Best Practices for C# Delegates

Scott Bain
Senior Consultant,
Net Objectives
slbain@netobjectives.com

Introduction

It has been rightly pointed out by many that C# bears a striking resemblance to Java in both syntax (many language constructs are identical) and overall strategy (compilation to an intermediate form, managed run-time environment with automatic garbage collection, etc...).

In addition to being Java-like, however, C# has added a number of language idioms. Some of these are elements that were available in C++ and were removed when Java was designed (operator overloading, early bound -- non-virtual -- methods as an option, etc...), and some could reasonably be called innovations created specifically for the .Net languages, of which C# is the primary.

The purpose of this article is to explore one of these more innovative language elements, delegates, in the hopes of investigating some of the opportunities they present as well as some of the dangers they expose us to.

Delegate Basics

The notion of the Class can be thought of this way: Define a set of methods that fulfill some application responsibility, and the state needed for those methods to operate, and give this arrangement a name. Similarly, a delegate is a named type that defines a particular kind of method. Just as a class definition lays out all the members for the give kind of object it defines, so does the delegate lay out the method signature for the kind of method it defines. Thus, the delegate declaration:

```
public delegate Foo ThingMaker(Bar b, int i);
```

...basically declares "all methods that take a Bar and an int as parameters, and return a reference to a Foo, qualify as ThingMaker methods, and can be used interchangeably."

A delegate can be declared for any given method signature. Note that the delegate declaration does not actually define what the delegate does, only what it looks like and how it is used. Any "real" method can be wrapped up in this delegate type, and

then called without the user of the delegate "knowing" where the actual method is, whether it's a static or an instance method, etc...
Here's an example:

```
namespace Delegate_Experiments
{
   // Delegate declaration for Strategy.  Any method that takes a String
   // and returns a String can be a wrapped in a Strategy delegate.
        public delegate string Strategy(string aString);

   // This class is designed to use a Strategy delegate without knowing
   // what method it actually wraps.
        public class Delegate_Strategy
        {
                // State needed for the class.  A string that will be processed
                // by the delegate
                private string stringToProcess;

                // The constructor for the class, setting the state
                public Delegate_Strategy(string aString)
                {
                        stringToProcess = aString;
                }

                // The method that uses the Strategy.  Note the type of the
                // myStrategy parameter is the delegate type defined above.
                public string ContextInterface(Strategy myStrategy)
                {
                        // The method calls the delegate just as if it were
                        // a local method in the current object
                        return myStrategy(stringToProcess);
                }
        }


        // Here is the client that will create different Strategy delegates and
        // Give them to the Delegate_Strategy class to use.
        public class Strategy_Client
        {
                // This method has the right signature for a Strategy delegate.
                // The fact that it's static does not matter
                public static string Capitalize(string rawString)
                {
                        return rawString.ToUpper();
                }

                // Here is another method, that has a different implementation
                // but also fits the Strategy delegate signature requirements
                public static string LowerCase(string rawString)
                {
                        return rawString.ToLower();
                }

                // The Main will test all this
                public static void Main(string[] args)
                {
                        // Build an instance of the class that uses a Strategy
delegate
                        // Constructor requires initial state
                        Delegate_Strategy ds = new
Delegate_Strategy("CamelCase");


                        // Make two Strategy delegates, one to wrap each local
method
                        Strategy cap = new Strategy(Capitalize);
                        Stragegy low = new Strategy(LowerCase);
```

```
                        // Now, call the method that uses a Strategy, once with
each
                        // of the two we've built, and observe the different
results
                        Console.WriteLine(ds.ContextInterface(cap));
                        Console.WriteLine(ds.ContextInterface(low));
                }
        }
}
```

For those of you who speak Design Patterns, this is indeed an implementation of the Strategy pattern (delegates make for interesting implementations of many of the GoF patterns, and create opportunities for some new ones).

Key things to note in the above example:

1. The delegate declared right after the namespace declaration creates a type in the system, much as a class or struct declaration creates a type. The difference is that a delegate defines a type of method.
2. The ContextInterface() method of the Delegate_Strategy class takes, as a parameter, an instance of type Strategy.
3. The Main() method of Strategy_Client creates two instances of the Strategy delegate using the new keyword and passing in local method names into what looks like a constructor. This is how delegate instances are built. The methods being wrapped must:
    a. Match the required method signature defined when the delegate was declared, and
    b. Must be accessible at the time the delegate is created. Does this mean you can wrap a private method in a delegate and allow some other entity to call it? Stay tuned.
4. The ContextInterface() method of Delegate_Strategy uses the Strategy it is passed, calling it just as if it was a local method of its own.

This code runs... give it a try.

Gotcha: Delegates and Type Safety

Delegates would appear to be type safe. The compiler will not allow you to instantiate a delegate by handing it a method with the wrong signature, and so the user of the delegate can be confident that the method is callable with the parameters it expects, and will return the type it expects.

However, this is not true type safety. The signature-checking is the only checking done by the compiler. If two delegate types are declared with the same signature, one can be substituted for the other, and the compiler will not care. Here's an example:

```
class Soldier
{
        // Delegate type declared within a class, so outside entities will
        // refer to it as type Soldier.Fireable
        public delegate void Fireable();

        // Class Soldier has a member of this type
        Soldier.Fireable myWeapon;
```

```
        // When you arm a Soldier you give it a delegate reference
        // of type Soldier.Fireable
        public void arm(Soldier.Fireable aWeapon) {
            myWeapon = aWeapon;
        }

        // When you tell the soldier to fight, he uses the Soldier.Fireable
        // delegate he holds a reference to
        public void fight(){
            myWeapon()
        }
}


class Payroll
{
        // This delegate has the same signature as Soldier.Fireable, but it is
        // a different type, being declared within Payroll.  It's type is
        // Payroll.Fireable
        public delegate void Fireable();
}
```

The point of concern for us is that it turns out that the arm() method in Soldier, which is declared as accepting a parameter of type Soldier.Fireable() will, in fact, take a reference to Payroll.Fireable(). Similarly, the fight() method will happily use it, even though it is the wrong type.

This is like taking a bad employee (a Fireable entity in the Payroll system) and handing it to Sergeant Rock as a weapon, and having him use the hapless fellow as a tommygun against the enemy. Not likely to have a good result.

This is because the signatures of these two delegate types are the same, and the signature is all the compiler checks.

Contrast this to the use of Interfaces, a la Java:

```
class Soldier {
    public interface Fireable {
        void fire();
    }
    Fireable myWeapon;
    public void arm(Soldier.Fireable aWeapon) {
        myWeapon = aWeapon

    }
    public void fight(){ myWeapon.fire() }
}

class Payroll {
    public interface Fireable {
        void fire();
    }
}
```

Now the arm() method of Soldier will only accept an implementation of the Soldier.Fireable interface, and not the Payroll.Fireable interface. C# has interfaces just like Java, so we can have this type-safety when we need it, but it's important to note that we lose it if we try to use delegates in this way.

Delegates for Security

For every gotcha there is a new feature.

When I first started experimenting with delegates, I discovered something that initially disturbed me in a very fundamental way. A delegate can wrap a private method in such a way that an outside entity can now call it. Here's an example of the phenomenon:

```
namespace Delegate_Experiments
{
        public delegate int Fireable(int rounds);

        public class Soldier
        {
                public Soldier()
                {
                }

                Fireable weapon;

                public void Arm(Fireable aWeapon) {
                        weapon = aWeapon;
                }

                public int Fire(int rounds) {
                        int damage = 0;
                        if(weapon != null) {
                            damage = weapon(rounds);
                          } else {
                              Console.WriteLine("No weapon available");
                        }
                        return damage;
                }
        }
}
```

Very similar to our previous example, except Fireable is now declared in the namespace, not as part of Soldier. Also, Fireable here takes an integer (number of rounds to fire) and returns an integer (the damage done).

At any rate, you can see how this would work. Any object could make a soldier instance, wrap a method that matches the Fireable signature into a Fireable delegate instance, and hand it into the Arm() method. Subsequent calls to Fire() would cause the solider to fire the Fireable it had been armed with. Pretty easy to follow.

Now, consider:

```
namespace Delegate_Experiments
{
        public class Army
        {
                private int myAmmo = 100;

                public int Pistol(int rounds)
                {
                        Console.WriteLine("Firing a Pistol");
                        myAmmo -= rounds;
                        return rounds * 1;
                }
```

```
                private int Rifle(int rounds)
                {
                        Console.WriteLine("Firing a Rifle");
                        myAmmo -= rounds;
                        return rounds * 5;
                }

                public void Attack(){
                        Soldier audie = new Soldier();
                        audie.Arm(new Fireable(Pistol));
                        Console.WriteLine("Attack: " + audie.Fire(5));
                        Console.WriteLine("Remaining ammo = " + myAmmo);
                        audie.Arm(new Fireable(Rifle));
                        Console.WriteLine("Attack: " + audie.Fire(5));
                        Console.WriteLine("Remaining ammo = " + myAmmo);
                }

                public static void Main(String[] args)
                {
                        Army a = new Army();
                        a.Attack();
                }
        }
}
```

The Main() method builds an Army and calls Attack(). So how does Attack() work? It makes a new Soldier, called audie (for Audie Murphy... eh, you're probably too young), and then wraps the local Pistol() method into a Fireable delegate, and arms audie with the Pistol. When Audie is told to Fire(), he is really using the Pistol() method in Army, through the Fireable delegate.

All well and good. However, next audie is re-armed with a new delegate, with wraps the Rifle() method in Army. The Rifle() method is private, and it also accesses private state in Army. Can audie call a method with is private in another class?

Yes. This would seem to be a huge, glaring mistake in the design of delegates... it breaks a very fundamental rule in encapsulation – private members may not be accessed by outside entities! Now, Army is not free to change Rifle because Soldier may be coupled to it. Oy!

It's not as bad as it looks. In fact, it's actually an opportunity disguised as a mistake.

First of all, only Army could have made the offending delegate instance. If another class tried to wrap Army's Rifle() method into a Fireable delegate instance, the compiler would not allow this. Only Army has the access rights or visibility to Rifle(), so only it can make the delegate in question.

So, if I change Rifle() in Army, then the line that makes the delegate (in the Attack() method) won't compile anymore. The compiler will catch me, in other words, so the coupling is not as bad as I thought.

...and when I think about it, I realize that this "feature" of delegates allow a class to make a method private, and then selectively grant access to it to particular entities. Without delegates, a method is either public (any entity can access it) or private (none can). Allowing access in this fine-grained way could enable some interesting patterns. This is not unlike "friends" in C++, but with finer-grained control. Consider this:

```
namespace Delegates_For_Security
{
        public interface Registerable
        {

        }
        public delegate String SecureInfo();


        public class Repository
        {
                public static String Open_Method()
                {
                        return "This is information anyone can have, "+
                               "anytime";
                }

                private static String Secure_Method()
                {
                        return "This is information that only " +
                               "authenticated entities can access";
                }

                public static SecureInfo GetAccess(Registerable requester)
                {
                        Registry r = Registry.GetInstance();

                        if(r.Auth(requester)){
                                return new SecureInfo(Secure_Method);
                        } else {
                                throw new Exception("Authenication Failure");
                        }
                }
        }
}
```

Repository here has two methods, Open_Method(), which is public and therefore callable by any entity, and Secure_Method(), which is private and would therefore, normally, not be accessible to any outside entity.

However, consider the GetAccess() method. It can return a delegate of SecureInfo type, which wraps the SecureMethod() method, and would then allow the outside entity to call it, and get to the secure information. It decides whether or not to do this by attempting to authenticate the entity before it hands back the delegate.

Just to round this out, the Client and Registry classes are supplied below. But this is just one authentication mechanism (very weak, it just requires you to sign up with the Registry before you can get the secure delegate) – any sort of authentication could be used here, and thus we gain method-by-method access control, which is a pretty neat trick.

```
namespace Delegates_For_Security
{
        public class Client : Registerable
        {
                public void Go()
                {
                        Registry r = Registry.GetInstance();
                        r.Add(this);
```

```
                        Console.WriteLine(Repository.Open_Method());

                        try {
                                SecureInfo mySI = Repository.GetAccess(this);
                                Console.WriteLine(mySI());
                        } catch(Exception e) {
                                Console.WriteLine("Problem: " + e.Message);
                        }
                }

                public static void Main(string[] args)
                {
                        Client c = new Client();
                        c.Go();
                }

        }

        public class Registry
        {
                private static Registry singleton = new Registry();
                private ArrayList registrars;

                private Registry()
                {
                        registrars = new ArrayList();
                }

                public bool Add(Registerable r)
                {
                        registrars.Add(r);
                        return true;
                }

                public bool Auth(Registerable r)
                {
                        bool rval = false;
                        if(registrars.Contains(r)) rval = true;
                        return rval;
                }

                public static Registry GetInstance()
                {
                        return singleton;
                }
        }
}
```

Multicast Delegates as Decorators

The decorator pattern (GoF) allows functionality to be added, in layers, to an existing object without changing its class.

Like all design patterns, the "pattern" is not a particular implementation, but the issues to be resolved, a particular way of resolving them, and a set of issues/opportunities that result from the resolution. In other words, a decorator is not about how you implement it, but rather about what it does for you.

Multicast delegates in C# are delegates that can wrap more than one method. When the multicast delegate is called, each of the wrapped methods is called in turn, one after the other. This is, in and of itself, an obvious implementation of the observer pattern (and, in fact, is how event handling is achieved in .Net).

It occurred to me that this is essentially decorating (adding on some number of layered behaviors on top of an existing behavior), albeit only decorating a single method. Decorators usually decorate the entire interface of an object, so using delegate to decorate is only useful if we're talking about a single method. Anyway, here's an example of the idea in code:

```
namespace DecoratorExample_Delegates
{
        // Only delegates with a void return type can be multicast
        // but we want the decorator to have an effect, so we make
        // sure its parameter is a reference type, either an object
        // or a value type using the ref keyword.

        // Here's the delegate declaration.  Note the use of ref
        // because string is a value type in C#.  We would not
        // need this if we were decorating a reference type.
        public delegate void NestedDec(ref string info);

        public class DelegationDecorator
        {
                // Here is one decorating method
                public static void m1(ref string info)
                {
                        info += " m1 decoration";
                }

                // Here is another decorating method
                public static void m2(ref string info)
                {
                        info += " m2 decoration";
                }

                public static void Main(string[] args)
                {
                        // Make the base delegate (note you must
                        // assign it to null)
                        NestedDec nd = null;

                        // Using the multicast overload of "+=",
                        // attach whatever delegate methods are desired.
                        // Note the order is significant
                        nd += new NestedDec(m1);
                        nd += new NestedDec(m2);

                        // Make the base (undecorated) item.
                        string info = "Base String";

                        // Invoke the decorator delegate.  All attached
                        // delegates will fire, changing the base item once
                        // per decorator method.
                        nd(ref info);

                        // This will produce:
                        // "Base String m1 decoration m2 decoration"
                        // on the console.
                        Console.WriteLine(info);
                }
        }
}
```

The details are in the comments, but to summarize:

1. Multicast delegates must have a void return type, so to propagate the decoration effects, you must use a reference type, or use the "ref" parameter keyword.

2. When you make a multicast delegate, you must initialize it to null explicitly. I'm not sure why this is, but the compiler complains otherwise.
3. You "attach" decorators to the delegate using the "+=" operator. Add as many as you like, but the order is significant. You can remove any decorator using the "-=" operator.
4. Only a single method can be decorated in this way. You could use multiple delegates to decorate an entire interface, but this would likely be hard to maintain. If the interface is complex, I'd use a more conventional decorator.

Summary

Delegates are an interesting and potentially useful language idiom in C# - and in all the .Net languages. They enable new patterns, and new ways of implementing old patterns, and create a level of decoupling-though-indirection that can add flexibility to your designs.

Those who are comfortable with function pointers in C++ probably see delegates as very similar in their role, the difference being the (admittedly somewhat weak) typing, and the fact that they can be multi-cast.

-Scott Bain-
February 2003