New C#4 Features – Part VI

Nico Ludwig (@ersatzteilchen)

TOC

- New C#4 Features – Part VI
  - Beyond Expando Objects: Home-brew Dynamic Dispatch
  - Hosting Scripting Languages

- Sources:
  - Jon Skeet, CSharp in Depth

If ExpandoObject is not enough

- We can get very far with *ExpandoObject*, but data driven APIs require more.
  - The possibility to encapsulate an existing API is often the core paradigm.
  - Adding user defined behavior to get more control (e.g. enforce readonly objects).
  - Sometimes you need or want to deal with operators.
  - The class *ExpandoObject* is sealed, and can not act as base class.
- Up to now we've exploited the DLR from the caller side, now we'll implement own dynamic behavior in C#.
  - We can create types, whose interfaces change during run time; and these interface changes are based on how we use the type.
  - That means we'll implement our own dynamic dispatch.

- *ExpandoObject* instances behave differently in VB and in C#, because in VB symbols are case insensitive, but in C# they are case sensitive!

DynamicObject in Action

See accompanying project <DynamicObjects>
This example shows home brew dynamic dispatch with the CLR type DynamicObject.

Key aspects of dynamic programming:
(1. Dynamic Typing)
(2. Late Binding)
(3. Duck Typing)
4. Dynamic Objects

Home-brew Dynamic Dispatch with DynamicObject

- If you need more than dynamic property bags you can handle the dynamic dispatching yourself in a type derived from *DynamicObject*.
  - E.g. to create methods and properties at run time to create explorative interfaces.
    - E.g. to wrap another API.
  - *DynamicObject* allows you to dispatch all (twelve) dynamic operations yourself.
  - We have the same abilities to handle unknown methods as in Smalltalk and Obj-C!
    - Instead of overriding "*doesNotUnderstand:*" (Smalltalk) we have to override "*TryInvokeMember()*".
- How to make use of *DynamicObject*:
  - Import the *System.Dynamic* namespace.
  - Inherit from *DynamicObject*.
  - Then you can override the behavior of dynamic operations or define new methods.
- If you can not derive from *DynamicObject*, implement *IDynamicMetaObjectProvider*.
  - But meta programming with Expression trees is not that simple...

5

- In the accompanying source code, you can find richly commented Smalltalk, Obj-C, Python and Ruby implementations of a property bag. (Please have a look, it is easy to understand in that languages!)
- It is planned to have a similar home-brew dynamic dispatch support in Scala.
- If you implement *IDynamicMetaObjectProvider* you have to program a *DynamicMetaObject*, in which you code how dynamic operations translate into Expression trees. This flexibility comes with a cost: complexity, as coding of Expression trees, can be very complicated.
  - In .NET 4.5 there'll be a new type, *JsonValue*, which allows simple serialization of Json-formatted data. In order to be programmable in a simple way, it implements *IDynamicMetaObjectProvide* and can be used like *ExpandoObject* to add new Json elements.
- So if the out-of-the-box abilities of the DLR are not sufficient for your scenario, you can write own binders and meta-objects.

Mechanics of Dynamic Dispatch

- The compiler lets us pass any kind of operation using the dynamic keyword.
  - And the real type has to handle the passed operation.
  - But the operation needs not to be a part of the real type's static interface.
  - The idea is to mix dynamic dispatch with static invocations in a single type.
  - This can be done with dynamic objects.

- Restrictions:
  - To inspect the run time type of *DynamicObject* you can't use reflection.
    - (Also you can not check, whether a variable was declared as dynamic reference at run time.)
    - As its type (e.g. the properties and methods) is part of its value.

- You absolutely have to document how your *DynamicObject* works!
  - No excuses! The compiler can't help here!
  - Best practice: write unit tests documenting your code!

Dynamic Languages based on .NET

- Meanwhile there exist a lot of dynamic .NET languages based on the DLR.
  - Iron*, like IronRuby, IronPython, IronScheme etc....
  - Inspired by the success of Jython on the JVM, IronPython was developed.
  - During run time, scripts get translated into IL code and then executed by the CLR.
- The new features of Expression trees support enabling scripting with the DLR.
  - The compiler/interpreter pipeline of a DLR-based scripting engine:
    - 1. Lex and parse the script => Expression trees, or abstract syntax trees (AST),
    - 2. pass the AST to the DLR,
    - 3. the DLR then compiles the AST to IL and executes it on the CLR,
    - 4. finally the CLR could JIT IL code to native machine code.
  - Expression trees are to the DLR what IL is to the CLR.
- Some scripting engines can run in compiler mode.
  - Then methods get compiled into IL code (w/o the Expression tree step).

- IronPython 1.0 was even faster than C-based Python, because .NET Delegate-invocations are very fast since .NET 2 (CPython, Jython do only have an own runtime (different from .NET)). So, yes, there exist also variants of the DLR (and IronRuby and IronPython) that run on .NET 2 (then you'll need to deploy the DLR (Microsoft.Scripting.dll, Microsoft.Dynamic.dll etc.) yourself).
- E.g. IronRuby has a compiler mode (default). In opposite to the compiler mode the interpreter mode has a better startup time, but the run time is worse.

The Iron Languages

- The DLR and IronPython have been developed by the same team until 2006.
  - The DLR, IronPython and IronRuby (2008) are all open source (Apache lic. v2).
- Iron* languages offer great integration with .NET:
  - Easy to use .NET libraries and other .NET languages.
  - Easy to use on .NET hosts and with .NET tools.
- Iron* languages compile to IL code during run time.
  - During run time: Created Expression trees will finally be compiled to IL code.
    - And the JIT compiler will compile the IL code to machine code.
    - Or you can compile your script code into an assembly!
- But we also have two type systems to deal with: .NET or the Iron* language's one.
  - Ruby has mutable strings, .NET don't (there is a to_clr_string() conversion method).

- IRON: (kidding) Implementation Running on .NET.
- For this presentation:
  - IronRuby 1.1.3 was installed as NuGet package for the project.
- The conversion between Iron* and CLR types does often happen automatically, when it should. E.g. IronRuby types are no CLR types, but they can act as CLR types, when required. Typically CLR types differ from Iron* types as they can not be modified at run time.
- E.g. creating assemblies from Python scripts: Use ipy.exe with the option "-X:SafeAssemblies". (In such an assembly you can inspect the DLR CallSite-code created for IronPython (e.g. in Reflector)!) At the time of this writing (September 2011) the Iron* compilers don't create CLS-compliant assemblies, this means they probably can't be used in other .NET languages.

Hosting Scripting Languages

- Scripting languages have been used as glue languages in past.
  - Hosting scenarios allow scripts to be a substantial part of an application.
  - The DLR enables seamless or restricted usage of scripting language as you configure!
  - Hosting can be controlled via App.config (requires no recompilation of your C# code).
- Exploit collaboration where required:
  - The communication between .NET and DLR-based languages requires no wrappers.
    - Dynamic dispatch enables this.
  - Different dynamic languages use one GC on the DLR!
- Establish isolation where required:
  - E.g. Iron* can be hosted in a dedicated AppDomain or process, with a different trust.
    - Also on a dedicated physical machine.
    - The intend is to run user scripts with a different security access level (SAL).

- The DLR is not able to host a static language within another static language or a static language within another dynamic language. But hosting of static (.NET) languages is in sight with MS's research on "compilers as a service" (CaaS) with the project "Roslyn".
- The DLR hosting API consists of two sets of APIs, one for language consumers and the other for language providers.
- The file App.config can e.g. control the discovering of script engines).
- Isolation: the relevant DLR scripting types inherit *MarshalByRefObject*, thus enabling .NET Remoting.

The "Big Iron* Hosting Example"

See accompanying project <IronRubyHosting>
This set of examples shows hosting of the scripting language IronRuby.

- IronRuby is to be understood as placeholder for any other DLR/.NET based scripting language, like IronPython or IronScheme.

DLR in the Web

- I.e. running the DLR within the browser.
- ASP-based DLR solutions exist as Codeplex projects, but w/o clear roadmap.
- Silverlight can script JavaScript via DLR.
  - Other languages (Iron* languages) can be scripted, e.g. via Chiron.
    - Chiron is a packager, packaging Iron* languages' code into Silverlight applications (like class libraries).
  - A modern approach is "Gestalt", which allows "just text" hosting of Iron* languages.
    - "Just text" allows scripting of HTML and XAML with Iron* languages much like JavaScript.
    - Gestalt exploits the Silverlight runtime to host scripting languages used in a site via the DLR.
- ASP.NET MVC allows using dynamic coding to wire Controllers and Views.

Other new Features in .NET 4

- Side-by-side (SxS) hosting of different CLRs within one process.
- New and updated .NET frameworks:
  - New: Parallel Extensions Framework (PXF)
    - Task Parallel Library (TPL), Parallel LINQ (PLINQ) and the types *Task* and *Parallel*.
  - New: Managed Extensibility Framework (MEF) and Code Contracts
  - Updates: the new Workflow Foundation (WF)
- New types:
  - *Tuple<T, ...>*, *IStructuralEquatable* and *IStructuralComparable*,
  - *ISet<T>*, *SortedSet<T>*, new types in namespace *System.Collections.Concurrent*,
  - the namespace *System.IO.MemoryMappedFiles*,
  - *Lazy<T>*, *IObserver<T>* and *IObservable<T>* and more ...
  - *Action<T>* and *Func<T>* are now taking up to 16 arguments.
  - New namespace *System.Numeric* containing the types *BigInteger* and *Complex*.

- SxS works for CLRs version > 2.0, the hosting works with different *AppDomain*s.
- Memory mapped files is a powerful means to operate on large data in memory (it is much faster than *MemoryStream*s) and to establish communication between processes.
- New overloads of *Action<T>* and *Func<T>*: They are present primarily to help support the DLR.

Thank you!