed us to tie the various blocks together by depending on an abstraction and allowing that abstraction to be fulfilled with different implementations. This creates a system that is much greater than the mere sum of its parts.

Encapsulation

True encapsulation is not just making fields private and hiding data from external objects, it's hiding implementation details from other objects, depending only on the abstractions and expected behaviors of those abstractions. LSP, DI, and SRP all work hand in hand to create true encapsulation in the new project structure. We've encapsulated our behavioral implementations in many individual objects, preventing them from leaking into each other while ensuring that the dependency on those behaviors is encapsulated behind a known interface. We've hidden the implementation details and allowed for any implementation to be put in place for that interface definition through DIP. At the same time, we've done the necessary due-diligence to ensure that we are not violating any of the individual abstraction's semantics or purpose (LSP), ensuring that we can properly replace the implementation as needed.

Copyright and Contact Information

For more information on SOLID software development and other best practices of software development, be sure to check out the LosTechies website: http://www.lostechies.com

You can also contact us directly through the website and through the individual blogs and profiles on the site. If you haven't already, please consider subscribing to the Los Techies Main Feed so that you can see the various post from the other Los Techies bloggers.

The main feed is here: http://feeds.feedburner.com/lostechies