ASP.NET, C# and Object Orientated Programming Primer

By Brett Hargreaves

Blog: http://bretthargreaves.wordpress.com
Email: bretthargreaves@hotmail.com

Table of Contents

Foreward

I started my career in IT as a hardware engineer, and Software support. My specialism was networked software and servers.

At the time Windows 3.1 had just been released and Microsoft didn't really have a networking product to speak of. If you wanted a low cost but effective small business solution back then you used Novell Netware.

Now, Novell had an accreditation program much likes Microsoft's today. At the age of 18 I started to study towards by CNE (Certified Netware Engineer). This set of exams covered in great depth a lot about networking and how servers and client communicated. At the time Novell used IPX/SPX as a communications protocol rather than TCP/IP – although they did eventually adapt to TCP/IP in later releases.

Even so the courses and books when into great detail about the various protocols including TCP/IP and so I got a good grounding.

On the client side I learnt how to use DOS to optimize PCs as much as possible (at the time 4MB was a lot of RAM. And no, that isn't a typo –I said MB as in Megabytes – even most hard disks didn't get to 1GB!) – So optimization was very important. Because of this I learn exactly how the OS used it's memory, physical and virtual.

The point is, during my early career, through both work related training, and my own desire to understand, I learnt how computers WORKED.

In theory, to do my job well, only I really needed to know was how to install Netware and Windows, install drivers and

create. None of these tasks required the knowledge I had learnt.

As the years went by the company grew, we took on more engineers and software became more varied and complex. And I started to notice something – I was becoming especially good at my job compared to other engineers (at least the newer ones!) – not because I had more experience but because I knew how things worked.

The new wave of engineers had learnt the higher level stuff – usually from books and the odd training course – but they never bothered with what made things tick.

So a problem would come along that they hadn't encountered before. Maybe it was a newer version of the software, maybe they'd discovered a bug, but for whatever reason they were now stumped. I would be asked to help out, I'd spend a it of time understanding the issue, and WHAT they were trying to achieve, and I would usually be able to either fix the issue or implement a workaround.

Fast-forward a few (OK quite a few) years and the story repeats more and more. I continually try to learn HOW things work. For example as an Web/Mobile focused developer I have learning what goes on the background, HOW internet pages are actually processed and server, how data gets transformed and communicated.

I recently overheard a colleague struggling getting some software upgrade to work. It was an WebSphere based web service, and had been deployed by a 3rd Party engineer who had since left site, and of course the upgrade had brought the system down and they couldn't get it back up.

I offered to help.

I have never heard of or seen the software before. I've never used WebSphere before, and yet I was able to get in, identify the issue and fix it. When I told the 3rd Party engineer what I had done he exclaimed – 'Oh – I didn't realize you could do that!'

You see I'd used my knowledge of how servers serve pages of data and how they communicate it out. How clients connect to and consume that data. And this basic understanding of HOW THINGS WORK enabled me to figure out not only where the problem was, but also how to fix it by looking though WebSphere's configuration pages until I found something that referred to what I was looking for.

That's all very nice, but what's the point?

The point is that more and more software developers are falling into the same trap. So many books, courses and tutorials focus so much on how to create a new 'Hello World' project, how to write for loops, if then else blocks, style your css, use correct HTML syntax – they completely forget the basics.

This is a twofold issue. The first is that, as I have pointed out in my nostalgic ramblings, if you don't know the basics, of how things work, it will make fixing issues later on far more difficult.

Second is that, with software development especially, if you don't implement your code in a well-structured format, making changes or correcting code soon becomes increasingly difficult.

A lot of less experienced developers are often put off learning concepts such as abstraction, encapsulation, poly-morphism, interface segregation, dependency inversion etc etc.

But I'm going to show you how simple these things are, and more importantly how they will help you save a lot of time and become a better programmer.

I won't baffle you with concepts you don't need to know, I'll stick to the basics, whilst still giving you a SOLID grounding (you'll learn that's a pun in a few pages!)

I'll show you real world examples of why they work, and conversely why not using them will make your life harder.

I really hope you get some value from my book, and please feel free to get in touch with questions and comments.

Thank you.

Who this book is for.

This book is aimed at novice and junior programmers. Although the concepts used don't necessarily require any previous programming experience, at least knowing some standard syntax would help, as what I WON'T be doing is showing you how to use loops, if then else constructs etc.

Experienced Developers may also get some use out of it – especially if you've never really worked on big teams, or learnt programming concepts before (which is quite a lot – don't be ashamed, it's how I started).

As I am primarily a Microsoft developer, the code samples are all done in C#. However the concepts themselves often still hold for other languages. C# is also very similar to Java and to a certain extent JavaScript.
It's also quite similar to Apple's new language Swift – however iOS apps work a bit differently in that it's more about delegation rather than inheritance.

Object Orientated Programming (OOP)

Introduction

I need to go through some basic theory, and a little bit of history, before we get into the nitty gritty.

This section may lose you a bit depending on how much experience you have – but battle through – the theory taught here will be re-iterated and expanded upon with greater detail as we go through the book.

I promise it will be worth it. So deep breath, here we go!

Why bother at all? (Or a bit of my programming history)

I learnt to program (sort of) on a Zx Spectrum. For young people and Americans – the Sinclair Zx Spectrum was one of the first home computers. It had a whopping 48Kilobytes of RAM. Disks were unheard of, there was no monitor – it was plugged up to my Black and White portable (allegedly) TV, and you had to save to tape…. which never actually worked properly.

The Spectrum used BASIC. And the first program I wrote

```
10 PRINT "BRETT"
20 GOTO 10
```

was this;

Voila, the screen fills with my name repeated ad-infinitum.

More complex programs, for example I wrote a tic-tac-toe game, worked in the same way. Hundreds and hundreds of rows of code. Jumping from one area of code to another mean trying to remember where in the hundreds of lines was the bit you want.

Then I went to college, and we learnt PASCAL.

PASCAL was a procedural language.

PASCAL is built from (you guessed it) procedures, and you have a main block of code that was essentially the startup block, and it called the various procedures depending on what keys the user pressed. E.g.

```
program Mine(output);

var i : integer;

procedure Print(var j : integer);
begin
...
end;

begin
...
Print(i);
end.
```

Next I started to learn Microsoft Visual Basic. This was a game changer. Rather than needing to learn C++ to build proper GUI's in Windows 3.1, you could just DRAW your GUI. Also, instead of having a 'main' program block, you just had the App – and then you hooked up your code to EVENTS – for example, you would click the 'Click Me' button and it would create a button1_clicked event.

You then just put your code in there. In many ways this was my first introduction to OOP, even though at the time I didn't actually know it.

However, the main thing is that we went a way from the spaghetti code of Basic, and even Pascal, and into a world where code was simpler to maintain, easier to update, and far far easier to debug!

So what is Object Orientated Programming?

Object orientated programming is about breaking down your code into objects. An object usually represents a thing that has some properties and maybe some methods.

In Microsoft.NET an object is synonymous with a Class – in other words to create an object you would create a new Class. Because of this class and object are often used interchangeably within the context of Microsoft.NET and Visual Studio.

Properties are bits of data – e,g. a persons name, a products Sale amount.

Methods are actions that may affect the data of that data in someway, for example an account action may have a Payment method that decreases the Balance property.

The idea is that everything in a piece of software is essentially a collection of 'things' that all interact with each other.

Sometimes these 'things' are representations of real word objects, for example a CUSTOMER, a PRODUCT or an ACCOUNT. In your program a CUSTOMER buys a PRODUCT,

which cause their ACCOUNT to change (e.g. the outstanding balance increases.

Some times these 'things' are more abstract representations of functionality. For example you might have a Stock Manager object that is responsible for acting on various other objects.

As basic class/object would look something like this;

```
public class MyClass
{
        string someInternalValue;

        public string SomeProperty { get; set; }

        public string SomeMethod (string possibleInputVar)
        {
                return "Some Value";
        }
}
```

First of all in Visual Studio when you add a new class it creates a file of that class name – e.g. in this case it would be MyClass.cs (cs stands for C Sharp). That file can actually contain multiple other classes – but generally you'd have one file per class. But this is more convention rather than a requirement.

As you can see we start by declaring 'public class' – public means the class is accessible from other classes. You could just have 'class MyClass' – but then it would only be accessible from another class WITHIN that particular file.

The public (or not) is called an access modifier. It is often overlooked but very important. The access modifier determines from WHERE you can access your.

The options are:

public
The type or member can be accessed by any other code in the same assembly or another assembly that references it.

private
The type or member can be accessed only by code in the same class or struct.

protected
The type or member can be accessed only by code in the same class or struct, or in a class that is derived from that class.

internal
The type or member can be accessed by any code in the same assembly, but not from another assembly.

protected internal
The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly. Access from another assembly must take place within a class declaration that derives from the class in which the protected internal element is declared, and it must take place through an instance of the derived class type.

Within the class we have an internal variable called 'someInternalValue'. This is a variable that is only used and accessible from within the class. This is because in order to get at values in a class we need getter and setters.

These are declared in the next few lines.

First we have a property called SomeProperty – we know it's a property because we have the { get; set; }

These basically tell the class that we can GET the value of the property and SET it.

For most properties this is all you want. But sometimes you may want to GET a calculated property or override what happens when you set SET the property.

Therefore you can place code within the getter and setters like this:

```
…
public SomeProperty
{
        get{  someInternalVar = value; }
        set{ return someInternalValue; }
}
```

Note when SETTING the value we can use 'value' to infer the data that was passed in.

Finally we have a method – Methods are declared as the return type (String, Int, Class) followed by the method name and finally parenthesis () - which can be empty or contain a variable to be passed in. e.g.

public string SomeMethod()

or

public string SomeMethod (string someInput)

we need to have a return at the end of the function to return a type that is defined within the declaration (also known as the method signature).

We can have methods that don't return ANYTHING – they might perform a calculation, the value of which can be retrieved through a property. In this case we simply declare the return type as void. E.g.

public void SomeMethod()

in which case we don't return anything.

Each method and property can again be declared with a different access modifier just like the class. And just like the class the same rules apply.

We can also add the 'virtual' keyword to the declaration – this marks the property or method as being 'overridable' – this is used in inheritance to allow your methods to have a different implementation in a derived class.

One final point on class/object construction. You may have noticed that sometimes I use capital letters and sometimes I don't.

There is no requirement to do this – however it is good convention to differentiate your internal and external methods, properties and variables. The convention is that internal variables start with a lowercase letter, whereas external methods and properties start with a capital letter.

Some developers even use _ at the start of internal variables and this is often hot debated as to whether you should or shouldn't! (I guess some developers have a lot of free time!)

Objects also have certain characteristics that are intrinsic to the whole OOP paradigm – Polymorphism, Encapsulation and Inheritance.

Encapsulation

Encapsulation is about ensuring an object groups together all the necessary properties and methods for that particular object. For example a Customer Object should contain the name and address of that customer, and not need to query another object.

However objects can CONTAIN other objects. So you might have an ADDRESS object and the CUSTOMER object has an ADDRESS property that is of the type ADDRESS. In this way CUSTOMER is a ROOT object and ADDRESS is a LEAF object.

Inheritance

Inheritance is the ability of objects to extend another object by gaining all the original objects base properties but then by adding its own.

For example, you may have a PERSON base class, and then CUSTOMER and SUPPLIER classes that inherit from PERSON. In this way common properties and methods (Address, Name) can be built into the PERSON base class, and specific properties and methods that only relate to the new object are only written into the new object.

```
Public class Person{
   Public string name;
}

Public class Supplier: Person{
  Public string Website;
}
```

As you can see Supplier inherits Person (by using : after the declaration of the class name).

The resultant object would have BOTH the ‘name’ AND the ‘Website’ properties.

Polymorphism

Polymorphism is the ability to create two different objects that can be used interchangeably. So the two objects would have the same properties and methods but the actual implementations of them *internally* would be different.

This is often achieved through the use of an Interface. The interface would define the properties and methods, and then each object would implement that interface accordingly.

An example of this could be a simple calculator. You could have a ‘Result’ object that has a single method called compute that returns a number. The method will take two inputs as numbers and then do something with them. One object would add them, another would subtract them, and another multiply them and another divide them.

Each object would implement an IResult interface (often interfaces are prefixed with I to denote interface). As long as the object implements all defined properties and methods of the interface it can be easily interchanged.

```
Public interface IResult{
   Int Compute (int first, int second)
}

Public class AddFunction : IResult{
   Int Compute (int first, int second){
     Return first + second;
  }
}

Public class SubtractFunction : IResult{
   Int Compute (int first, int second){
     Return first - second;
  }
}
```

The objects that implement the IResult interface are often referred to as a Concrete Implementation. And any code that uses IResult does not need to be aware of the internal implementations – they can be swapped out accordingly.

So, in any main code block that wants to use the interface – such as a calculator program, would reference the interface not the concrete object.

```
IResult r;

If(mode==”Add”){
        r = new AddFunction();
}
If(mode==”Subtract”){
        r = new SubtractFunction();
}
Int answer = r.Compute(5, 5);
```

Namespaces and References

It would be feasible to creating an entire program in one big file. And that entire file would contain all our objects. However this is neither practical, not possible these days. Even your base functionality for everything you want to do comes from another object.

For example when you create an application Visual Studio automatically references a bunch of other prebuilt objects that contain many functions (for example, outputting to the screen).

So we have two problems the first is how to tie them together, the other is how to prevent conflicts.

First the conflict issue. How do we ensure, given that in a large application we may want to link to hundreds of other objects, that two objects with the same name don't conflict? The answer is namespaces.

A namespace a bit like your full name, or house address. Rather than saying you're live at Blogger Street – which would host a host of issues for your postman – you say you live at 35 Blogger Street, Some Town, England.

So it is with name spaces – you define a namespace for your class to ensure it is unique. Of course you could conceivably create two namespaces the same, but generally software you are releasing is pre-fixed with your company name – then it's just up to you internally to ensure you adhere to common standard.

So for example Microsoft have a bunch of core objects all under the Microsoft.CSharp namespace.

So now when you want to link an external object into your object you simply add the namespace to the top of your object with the using keyword – e.g.
using Microsoft.CSharp;

Using Objects

When you create a class you're only defining the class itself. It can't be used until you instantiate it. This is done using the new keyword.

For example – MyClass myvarref = new MyClass();

(The exceptions are static classes).

So what happens here is that an INSTANCE of the object is created, and assigned to your variable name (in this example myvarref). Once instantiated it is now in memory and can be used.

This fact can cause problems. Because if you leave the object in memory and then just go on creating other objects eventually you might run out of memory.

A lot of development platforms are quite intelligent these days. Once an object passes out of scope then all objects are deemed no longer in use. .NET has something called a garbage collector that tidies up these killed off objects, however with some objects it can be good practice to manually kill them.

What do we mean by out of scope? Scope is when you create a section of code, such as a loop or block. Anything inside that block is in scope as is anything outside the block. But once we move OUT of the block, everything created inside it is now out of scope.

Here's an example:

```
GlobalObject globalObject = new globalObject();

If(a==b)
{
        ScopeObject scopeobject = new ScopeObject();
        scopeobject.candosomething();
        globalobject.candosomthing();
}
```

GlobalObject is global – which means it can be used inside AND outside the if block.

But the LocalObject is declared within the if block and can therefore only be used in the if block. Once our code has moved out of the if block it is now out of scope. If we tried to use it, it would simply throw a compiler error.

Constructors and Deconstructors

Objects have what are called constructors and deconstructors. These are special areas of code that are responsible for running when an object is created or destroyed.

In c# constructors are declared thus:

```
Public class MyClass{
        Int globalvar;
        Public MyClass(int someparameter)
        {
                this.globalvar = someparameter;
        }
        …other methods and properties
}
```

As you can see we simply have a Public followed by the class name. We can then optionally tell it to receive a bunch of

parameters. If these are declared then you MUST pass them in when instantiating the object, otherwise an error will be thrown.

Because the passed in parameters are only valid for that block of code, if you need to use them elsewhere in your object you must assign them to a variable listed outside that block. The term 'this.' is a handy way of stating that you mean the object itself. An internal property called globalvar can be referenced within the class as this.globalvar – this isn't necessary, but is good practice to prevent variable name conflicts.

A Deconstructor is defined using the ~ (tilde)

```
Public class MyClass{
        ~ MyClass()
        {
                //any internal cleanup here
        }
}
```

Reference Types versus Value Types

Reference Types are most objects. When you create a class it is a reference type. Reference types get PASSED AROUND because they contain a reference to a memory location that holds our object, rather than the actual object itself. This means if you copy an object it is actually just a copy of the reference, the actual memory location it references stays the same.

Value types contain the actual value. Therefore if you copy one to another the value itself is duplicated, so you now have two independent copies. E.g.

```
public class MyClass{
      public int aVlaue { get; set; }
   }

   public class Worker{

      public void DoSomeThing(MyClass refClass, int valueint)
      {
         refClass.aVlaue = 1;
         valueint = 1;
      }

   }

   class PlayGround
   {
      public PlayGround()
      {
         MyClass myClass = new MyClass();
         myClass.aVlaue = 0;
         int localint = 0;

         Worker worker = new Worker();
         worker.DoSomeThing(myClass, localint);

         Console.WriteLine(myClass.aVlaue);
         Console.WriteLine(localint);
      }
   }
```

In this example we have a MyClass object with a single property called aValue.
We then have a Worker class that has a DoSomethingMethod – this takes in a MyClass object and an int.

In the Playground we do the actual code.
So first we instantiate MyClass. We then assign 0 to its property aValue.
We then create an int and assign it a value of 0.

Now the worker takes in our myClass object an our integer variable.
It then updates the int to 1 and the aValue of myClass to 0.

We then output the values in the main Playground class.
What will the outputs be?

myClass.aValue is 1. This is because myClass is a REFERENCE type. When we sent it to the worker class we were actually send a reference to our object, so any actions on the object apply to our instance of myClass.

localInt is 0. Int is a Value Type, so when we passed it into the worker class all we were really doing was sending in the value of 0 – we DIDN'T send the variable itself, just a COPY. Because of this any changes to localint within the actual DoSomething method are effectively lost because it applied to a copy rather than the actual variable from our main code.

To get the value of 1 back to our localint we'd have had to make the DoSomething return a value and assign it back to the original var. e.g.

```
public class MyClass{
      public int aVlaue { get; set; }
   }

   public class Worker{

      public int DoSomeThing(MyClass refClass, int valueint)
      {
         refClass.aVlaue = 1;
         valueint = 1;

         return valueint;
      }

   }

   class PlayGround
   {
      public PlayGround()
      {
         MyClass myClass = new MyClass();
         myClass.aVlaue = 0;
         int localint = 0;

         Worker worker = new Worker();
         localint = worker.DoSomeThing(myClass, localint);

         Console.WriteLine(myClass.aVlaue);
         Console.WriteLine(localint);
      }
   }
```

In this example both localint and myClass.aValue will be 1.

SOLID Principals

Power is nothing without control

OOP provides a very powerful framework for writing code. It allows for software to built in a modular fashion, enabling easier code re-use, faster bug finding, and easier to maintain code bases.

However it still needs to be used correctly. Even with OOP it is common to build software that quickly becomes unmanageable as it grows. This is often a result of updates and bug fixes.

Traditionally one way to keep things under control was to have a rigid and well-documented application architecture in place. The whole solution from start to finish would be well mapped out, and any changes had to be agreed and documented before implementation.

Unfortunately this rigidity causes more problems than it fixes. Project delays and spiraling costs are all too common in larger projects, and history is full of billion dollar failures all over the world.

One of the problems was that software would take so long to build, and by the time the first releases were available requirements might have changed, or maybe different teams had misunderstood something and built code that wasn't fully interoperable. Or perhaps once the customer started to use the software they realized that actually wasn't the best way to do it – and so they needed it changing.

What was needed was a more flexible approach – one that not only allowed changes but actually embraced them. Software needed to be released for testing earlier – with constant feedback and testing.

Enter Agile – a management methodology that does just that. Now software is written in smaller chunks, the customer tests it, give feedback, and then changes are made.

But working in this matter means code and even the overall architecture of an application is forever changing – so how do you keep the code base clean?

Enter (among others) SOLID Principles – there are a number of similar guidelines but my favorite is SOLID.

SOLID Principles provide a set of 7 guidelines that dictate how software and objects should be built in order to create a robust, flexible and extensible application.

The aim is not just about breaking down an app into modules but to make those modules INDEPENDENT of each other – or at least as much as possible.

So for example. Let's say we want to create a Logger object.

The logger object will be responsible for logging debug and error information to a console.

The logger has a number of methods – Debug, Warn, and Severe – and each of these methods takes a string.

Throughout our application we want to use the logger object to log out some kind of error to the console.

Each bit of our program, be it a main block or another object, needs to instantiate an instance of our Logger object - instantiate means create an instance of that object, usually by using the NEW keyword – e.g. Logger logger = new Logger().

Now comes the problem – what if we need to change the implementation of Logger? What if we have found a newer logger that does things better? Maybe it outputs to SQL, or emails logs. Whatever, the point is we might now have a new logger we want to use – let's say it's called SuperLogger. So now every time we want to use our new SuperLogger we'd need to go through every bit of our program and manually swap out Logger for SuperLogger.

Incidentally you can't just having the same name wouldn't cut it e.g. if you had two different implementations called Logger they would have to be in difference namespaces, and so even if you didn't need to change Logger for SuperLogger you'd still have to go through each class updating the namespace references.

Another issue of doing it this way is that it would use more memory. Because each of your objects is creating its OWN instance of the logger. If the logger is writing to a file this could also cause file-locking issues because you now have multiple instances of the logger trying to write to the same file.

So the first step is to pass down a reference to an existing running instance of our logger app rather than creating a new one each time. This will be done via each object's constructor.

But this still leaves us with a concrete implementation, even though we're instantiating it once – each class still needs to know about the concrete implementation to work.

Instead we use an Interface. Remember an interface defines the structure of an object. So each object will take in an interface, and only in the actual main program will we create and assign our concrete implementation to the interface. E.g.

```
public interface ILogger {
        void Debug(string message);
    }
    public class Logger : ILogger{
        public Logger() {

        }

        public void Debug(string message) {
            Console.WriteLine(message);
        }
    }

    public class MyObject{
        public ILogger logger;
        public MyObject(ILogger logger)
        {
            this.logger = logger;
        }

        public void Something(string param) {
            logger.Debug(param);
        }

    }

    class PlayGround
    {
        public PlayGround()
        {
            ILogger logger = new Logger();
            MyObject myObject = new
MyObject(logger);

        }
    }
```

As you can see out Logger class implements ILogger – and it's this that gets passed into any objects that need to use it. That way if we ever need to change the Logger – we only need to change it in the one place (provided the new logger class also implements the same interface).

Incidentally, especially with things like loggers, you often implement 3rd Party plugins. So the way round this would be to *wrap* the plugin with our own logger. So our logger class would implement our interface, and internally our logger class will utilize the 3rd party software. Then if at some point we need to use a different logger we can just update our wrapper.

This is just one simple example of how adhering to SOLID principles can make your solutions more flexible and easier to manage.

To go more in depth is beyond the scope of this book, however I hope you have enjoyed and gained some help from the contents.

For a more in depth look at SOLID Principles and how to put them into practice please checkout my blog at

http://bretthargreaves.wordpress.com