

# What is Microsoft .NET and Why Should I Care?

Michael Stiefel

Reliable Software, Inc.

co-author “Application Development Using C# and .NET”

# Why Microsoft .NET?

- .NET provides a new development paradigm that
  - is identical for all languages and technologies.
  - allows for the development of solutions for heterogeneous computing environments.
  - can be expanded to encompass future development strategies, technologies, and customer demands.

# Past Development Paradigms

- Do I use MFC? Visual Basic or C++?
- Scripting languages for Web development
- COM interface or C style API?
  - Use IDispatch, dual, or pure vtable interfaces?
- Fragile Registry
  - No Versioning strategy, component dependencies
  - DLL Hell, new installations break old ones
  - Registry, SQL Server, IIS Metabase alternative configuration stores.
- Security
- MTS/COM+ - another way to do things.

# .NET Interoperability

- A Variety of Computing Platforms Exist:
  - Microsoft Platforms
  - Unix of Varying Types (including Linux)
  - Legacy Systems (IBM Mainframes)
- A Variety of Application Environments:
  - Web Services
  - Web Applications
- Use Industry Standards: XML, HTTP, SOAP, etc.

# Future Developments

- Increase Coverage of Win32 API
- PDA
- Tablet PC
- Smart Devices
- ???

# .NET Programming Model

- The .NET platform has a virtual execution environment.
  - Common Language Runtime (CLR)
  - Common Type System (CTS)
  - Common Intermediate Language (CIL)
- Nonetheless, a program executes as native code of the underlying hardware platform.

# Common Type System

- Defines legal types and rules
  - Single implementation, multiple interface inheritance
  - Standard exception handling
  - Primitive types
  - Reference and value types
  - Interfaces, delegates, events
- Metadata (required)
  - describes types independent of physical layout
  - specifies assembly version and dependencies
  - extensible through attributes
  - can be queried at runtime (reflection)

# Everything can be an Object

- Reference types inherits from the System.Object base type.
- Value types (int, long, float, etc.) are converted to objects when necessary
  - Boxing and Unboxing
  - No object overhead if not needed.
- Both reference and value types can have methods and fields
- System.ValueType inherits from System.Object.

# Verification for Type Safety

- A subset of the CTS can be used to write type-safe code that cannot be subverted.
  - No buffer overwrites
  - Method entry and exit at well defined points.
- Code verified before compilation.
- Security Policy applies to type safe code.
- Allows for application domains.
- CTS has unverifiable pointer arithmetic operations to handle legacy code.

# Common Language Specification

- No language supports all CTS features.
- The CLS defines a subset of the CTS to allow languages to interoperate
  - Applies to public features only
  - not all languages are case-sensitive so do not define class names that differ only by case.
  - no need to support methods with variable args
- Allows:
  - A VB.NET class inherits a COBOL.NET class
  - C# code calls a VB.NET method.
  - Interlanguage debugging.

# Common Intermediate Language

- The CIL has instructions to express CTS types, methods, and metadata.
  - stack based, operations based on type, no offsets or other physical characteristics in CIL
  - compilers produce CIL code
  - some produce type safe code only, others can use unverifiable constructs (i.e. pointers) to handle legacy code
  - verification operates on the IL code so that type-safety can be assured irrespective of ultimate hardware platform

# Sample IL to Add Two Numbers

```
.assembly extern mscorlib {}  
.assembly addtest { .ver 1:0:0:0 }  
.module add.exe  
.method private static void Main() cil managed {  
  .entrypoint  
  .maxstack 2  
  .locals init (int32 a, int32 b, int32 sum)  
  ldc.i4.s 100  
  stloc a  
  ldc.i4.s 25  
  stloc b  
  ldloc a  
  ldloc b  
  add  
  stloc c  
  ldloc c  
  call void [mscorlib]System.Console::WriteLine(int32)  
  ret }  
}
```

# Conversion to Native Code

- MSIL has to be converted to native code.
  - Logical to physical translation occurs here.
- Just-In-Time (JIT) compilation.
  - Knows execution environment (registers)
  - No optimizations
  - Only executed code is translated.
- Native Image Generator (Ngen).
  - Optimizations (not in first release)
  - Faster Start up

# ILDASM

- Intermediate Language Disassembler displays metadata and MSIL instructions.
- Useful for debugging and understanding.
  - Can use it to understand Framework code.
  - Can display metadata associated with code and assembly.

# Common Language Runtime

- Code using the services of the CLR is called managed code.
- Provides managed execution services:
  - garbage collection (managed data)
  - code access security
  - loading the correct component version
- Metadata gives CLR complete knowledge about executing program.
- Managed code is not automatically type-safe.
  - C++

# Win32 Application Isolation

- Win32 has each application in a separate process.
  - Each process has its own address space.
  - A process context switch is implemented in the processor.
- Processes are inefficient for really scalable solutions (tens to hundreds of thousands).
  - Process switches are slow.
  - Interprocess communication is much slower than intraprocess communication.

# Application Domains

- The CLR uses Application Domains to provide isolation in software.
  - A process can have many app domains.
- Type-safety means methods can only be accessed in well-defined ways. The CLR prevents direct access to types from one app domain to another.
  - Application Domains can use multiple assemblies.
- Each Application Domain can have its own:
  - Security Evidence
  - Configuration Information
- ASP.NET is an example.

# CLR Hosting

- If you need to use .NET from within a legacy application, you can write a CLR Host.
- A CLR Host must load the CLR, setup the application domains, and then execute their code.
- ASP.NET uses an ISAPI filter to start the CLR and load Web applications.
- Yukon will use CLR Hosting so that you can write stored procedures in .NET languages.

# Assemblies

- Unit of Deployment as one or more files
- Contain metadata, code, and resources
  - Metadata stored with assembly (self-describing)
- Metadata includes version and dependencies on other components and complete description of types in the assembly.
- Type definitions are assembly relative.

# Assembly Version Policy

- Version is part of the assembly name.
  - Unique name based on public/private keys.
  - Version equivalences specified in configuration files.
  - Makes side-by-side deployment possible.
- Private deployment
  - Copy all files to application directory.
  - No need for versioning or unique names.
- Public deployment in *Global Assembly Cache (GAC)* requires strong name.
  - No more “DLL Hell”.

# .NET Framework Class Library

- Base class library (strings, arrays, formatting).
- Networking
- Security
- Remoting
- Diagnostics
- I/O
- Database
- XML
- Web services and Web programming
- Windows User Interface

# Legacy Interop

- .NET code can make calls to Win32 functions including the standard Win32 DLLs.
- .NET can make calls to COM components.
- COM components can make calls into to .NET code.

# Development Tools

- Visual Studio.NET
  - compilers, debuggers, tools, editor, wizards
  - drag and drop design for Web and Windows Forms that work identically for all languages
- XML graphical tools
- Database Server Explorer
- Platform SDK with samples

# Serialization Example

- Common programmer task.
- Sample code serializes a customer class.
  - two customer objects are created.
  - objects are added to a collection.
  - objects are saved to and restored from disk.
- No serialization code written
  - only disk format and location were specified
- No save/restore code was written.

# Customer Value Type

```
[Serializable]
class Customer
{
    public string name;
    public long id;
}
```

# Main Code

```
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);
```

```
...
```

```
FileStream s = new FileStream("cust.txt", FileMode.Create);  
IFormatter f = new SoapFormatter();  
SaveFile(s, f, list);
```

```
s = new FileStream("cust.txt", FileMode.Open);  
f = new SoapFormatter();  
ArrayList list2 = (ArrayList)RestoreFile(s, f);
```

# Persistence Code

```
public static void SaveFile(Stream s, IFormatter f, IList l)
{
    f.Serialize(s, l);
    s.Close();
}
```

```
public static IList RestoreFile(Stream s, IFormatter f)
{
    IList l = (IList)f.Deserialize(s);
    s.Close();
    return l;
}
```

# Attribute Based Programming

- No code was written to save/restore data.
  - Just specified format and location.
- Customer class has **Serializable** attribute.
  - Type creator uses this attribute to indicate that the framework can persist an instance of this type.
  - If attribute is missing, an exception is thrown.
- Attributes used through .NET.
  - Security settings
  - Multithreading synchronization.

# Metadata

- Framework classes use the type's metadata.
  - Serialize method uses metadata to save and restore an instance of the collection type and the type instances stored in the collection.
  - No attribute framework code throws an exception.
- Metadata contains type information.
  - Name, visibility
  - Fields, Methods, Properties, Events
  - Layout (not byte location)
  - Attributes (like Serializable)

# Everything is an Object

- Every reference in .NET derives from *System.Object*.
  - The `ArrayList.Add` can be used to create a heterogeneous collection.
  - In the sample, the framework can walk through the `ArrayList` of `Customer` objects and manipulate each member of the collection.
- All access to objects in .NET is through object references.
  - No more random pointers causing memory corruption.

# Framework Class Libraries

- This simple example uses three Framework classes that simplify your application development.
  - ArrayList
  - FileStream
  - SoapFormatter

# Interface Based Programming

- Interfaces allow you to work with abstract types in a way that allows for extensible programming.
  - Interfaces are a type in the Common Type System.
  - The *SaveFile* and *RestoreFile* routines are written using the *IList* and *IFormatter* interfaces.
  - Routines work with all collection class (**IList**) and all formatters (**IFormatter**).

# Interface Inheritance

- Interface inheritance permits code reuse without breaking the encapsulation of a base class.
- Many framework classes implement standard interfaces, program to the interface, rather than against an implementation.
  - allow for polymorphism
  - clients ignorant of implementation, if it changes, client programs unchanged
  - separate object creation from object use

# Custom Serialization

- If you do not want to rely on the Framework's serialization...
- Mark type with `Serializable` and override the **`ISerializable`** interface.
- The `Persist` method can query the metadata to see if the interface is implemented. If it is, the type's serialization is used rather than the default.

# ILDASM

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 3
    ....
    IL_0000: newobj      instance void
               [mscorlib]System.Collections.ArrayList::.ctor()
    IL_0005: stloc.0
    IL_0006: newobj      instance void Customer::.ctor()
    IL_000b: stloc.1
    IL_000c: ldloc.1
    IL_000d: ldstr       "Charles Darwin"
    IL_0012: stfld      string Customer::name
    IL_0017: ldloc.1
    IL_0018: ldc.i4.s    10
    IL_001a: conv.i8
    IL_001b: stfld      int64 Customer::id
    IL_0020: ldloc.0
    IL_0021: ldloc.1
    IL_0022: callvirt   instance int32
               [mscorlib]System.Collections.ArrayList::Add(object)
    ...
    IL_015a: ret
}
```

# Managed Memory Management

- In the serialization example we never deallocated any memory.
- .NET uses automatic garbage collection.
  - Memory passes out of scope or is orphaned, placed on a list of memory locations.
  - Periodically the system reclaims memory.
  - Generally faster allocation.
- Eliminates common programming errors.
- Objects are references, not object pointers.

# Assemblies

- Assembly metadata about the entire assembly is stored in the assembly's manifest.

```
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 1:0:2411:0
}
.assembly extern System.Runtime.Serialization.Formatters.Soap
{
    .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )
    .ver 1:0:2411:0
}
.assembly Serialize
{
    ...
    .hash algorithm 0x00008004
    .ver 1:0:606:39243
}
.module Serialize.exe
...
```

# Components

- Assembly deployment with language interoperability makes for easy component development.
  - An assembly is a component
  - No COM infrastructure to implement.
- *As*, *Is* C# constructs allow interface query.
  - *As* operator converts one interface type to another
  - *Is* operator checks if interface is supported
  - IsInst IL keyword, other languages, other syntax

# ASP.NET and Web Services

- ASP.NET applications can be written with any .NET language: C#, C++ with managed extensions, VB.NET, etc.
- You can use a Visual Studio.NET forms-based interface to create ASP.NET web pages.
  - This is the same interface used by .NET Windows Forms.
- The .NET Framework classes make it straightforward to build Web services using industry-standard protocols such as SOAP and WSDL.
  - You can also modify the standard implementations by using attributes, or plugging in your own support classes.

# Windows Forms

- NET also changes the programming paradigm for rich client applications.
- The .NET Windows Framework classes have a set of classes that can be used by all applications to build classic windows applications.

# Security

- Framework classes can encapsulate standard user identity-based authentication and authorization.
- Code Access Security allows you to place restrictions on executing code depending on its characteristics (evidence) such as name, URL of origin, or publisher.
- Code Access Security makes it easier to secure components from diverse sources that run under different user identities.
  - Security Policy is set by administrators by assigning permissions to assemblies based on its evidence.

# Implications for Managers

- Developers need to understand object-oriented development and design.
- Eventually, language issues recede into the background. You will hire .NET developers not, C# or VB.NET developers.
  - languages such as Perl or APL will continue to exist.
- Developers need to understand Code Access Security because administrators will be able to control whether your code will run.
- Since memory overwrites and memory leaks are the major cause of programming errors, development should be faster.
- Application deployment problems should diminish.

# Summary

- One development paradigm for all languages.
  - Metadata
  - Security
  - Type Safety
  - Attributes
  - Garbage Collection
- Language Interoperability
- Framework Class Library
- Rational deployment and versioning.