Reflection

Leveraging the Power of Metadata

www.csse.monash.edu.au/courseware/cse5510/Lectures/lecture3.ppt
Objectives

- Provide an introduction to .NET Reflection
- Explain how applications can use Reflection to explore type information and how they can build types at runtime
Contents

- Section 1: Overview
- Section 2: Exploring Metadata
- Section 3: Detail Information
- Section 4: Building Types at Runtime
- Section 5: Putting it together
- Summary
Section 1: Overview

- .NET Compile-Execution Cycle
- .NET Reflection Core Concepts
Execution model

Cobol | VB | C++ | C# | .NET languages

Language compilers

CIL code (+ metadata)

Loader/verifier

JIT compiler

Managed Code

Execution

Uncompiled method call
.Net Architecture

- Common Language Specification
- Base Class Library
- Data and XML
- Web Services, ASP.Net
- User Interface WinForms
- Visual Studio .NET
All types in the CLR are self-describing
- CLR provides a reader and writer for type definitions
- You can ‘read’ programs
- You can map between type systems
- You can interrogate objects and types inside a running program
.NET Reflection Core Concepts

- **Metadata**
  - Single location for type information and code
  - Code is literally contained within type information
  - Every .NET object can be queried for its type
  - Types' metadata can be explored with Reflection

- **Dynamic Type System**
  - Highly dynamic and language independent
  - Types may be extended and built at run-time
  - Allows on-the-fly creation of assemblies
  - .NET Compilers use .NET to emit .NET code
Reflection is fundamental to the CLR

- JVM
- COM
- Directory
- DBMS
- XML

- RMI
- Proxies
- Serialization
- Code Generation
- Analysis, Documentation
Metadata uses

- Allows type developed in a PL to be used by code in other PL
- GC uses to traverse objects
- Serialization (essential for Web Services)
- IDE (e.g. IntelliSense)
Section 2: Exploring Metadata

- Who are you?
- What are you?
- Anything special about you?
- Now tell me what you have!
- Types and Instances
Type Reflection

- `Savings.BaseType == Account`
- `Savings.BaseType.BaseType == Object`
- `Ixfer.BaseType == null`
- `Account.IsSubClassof(IEnquire) == false`
- `Savings.IsSubClassof(Account) == true`
- `Savings.GetInterfaces() == { Iprint, IXfer, Ienquire }`
- `Ienquire.IsAssignableFrom(Savings) == true`
Reflection and the CLR type system
public class Person {
    public event OnSaveChange onsv;
    public Date DOB;
    public string FirstName;
    public string LastName;
    public string Name {
        get {
            return FirstName + " " + LastName;
        }
    }
    public Person(string First, string Last) {
        FirstName = First; LastName = Last;
    }
    public bool Save() {
        System.Type t = this.GetType();
        foreach (FieldInfo f in t.GetFields()) {
            ... }
    }
}
Traverse every element in an assembly

```csharp
Using System.Reflection;

Public static void WalkAssembly(String assemblyName)
{
    Assembly assembly = Assembly.Load(assemblyName);
    foreach (Module module in assembly.GetModules())
    {
        foreach (Type type in module.GetTypes())
        {
            foreach (MemberInfo member in type.GetMembers())
            {
                Console.WriteLine("{0}.{1}", type, member.Name;
            }
        }
    }
}
Serialization

- Save state of an object on disk or send to a stream:

```java
public class Point {
    public double x;
    public double y;
    public double length; // computed from x, y
};
```
Solutions

- **Inherit from a base type:**
  ```
  class Point : Serializable {
      ...
  }
  ```
  but …
  - base type does not know derived type
  - requires multiple inheritance

- **Java solution:**
  ```
  class Point implements Serializable {
      ...
  }
  ```
  but …
  - method granularity
  - customization requires full rewrite of WriteObject, ReadObject
Solution: Attributes

```java
[Serializable]
public class Point :
{
    public double x;
    public double y;
    [NonSerialized]
    public double length; // computed from x, y
};
```
protected virtual void WriteMember(String memberName, Object data) {
    if (data == null) {
        WriteObjectRef(data, memberName, typeof(Object));
        return;
    }
    Type varType = data.GetType();
    if (varType == typeof(Boolean)) {
        WriteBoolean(Convert.ToBoolean(data), memberName);
    } else if (varType == typeof(Char)) {
        WriteChar(Convert.ToChar(data), memberName);
    } else if (varType == typeof(Byte)) {
        WriteByte(Convert.ToByte(data), memberName);
    } ...
    else {
        if (varType.IsArray) {
            WriteArray(data, memberName, varType);
        } else if (varType.IsValueType) {
            WriteValueType(data, memberName, varType);
        } else {
            WriteObjectRef(data, memberName, varType);
        }
    }
}
Reflection Emit

Abstractions correspond closely to the CTS that underlies the CLR:

- AssemblyBuilder
- ConstructorBuilder
- CustomAttributeBuilder
- EnumBuilder
- EventBuilder
- FieldBuilder
- ILGenerator
- Label
- LocalBuilder
- MethodBuilder
- ModuleBuilder
- ParameterBuilder
- PropertyBuilder
- TypeBuilder
Who are you?

- **Accessing meta-data: System.Object.GetType()**
  - All .NET classes (implicitly) inherit System.Object
  - Available on every .NET class; simple types too

- **Explicit language support for type meta-data**
  - C#, JScript.NET: `typeof(…)`
  - VB.NET: `If TypeOf … Is … Then …`

- **Determining Type Identity**
  - Types have unique identity across any assembly
  - Types can be compared for identity
    - `if (a.GetType() == b.GetType()) { … };`
System.Type

- Access to meta-data for any .NET type
- Returned by System.Object.GetType()
- Allows drilling down into all facets of a type
  - Category: Simple, Enum, Struct or Class
  - Methods and Constructors, Parameters and Return
  - Fields and Properties, Arguments and Attributes
  - Events, Delegates and Namespaces
What are you?

- **Value, Interface or Class?**
  - IsValueType, IsInterface, IsClass

- **Public, Private or Sealed?**
  - IsNotPublic, IsSealed

- **Abstract or Implementation?**
  - IsAbstract

- **Covers all possible properties of a managed type**

- **Very intuitive API, no "Parameter Hell"**
Anything special about you?

- **Special Memory Layout?**
  - IsAutoLayout
  - IsExplicitLayout
  - IsLayoutSequential

- **COM Objects?**
  - IsCOMObject

- **More...**
  - IsUnicodeClass
  - IsSpecialName, etc.
Now tell me what you have!

- **Finding and Exploring Members**
  - MemberInfo: GetMembers(), FindMembers()

- **Exploring Fields and Properties**
  - FieldInfo: GetFields(), PropertyInfo: GetProperties()

- **Exploring Constructors, Methods and Events**
  - GetConstructors(), GetMethods(), GetEvents()

- Exploring attributes, determining implemented interfaces, enumerating nested types, ...

- Summary: Everything you may ever want to know
Type and Instances

● **Type Safety First! Type checking at runtime**
  - C#: if (o is Customer) { ... }
  - VB: If TypeOf o Is Customer Then ... End If

● **Dynamic Invocation through Reflection**
  - Support for late binding
    • MethodInfo.Invoke()
    • FieldInfo.SetValue()
    • PropertyInfo.SetValue()
Section 3: Detail Information

- MemberInfo
- FieldInfo, PropertyInfo
- ConstructorInfo, MethodInfo
MemberInfo

- Base class for all "member" element descriptions
  - Fields, Properties, Methods, etc.
- Provides member kind, name and declaring class

Diagram:

```
MemberInfo
  ↓
MethodBase  ParameterInfo  FieldInfo  EventInfo  PropertyInfo
  ↓
MethodInfo  ConstructorInfo
```
FieldInfo, PropertyInfo

- **FieldInfo**
  - Field data type and attributes
  - Static or Instance field, protection level
  - Can set value through reflection
    - Provides low-level, direct access with SetValueDirect()

- **PropertyInfo**
  - Property type and attributes
  - Test methods for readability, writeability
  - "get" and "set" MethodInfo
  - Indexer ParameterInfo
  - Can invoke "set" and "get" through Reflection
MethodInfo, ConstructorInfo

- **MethodInfo**
  - Return type and attributes
  - List of all parameters as ParameterInfo array
  - Detail implementation information through flag-field
  - Can invoke method through Reflection

- **ConstructorInfo**
  - Same features asMethodInfo, just for constructors
Attributes

- Custom attributes are the killer-app for Reflection!
- Attributes enable declarative behavior
- Attributes allow data augmentation

```csharp
[dbcolumn("Address1")]
string Street;

[dbcolumn("Postcode")]
string ZIP;
```

Mark class as serializable
Type.GetCustomAttributes()

Map fields to database columns with
FieldInfo.GetCustomAttributes()
Custom Attributes

[BugFix(121, "GA", "01/03/05")]
[BugFix(107, "GA", "01/04/05", Comment="Fixed off by one errors")]

public class MyMath {
    public double DoFunc1(double param1) {
        return param1 + DoFunc2(param1);
    }

    public double DoFunc2(double param1) {
        return param1 / 3;
    }
}

public class BugFixAttribute : System.Attribute {

    public BugFixAttribute(int bugID, string programmer, string date) {
        this.bugID = bugID;
        this.programmer = programmer;
        this.date = date;
    }

    // Named parameters are implemented as properties:
    public string Comment {
        get { return comment; }
        set { comment = value; }
    }
}
Using Attributes

```csharp
MyMath mm = new MyMath();
Console.WriteLine("Calling DoFunc(7). Result: {0}", mm.DoFunc1(7));
// get the member information and use it to retrieve the custom attributes

System.Reflection.MemberInfo inf = typeof(MyMath);
BugFixAttribute[] attributes =
    (BugFixAttribute[])inf.GetCustomAttributes(typeof(BugFixAttribute), false);

// iterate through the attributes, retrieving the properties
foreach(BugFixAttribute attribute in attributes) {
    Console.WriteLine("BugID: {0}", attribute.BugID);
    Console.WriteLine("Programmer: {0}", attribute.Programmer);
    Console.WriteLine("Date: {0}", attribute.Date);
    Console.WriteLine("Comment: {0}", attribute.Comment);
}

Output:
Calling DoFunc(7). Result: 9.33
BugID: 121 Programmer: GA Date: 01/03/05 Comment:
BugID: 107 Programmer: GA Date: 01/04/05 Comment: Fixed off by one errors
```
Dynamic Method Invocation

Type theMathType = Type.GetType("System.Math");
Object theObj = Activator.CreateInstance(theMathType);
Type[] paramTypes = new Type[]
{ Type.GetType("System.Double")};

MethodInfo CosineMeth = theMathType.GetMethod("Cos", paramTypes);

Object[] parameters = new Object[1];
parameters[0] = 45;
Object returnVal = CosineMeth.Invoke(theObj, parameters);
Loop unrolling

```java
public int DoSumLooping(int n) {
    int result = 0;
    for(int i = 1; i <= n; i++)
        result += i;
    return result;
}
```
public double DoSum(int n) {
    if (adder == null)
        GenerateCode(n);
    // call the method through the interface
    return (adder.ComputeSum( ));
}

public void GenerateCode(int n) {
    Assembly theAssembly = EmitAssembly(n);
    adder = (IComputer) theAssembly.CreateInstance("UnrolledSum");
}

private Assembly EmitAssembly(int n) {
    assemblyName = new AssemblyName();
    assemblyName.Name = "DoSumAssembly";
    AssemblyBuilder newAssembly =
        Thread.GetDomain().DefineDynamicAssembly(
            assemblyName, AssemblyBuilderAccess.Run);
    ModuleBuilder newModule =
        newAssembly.DefineDynamicModule("Sum");
    TypeBuilder myType =
        newModule.DefineType("UnrolledSum",
            TypeAttributes.Public);
    myType.AddInterfaceImplementation(typeof(IComputer));
Create the Assembly

// Define a method on the type to call. Pass an array that defines the types of the parameters, the type of the return type, the name of the method, and the method attributes.
Type[] paramTypes = new Type[0];
Type returnType = typeof(int);
MethodBuilder simpleMethod = myType.DefineMethod( "ComputeSum", MethodAttributes.Public | MethodAttributes.Virtual, returnType, paramTypes);
ILGenerator generator = simpleMethod.GetILGenerator();
// Emit the IL
// Push zero onto the stack. For each 'i' less than 'theValue',
// push 'i' onto the stack as a constant
// add the two values at the top of the stack.
// The sum is left on the stack.
generator.Emit(OpCodes.Ldc_I4, 0);
for (int i = 1; i <= theValue; i++) {
    generator.Emit(OpCodes.Ldc_I4, i);
    generator.Emit(OpCodes.Add);
}
// return the value
generator.Emit(OpCodes.Ret);

computeSumInfo = typeof(IComputer).GetMethod("ComputeSum");
myType.DefineMethodOverride(simpleMethod, computeSumInfo);

// Create the type.
myType.CreateType();
return newAssembly;
Generated IL Code

Ldc_I4 0
Ldc_I4 1
Add
Ldc_I4 2
Add
Ldc_I4 3
Add
Ldc_I4 4
Add
Ldc_I4 5
Add
Ret
Performance

**Output:**

Elapsed ms: 11468.75 for 1000000 iterations

Elapsed ms: 406.25 for 1000000 iterations
The Bigger picture

- Types know their Module, Modules know their types
- Modules know their Assembly and vice versa
- Code can browse and search its entire context
The Unmanaged Spy: Metadata API

- Unmanaged (COM) Version of Reflection
- Used by VisualStudio.NET and Compilers
- Full Access to all Reflection Information for Tools
- Fully documented in the "Tool Developer's Guide"
- Buddy: Assembly Metadata API
  - Reads/writes Assembly Manifests
Section 4: Building Types at Runtime

- Introducing System.Reflection.Emit
- Why? Some scenarios
- Dynamic Modules and Assemblies
- Creating Types and Classes
- Writing IL
Introducing System.Reflection.Emit

- Full representation of physical structure
- Allows building modules and assemblies at runtime
  - Transient code only used at runtime
  - Persistent code for reuse
- Create classes, types and emit IL
- Used by .NET compilers to build .NET apps
Why? Some Scenarios…

- **Build classes dynamically from script-like code**
  - ASP.NET, Regular Expressions do just that!
- **Generate code from visual development tools**
  - e.g. build interfaces, base classes from UML
- **Create dynamic wrappers for existing code**
- **Transfer code-chunks to remote machines**
  - Distributed processing scenarios (like SETI@home)
Building Assemblies

- System.Reflection.Emit.AssemblyBuilder
- Dynamically create new assemblies
  - Create manifest and resources
  - Add modules
  - Specify security requirements
- Can be persisted as files or held in memory
- Act and behave like any other assembly
Building Modules

- Modules contain types, classes and code
- Allows creating fully functional modules with code
- Can associate source code
- Emit debug symbols
- Acts like any module created by any .NET compiler
Writing IL

- Emit accepts IL (Intermediate Language) code
- Same code as emitted by C#, VB.NET, Eiffel#
- IL is optimized and translated into machine code
- Reflection.Emit puts you "on-par"
  - Same backend as .NET compilers
  - Same access to code-generation
  - Languages are mostly scanners and parsers
- For more info see System.CodeDOM
Section 5: Putting It Together

- VisualStudio.NET and Reflection
- What ASP+ really does with a page
VisualStudio.NET and Reflection

Component Class

- Description
- DefaultValue
- Help
- Localizable
- ReadOnly

ComponentModel Attributes

Toolbox
Properties Window
Designers
Help
What ASP.NET does with a Page

1. File exists?
   - Yes
   - No? Compile

2. Assembly exists, same timestamp?
   - Yes
   - No? Run

IIS

ASP.NET runtime

ASP.NET compiler

Reflection

Page Assembly

```csharp
class pg : Page
{
    ...
}
```
Summary

- Reflection = System.Type + GetType()
- Explore Type Information at Runtime
- Enables Attribute-Driven Programming
- Use Emit Classes to Produce .NET Assemblies
- Bottom Line: Fully Self-Contained Structural Model