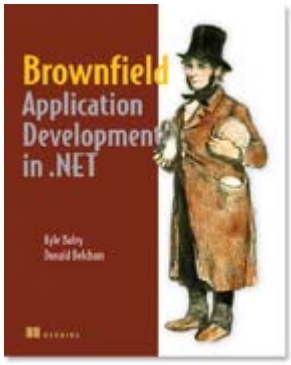
MANNING PUBLICATIONS

Separation of Concerns/Single Responsibility Principle

Excerpted from



[Brownfield Application Development in .NET]

EARLY ACCESS EDITION

Kyle Baley and Donald Belcham
MEAP Release: March 2008
Softbound print: June 2009 (est.) | 550 pages
ISBN: 1933988711

This article is taken from the book Brownfield Application Development in .NET from Manning Publications. As part of a chapter on bringing better object-oriented principles into your Brownfield project, this segment discusses the benefits (and limitations) of the Single Responsibility Principle, which makes classes easier to maintain by delegating responsibility. For the table of contents, the Author Forum, and other resources, go to http://manning.com/baley/.

If you remember nothing else from this article, you should remember the Single Responsibility Principle (SRP) and the notion of Separation of Concerns (SoC). SoC is the process by which we break down the functionality of an application into distinct modules that do not overlap. It is a key concept that we will revisit often.

In and of itself, SoC is somewhat abstract. There is no guidance on *how* to separate the concerns or even how to define the concerns. A concern is simply defined as an area of interest in the application. And by and large, your definition of a concern may not match someone else's.

This is where the Single Responsibility Principle can help. It is a more concrete concept. It states:

> "A class should have only one reason to change."
>
> Robert C. Martin & Micah Martin, *Agile Principles, Patterns, and Practices*

The key to this definition is "reason to change." Often this can be hard to pin down. For example, one could argue that a class called `PrintFunctions` needs updating only when the application's printing behavior changes, and therefore, it does not violate SRP. Conversely, you could say that the class is handling the printing of Jobs AND the printer configuration screen. Thus, it *does* violate SRP.

By and large, the scope of your application will define what a "reason to change" is. It could be that the `PrintFunctions` class is a façade over a third-party printing subsystem, in which case, the only thing wrong with it is the misleading name. Or it could contain half a dozen disparate print-related functions, none of which depend on any others, in which case, consider breaking the class up. For example, perhaps you might separate it into a `PrinterConfiguration` class, a Printer base class, and a couple of entity-specific subclasses (i.e. `JobPrinter`, `CustomerPrinter`, etc.).

Listing 1 shows a class that probably does too much.

Listing 1: Class with too many responsibility

```
public class PrintFunctions
{
    public void ResetPrinter( )
    {
        // ... code to reset printer
    }

    public void SendPrintCompleteEmail( PrintJob printJob )
    {
        // ... code to send e-mail to the person who printed
    }

    public void PrintToFile( Document documentToPrint, string filename )
    {
        // ... code to print document to file
    }

    public void PrintToPrinter( Document documentToPrint, Printer printer )
    {
        // ... code to print document on printer
    }

    public void SetPrinterConfiguration( Printer printer,
            PageSize pageSize )
    {
        // ...
    }

    public void SetPrinterConfiguration( Printer printer,
            PageSize, ColourScale colourScale )
    {
        // ...
    }

    public void SetPrinterConfiguration( Printer printer,
            ColourScale colourScale )
    {
        // ...
    }

}
```

This code has a lot of responsibilities. It is resetting printers, printing to printers, printing to files, and making configuration settings.

Listing 2 shows the same class with the responsibilities delegated

Listing 2: PrintManager with one responsibility

```
public class PrintManager
{
    private IPrintNotifier _printNotifier = new EmailNotifier( );
    private IPrintConfiguration _configuration =
            new DefaultPrintConfiguration( );
    private IPrintDestination _destination =
            new PrintFileDestination( DEFAULT_PRINT_FILE );
    private IDocumentPrinter _printer = new Default( );

    public IPrintNotifier PrintNotifier { get; set; }
    public IPrintConfiguration PrintConfiguration { get; set; }
    public IPrintDestination Destination { get; set; }
    public IDocumentPrinter Printer { get; set; }

    public void Print( PrintJob printJob )
    {
        _printer.Print( printJob.Document );
        _printNotifier.Notify( printJob.Sender );
    }
}
```

You'll notice that this is quite a bit leaner than the previous version. That's because it doesn't deal with extraneous functions like e-mailing the sender or setting the configuration. All of that has been moved out to other classes like `DefaultPrintConfiguration` and `EmailNotifier`.

By default, this class will print to a file but the client can override it with another destination if it wants. Ditto for the print configuration. This class is concerned only with the actual print action (and even then, that is delegated off to an `IDocumentPrinter` object).

It should be obvious why this class is easier to maintain than the previous version. It doesn't do very much. Its sole responsibility is to print documents. If we change how printers are configured, we don't need to touch this class. Instead we would modify the object passed to the PrintConfiguration property and implementing IPrintConfiguration. If we add a new method to notify senders that a print job was complete, again, we don't need to change this class. That change would be implemented in the object provided to the PrintNotifier property. This class changes only if the printing process changes.

Note, however, that the class is still very highly coupled. It contains a lot of default objects. For instance, it will notify senders by e-mail if a print job is complete. This can be overridden and indeed, it would be a good practice always to do this so that we don't rely on the default behavior. But if someone is relying on the fact that e-mails are sent out and we decide to change to SMS notifications, we will have some broken code.

In short, this code is still rather dependent on a lot of other objects, so we'll have to learn how to break those dependencies.

SRP and Brownfield applications

In our opinion, moving your application to be more SRP-oriented is the best road to making it more maintainable. With each class responsible for one and only one piece of functionality, the classes tend to be much smaller and more manageable. Granted, you will have many more classes than you may be used to, but this can be mitigated through careful organization of your solution structure and by taking advantage of code navigation abilities in Visual Studio, or a third-party add-in such as ReSharper or Code Rush.

In any case, the benefits, a less coupled, more cohesive application, far outweigh any trauma you may feel at having "too many files." In addition, you'll usually find that code that adheres to SRP is easier to maintain because the classes are easier to change. By definition, classes with a single responsibility have only one reason to change so when that reason arises, it's very easy to determine where the change needs to be made.

This also helps your code's "reversibility." That is, any changes you make can easily be backed out. This may not sound like

Unfortunately, Brownfield applications are often notoriously coupled and when you are staring a 6000-line method in the face, single responsibility often gets relegated to the status of "would be nice but I have *real* work to do."

In addition, we recommend not writing new code without backing tests. This can seem like a lofty goal if you are writing a test to expose a bug in a method that spans four printed pages. It is often difficult to implement tests on code without having to refactor first. In his book, *Working Effectively with Legacy Code*, Michael Feathers calls this the Legacy Code Dilemma:

> "When we change code, we should have tests in place. To put tests in place, we often have to change code."

This most often manifests itself when you want to break a piece of code up into separates pieces of responsibility. That is, we are separating out dependencies into individual classes.

The reality is that unit tests may not be possible. And when that is the case, you need to tread lightly when you need to modify the code. If you can't write a unit test (and you should be *very* sure that you can't), chances are it is because the code has a lot of external dependencies that need to be set up. In this case, an integration test may be possible.

If an integration test is possible, your work is much easier. Now you have a safety net under you for when you perform the real work. You can start breaking out the dependencies into separate classes/responsibilities and know that you are not breaking anything.

In addition, as you separate out responsibilities, you can also write unit tests for them along the way. Even though you have an integration test backing you up, you still want unit tests for individual classes because they are run more regularly. The last thing you need is to have someone break one of your new classes and not have it caught until the integration tests are run overnight.