.NET Fundamentals

What kind of problems is .NET designed to solve? .NET solves problems that have plagued programmers in the past. .NET helps programmers develop the applications of the future. This chapter is designed to present an overview of Microsoft .NET by looking at a simple program rather than talking in vague generalities. While we will start discussing Microsoft .NET in detail in Chapter 6, this chapter will enable you to get a feel for the big picture right away.

Problems of Windows Development

Imagine a symphony orchestra where the violins and the percussion sections had different versions of the score. It would require a heroic effort to play the simplest musical composition. This is the life of the Windows developer. Do I use MFC? Visual Basic or C++? ODBC or OLEDB? COM interface or C style API? Even within COM: do I use IDispatch, dual, or pure vtable interfaces? Where does the Internet fit into all of this? Either the design had to be contorted by the implementation technologies that the developers understood, or the developers had to learn yet another technological approach that was bound to change in about two years.

Deployment of applications can be a chore. Critical entries have to be made in a Registry that is fragile and difficult to back up. There is no good versioning strategy for components. New releases can break existing programs often with no information about what went wrong. Given the problems with the Registry, other technologies used other configuration stores such as a metabase or SQL Server.

Security in Win32 is another problem. It is difficult to understand and difficult to use. Many developers ignored it. Developers who needed to apply security often did the best they could with a difficult programming model. The rise of Internet-based security threats transforms a bad situation into a potential nightmare.

Despite Microsoft's efforts to make development easier problems remained. Many system services had to be written from scratch, essentially providing the plumbing code that had nothing to do with your business logic. MTS/COM+ was a giant step in the direction of providing higher level services, but it required yet another development paradigm. COM made real component programming possible. Nonetheless, you either did it simply, but inflexibly in Visual Basic, or powerfully, but with great difficulty in C++, because of all the repetitive plumbing code you had to write in C++.

Applications of the Future

Even if .NET fixed all the problems of the past, it would not be enough. One of the unchanging facts of programming life is that the boundaries of customer demand are always being expanded.

The growth of the Internet has made it imperative that applications work seamlessly across network connections. Components have to be able to expose their functionality to other machines. Programmers do not want to write the underlying plumbing code, they want to solve their customers' problems.

.NET Overview

The Magic of Metadata

To solve all these problems .NET must provide an underlying set of services that is available to all languages at all times. It also has to understand enough about an application to be able to provide these services.

Serialization provides a simple example. Every programmer at some time or another has to write code to save data. Why should every programmer have to reinvent the wheel of how to persist nested objects and complicated data structures? Why should every programmer have to figure out how to do this for a variety of data stores? .NET can do this for the programmer. Programmers can also decide to do it themselves if required.

To see how this is done, look at the **Serialize** sample associated with this chapter. For the moment ignore the programming details of C# which will be covered in the next three chapters, and focus on the concepts.

```
[Serializable] class Customer
{
  public string name;
  public long id;
}
class Test
{
  static void Main(string[] args)
  {
    ArrayList list = new ArrayList();

    Customer cust = new Customer();
    cust.name = "Charles Darwin";
    cust.id = 10;
    list.Add(cust);

    cust = new Customer();
    cust.name = "Isaac Newton";
    cust.id = 20;
    list.Add(cust);

    foreach (Customer x in list)
      Console.WriteLine(x.name + ": " + x.id);

    Console.WriteLine("Saving Customer List");
    FileStream s = new FileStream("cust.txt",
             FileMode.Create);
    SoapFormatter f = new SoapFormatter();
    f.Serialize(s, list);
    s.Close();

    Console.WriteLine("Restoring to New List");
    s = new FileStream("cust.txt", FileMode.Open);
    f = new SoapFormatter();
    ArrayList list2 = (ArrayList)f.Deserialize(s);
    s.Close();

    foreach (Customer y in list2)
      Console.WriteLine(y.name + ": " + y.id);
  }
}
```

We have defined a **Customer** class with two fields: a **name** and an **id**. The program first creates an instance of a collection class that will be used to hold instances of the **Customer** class. We add two **Customer** objects to the collection and then print out the contents of the collection. The collection is then saved to disk. It is restored to a new collection instance and printed out. The results printed out will be identical to those printed out before the collection was saved.[1]

We wrote no code to indicate how the fields of the customer object are saved or restored. We did have to specify the format (SOAP) and create the medium to which the data was saved. The .NET Framework classes are partitioned so that where you load/save, the format you use to load/save, and how you load/save can be chosen independently. This kind of partitioning exists throughout the .NET Framework.

The **Customer** class was annotated with the **Serializable** attribute in the same way the **public** attribute annotates the name field. If you do not want your objects to be serializable, do not apply the attribute to your class. If an attempt is then made to save your object, an exception will be thrown and the program will fail.[2]

Attribute-based programming is used extensively throughout .NET to describe how the Framework should treat code and data. With attributes you do not have to write any code; the Framework takes the appropriate action based on the attribute. Security can be set through attributes. You can use attributes to have the Framework handle multithreading synchronization. Remoting of objects becomes straightforward through the use of attributes.

The compiler adds this **Serializable** attribute to the *metadata* of the **Customer** class to indicate that the Framework should save and restore the object. Metadata is additional information about the code and data within a .NET application. Metadata, a feature of the Common Language Runtime, provides such information about the code as:

- Version and locale information
- All the types
- Details about each type, including name, visibility, and so on
- Details about the members of each type, such as methods, the signatures of methods, and the like
- Attributes

Since metadata is stored in a programming-language-independent fashion with the code, not in a central store such as the Windows Registry, it makes .NET applications self-describing. The metadata can be queried at runtime to

[1] The sample installation should have already built an instance that you can run. If not, double-click on the Visual Studio.NET solution file that has the .sln suffix. When Visual Studio comes up, hit Control-F5 to build and run the sample.

[2] Comment out the **Serializable** attribute in the program (you can use the C/C++/* */ comment syntax) and see what happens.

get information about the code (such as the presence or absence of the **Serializable** attribute). You can extend the metadata by providing your own custom attributes.

In our example, the Framework can query the metadata to discover the structure of the **Customer** object in order to be able to save and restore it.

Types

Types are at the heart of the programming model for the CLR. A type is analogous to a class in most object-oriented programming languages, providing an abstraction of data and behavior, grouped together. A type in the CLR contains:

Fields (data members)
Methods
Properties
Events

There are also built-in primitive types, such as integer and floating point numeric types, string, etc. We will discuss types under the guise of classes and value types when we cover C#.

.NET Framework Class Library

The **Formatter** and **FileStream** classes are just two of more than 2500 classes in the .NET Framework that provide plumbing and system services for .NET applications. Some of the functionality provided by the .NET Framework includes:

- Base class library (basic functionality such as strings, arrays, and formatting)
- Networking
- Security
- Remoting
- Diagnostics
- I/O
- Database
- XML
- Web services that allow us to expose component interfaces over the Internet
- Web programming
- Windows User Interface

Interface-Based Programming

Suppose you want to encrypt your data and therefore do not want to rely on the Framework's serialization. Your class can inherit from the **ISerializable** interface and provide the appropriate implementation. (We will discuss how to do this in a later chapter.) The Framework will then use your methods to save and restore the data.

How does the Framework know that you implemented the **ISerializable** interface? It can query the metadata related to the class to see if it implements the interface! The Framework can then use either its own algorithm or the class's code to serialize or deserialize the object.

Interface-based programming is used in .NET to allow your objects to provide implementations to standard functionality that can be used by the Framework. Interfaces also allow you to program using methods on the interface rather than methods on the objects. You can program without having to know the exact type of the object. For example, the formatters (such as the SOAP formatter used here) implement the **IFormatter** interface. Programs can be written using the **IFormatter** interface and thus are independent of any particular current (binary, SOAP) or future formatter and still work properly.

Everything Is an Object

So if a type has metadata, the runtime can do all kinds of wonderful things. But does everything in .NET have metadata? Yes! Every type, whether it is user defined (such as **Customer**) or part of the Framework (such as **FileStream**), is a .NET object. All .NET objects have the same base class, the system's **Object** class. Hence everything that runs in .NET has a type and therefore has metadata.

In our example, the serialization code can walk through the **ArrayList** of customer objects and save each one as well as the array it belongs to, because the metadata allows it to understand the object's type and its logical structure.

Common Type System

The .NET Framework has to make some assumptions about the nature of the types that will be passed to it. These assumptions are the *Common Type System* (CTS). The CTS defines the rules for the types and operations that the Common Language Runtime will support. It is the CTS that limits .NET classes to single implementation inheritance. Since the CTS is defined for a wide range of languages, not all languages need to support all features of the CTS.

The CTS makes it possible to guarantee type safety, which is critical for writing reliable and secure code. As we noted in the previous section, every object has a type and therefore every reference to an object points to a

defined memory layout. If arbitrary pointer operations are not allowed, the only way to access an object is through its public methods and fields. Hence it's possible to verify an object's safety by analyzing the object. There is no need to know or analyze all the users of a class.

How are the rules of the CTS enforced? The Microsoft Intermediate Language (MSIL or IL) defines an instruction set that is used by all .NET compilers. This intermediate language is platform independent. The MSIL code can later be converted to a platform's native code. Verification for type safety can be done once based on the MSIL; it need not be done for every platform. Since everything is defined in terms of MSIL, we can be sure that the .NET Framework classes will work with all .NET languages. Design no longer dictates language choice; language choice no longer constrains design.

MSIL and the CTS make it possible for multiple languages to use the .NET Framework since their compilers produce MSIL. This one of the most visible differences between .NET and Java, which in fact share a great deal in philosophy.

ILDASM

The Microsoft Intermediate Language Disassembler (ILDASM) can display the metadata and MSIL instructions associated with .NET code. It is a very useful tool both for debugging and for increasing your understanding of the .NET infrastructure. You can use ILDASM to examine the .NET Framework code itself.[3] Figure 2–1 shows a fragment of the MSIL code from the **Serialize** example, where we create two new customer objects and add them to the list.[4] The **newobj** instruction creates a new object reference using the constructor parameter.[5] **Stloc** stores the value in a local variable. **Ldloc** loads a local variable.[6] It is strongly recommended that you play with ILDASM and learn its features.

[3] ILDASM is installed on the Tools menu in Visual Studio.NET. It is also found in the Microsoft.NET\FrameworkSDK\Bin subdirectory. You can invoke it by double-clicking on its Explorer entry or from the command line. If you invoke it from the command line (or from VS.NET) you can use the /ADV switch to get some advanced options.

[4] Open Serialize.exe and Click on the plus (+) sign next to Test. Double-click on Main to bring up the MSIL for the Main routine.

[5] Technically it is not a parameter. IL is a stack-based language, and the constructor is a metadata token previously pushed on the stack.

[6] You can read all about MSIL in the ECMA documents, specifically the Partition III CIL Instruction Set.

Figure 2–1 *Code fragment from Serialize example.*

Language Interoperability

Having all language compilers use a common intermediate language and common base class make it *possible* for languages to interoperate. But since all languages need not implement all parts of the CTS, it is certainly possible for one language to have a feature that another does not.

The *Common Language Specification* (CLS) defines a subset of the CTS representing the basic functionality that all .NET languages should implement if they are to interoperate with each other. This specification enables a class written in Visual Basic.NET to inherit from a class written in COBOL.NET or C#, or to make interlanguage debugging possible. An example of a CLS rule is that method calls need not support a variable number of arguments, even though such a construct can be expressed in MSIL.

CLS compliance applies only to publicly visible features. A class, for example, can have a private member that is non-CLS compliant and still be a base class for a class in another .NET language. For example, C# code should not define public and protected class names that differ only by case-sensitivity, since languages such as VB.NET are not case-sensitive. Private fields could have case-sensitive names.

Microsoft itself is providing several CLS-compliant languages: C#, Visual Basic.NET, and C++ with Managed Extensions. Third parties are providing

additional languages (there are over a dozen so far). ActiveState is implementing Perl and Python. Fujitsu is implementing COBOL.

Managed Code

In the serialization example a second instance of the Customer object was assigned to the same variable (**cust**) as the first instance without freeing it. None of the allocated storage in the example was ever deallocated. .NET uses automatic garbage collection to reclaim memory. When memory allocated on the heap becomes orphaned, or passes out of scope, it is placed on a list of memory locations to be freed. Periodically, the system runs a garbage collection thread that returns the memory to the heap.

By having automatic memory management the system has eliminated memory leakage, which is one of the most common programming errors. In most cases, memory allocation is much faster with garbage collection than with classic heap allocation schemes. Note that variables such as **cust** and **list** are object references, not the objects themselves. This makes the garbage collection possible.

Garbage collection is one of several services provided by the *Common Language Runtime* (CLR) to .NET programs.[7] Data that is under the control of the CLR garbage collection process is called managed data. Managed code is code that can use the services of the CLR. .NET compilers that produce MSIL can produce managed code.

Managed code is not automatically type safe. C++ provides the classic example. You can use the __gc attribute to make a class garbage collected. The C++ compiler will prevent such classes from using pointer arithmetic. Nonetheless, C++ cannot be reliably verified.[8]

Code is typically verified for type safety before compilation. This step is optional and can be skipped for trusted code. One of the most significant differences between verified and unverified code is that verified code cannot

[7] Technically, metadata, the CTS, the CLS, and the Virtual Execution System (VES) are also part of the CLR. We are using CLR here in the sense that it is commonly used. The VES loads and runs .NET programs and supports late binding. For more details refer to the Common Language Infrastructure (CLI) Partition I: Concepts and Architecture document submitted to ECMA. This document is loaded with the .NET Framework SDK.

[8] The most immediate reason for this is that the C Runtime Library (CRT) that is the start-up code for C++ programs was not converted to run under .NET because of time constraints. Even if this were to be done, however, there are two other obstacles to verifying C++ code. First, to ensure that the verification process can complete in a reasonable amount of time, the CLR language specifications require certain IL language patterns to be used and the managed C++ compiler would have to be changed to accommodate this. Second, after disallowing the C++ constructs that inhibit verification (like taking the address of a variable on the stack, or pointer arithmetic), you would wind up with a close approximation to the C# language.

use pointers.[9] Code that used pointers could subvert the Common Type System and access any memory location.

Type safe code cannot be subverted. A buffer overwrite is not able to corrupt other data structures or programs. Methods can only start and end at well-defined entry and exit points. Security policy can be applied to type safe code.[10] For example, access to certain files or user interface features can be allowed or denied. You can prevent the execution of code from unknown sources. You can prevent access to unmanaged code to prevent subversion of .NET security. Type safety also allows paths of execution of .NET code to be isolated from one another.[11]

Assemblies

Another function of the CLR is to load and run .NET programs.

.NET programs are deployed as assemblies. An *assembly* is one or more EXEs or DLLs with associated metadata information. The metadata about the entire assembly is stored in the assembly's manifest. The manifest contains, for example, a list of the assemblies upon which this assembly is dependent.

In our Serialize example there is only file in the assembly, **serialize.exe**. That file contains the metadata as well as the code. Since the manifest is stored in the assembly and not in a separate file (like a type library or registry), the manifest cannot get out of sync with the assembly. Figure 2–2 shows the metadata in the manifest for this example.[12] Note the **assembly extern** statements that indicate the dependencies on the Framework assemblies **mscorlib** and **System.Runtime.Formatters.SOAP**. These statements also indicate the version of those assemblies that serialize.exe depends on.

Assemblies can be versioned, and the version is part of the name for the assembly. To version an assembly it needs a unique name. Public/private encryption keys are used to generate a unique (or strong) name.

Assemblies can be deployed either privately or publicly. For private deployment all the assemblies that an application needs are copied to the same directory as the application. If an assembly is to be publicly shared, an entry is made in the *Global Assembly Cache* (GAC) so that other assemblies can locate it. For assemblies put in the GAC a strong name is required. Since the version is part of the assembly name, multiple versions can be deployed

[9] It would not be correct to say that code written in MSIL is managed code. The CTS permits MSIL to have unmanaged pointers in order to work with unmanaged data in legacy code. The reverse is not true; unmanaged code cannot access managed data. The CLS prohibits unmanaged pointers.

[10] This is discussed in more detail in Chapter 12.

[11] See the discussion of Application Domains in Chapter 8.

[12] Open **serialize.exe** in ILDASM and double-click on the MANIFEST item.

Figure 2–2 *Manifest for the Serialize assembly.*

side by side on the same machine without interfering with each other. Whether you use public or private deployment there is no more "DLL Hell."[13]

Assembly deployment with language interoperability makes component development almost effortless.

JIT Compilation

Before executing on the target machine, MSIL has to be translated into the machine's native code. This can either be done before the application is called, or at runtime. At runtime, the translation is done by a just-in-time (JIT) compiler. The Native Image Generator (Ngen.exe) translates MSIL into native code so that it is already translated when the program is started.

The advantage of pretranslation is that optimizations can be performed. Optimizations are generally impractical with JIT because the time it takes to do the optimization can be longer than it takes to compile the code. Start-up time is also faster with pretranslation because no translation has to be done when the application starts.

[13] This is discussed in much more detail in Chapter 7.

The advantage of JIT is that it knows what the execution environment is when the program is run and can make better assumptions, such as register assignments, when it generates the code. Only the code that is actually executed is translated, code that never gets executed is never translated.

In the first release of .NET, the Native Image Generator and the JIT compiler use the same compiler. No optimizations are done for Ngen, its only current advantage is faster start-up. For this reason we do not discuss Ngen in this book.

Performance

You may like the safety and ease-of-use features of managed code but you might be concerned about performance. Early assembly language programmers had similar concerns when high-level languages came out.

The CLR is designed with high performance in mind. With JIT compilation, the first time a method is encountered, the CLR performs verifications and then compiles the method into native code (which will contain safety features, such as array bounds checking). The next time the method is encountered, the native code executes directly. Memory management is designed for high performance. Allocation is almost instantaneous, just taking the next available storage from the managed heap. Deallocation is done by the garbage collector, which has an efficient multiple-generation algorithm.

You do pay a penalty when security checks have to be made that require a stack walk as we will explain in the Security chapter.

Web pages use compiled code, not interpreted code. As a result ASP.NET is much faster than ASP.

For 98% of the code that programmers write, any small loss in performance is far outweighed by the gains in reliability and ease of development. High performance server applications might have to use technologies such as ATL Server and C++.

Summary

.NET solves the problems that have plagued Windows development in the past. There is one development paradigm for all languages. Design and programming language choices are no longer in conflict. Deployment is more rational and includes a versioning strategy. While we will talk more about it in later chapters, metadata, attribute-based security, code verification, and type-safe assembly isolation make developing secure applications much easier. The plumbing code for fundamental system services is provided, yet you can extend or replace it if you must.

The Common Language Runtime provides a solid base for developing applications of the future. The CLR is the foundation whose elements are the Common Type System, metadata, the Common Language Specification, and the Virtual Execution System (VES) that executes managed code.[14] As we shall see in future chapters, .NET makes it easier to develop Internet applications for both service providers and customer-based solutions. With the unified development platform .NET provides, it will be much easier than in the past for Microsoft or others to provide extensions.

All this is made possible by putting old technologies together in the CLR creatively: intermediate languages, type-safe verification, and of course, metadata. As you will see, metadata is used in many features in .NET.

We shall expand on these topics in the course of the book. We next cover the C# language. Depending on your knowledge of C#, you might be able to skim Chapter 3 and skim Chapters 4 and 5. Chapter 4 introduces the Acme Travel Agency case study, which is used throughout the book. Chapter 5 covers important topics about the interaction of C# and the .NET Framework.

[14] The Base Class Libraries classes (BCL) are also part of the CLR.