Dot Net

An Overview for those
Coming From VO

Development For dot Net

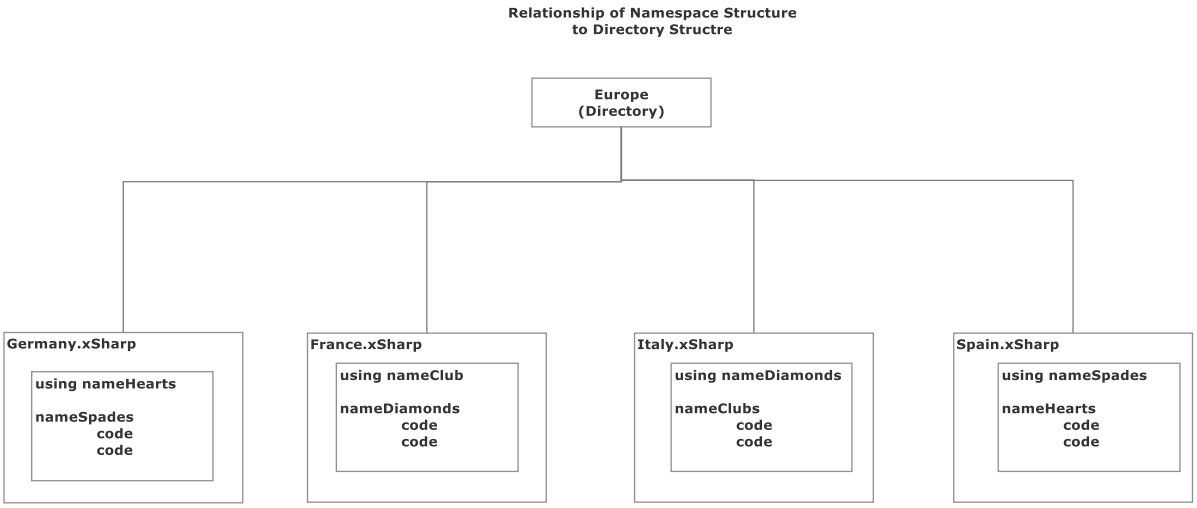
Development coding for .Net is File Based. Unsurprisingly that means it is hosted in the Directory/ File System with which we are all familiar. In a sense it is not unreasonably to consider it as the repository of VO.

But that structure is far from convenient when, at the end of the development process, we need to produce one readable file for submission to the CLR.

Some way of identifying “grouping” code so that it can be extracted logically, unambiguously and automatically from the file structure is needed. Furthermore, it is up to us, as developers, to identify, in our code, how we want things to be brought together.

We do all that with the .Net concept of namespaces (which embrace our code), and the keyword “using” which references them. This results in a sort of “holding” structure which sits atop of the filing system.

By judicious naming we can illustrate the situation, as shown in the diagram below:



.Net is based on Types

This is true, but do not confuse .Net Types with types you are already familiar with. Everything in .Net is considered to be a Type. It is the CLR concept of a type. And there is reason for it – that is to do with the way the CLR works. Basically, it is a way of packaging things that work the same way.

The CLR is extremely complex. To fulfil its primary task of facilitating (JIT) compilation on the user machine, it has to regenerate our program. It does this from highly efficient compacted code.

Fortunately, we don't need to go into the details at this time. But we can easily understand the principle by relating the process to what we do every day of the week. We could go to the baker, armed with ingredients and get him to bake us a cake, we could go to the doctor and get him to give us some pills (to cure the headache we've got from reading this), we can take a method to the CLR and get it to process that and so on.

In fact, the Types that can be accepted by the CLR are fairly limited: class, fields, enumerations being among them. These will all have been "mapped down" from our source code.

Much of our source code will have become superfluous by the time the CLR output is read by the JIT compiler. Its instructional meaning will be inherent in the way in which the read is done. This means the operation is quick and limited to those instructions that are required at that moment.

It is wrong to think that this compilation process must necessarily impact performance. The first time it takes place it will, but after that an already compiled version of the code, derived from an Intermediate Language (IL) will be used. Any performance hit is not likely to be a concern in an X-Sharp development. (and there are ways round it anyway)

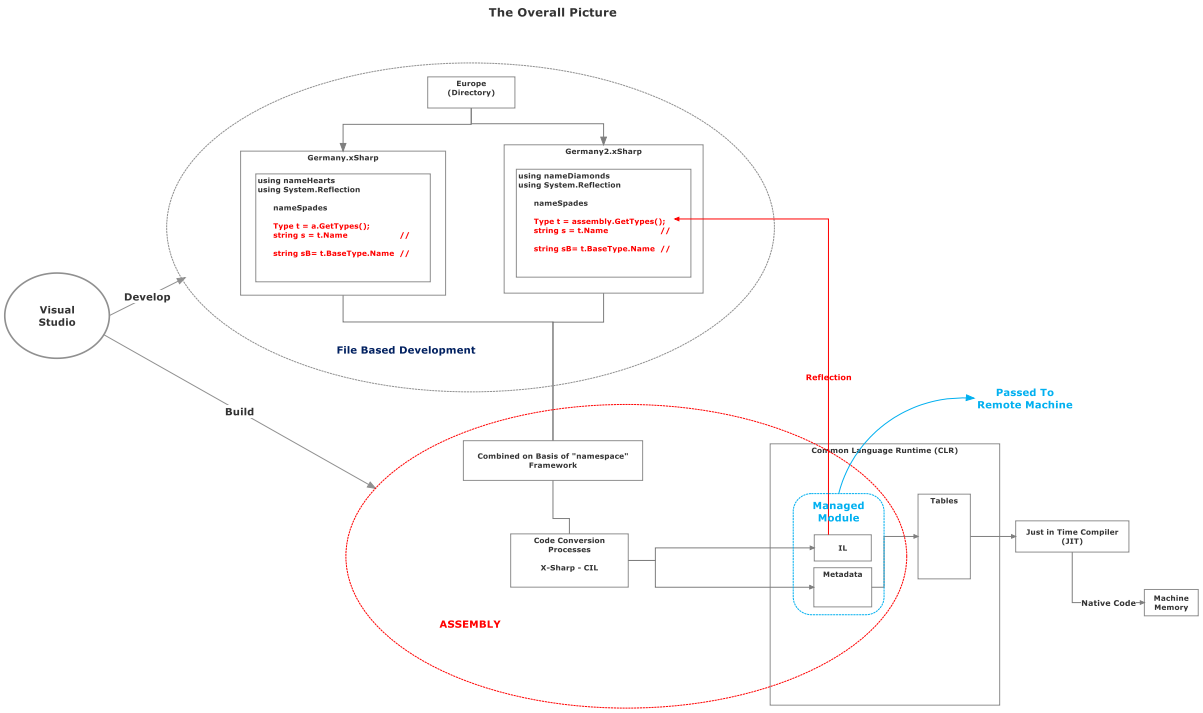
But there is more to the CLR than we've covered so far. In providing an input to the JIT compiler it has to regenerate the program on the user machine. This means it must have all the information from our program close at hand. Which brings us to the next topic: -

Meta Data and the CLR

`Metadata (Data about Data) is stored in the CLR in highly efficient code constructs which store every detail needed for the CLR itself to do its job. Whilst this information is essential for the compilation output, it is also useful (very useful) for a number of things -tools etc which may be hosted within Visual Studio and also for us, as developers, to use as we see fit. It represents a Reflection of our program and is available for us to use in – you've guessed it: the System. Reflection Namespace!

Above all .Net is a Component Integration Technology. It allows us to integrate components developed by third parties into our own programs, as well as allowing others to use components we may ourselves develop. This means we need some way of packaging things up; to do this we package code into Assemblies and Modules.

I have tried to show how all this ties up in the diagram below:



You may well think that wit all this going on our code and any component code we are using would have been exhaustively checked for consistency and you'd be right; it has been checked and rechecked every step of the way. But as they say, "the proof of the pudding is in the eating" so we'll move on to that and introduce the star of the show: **TIME**.

Proof of the Pudding

There is not much point in writing a program if it is never used. Likewise, there is not much point in writing a book if it never to be read. In both these cases, using and reading takes **Time**. In both these cases something has to prompt either the program to do something or a reader to read.

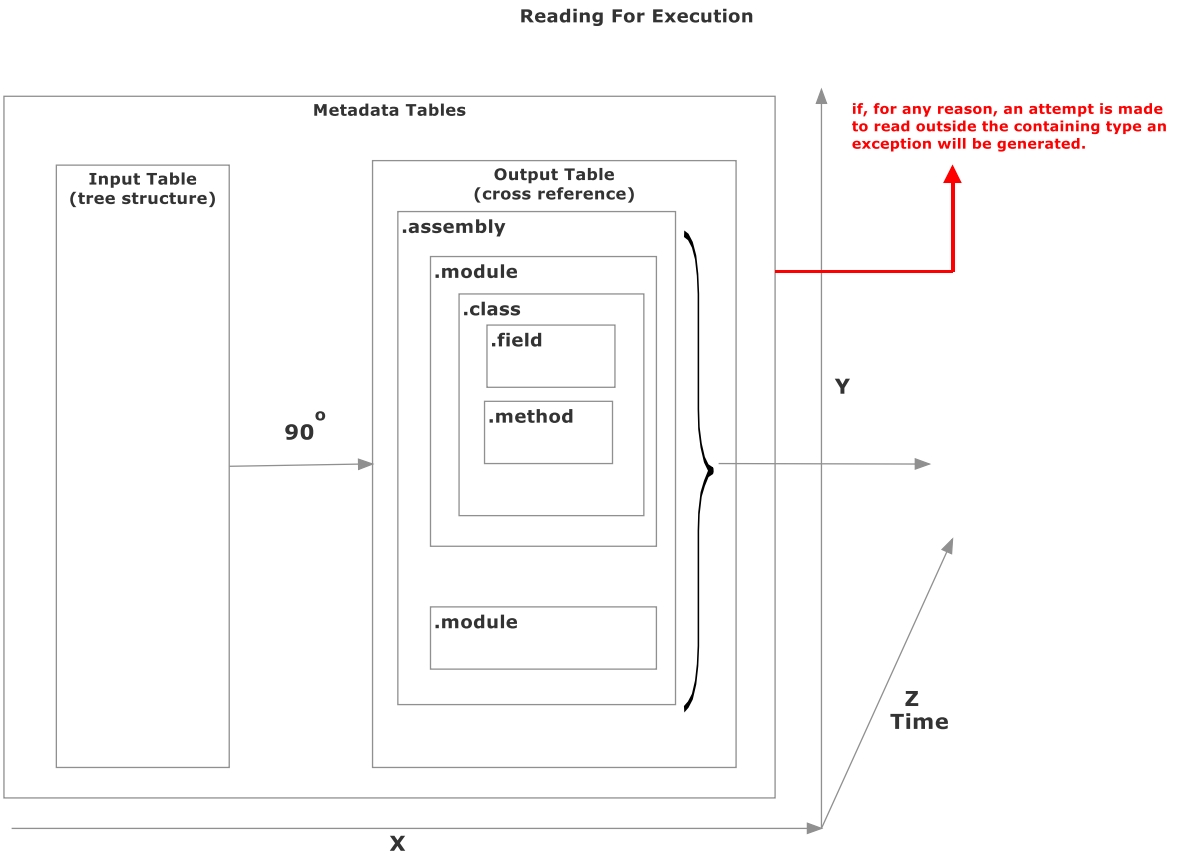
In the former case, this prompting can be done either automatically or by it manifesting itself as a UI giving the user the choice of what to read next.

In the latter case it simply putting the book down and having a cuppa.

In both cases it is the human who ultimately makes the choice.

In the case of a (technical) book we will read things sequentially much of the time, but there will come a point where things do not follow in sequence, to allow for this we "scope" our text into chapters, paragraphs etc giving us an easy way to cross reference.

The CLR scopes code on the basis of namespaces and types as shown below: -



There is just one other thing I'd like to mention here. That is the idea of a "Class Variable". This will come up when, as I suspect, you start using WPF. Or, indeed in your own code. It originates from being able to "scope" across a full class type.