The .NET Framework and C#

What is .NET?

When I first came across the Microsoft .NET concept some years ago, I found difficult to grasp the idea behind the pile of methodologies and technologies that Microsoft tried to introduce as .NET.
At first, the term .NET was used to identify a complete set of new products that Microsoft was working on: a new developer environment (Visual Studio .NET), a new operating system (Windows .NET server), a new database (SQL Server .NET), and most important a new way of envisioning the creation of software.
Five years later since my first contact with .NET, I have found that then branding use of .NET has been dropped so we shouldn't expect to see any Windows .NET server operating system of database. The actual official definition is the following: "Microsoft® .NET is a set of software technologies for connecting information, people, systems, and devices. This new generation of technology is based on Web services—small building-block applications that can connect to each other as well as to other, larger applications over the Internet."[1]
This definition is still a bit general and in my opinion does not tell us much about what .NET is really about.

The .NET Framework is a platform for building and running applications (Prosise, 2002). As its core, the framework has a common language runtime (CLR) and the .NET Framework class library (FCL). (Robinson, 2001)

The **CLR is like a virtual machine that abstracts operating system services** and serves as an execution engine for managed

[1] http://www.microsoft.com/net/basics/

applications. The **FCL is a library of classes** that provides the object-oriented API that managed applications write to. When writing .NET Framework applications, all the Windows API, and technologies like MFC, ATL and COM are left behind and the FCL is used instead (Prosise, 2002).

The framework brings a new API for programming on Windows, totally apart from all the technologies that had been evolving during the years and that suffered the burden of trying to maintain backward compatibility with its predecessor technologies (Robinson, 2001).

Why .NET?

> "A running program is often referred to as a virtual machine - a machine that doesn't exist as a matter of actual physical reality. The virtual machine idea is itself one of the most elegant in the history of technology and is a crucial step in the evolution of ideas about software. To come up with it, scientists and technologists had to recognize that a computer running a program isn't merely a washer doing laundry. A washer is a washer whatever clothes you put inside, but when you put a new program in a computer, it becomes a new machine.... The virtual machine: A way of understanding software that frees us to think of software design as machine design."
> From David Gelernter's "Truth, Beauty, and the Virtual Machine," *Discover Magazine*, September 1997, p. 72.

It didn't take long before somebody realized that the idea of the virtual machine could be applied to a whole operating system. This idea is what's behind technologies like the Java Virtual Machine developed by Sun Microsystems. There are many advantages of abstracting a whole operating system, for example portability and ease of design for software targeted for the virtual OS.

In today's world of massive information sharing and radical hardware technology changes, it becomes really handy to have common platforms for running software that are not tied to a particular kind of hardware.

The .NET framework is Microsoft's respond to this need for abstraction of a virtual operating system.

.NET Framework Components

The first component of the framework is a library, an API as extensive as the Windows API. This API can be used to call up all the same sorts of features that have traditionally been the role of the Windows operating system, for example: displaying of dialog boxes, windows, creating threads, etc (Petzold, 1998).

In addition to the basic functionality, the API addresses newer areas such as database access and web service providing. Unlike prior Windows APIs which consisted of huge sets of C function calls, the .NET library is fully object oriented. It is exposed as a set of objects, each of which implements a number of methods (Prosise, 2002).

The second component that completes the framework is the environment in which the programs run. This environment is known as the Common Language Runtime (CLR). When a program intended for the .NET framework is executed, it will be the .NET runtime which starts up the code, manages the running threads, provides various background services, and in a real sense is the immediate environment seen by the program. Therefore the .NET framework from the code point of view is an abstraction of the operating system.[2] The code that runs under the CLR is referred as “managed code”. The CLR also provides means for “managed code” to interact with pieces of “unmanaged code” that don’t run under the framework’s boundaries. Another important feature of the CLR is its garbage collector which constitutes the .NET’s answer to memory management, and in

[2] http://www.microsoft.com/net/basics/framework.asp

particular to reclaiming memory that is not longer needed by running applications.[3]

Advantages of the .NET framework

1. Object Oriented Programming

The .NET framework and C# are entirely based on object oriented principles right from the start (Ritchter, 2002). In particular, the framework's library is a library of classes instead of a library of functions. The classes can be instantiated and their member methods can be called. These object orientation makes it easier to write well structured and maintainable code.

2. Language Independence and Interoperability

Although in the past COM components could communicate with each other no matter which COM aware languages each one had been written in, there was still a gap between Visual Basic, Visual J++ and Visual C++. The data types were different, and making a COM component available to any language meant putting severe restrictions on its method signatures, often in a way that would affect performance. And it was certainly not possible to mix the various languages in the same code module. With .NET all of this has changed, in the framework, all languages compile to a common subset called the Intermediate Language (IL). In this sense .NET Framework is language-agnostic. It matters little what language is chosen to write code targeted to run under .NET, because in the end all languages exercise the same set of features in the Framework.

---

3 http://msdn.microsoft.com/netframework/using/understanding/netframework/default.aspx?pull=/library/en-us/dnnetdep/html/sidexsidenet.asp

.NET has common type specification, which means that the Intermediate Language comes with a set of predefined data types that are shared among all languages targeting the framework. The reason why the common type specification is important for language interoperability is that if a class is to derive from another class or contains instances of other classes; it needs to know about all data types used by these other classes, even if they were written in a different language (Prosise, 2002).

3. End to the “DLL Hell”

When first introduced, Dynamic link libraries (DLLs) presented a great way of saving disk space and memory and allowed different processes to share code. In time it has become apparent that DLLs have also given rise to a number of problems, largely due to both their lack of any formal system for versioning, and the fact that newer versions of a DLL usually overwrite older versions (Petzold, 1998).

The problem rises when some software overwrites a shared DLL with an updated version. If the updated version turns out not to be fully backward compatible, the result is that existing software on the machine that used the same DLL may no longer work.

In the .NET framework the way code is shared between applications has been completely revamped with the introduction of the concept of the “assembly”, which replaces the traditional DLL. Assemblies have formal facilities for versioning, and most important, different versions of the same assembly can co-exist in the same system (Macdonald, 2001).

4. C#

C# is the programming language most commonly associated with programming for the .NET framework. The C# compiler specifically targets .NET. The architecture and methodologies of C# reflect the underlying methodologies of .NET. For example, C# is based around single inheritance of classes, and has a type system that is based on the distinction between value and reference types, just as is the case for .NET. In many cases, specific language features of C# actually depend on features of .NET's base classes. One example of this is that all the basic data types in C# - int, float, string, etc are actually .NET base class types, which C# simply represents using a more convenient syntax (Robinson, 2001).

C# is a modern language because it supports the notion of data types, flow of control statements, operators, arrays, properties and exceptions. It is also an object-oriented language because it supports the notion of classes and their nature including encapsulation, inheritance and polymorphism. The notion of indexers is also supported in C#, allowing the manipulation of objects as arrays. Another important feature in C# is the introduction of delegates which can be described as method callbacks (Robinson, 2001).

Compilation and execution of code

Before the program can be run it must be compiled, and the compiled code presumably shipped. However, unlike previous executable and DLL files, the compiled code does not contain assembly language instructions. Rather, it contains instructions in Microsoft Intermediate Language (MSIL or simply IL).

IL has some similarities with the ideas of Java byte code. It is fairly low level language that has been designed, so it can be very rapidly converted into native machine code by a just-in-time (JIT) compiler (Prosise, 2002).

The compiled program consists of a number of assemblies. Each assembly contains the IL code, but it also contains metadata that describes the data types and methods in each assembly. The metadata also includes a simple hash of the assembly contents, which can be used to verify that the integrity of the assembly; it also includes version information and details of other assemblies that are called within the assembly (Ritchter, 2002).

The assembly that contains the main program entry point is marked as executable while the others assemblies are designated as libraries. When the program is executed, the .NET runtime first loads the assembly containing the main entry point. It uses the hash to verify the assembly's integrity and it uses the metadata to check through the defined types to ensure that it will be able to run the assembly (Ritchter, 2002).

The CLR then actually runs the code. It creates a process for the code to run in, and also marks out an application domain, in which it places the program's main thread. In some cases, a program may request instead, to be placed in the same process as some other running program, in which case the CLR will only create a new application domain for it.

The CLR takes the first portion of code that it needs to run and compile it from Intermediate Language into assembly language, and execute it from the appropriate program thread. Each time execution flow enters a new method that has not been executed before, that method will be compiled into executable code. This compilation however will only take

place once. Once the method has been compiled, the address of the entry points to that method will be replaced by the address of the compiled executable code. In this way, performance is maintained, because only those portions of the code that are actually used will be compiled. This process is known as just-in-time compiling. The JIT compiler may - depending on the compilation settings specified in the assembly-, optimize the code as is compiles, for example, by in lining some methods.

While the code is running, the CLR monitors its memory usage. Based on this monitoring it temporarily halts execution for short periods of time at certain points and call up the garbage collector, which examines the variables in the program in order to determine which areas of memory are still actively being used by the code, so that it can free up any unused areas (Robinson, 2001).

References

[Prosise, 2002] J. Prosise, “Programming Microsoft .NET (core reference),” Microsoft Press. ISBN 0-7356-1376-1

[Petzold, 1998] C. Petzold, “Programming Windows,” Microsoft Press. 5th Ed. ISBN 1-57231-995-X

[Ritchter, 2002] J. Ritchter, “Applied Microsoft .NET Framework Programming,” Microsoft Press. ISBN 0-7356-1422-9

[Robinson, 2001] S. Robinson et al, “Professional C#”, Wrox Press, 2001. ISBN 0-7645-4398-9

[Macdonald, 2001] R. Macdonald, “Understanding Assemblies”, Visual Basic Developer magazine July 2001. Pinnacle Publishing, Inc.