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

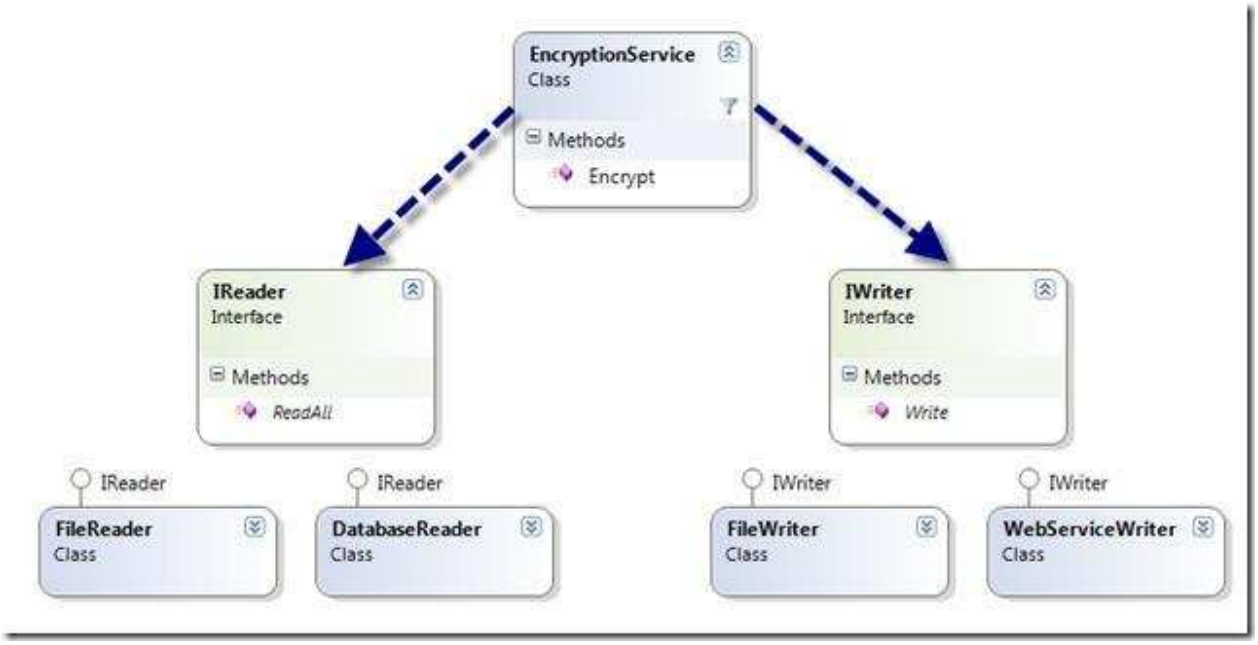
Theory: the *dependency inversion* principle states:

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.