Laboratorio di Ingegneria del Software L-A

Metadati e introspezione

Metadata

- "Metadata is data that describes other data. For example, the definition of a class is metadata"

Rumbaugh, J. et al, *Object Oriented Modeling and Design* [Prentice Hall, 1991]

Why Have Metadata?

- "Provided that a component comes with enough information to be self-describing, the interfaces supported by a component can be dynamically explored"

Szyperski, C.,
Component Software [Addison-Wesley, 1998]

C++ Metadata

- A C++ **header file** may be considered **metadata**
- Clients can include this file at compile time to use the types it declares
- Clients then link with the types' definition
- C++ has also added support for **RTTI** (*Run-Time Type Information*), a very limited runtime metadata facility

Interface Definition Language

- C++ headers files are language specific
- Providing information across different languages is a difficult issue
- COM and CORBA use the **IDL** (*Interface Definition Language*) to provide metadata
  - COM – Type Libraries
  - CORBA – Interface Repository

COM IDL

```
import "oaidl.idl";
import "ocidl.idl";
#include "olectl.h"
[ object,
   uuid(29AABB7F-E702-11D2-89CF-004033412CFC),
   dual, helpstring("IPolyCtl Interface"),
   pointer_default(unique) ]
interface IPolyCtl : IDispatch
{
   [ propget, id(1),
     helpstring("property Sides") ]
   HRESULT Sides([out, retval] short *pVal);
   [ propput, id(1),
     helpstring("property Sides") ]
   HRESULT Sides([in] short newVal);
};
```

IDL → Reflection

- **IDL**s are an additional requirement for developers to understand
- **Interface Repositories** and **Type Libraries** can be housed in separate files to the type they describe

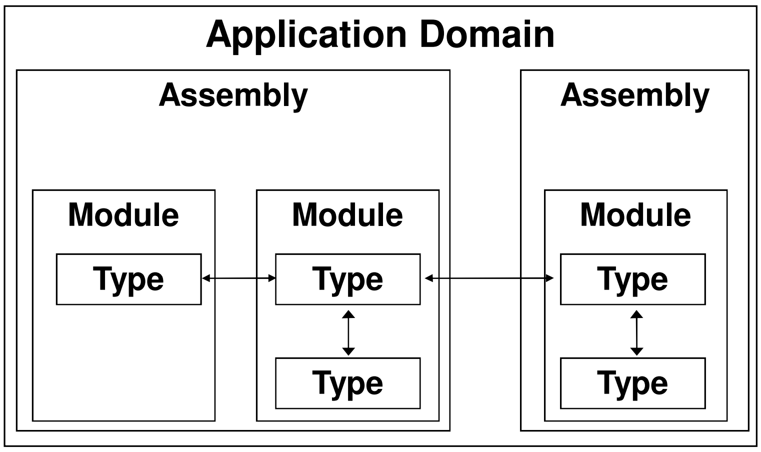
- Java/.NET use ***reflection***
- The metadata is generated from the type's definition
- The metadata is stored with the type's definition
  → if you have the definition you have the metadata and vice versa

Reflection

- Reflection can be used
  - **To examine the details of an assembly**
  - **To instantiate objects and call methods discovered at runtime**
  - **To create, compile, and execute assemblies on the fly**

- .NET classes that deal with providing reflection can be found in:
  - **`System`**
  - **`System.Reflection`**
  - **`System.Reflection.Emit`**

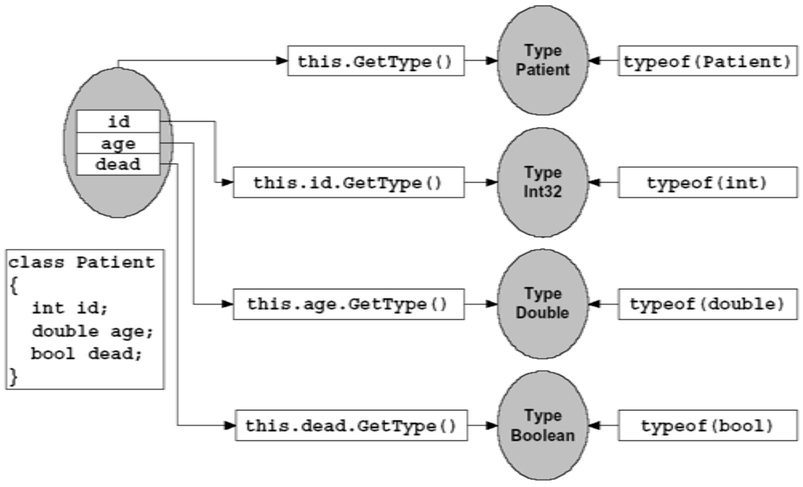
Assemblies, modules and types



`System.Type`

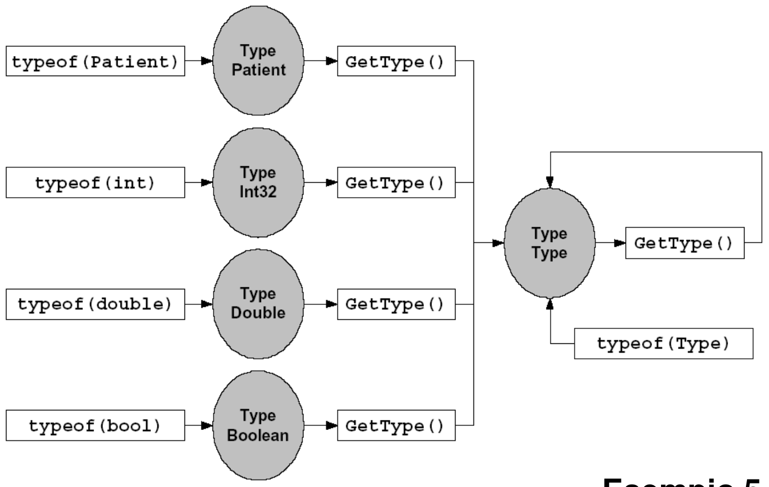
- **`System.Type`** is the focal point of reflection
  - All objects and values are instances of types
  - Can discover type of object or value
```
Type t0 = obj.GetType();
Type t1 = "Pippo".GetType();
```
  - Can reference type by symbolic name
```
Type t2 = typeof(System.String);
Type t3 = Type.GetType("System.String");
```
  - Types are themselves instances of type **`System.Type`**
- There is a single **`Type`** object for each type defined in the system

System.Type



```
class Patient
{
  int id;
  double age;
  bool dead;
}
```

System.Type

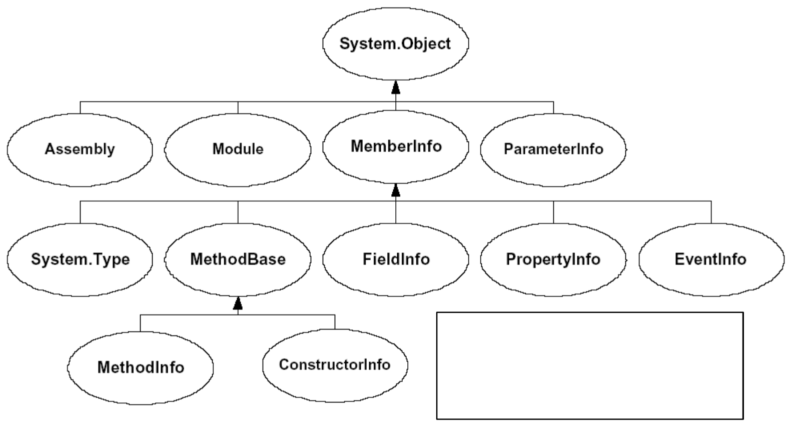


Esempio 5.1

System.Type

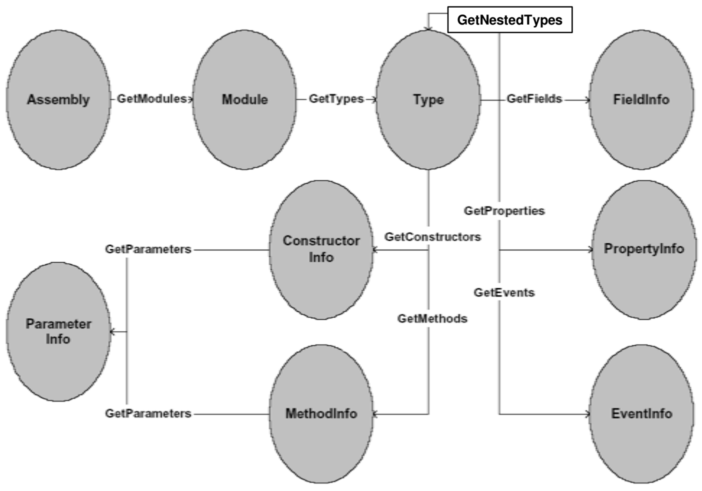
- Some methods:
  - `Type[] GetInterfaces();`
  - `MemberInfo[] GetMembers();`
  - `ConstructorInfo[] GetConstructors();`
  - `MethodInfo[] GetMethods();`
  - `FieldInfo[] GetFields();`
  - `PropertyInfo[] GetProperties();`
  - `EventInfo[] GetEvents();`
  - `object[] GetCustomAttributes();`

Reflection and the CLR type system



The reflection object model

Laboratorio di Ingegneria del Software L-A



Esempio 5.2

5.15

Esempio
Enumerating all types in an Assembly

Laboratorio di Ingegneria del Software L-A

1. Use **`Assembly.Load`** to load a .NET assembly returns an **`Assembly`**
2. **`Assembly.GetModules`** returns an array of **`Module`**
3. For each **`Module`**, call **`Module.GetTypes`** returns an array of **`Type`**
4. For each **`Type`**, …

Esempio 5.3

5.16

Very late binding

- Types may be instantiated and/or members accessed in a **very late bound manner**
  - Can instantiate type in memory, choosing constructor to call
    - `Activator.CreateInstance(…)`
  - Can invoke methods
    - `methodInfo.Invoke(…)`
  - Can invoke property getters and setters
    - `propertyInfo.GetValue(…)`
    - `propertyInfo.SetValue(…)`
- Public members always accessible
- Non-public members accessible if callers hold sufficient permissions

Esempio 5.4

`System.Activator`

- Dynamically create instances
- `Activator.CreateInstance` is the late-bound equivalent to operator `new`
  - Allocates storage for new type instance
  - Calls specified constructor
  - Returns generic object reference

```
T1 t = (T1) Activator.CreateInstance(typeof(T1));
T1 t = (T1) Activator.CreateInstance(typeof(T1),
     object[] args);
```

Esempio 5.5

Meta-Programming

- "... the fundamental problem is always the same: preserve information available at compile time for inspection at runtime. Making such information about a system available within that system is called **reification**.
  Programming a system to not only use reified information but also to manipulate this information is called ***meta-programming***.
  …*meta-programming* can be used to dynamically create new classes, insert them into an existing inheritance graph and instantiate them"

  Szyperski, C.,
  Component Software [Addison-Wesley, 1998]

- **Reificazione**: Concretizzazione di un'astrazione

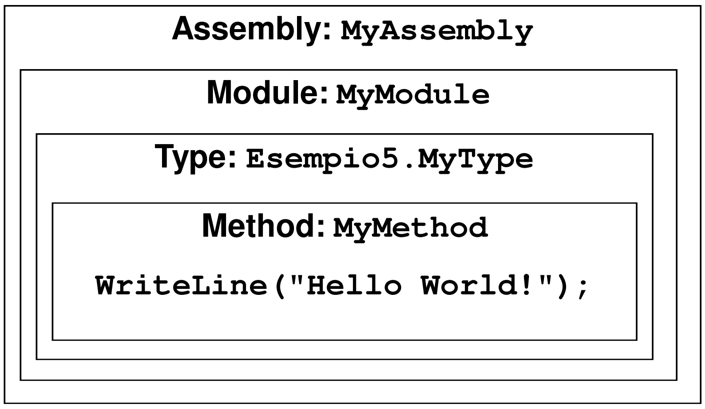
Meta-Programming in .NET

- A number of classes function together to achieve this goal in .NET
- By using the previous objects, and others, you can **build an assembly on the fly**
  - `Reflection.Emit` allows you to write out the IL necessary to **create and compile the assembly**
  - You can then **call this assembly** from with the program that created it
  - **The assembly can be stored to disk** so that other programs can use it

Meta-Programming in .NET

- **`System.Reflection`**
  - **`AssemblyName`**
    Fully describes an assembly's unique identity
- **`System.Reflection.Emit`**
  - **`AssemblyBuilder`**
    Defines and represents a dynamic assembly
  - **`ModuleBuilder`**
    Defines and represents a module
  - **`TypeBuilder`**
    Defines and creates new instances of classes during runtime
  - **`MethodBuilder`**
    Defines and represents a method (or constructor) on a dynamic class
  - **`ILGenerator`**
    Generates Microsoft intermediate language (MSIL) instructions

Dynamically Creating a Type



Esempio 5.6

Custom Attributes

- Are an easy way **to add information to the metadata** for any application element
  - Can be applied to an assembly using special syntax
- Can be used so that **clients can automatically pick up on certain functionality**
  - Are visible via reflection
- Are supported in any .NET language
- Are really just **common classes** that derive from `System.Attribute`
  - Can contain methods and properties

Creating Custom Attributes

- Declare the attribute class

```
public class AuthorAttribute : System.Attribute
```

- Declare constructors
- Declare properties
- Apply the `AttributeUsageAttribute` (opzionale)
  Specifies some of the characteristics of the class
  - The target of the attribute (`AttributeTargets`) – a quali elementi l'attributo è applicabile
  - Whether or not the attribute can be inherited (`Inherited`)
  - Whether or not multiple instances of an attribute can exist for an element (`AllowMultiple`)

Esempio 5.7 – AuthorAttribute

Using Custom Attributes

- C# uses IDL-like syntax with [ ] prior to the definition of the target
- Attribute parameters passed
  - by **position** or
  - by **name**

Primo argomento del costruttore

```
[ Author("Bellavia",
Contact="giuseppe.bellavia@unibo.it") ]
```

Nome di una proprietà

Esempio 5.7 – MyClass

Accessing the Custom Attributes

- Once the custom attributes have been created, you use Reflection in order to read them
- You can get a list of custom attributes by calling the **`GetCustomAttributes`** method

```
object[] X.GetCustomAttributes(inherit);
object[] X.GetCustomAttributes(attributeType,inherit);
```

  `inherit` specifies whether to search this member's inheritance chain to find the attributes
- **`X`** è
  - un'istanza di
    - **`Assembly, Module`**
    - **`MemberInfo`**
    - **`ParameterInfo`**

Esempio 5.7