**How to enumerate solution and source code items using DXCore**

April 6th, 2011

One of the trivial tasks when developing a [DXCore plug-in](http://www.skorkin.com/2010/08/dxcore-plug-ins-overview/) is the enumeration of the active solution items, such as projects, source files, then interfaces, classes, methods, properties, statements, etc. A similar task is to get an active element (in other words, the element where the editor caret is located) inside the active source file to start working with one.

All of the items of the solution are represented by the [DXCore](http://www.skorkin.com/2010/06/devexpress-dxcore-framework/) classes, located in the “*DevExpress.DXCore.Parser*” assembly inside the “*DevExpress.CodeRush.StructuralParser*” namespace. Consider, we have a standard Visual Studio solution, **DXCore** uses the following classes to represent its hierarchy:

|  |  |
| --- | --- |
| **Class Name** | **Description** |
| SolutionElement | Represents a Visual Studio solution. |
| ProjectElement | Represents a Visual Studio project. Holds a list of all files belonging to the project. |
| SourceFile | Represents a file (source code, resource file, bitmap file, XML page, HTML page, etc). |

All elements of the file, containing source code, are parsed as [DXCore abstract source tree structure](http://www.skorkin.com/2011/01/dxcore-abstract-source-tree-structure/), maintained by the [language elements](http://www.skorkin.com/2011/01/ide-tools-terminologykey-concepts/). These are the most often used elements inside a source code file:

|  |  |
| --- | --- |
| **Class Name** | **Description** |
| Namespace | Represents a namespace declaration, e.g. “namespace N { }” (C#), “Namespace N … End Namespace” (VB). |
| NamespaceReference | e.g. “using System;” (C#), “Imports System” (VB) |
| Class, Struct, Interface, Enumeration | These are different kinds of type declarations. |
| Method, Property, Event, Variable | These are different kinds of type member declarations. |
| Hundreds of the elements consisting of statements and expressions, such as: *For, ForEach, If, Return, Throw, ElementReferenceExpression, TypeReferenceExpression, PrimitiveExpression* | These are different kinds of code blocks inside members. |

To get an element of the solution, use the [SourceModel DXCore service](http://www.skorkin.com/2010/09/dxcore-services-list/); it has numerous properties to work with the solution and code structure, such as: *ActiveSolution*, *ActiveProject*, *ActiveClass*, *ActiveMethod*, *ActiveProperty*, etc. For example, let’s count the number of properties and constants declared inside the entire solution. Here’s a sample code:

**CSharp** code:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19 | void DoCount()  {    int propertiesCount = 0;    int constantsCount = 0;      SolutionElement activeSolution = CodeRush.Source.ActiveSolution;    foreach (ProjectElement project in activeSolution.AllProjects)      foreach (SourceFile sourceFile in project.AllFiles)        foreach (TypeDeclaration typeDeclaration in sourceFile.AllTypes)        {          foreach (Property property in typeDeclaration.AllProperties)            propertiesCount++;          foreach (Const constant in typeDeclaration.AllConstants)            constantsCount++;        }      Console.WriteLine("Number of properties inside solution: " + propertiesCount);    Console.WriteLine("Number of constants inside solution: " + constantsCount);  } |

[Show Visual Basic code… »](http://www.skorkin.com/2011/04/how-to-enumerate-solution-and-source-code-items-using-dxcore/#SID3099_1_tgl)

As you can see, we get an instance of the active solution, then enumerating over all of its projects, then all files of all projects, then all types of all files and, finally, the members we need – properties and constants. Bear in mind, that if there’s no solution opened, the *ActiveSolution* will return *null* (C#) or *Nothing* (VB).

For enumerating inner code blocks of members, there are properties like *AllStatements, AllExpressions, AllVariable* on the instance of a member language element. But these properties might be not enough if you would like to enumerate every piece of the code inside member in detail. To enumerate everything, you can use the **Nodes** and **DetailNodes** properties of the particular language element. **Nodes** property contains high-level root elements like statements and the **DetailNodes** property contains granular details specific to this particular element. For example, let take a look at this ‘for’ loop in **CSharp**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | for (int i = 0; i < 100; i++)  {    Console.WriteLine(i);    if (i > 50)     break;  } |

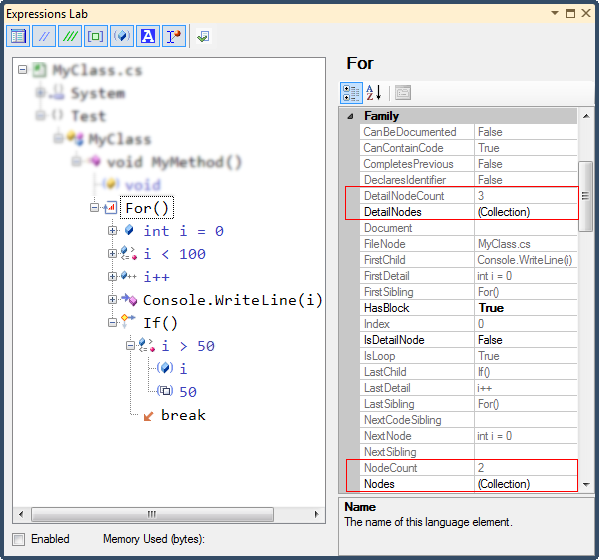
The **Nodes** property will contain two items:

* Language element named “*MethodCall*” – “Console.Write(i);”.
* Language element named “*If*” – “if (i < 50) break;”

The **DetailNodes** property, in turn, will contain three items:

* Language element named “*InitializedVariable*” – “int i = 0;”
* Language element named “*RelationalOperation*” – “i < 100;”
* Language element named “*UnaryIncrement*” – “i++”.

To better understand the difference between **Nodes** and **DetailNodes** properties, use the [Expression Lab](http://www.skorkin.com/2011/08/dxcore-expression-lab-tool-window-for-observing-the-parsed-source-trees/) tool window available via **DevExpress | Tool Windows | Diagnostics | Expression Lab** menu item. Here is what the ‘for’ loop language element looks like inside this tool window:



The other way to enumerate elements inside of a scope is to use the *ElementEnumerable* class, declared in the “*DevExpress.DXCore.Parser*” assembly. It may take one, two or three parameters, such as:

* The **Scope** – the *LanguageElement* instance inside of which you are going to enumerate inner elements, for example, a *SourceFile* or a *Method* instance.
* The optional **Type** of the inner elements parameter or a specific **IElementFilter** interface descendant. The “*Type*” parameter can be specified by the *LanguageElementType* enumeration (e.g. *LanguageElementType.Comment*) or a *System.Type* instance. For example:

**CSharp** code:

|  |  |
| --- | --- |
| 1  2  3 | Class activeClass = CodeRush.Source.ActiveClass;  ElementEnumerable xmlCommentsEnumerable = new ElementEnumerable(activeClass, LanguageElementType.XmlDocComment);  ElementEnumerable throwExceptionsEnumerable = new ElementEnumerable(activeClass, typeof(Throw)); |

[Show Visual Basic code… »](http://www.skorkin.com/2011/04/how-to-enumerate-solution-and-source-code-items-using-dxcore/#SID3099_2_tgl)

You are also able to pass an array of the *LanguageElementType* or *System.Type* instances.

The *IElementFilter* descendant allows you to specify more conditions for the element than just its type which specifies whether a particular element will be returned during an enumeration process. There are two methods on the *IElementFilter* interface to implement: *Apply* and *SkipChildren*.

The *Apply* method returns the value, which specifies whether the element will be returned when enumerating the scope. The *SkipChildren* specifies whether the enumerator should look inside the element that is passed the *Apply* procedure, or move on to the next element skipping its nodes. For example, here’s the code of the filter that looks for the field members with the name starting with an underscore “\_”:

**CSharp** code:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11 | public class FildsWithUnderscoreFilter : IElementFilter  {    public bool Apply(IElement element)    {      return element is IFieldElement && element.Name.StartsWith("\_");    }    public bool SkipChildren(IElement element)    {      return true;    }  } |

[Show Visual Basic code… »](http://www.skorkin.com/2011/04/how-to-enumerate-solution-and-source-code-items-using-dxcore/#SID3099_3_tgl)

There is also the *DefaultElementFilters* class, defined with a few default element filters, such as *TypeOrNamespace* filter, *NonPrivateMember*, etc.

* The third optional parameter is the “UseRecursion” boolean parameter, which specifies whether the enumeration search should be performed recursively.

**Conclusion**

Enumeration of the solution and code blocks elements is one of the basic tasks for [DXCore plug-in](http://www.skorkin.com/2010/08/dxcore-plug-ins-overview/), which allows you to select elements to begin work with. Any consequent task, such as modifying elements, navigating to elements almost always needs to get list of an active or specific elements to start working with. Getting an active element allows you to define the condition whether a particular feature should be available, for example. E.g. the [IsNullOrWhiteSpace contract provider](http://www.skorkin.com/2011/02/how-to-add-new-contracts-for-the-add-contract-code-provider/) needs to check if the active element is a *Parameter* element instance, and if it is not a *Parameter*, then it is not available.