1 Introduction

We learn from failure, not from success!

— *Bram Stoker's* Dracula

The Microsoft Windows platform has evolved substantially over time. There have been clear ups and downs along the way, but Microsoft's platform has generally maintained a leadership position in the industry. The downs have been responsible for birthing the technologies that compose this book's table of contents. This chapter briefly discusses this inflection point and provides an overview of the architecture of technologies we discuss throughout the book. Chapter 2 begins our exploration with a look at the *Common Type System*, the foundation on top of which all code on the platform is built.

The History of the Platform

The introduction of Windows to the IBM-PC platform in 1985 revolutionized the way people interact with their computers. Most people think of this in terms of GUIs, mouse pointers, and snazzy new application interfaces. But what I'm actually referring to is the birth of the Windows application program interface (API). The 16-bit Windows APIs enabled you to do powerful new things to exploit the capability of the Windows platform, and offered new ways to deploy applications built on top of dynamic linking. About eight years later, in 1993, Windows NT was released, which had the first version of what is now known as the Win32 API. Aside from supporting 32-bit and thousands of new functions, the Win32 APIs were nearly identical to the Windows 1.0 APIs.

Traditionally, programming on the early Windows platform has been systems-level programming in C. But Windows programming in the late 1990s could be placed into one of three distinct categorized: systems, applications, and business scripting programming. Each category required the use of a different set of languages, tools, and techniques. This polarization grew over time, causing schisms between groups of Windows developers, and headaches for all involved.

For systems programming and very complex and robust applications, you wrote your code in C or C++, interacted directly with the Win32 programming model, and perhaps used something like COM (Component Object Model) to architect and distribute your reusable components. Memory management was in your face, and you had to be deeply familiar with the way Windows functioned. The separation between kernel-space and user-space, the difference between `USER32` and `GDI32`, among other things were need-to-know topics. Stacks, bits, bytes, pointers, and `HANDLE`s were your friend. (And memory corruption was your foe.)

But when they wanted to do applications development, many software firms utilized Visual Basic instead, which had its own simpler language syntax, a set of APIs, and a great development environment. Furthermore, it eliminated the need to worry about memory management and Windows esoterica. Mere mortals could actually program it. It interoperated well with COM, meaning that you could actually share code between systems and applications developers. *Visual Basic for Automation* (VBA) could be used to script business applications on top of *OLE Automation*, which represented yet another technology variant requiring subtly different tools and techniques.

And of course, if you wanted to enter the realm of web-application development, it meant using even different languages and tools, that is, VBScript or JScript, along with new technologies and APIs, that is, "classic" *Active Server Pages* (ASP). Web development began to rise significantly in popularity in the late 1990s in unison with the Internet boom, soon followed by the rise of XML and web services. The landscape became even more fragmented.

Meanwhile, Sun's Java platform was evolving quite rapidly and converging on a set of common tools and technologies. Regardless of whether you were writing reusable components, client or web applications, or scripting, you used the same Java language, Java Development Kit (JDK) tools and IDEs, and Java Class Libraries (JCL). A rich ecosystem of open source libraries and tools began to grow over time. In comparison, it was clear that Windows development had become way too complex. As this realization began to sink in industry-wide, you began to see more and more Microsoft customers moving off of Windows-centric programming models and on to Java. This often included a move to the L-word (Linux). Worse yet, a new wave of connected, data-intensive applications was on the horizon. Would Java be the platform on which such applications would be built? Microsoft didn't think so. A solution was desperately needed.

Enter the .NET Framework

The .NET Framework was an amazing convergence of many technologies — the stars aligning if you will — to bring a new platform for Windows development, which preserved compatibility with Win32 and COM. A new language, C#, was built that provided the best of C++, VB, and Java, and left the worst behind. And of course, other languages were written and implemented, each of which was able to take advantage of the full platform capabilities.

Two new application programming models arose. Windows Forms combined the rich capabilities of MFC user interfaces with the ease of authoring for which Visual Basic forms were heralded. ASP.NET reinvented the way web applications are built in a way that Java still hasn't managed to match. Extensible Markup Language (XML), this whacky new data interchange format (at the time), was deeply integrated into everything the platform had to offer, in retrospect a very risky and wise investment. A new communication platform was built that used similarly crazy new messaging protocols, labeled under the term *web services*, but that still integrated well with the COM+ architecture of the past. And of course, every single one of these technologies was built on top of the exact same set of libraries and the exact same runtime environment: the .NET Framework and the Common Language Runtime (CLR), respectively.

Now, in 2006, we sit at another inflection point. Looking ahead to the future, it's clear that applications are continuing to move toward a world where programs are *always connected* in very rich ways, taking advantage of the plethora of data we have at our disposal and the thick network pipes in between us. Presentation of that data needn't be done in flat, boring, 2-dimensional spreadsheets any longer, but rather can take advantage of powerful, malleable, 3-dimensional representations that fully exploit the graphics capabilities of modern machines. Large sets of data will be sliced and diced in thousands of different ways, again taking advantage of the multiprocessor and multi-core capabilities of the desktops of the future. In short: the .NET Framework version 2.0 released in 2005 will fuel the wave of Windows Vista and WinFX technologies, including the Windows Presentation, Communication, and Workflow Foundations that are in the not-so-distant future. It enables programmers on the CLR to realize the *Star Trek* wave of computing that sits right in front of us.

.NET Framework Technology Overview

The .NET Framework is factored into several components. First, the Common Language Runtime (CLR) is the *virtual execution environment* — sometimes called a *virtual machine* — that is responsible for executing *managed code*. Managed code is any code written in a high-level language such as C#, Visual Basic, C++/CLI, IronPython, and the like, which is compiled into the CLR's binary format, an *assembly*, and which represents its executable portion using *Intermediate Language* (IL). Assemblies contain self-descriptive program metadata and instructions that conform to the CLR's type system specification. The CLR then takes this metadata and IL, and compiles it into executable code. This code contains hooks into CLR services and Win32, and ultimately ends up as the native instruction set for the machine being run on. This happens through a process called *just-in-time* (JIT) compilation. The result of that can finally be run.

Then of course, the .NET Framework itself, a.k.a. WinFX, or commonly referred to simply as "the Framework," is the set of platform libraries and components that constitute the .NET API. In essence, you can think of WinFX as the next Win32. This includes the Base Class Libraries (BCL), offering ways to utilize Collections, I/O, networking, among others. A complex stack of libraries is built on top of the BCL, including technologies like ADO.NET for database access, XML APIs to manipulate XML data, and Windows Forms to display rich user interfaces (UIs).

Lastly, there are hosts that can run managed code in a specialized environment. ASP.NET, for example, is a combination of hosted environment and libraries that sit on top of the BCL and CLR. The ASP.NET host extends the functionality of the CLR with runtime policies that make sense for web applications, in addition to offering services like integration with Internet Information Services (IIS) so that IIS can easily dispatch a request into ASP.NET's web processing pipeline. SQL Server 2005 and Internet Explorer are two other examples of native applications that can host the CLR in process.

Figure 1-1 depicts the stack of technologies at a very broad level, drilling into the CLR itself in a bit more detail. This diagram obviously simplifies the number of real working parts, but can help in forming an understanding of their conceptual relationships.

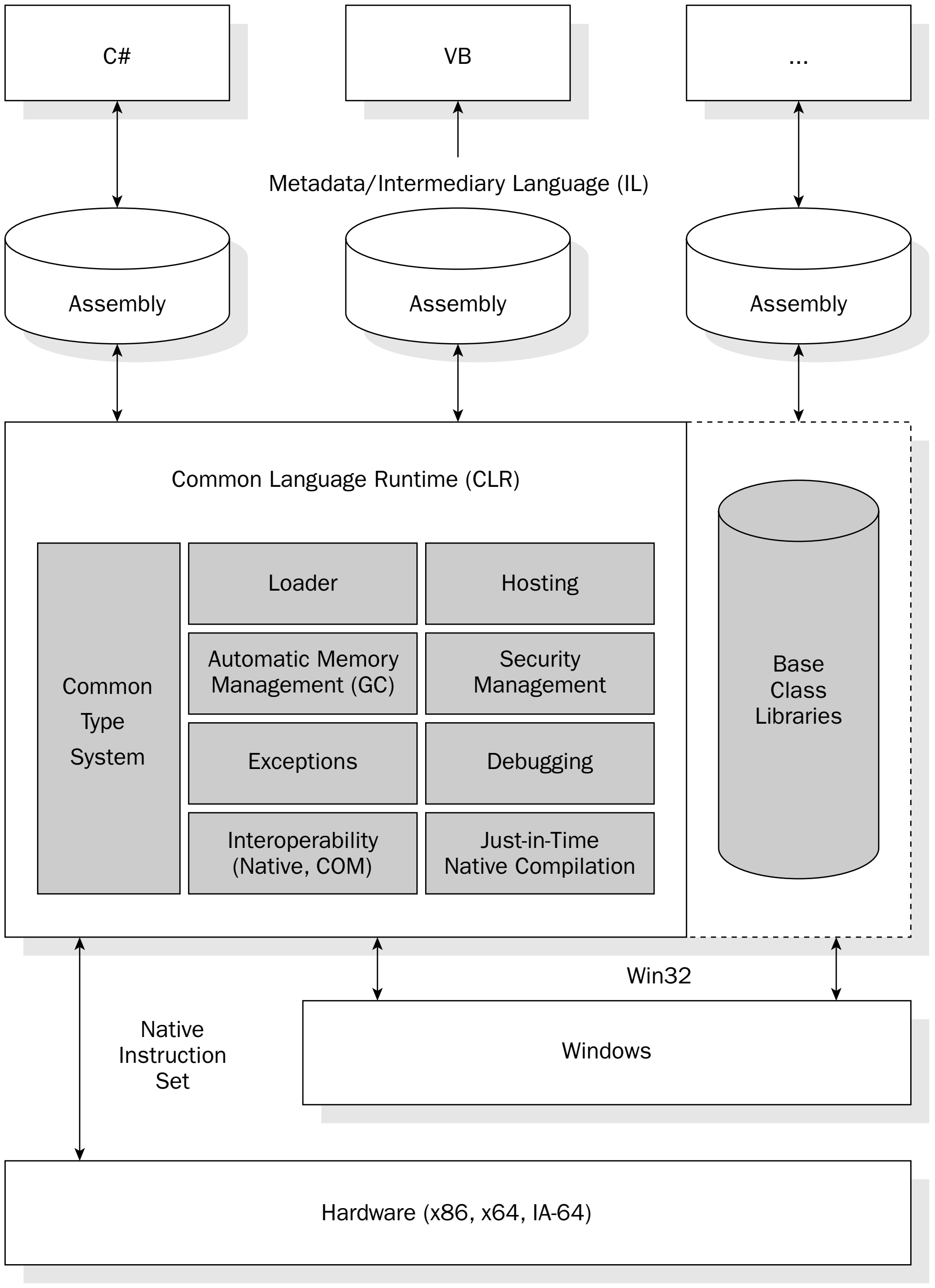
C#
VB
...
Metadata/Intermediary Language (IL)
Assembly
Assembly
Assembly
Common Language Runtime (CLR)
Common Type System
Loader
Hosting
Automatic Memory Management (GC)
Security Management
Exceptions
Debugging
Interoperability (Native, COM)
Just-in-Time Native Compilation
Base Class Libraries
Win32
Native Instruction Set
Windows
Hardware (x86, x64, IA-64)

Figure 1-1: Overview of the Common Language Runtime (CLR).

This book looks in depth at every single component on that diagram.

Key Improvements in 2.0

There are many key improvements in version 2.0 of the .NET Framework and CLR. Listing them all here would be impossible. They are, of course, mentioned as we encounter them throughout the book. I'll highlight some of the "big rocks" right here — as they're called inside Microsoft — that consumed a significant portion of the team's effort during the 2.0 product cycle.

- *Reliability*: Along with 2.0 of the CLR came hosting in process with SQL Server 2005. A host like SQL Server places extreme demands on the CLR in terms of robustness and failure mechanisms. Nearly all of the internal CLR guts have been hardened against out-of-memory conditions, and a whole set of new reliability and hosting features has been added. For example, `SafeHandle` makes writing reliable code possible, and constrained execution regions (CERs) enable the developers of the Framework to engineer rock-solid libraries, just to name a few. The hosting APIs in 2.0 enable a sophisticated host to control countless policies of the CLR's execution.
- *Generics*: This feature has far-reaching implications on the type system of the CLR, and required substantial changes to the languages (e.g., C# and VB) in order to expose the feature to the programmer. It provides a higher-level programming facility with which to write generalized code for diverse use cases. The .NET Framework libraries have in some cases undergone substantial rewrites — for example, `System.Collections.Generic` — in order to expose features that exploit the power of generics. In most cases, you'll notice the subtle mark, for example in the case of `System.Nullable<T>`, enabling things that simply weren't possible before.
- *64-bit*: Both Intel and AMD have begun a rapid shift to 64-bit architectures, greatly increasing the amount of addressable memory our computers have to offer. There is a 64-bit native .NET Framework SKU, which now ships with JIT compilers targeting the specific 64-bit instruction sets. Your old code will just work in the new environment thanks to WOW64 (Windows-on-Windows64), which is substantially more reliant when compared to the migration nightmares from 16-bit to 32-bit just over 10 years ago.

Of course, there is much, much more. While this book takes a broad view of the platform, plenty of 2.0 features are highlighted and discussed in depth in this book.