Breaking the Language Barrier.

By Software FX

The release of the .NET platform posed an exciting challenge to us here at Software FX, in order to maintain our position as industry leader, we could not simply distribute our product in a .NET wrapper; we had to totally rewrite it in C# to take advantage of the platform's new features and improvements.

This required us to explore and work with the .NET platform in intimate detail, revealing the closeness between .NET and J2EE. Both object models are very similar; in particular the graphics APIs, heavily used in Chart FX, are nearly identical in functionality. Additionally, the C# and Java languages are very similar.

The possibility of a Chart FX for Java was growing parallel to the reality of Chart FX for .NET. As long as we were redesigning our product, why not expose Chart FX to Java users?

Although similar, the platforms are not identical. There are some fundamental differences. Thus, the burden of delivering the most powerful charting component available and maintaining different sets of source code without diverting a huge amount of our resources into porting needed to be lifted. Dual development would make it very difficult to create new features and extensions as they would need to be written, tested and integrated for both .NET and Java.

An idea began to take shape, suppose we were to build an engine that would allow us to write code only once, in C#, it being the language we started with, yet deliver it for .NET, Java and possibly even other platforms such as COM.

This would accelerate our time-to-market and ensure that our focus remain on building better products, giving us the freedom to improve as the burden of porting from one language/platform to the other would be virtually eliminated. So we did it.

The Architecture of the Translator

The translation process follows the illustrated scheme:

First, original C# code is passed to a C# Parser and transformed it into a CodeDom, a structured representation of the code within the .NET framework. Next, the CodeDom is passed to a Java Code Writer and the desired Java code is produced.

This modular design, based on the innovative CodeDom, allows us to translate other .NET languages (e.g. VB.NET, Delphi, etc.) by creating a parser for them. It also allows code generation in other languages by plugging in a different Code Writer (e.g. C++).

About the author: Francisco Padron is a senior Software Architect and co-founder of Software FX, Inc., founded in 1993 and the publisher of Chart FX.

The parsing process is fairly straight forward thanks to the intermediate CodeDom. Translating the CodeDom to Java is the next step. Control structures and basic statements are almost identical in these languages but there are some fundamental differences:

1) The concept of an assembly in .NET has no direct representation in Java.
2) By reference parameters: out and ref in C# are not supported in Java.
3) The concept of Value Types is not present in Java.
4) .NET Boxing does not occur automatically in Java.
5) Attributes for classes, properties, etc. are not supported in Java.
6) Properties and Events are not supported by the Java language itself, however there are naming conventions that allow a component to expose them, namely the Java Bean Specs.
7) Operator overriding is not supported in Java.
8) There is no Licensing Standard for Java Components.

> "Suppose we were to build an engine that would allow us to write code only once."

Bridging the Gap

As we know, there are some fundamental differences between .NET languages and Java. These differences generate the need for ingenious translation strategies. Here are some of the most interesting ones:

Value Types

As opposed to objects (Class Types), Value Types are allocated in the stack as they are declared and are copied when they are assigned or passed as parameters.

For example:

```
[C#]
Rectangle r1 = new Rectangle(0,0,10,20);
Rectangle r2 = r1;
r2.Width = 100;
```

Since Rectangle is a Value Type, this code results into two objects, r1, having a width of 10, and r2, having a width of 100, as opposed to just one object having a width of 100 (which would happen if Rectangle was declared as a class).

To translate this code to Java requires the following:

[Java]

```
Rectangle r1 = new Rectangle(0,0,10,20);
Rectangle r2 = r1.Clone();
r2.setWidth(100);
```

Clone() is a method that makes a deep cloning of all the members of Rectangle. All value types must implement this method and the translator generates the code for it.

Parameters by Reference

All parameters in Java are passed by value. This is not the case in .NET, the ref and out keywords in C# allows you to pass parameters by reference. Combine this with Value Types and it becomes very interesting. For example, let's consider the following fragment of code:

[C#]

```
void Func1 (ref Rectangle r1, Rectangle r2,
   ref object obj)
{
       r1.Width = 100;
       r2.Width = 200;
       object obj2 = obj;
       obj = new MyObject(2);
}
void Func2 ()
{
Rectangle r1 = new Rectangle(0,0,10,20);
Rectangle r1 = new Rectangle(0,0,100,200);
object obj = new MyObject(1);
Func1(ref r1, r2, ref obj);
}
```

The translation looks like this:

[Java]

```
void Func1 (Rectangle r1, Rectangle r2,
   ObjectByRef _obj)
{
       object obj = _obj.obj;// Unpack byref
          param for input
       r1.Width = 100;
       r2.Width = 200;
       obj = new MyObject(2);

       _obj.obj = obj; // re-pack byref
          param for output
}
void Func2 ()
{
Rectangle r1 = new Rectangle(0,0,10,20);
Rectangle r1 = new Rectangle(0,0,100,200);
object obj = new MyObject(1);
ObjectByRef objByRef = new ObjectByRef(obj);
Func1( r1, r2.Clone(), objByRef);
obj = objByRef.obj;
}
class ObjByRef {
       public ObjByRef(object obj)
{
       this.obj = obj;
}
       public object obj;
}
```

Notice that r1, being a Value Type (Rectangle), is passed normally since in Java it is a class and a reference (not a copy) is passed, r2 on the other hand has to be cloned because it is passed by value in the original code (a copy of it was created automatically in .NET).

When passing an object (Class Type) by reference an extra level of indirection is needed, for this purpose, the class ObjByRef is generated on the fly and encapsulates an object. The appropriate un-packing and re-packing code is added to the beginning and end of the method and additional code if added after the call is made to return the values to the corresponding variables.

Boxing

Boxing is performed when a primitive type needs to be converted to an object. In .NET this occurs automatically. For example:

[C#]

```
void Func1 (object obj)
{
}
void Func2 ()
{
       Func1(100);
}
```

[Java]

```
void Func1 (object obj)
{
}
void Func2 ()
{
       Func1(new Integer(100));
}
```

Notice that the primitive value 100 is encapsulated into an Integer object (Integer class), this conversion needs to be smart and each of the primitive types needs to be encapsulated using the appropriate Java class.

As these examples illustrate, even though the languages are very similar, some situations require sophisticated translation. Combining these situations increases the complexity exponentially. All of the solutions presented preserve the original design of the code and maintain the same level of performance; this is of the utmost importance.

The Finishing Touch

Having overcome each of these hurdles, we are able to translate our .NET code into very clean, efficient Java code.

Porting from the .NET Framework object model to J2EE is the next and most time consuming step. Limiting this stage to the libraries and classes that we use makes this daunting task more manageable. We concentrate on porting only our Server Side Components as we believe it is on the server where Java's strength resides. This eliminates the need for porting all of the Windowing System (AWT / Swing) and leaves us with the drawing API, file IO, Reflection and standard classes such Collections, Arrays, Hash Tables, etc.

Keeping in mind our requirement of generating code that looks and performs as code hand written in Java, converting from one object model to the other involves not merely wrapping the J2EE classes into a .NET Framework shell but also creating some sophisticated code conversions making direct use of fundamental Java classes without wrapping them. For example, we keep the string handling in native Java instead of going through wrappers. The same is done with many other key performance related classes.

The Final Product

We invested in the future and it paid off. Instead of taking the easy, short road and building Chart FX for Java from scratch once, we opted for a more ambitious approach: building an engine that ports our current version of Chart FX to Java and ensures that we stay focused on developing more functionality and that this functionality will, with very little effort, be ported to our Java versions.

Due to the success of this project, we are developing parsers and code writers to port .NET code to COM (C++) to bring new developments to more customers while we continue to improve our existing translation engines as well include support for new functionality in all of the platforms involved.