**The Inside Track**
"See Farther, See Sharper"
December 2000

Do you have a favorite ax? You know, musicians call their instruments an "ax". The equivalent for a programmer must be a programming language. I've used a lot of languages in my career, I'm sure you've used a few yourself. The progress of programming languages is a long and interesting topic but a bit esoteric for this column. I have an ax or two that I turn to repeatedly. Learning a new language is fun. OK, my wife tells me I'm a bit strange too. But I think its fun to see how I can write better programs more quickly with a new language.

As part of the .NET technologies, Microsoft has created a new language. C# is the name. C# introduces a number of interesting language innovations while preserving familiarity. While C# is almost as completely an object oriented language as the prototypical OO language SmallTalk, it talks and walks a whole lot like C++.

Before getting into some details about C#, let's take a quick look at the alternatives. Microsoft could have just enhanced C++ or VB or Java. In fact, if you look at .NET you'll see that there is something called Managed C++ and Visual Basic 7 (or VB.NET for short). The extensions to C++ are ugly since they were added using the double underscores that are required by the C++ standard for vendor specific extensions.

Likewise, the new VB.NET has the flavor of VB6, but there are LOTS of new features and concepts to make VB fully object oriented that make it more of a new language. The differences are even greater than the VB3 to VB4 transition.

Java could have been an alternative. Microsoft has been limited legally by lawsuits and technically (you can't call it Java if it doesn't follow certain strict limitations) in their effort to extend that language. However, JavaScript is one of the .NET languages, so you don't need to feel left out if you're a Java maven.

C# is more evolutionary than revolutionary, built on that rich language history. Borrowing the syntax of C++, C# integrates new concepts in a clean, modern language making C# an extremely productive language for writing real applications.

## Common Language Runtime

One of the features of the .NET technologies is the Common Language Runtime (CLR). The CLR is similar to the Java Virtual Machine (JVM). Like the JVM, the CLR is an execution environment for your program that is machine and operating system independent. Programs destined for the CLR are compiled into Intermediate Language (IL) the "machine" language of the CLR.

Unlike the JVM, the CLR was conceived of as an execution environment for any number of languages. This isn't an original idea to Microsoft, compilers have been built that produce code for the JVM and cross-compilers are as old as the hills. And what is a compiler of not a translator from one language to another? The CLR currently has implementations for a number of languages and, not surprisingly, so does the JVM. The difference is that the CLR was designed to support a broad cross-section of language features; the JVM was only designed to support Java's features. To enable multiple programs to interoperate, the Common Language Specification (CLS) says which features of the CLR a program may use and interoperate with other programs running in the CLR. For instance, C# and the CLR support an unsigned 64 bit integer, but the CLS says that only signed 64 bit integers MUST be supported. The CLS reduces the burden on language developers and implementers that wish to allow their languages to interoperate.

One of the really nice things Microsoft has said about the .NET technologies is that they will be open and follow standards. And C# would not be very standard if it was only under the control of

Microsoft. Microsoft, Hewlett-Packard and Intel submitted C# to the international standardization body ECMA (see http://www.ecma.ch and http://msdn.microsoft.com/net/ecma/).

## Components

I explain this much of the CLR so that you'll understand the significance and motivation for the language changes and extensions made to .NET languages and to motivate the real topic of this month's The Inside Track. Let's look at how C# supports building components!

What is a component? It's a piece of a software system. As a piece, it must work with other pieces. You need rules to allow pieces to work together. So now you see way the CLR and CLS are so important. If developer A wants to develop a component of a software system with C++ (say a heavy duty server component), another with VB (say a chunk of business logic) and another with JavaScript (say some code to handle Web pages), then they need well defined rules to allow all these parts to work together. Has this been possible in the past? Sure! Was it easy? Not on your life! Will it be tons easier with .NET? Yup, you just got your life back!

As I've mentioned before, Great Plains plans to build its next generation of products using Microsoft .NET technologies. At this point you may have guessed that we plan to write those products in C#. C# takes all the learning from VB, C++, Java, Delphi, COM and undoubtedly a lot of other source (hey, I don't work for Microsoft, I have to guess just like everyone else) to come up with a clean, new language for building components. I'm going to talk about 3 basics of components: properties, methods and events. If you've written a COM component (any language), written any VB or used an ActiveX component (.OCX) then you know why properties, methods and events are the key.

Properties represent the state of a component to the outside world. Methods are the means that the outside world uses to ask a component to perform some action. Events are the way a component lets the outside world know that something interesting has just happened. These three mechanisms are about 95% of what components use to work together to implement a software system. So lets look at how C# makes implementing and using each of these mechanisms effortless.

## Methods

A method is nothing more than a function. Functions are basic building blocks of any software system. When writing a large software system, you have parts of the system you want exposed to other parts and parts that you don't want exposed. You might call these parts "public" and "internal". The public parts are those that another component in the software system may use, and the internal parts are those that they may not. The internal parts are part of your implementation of your component. In object-oriented terms, the public part is your interface. Encapsulation is the object-oriented concept that says you hide the internal details of your implementation. In the same sense then a component should expose a specific interface to other components and hide the internal details of how it implements that functionality. C# supports exactly this concept. Take a look at the following C# class definition:

```
public class TaxCalculator
{
	public TaxCalculator(Location location)
	{
		this.location = location;
	}

	internal decimal TaxRate(TaxableType taxableType)
	{
		return TaxRate(DateTime.Today,taxableType);
	}

	internal decimal TaxRate(DateTime when, TaxableType taxableType)
	{
		TaxInfo taxInfo = new TaxInfo(when);
		TaxTable taxTable = taxInfo.RateTable(location);
		TaxRate taxRate = taxTable[taxableType];

		return taxRate.Rate;
	}

	public decimal TaxAmount(TaxableType taxableType,decimal taxableAmount)
	{
		return TaxRate(taxableType) * taxableAmount;
	}

	Location location;
}
```

The TaxCalculator class defines a public constructor called TaxCalculator () that sets the location. A constructor is used to initialize the state of a new object. The public method TaxAmount() relies on two internal methods to complete its work. The first TaxRate() method takes TaxableType and uses the current date to call the second helper method. The second TaxRate() takes two parameters and uses those parameters along with the location passed into the constructor to find the correct tax rate. If TaxCalculator where built into a .NET component, you would only see the two public methods, the internal helper methods would not be accessible outside the component.

By the way, having two TaxRate() methods is perfectly legal. Its called method overloading and C# knows which method to call based on the number and type of parameters used by the caller.

## Properties

Properties of a component represent the data values that a component supports. Just like methods, properties can have the same visibility attributes. In addition, properties may be read/write, read-only or write-only. Components may need to react to a new property value or returning a property value may require some processing. To support these behaviors, C# includes syntax just for properties. Consider the following C# class:

```
class Temperature
{
	public double Celsius
	{
		get
		{
			return (Fahrenheit - 32) * 5 / 9;
		}
		set
		{
			Fahrenheit = value * 9 / 5 + 32;
		}
	}
	public double Fahrenheit;
}
```

The Temperature class allows the user to save and get a temperature in either Celsius or Fahrenheit and always returns the correct value. Strictly speaking C# calls Celsius a property and Fahrenheit a field. However, if Temperature were built into a component, two “properties” would be visible. Temperature works by always keeping the temperature value in Fahrenheit and converting to and from Celsius as needed. Though this is a very simple example, you can see the power to transform values, perform processing in response to changes and obtaining values. Perhaps getting a value might be implemented by reading a database!

## Events

The last component technology that we will look at in C# is the event. Events, as I mentioned, can be used by a component to signal to the component’s client(s) that something interesting has happened. Think about this: you have “customer” implemented by a component. You would like to notify the credit department whenever a customer has exceeded a credit limit. Good object oriented design says that Customer and CreditDepartment should be separate objects (components) that use well-defined interfaces to communicate. You don’t want them tightly coupled. You could have your Customer component call a method in the CreditDepartment component, but that means that the Customer component must “know” about CreditDepartment objects. Not good, that’s tight coupling. Alternatively, you could have your CreditDepartment object check the credit limit and current balance properties on each Customer object. The question is how often? If you have lots of customers, that’s a lot of wasted effort. The solution is events! The Customer object could implement an OnOverCreditLimit() event or even an OnBalanceChange() event. Then the CreditDepartment object could “register” with the Customer objects its interest in being notified when the customer goes over his credit limit. Let’s see how this would look:

```
using System;

public class Customer
{
	public delegate void
		OverCreditLimitHandler(Customer customer,decimal amount);

	public event OverCreditLimitHandler OnOverCreditLimit;

	public Customer(int id,string name,decimal creditLimit)
	{
		this.id = id;
		this.name = name;
		this.creditLimit = creditLimit;
		this.currentBalance = 0;
	}
	public bool Purchase(decimal amount)
	{
		if(amount + currentBalance <= creditLimit)
		{
			currentBalance += amount;
			return true;
		}
		else
		{
			if(OnOverCreditLimit != null)
				OnOverCreditLimit(this,amount);
			return false;
		}
	}

	public int id;
	public string name;
	public decimal creditLimit;
	public decimal currentBalance;
}

class Test
{
	static void Over(Customer cust,decimal amt)
	{
		Console.WriteLine("Customer {0}:", cust.name);
		Console.WriteLine("\tTransaction Amount = {0}",amt);
		Console.WriteLine("\tCurrent Balance = {0}",cust.currentBalance);
		Console.WriteLine("\tCredit Limit = {0}",cust.currentBalance);
	}

	static public void Main()
	{
		Customer cust = new Customer(100,"Karl Gunderson",100m);

		cust.OnOverCreditLimit +=
				new Customer.OverCreditLimitHandler(Over);

		cust.Purchase(50m);
		cust.Purchase(60m);
		cust.Purchase(120m);
		cust.Purchase(50m);
	}
}
```

I've included a complete working program in this case. I've implemented the Customer class as described, but not the CreditDepartment. The first thing to notice is the line that starts "public delegate". What's a delegate?? A delegate is a pattern for a method. You'll notice that the delegate is used in the next line to declare an event. The delegate describes the parameters of the event method. The event method is the method that will be called when the event is triggered. In this example, the method is called Over() in the Test class. Notice how the parameters of the Over() method match those of the delegate.

The next interesting piece of this program is the line in Main() that starts "cust.OnOverCreditLimit +=". This line subscribes Over() to the OnOverCreditLimit event. Now look at the Purchase() method in Customer. The second "if" statement checks to see if there are any subscribers to the OnOverCreditLimit and if there are, it calls them with two parameters: this and amount. This refers to the current object, the current Customer object in this case. You can see that the first "if" statement checks to see if the purchase would exceed the credit limit and either completes the transaction or triggers the event. Pretty neat!

## Do it Yourself

I'm sure you're as excited as I am about the prospects of writing component-based software with C#. So go to http://msdn.microsoft.com/net/ where you'll find the ".NET Framework SDK Beta 1" download. The download contains everything you need to compile the code in this column. If you're more ambitious, you can visit: http://msdn.microsoft.com/vstudio/nextgen/beta.asp where you can order then entire Visual Studio .NET Beta 1 (including the Framework SDK if you don't want to download 111 MB!) on CD-ROM for the price of shipping, $12.99.

See sharper and see you next month,
Karl Gunderson
Technical Evangelist