Enumerating the Control Hierarchy

Given a WPF window, it seems a reasonable question to ask, programmatically, *What does it contain?* After all, if an objective were, for example, to attach an event handler to a text block to handle, for example, internationalisation, then one would like to know what and where any text blocks are! The same, as another example, for a “generic” button click or validation event handler.

As it happens there are a number of ways of viewing controls - from a parent/child perspective, remember we’re dealing with a .NET Framework entity and they’re not in the habit of employing recursive IEnumerable collections, any hierarchy is generally dealt with by a single step of an ancestry/dependency basis. There is, however, one WPF method that is used for selecting a single control rather than browsing through a one-step hierarchy and that is the method `FindName` of the `FrameworkElement` class. Recall, also, that `FrameworkElement` appears in the inheritance hierarchy of every WPF element be it a `TextBlock`, `Window`, `UserControl` etc. so it can be considered “universal”. Another useful aspect of `FindName` is that it is actually recursive! One of a very few number of .NET Framework methods that are. Therefore, when one invokes `FindName`, it will find any descendant element whose name matches the string argument with which it is invoked. This gives rise to an important fact to remember...

The `FrameworkElement` method `FindName` will only work for framework elements that sport a `Name` attribute. That is, the Xaml element must contain the `Name` attribute with the `x` namespace prefix.

`Name` is an attribute defined in the Xml Microsoft Schemas namespace of `winfx/2006/xaml` that is usually aliased in your Xaml source by `x` so it will “appear” as `x:Name`. `Name` is not therefore a “native” Xaml attribute of a Xaml element; it is, rather, an artefact introduced by WPF in order to identify a Xaml element.
In line with the best traditions of the .NET Framework, if a framework element sought for via the string argument of `FindName`...

- Is not found then no exception will be raised but the method will return a `null` value.
- The output is not strongly typed - it will always have a type of `obj` so, inevitably, an up-cast will be required. The only, relatively “safe” up-cast you can try is that for a `FrameworkElement`.

Before we consider the issues of being able to enumerate framework elements’ descendants, I want to raise another warning: There is a distinction that needs to be made between dealing with Xaml, BAML and WPF. Xaml is “just” Xml and we’ve already seen how we can use various techniques, which we will investigate further, later, when considering a variation of using RESX resources, to enumerate elements in an Xml document. However, one would have to bear in mind that doing such as XPath queries against a `Name` attribute, that said attribute does not appear in the default XML namespace. This would therefore require us to “adjust” the query domain accordingly. This can be done but it does introduce a further level of consideration and complexity into the enumerating of Xml elements upon the basis of such an attribute.

Certainly, we could use the underlying Xml to build a sequence, or list, of named framework elements but all that yields is a list of names that one may subsequently, safely use as an argument for the `FindName` method. This is therefore a three-step process…

1. Parse the Xaml as Xml and then build a list of named elements.
2. Use such a list element name to assign the output of a `FindName` to a value type that is then up-casted, at least to a `FrameworkElement`.
3. Perform any desired manipulation, such as setting an event handler, against this up-casted, “temporary” value type.

Even so, this approach exposes the issue that if the library we’re dealing with is from WPF the manifest `g.resources` are BAML, not Xaml. We would therefore have to “convert” the BAML back to Xaml before being able to use Xml enumeration to find named elements. You can do this; it requires that after one does a `XamlReader.Load` against the `Baml2006Reader` stream, it is then necessary to “serialize” the framework element using the `XamlDesignerSerializationManager` class in the `System.Windows.Markup` namespace. If your curiosity is piqued then it goes something like this…

```fsharp
let xaml = new StringBuilder()
let serialiser =
    new XamlDesignerSerializationManager()
```
I've tried this and it's a wasteful process I'd not recommend - apart from anything else, whilst BAML may just be optimised Xaml, once you have invoked \texttt{XamlReader.Load} (which, remember, is also implicitly invoked by \texttt{XamlReader.Parse} for parsing a string of Xaml - it first optimises the Xaml into BAML then, “under the covers”, invokes \texttt{XamlReader.Load}, what results is not really Xaml; it is what's known as a WPF \textbf{Object Graph} which is an operating system and device specific representation of how the Xaml will appear on the runtime system and device. Consequently, although a Xaml derived list of element names may provide some “linkage” between the world of Xaml and the WPF world of rendering, this is a shallow reference. In order to do anything “constructive” at runtime, one must ultimately address the WPF object graph - and do so before the object graph is rendered on-screen. Essentially, let's get our information first-hand from the horse's mouth rather than rely upon hints from the horse's arse - which we then have to feed back into the horse's mouth anyway!

If that analogy were “realistic” then it would be reversed so that Xaml is the horse's mouth and WPF the horse's arse - however, I can't, in good faith associate WPF (or Xaml) with a horse's arse; WPF is a vast and complex entity that spans many useful and even mystical cross-platform display features and functions. However, the WPF and, to a lesser extent, Xaml application programming interfaces are hugely complex, obtuse and, to my mind, overly biased towards object-oriented, single instance usage with annoying and oblique class “wrappers”, inheritance and artefacts to get around object-oriented limitations in dealing with hierarchical structures. Within these two .NET API domains there is, necessarily, some overlap but you should appreciate that one cannot “arbitrarily” switch between the WPF and Xaml API's in dealing with an entity that has “a foot in both worlds” since they are, largely, fundamentally different representations of the “same” entity and each deals with the entity in a different manner.

Thus, for our sins, we must now consider aspects of the WPF API that may be regarded as painful…

The WPF Visual and Logical Trees

There is much documentation and even books have been written on these subjects since Avalon was introduced to .NET in 2003. For me I have one goal in mind - “how do I create a recursive F# type that characterises a framework
element hierarchy - the name, if it exists, the type and a ‘reference’ to the underlying framework element instance’? As that’s all I want, I’m going to disregard everything else about the subject matter - especially given its object-oriented bias and lack of hierarchical structure. If you “like” C# and the subject interests you then I’d point you to an article written in December 2007 on http://codeproject.com by Josh Smith. This time, I’m not going to warn you - I leave that to a quote by Josh in his article: “The existing documentation about the visual tree and logical tree in the Windows SDK leaves much to be desired”.

The Visual Tree

Let me state, here and now, I’m not interested in the Visual Tree - ever. Such a statement warrants some explanation - as follows: By example, I want you to re/open your BamlReader.fsx script and execute everything in it down to the point of declaring the new Application and running it for the window WinPg or WinUc - that is, up to and including the declarations, only, for WinPg and WinUc. Then skip to the block of code where we’re dealing with Silverlight. In this block, execute everything from the declaration of slSample, slIsLoaded to the declaration of suc wherein we do a tryUnbox for the UserControl - inclusive.

Now modify the application declaration and run to the following...

```fsharp
let slWin = HostWindow suc.Value
let cc0 = System.Windows.Media.VisualTreeHelper.GetChild(slWin, 0)
let slApp = new Application()
[<STAThread>] slApp.Run(slWin)
let cc1 = System.Windows.Media.VisualTreeHelper.GetChild(slWin, 0)
let cc = System.Windows.Media.VisualTreeHelper.GetChild(slWin, 0)
```

Execute lines #1 and #2 of the above and the FSI output for the cc0 declaration will show...

```
val cc0 : int = 0
```

Therefore, as far as our visual tree is concerned, at this point our window has no child framework elements! Now run lines #3 and #4 to show the WPF host window and, after the window is shown, close it from the system menu. This will release the [<STAThread>] generated thread lock on your FSI code, so you can then execute lines #5 and #6. FSI output should resemble...
Now, having “shown” (and closed) the window hosting the Silverlight control, it appears to have one child framework element. Do you expect that to be the hosted Silverlight user control? Forget the `DependencyObject` type - just regard it as a framework element. Have a look at FsEye for the value type named `cc` as in Figure 93. What we actually have is a `Border` control - not the hosted Silverlight `UserControl`!

There is therefore one immediate “restriction” as regards using the visual tree - it will only allow one to access child controls, not descendants and will only do so after the window has been rendered on-screen. Before rendering, we have nothing since no “visual” control yet exists! This, to me, is problematic; I’d like to be able to enumerate framework elements descendants not just “first-level” children and I’d like to be able to do it before a window is displayed on-screen! Furthermore, whilst I accept that a `Border` is a “valid child control”, I’d prefer the “first” visual tree child of the window to have been something that would have some “relevance” to the Xaml I used; I created an empty window, set its dimensions and title then set the content to be a `UserControl` and in none of that have I ever explicitly encoded a Border control!

It’s fairly clear to me that the Visual Tree is therefore just an artefact that can be used to iteratively enumerate “visual” elements of a WPF window and the “visibility” of these elements can only be determined post-rendering. I’m therefore going to disregard the Visual Tree and anything it may have to offer.

The Logical Tree

Let’s hope that the logical tree has something to offer that is more in-line with the encoded source Xaml: Add the following to the end of your `BamlReader.fsx` script...

```fsharp
let ltc = System.Windows.LogicalTreeHelper.GetChildren(slWin)
```
Run that and, in FsEye, you’ll see that it “resolves” to a sequence of UserControl - as in Figure 94; this looks a lot more promising - after all, our hosted Silverlight control is actually a UserControl! Unfortunately, FsEye is unable to show the output of GetEnumerator so, in your script change the ltc declaration so, in your script change the ltc declaration to the following…

```fsharp
let ltc =
    System.Windows.LogicalTreeHelper.GetChildren(slWin)
    |> Seq.cast<System.Windows.Controls.UserControl>
    |> Seq.tryHead
```

FsEye will then show you this control as in Figure 95. Can we be sure that this is our Silverlight user control? I suppose one characteristic is that the child of the user control is a Grid with a name of LayoutRoot. Unfortunately, FsEye can’t invoke most of the methods of this entity - I imagine that is because it’s running on a different thread hence it will throw many exceptions in trying to evaluate members of ltc. Instead, try this in your script…

```fsharp
System.Windows.LogicalTreeHelper
    .GetChildren(ltc.Value)
    .OfType<Controls.Grid>()
    .Min()
    .Name
```

It should show you that the "next" child, being a Grid, has the desired name of LayoutRoot. Thus, the logical tree helper, System.Windows.LogicalTreeHelper, method GetChildren, in nearly gets us what we want - it’s just not recursive. However, one final point; we’ve tacked this code onto the end of our script and run it after we’ve already rendered and closed the host window - when we were testing the visual tree.

To be sure, you should reset your FSI session then re-execute everything up to the definition of the slWin - excluding the showing of the window for WinUc (or WinPg) then try to declare ltc after the window declaration of slWin - that is, before the declaration of a new Application. You’ll
find we get the same result so there is no need for the window to be rendered before we can try to enumerate its elements.

A Recursive Element Enumerator

The question of recursion is fairly easily solved via an F# recursive function. Consider the following - which you can place at the end of your BamlReader.fsx script...

```fsharp
let rec basicLogicalTree (element : FrameworkElement) =  
    [ for child in LogicalTreeHelper.GetChildren(element)  
        .OfType<FrameworkElement>() do  
        yield child  
        yield! basicLogicalTree child ]

let sllt = basicLogicalTree slWin
```

For a given element, we initiate a list comprehension that yields a child control as found by the logical tree helper for the given element. However, for that child we also want to include in the list its child controls, consequently we use a `yield!` to produce a list of lists. Once the recursive iteration of child controls is completed then the action of the `yield!` is to “compress” the list of lists into a single list. Running the above for the hosting window of our Silverlight user control will produce the list that we show in Figure 96 from FsEye.

Notwithstanding that, this is a list - because I want to be able to immediately see its content in FSI, whereas, in the source I'd be inclined to use a sequence, I do have issues with this representation...

1. There’s no indication of the “root” - that is the Window to which these controls belong - although we could invoke recursion from the starting point of the Grid, say.

2. In any event, the list is “flat” there’s no indication of the underlying hierarchy.

If we really want to display the hierarchy from a “root”, then we have to look at “trees” to model our hierarchical structure.
Recursive Types

Until now we’ve only dealt with recursive functions though I have previously implied that the concept of recursion can be applied to a type and I’ve waited up to this point since here we have a “real” example of what we can model as a hierarchy that would benefit from recursion.

Modelling a Tree

To understand a hierarchy we model it as a “tree”; mathematically this may be called a “graph”. When I use the terms tree and graph I do so in the “loosest possible way” as I’m not going to consider the mathematics behind such representations and I’m going to limit consideration to the kind of tree presented by a WPF control hierarchy. If we were to view our WPF hosted Silverlight user control in a TreeView control – as is done, for example, in FsEye or as we did in our Windows Forms XmlViewer, then we expect to see a “structure” resembling that depicted in Figure 97.

I added a “unidirectional” pointer between each control and its children to highlight a “relationship” between controls. Still, it doesn’t really look like a “tree” - more like a badly made sandwich! However, let’s generalise this by representing a “control” as a numbered “blob” and turn the “sandwich” on its side to resemble what we depict in Figure 98. At the bottom of the tree, we have the “root node” and, going up the branches, we have a series of unidirectional linked nodes.

I’ve numbered the nodes as a tuple; the second element of the tuple represents the “level” of the node - starting at 0 for the root node. Here we have a tree of 8 levels so the outermost nodes are at level 7. The first element of the numbering tuple is an index which is an integer monotonically increasing from 0.

Figure 97: A simplistic picture of the parent/child relationship underlying our Silverlight UserControl control hierarchy with a WPF Host Window.

Figure 98: A depiction of a simple tree with numbered nodes.
The root node number is thus \((0,0)\). For the index, you increment it as you traverse the tree from bottom-to-top and left to right. Thus, whilst index 1 has four children with indexes 2, 3, 10 and 15 the node with index 2 has no children so its sibling on the right is given index number 3. If it didn’t have a sibling then 3 would be assigned to its right-most cousin. Since the node with index number 3 does have children, we increment the index along its left-most branch. If you don’t like this convention don’t complain to me - it is, as said “a convention” and it happens to work very well!

I have found the hardest thing about this convention is that it imposes a mind-set about modelling trees and, when it comes to populating a tree in code, trying to assign node numbers can lead to confusion, complication and frustration. In fact, in my experience populating a tree is the hardest thing you can do with a tree. I was, for a long time frustrated by “internet examples” - whatever few of them there are for F# trees because when it comes to declaring nodes on a tree the “example” hard-codes node values and this is useless to me; I have a source - the WPF control hierarchy and I’m damned if I’m going to hard-code each node manually - what the hell is the point of that? In the event, “initial” population of our tree from a WPF window will require four lines of “easy” code - depending on how much word/expression wrapping you want. The code may be “easy” but it takes a while to wrap your thoughts around it - beer may help!

Given this representation of a tree then there’s a number of things I want to note about it…

- You’ll note that each node is connected to another although, if we only had one node, say a window without any controls, then it has nothing else to connect to as there is no control hierarchy so it would exist as an isolated root node.

- Nodes are always connected via “connectors” except in the case of us having only an isolated root node. Although our connectors exhibit the attribute that each node has, at most, a single input connector this need not, generally be the case. Generally, a non-root node must have at least one input connector and zero or more output connectors.

  - The “root” of our tree - which is, itself a “node”, is characterised by the fact that it has no “incoming” connector - it has no “input”.

  - Tracing a connection path from the root will eventually cause us to “arrive” at a node that has no outgoing connector - assuming the tree is finite although there’s no reason why this “must” generally be the case; this characterises a “terminating node”. In our model of a WPF control hierarchy such terminating nodes must exist - they are the WPF controls that have no children - even if a root node is, itself, a terminating node - our “isolated root node”.

• The connection between nodes has a “direction” which I’ve represented with an arrow; this tells me how to navigate from one node to the other - for example, how to get from the \((3,2)\) to \((4,3)\). I’ve said our connectors are unidirectional but it is possible, generally, for a connector to have no direction (no start and end arrow) or even bidirectional with both a start and end arrow. Furthermore, in F#, we could represent a connector via a type that exposes “directionality”, furthermore, such a type may have, for example, a `Scale` property - say a percentage such as we have already used for scaling a window - so as to identify a path preference or a “weight” when traversing a tree.

• Each node has a unique identifier; we’ve used a numbering tuple that implicitly contains logic about position. Now, while this numbering also serves as a useful indicator of position in the hierarchical view, when we deal with the tree in code initially it is fairly intractable in terms of evaluation. Therefore, as having a unique node identifier is generally a “good thing” one may use a GUID to represent such (see “Unique Identifiers” on page 383).

• When we tag each node with a unique identifier we don’t really care, at the moment, what the node “contains” - it could just be a string or it could be a tuple so the “blob” behind the node could, effectively be anything - even a user-defined type.

• A node may have one or more children - a terminating node will have zero children since, by our definition, it has no output connector. By children, we explicitly mean the nodes nearest neighbour that is traversed by following a single output connector from that node path one level only. Now, a node may have more than one child; for example, \((17,4)\) has 4 children - it has four output connectors to child nodes. In general, the children of a node is a set of zero, one or more nodes.

• In a similar manner, we can define a nodes’ parent; in our WPF control model the parent (note the singular usage) is the level-adjacent node that is arrived at by traversing an input connector in “reverse”. By definition, the root node has no input connector so it cannot therefore have a parent. So, in our WPF control model a nodes’ parent is a single node or none. In general, a node can have “parents” as opposed to a single parent or none but, conceptually, for the WPF control hierarchy, this doesn’t make sense since it implies that a single WPF control is a child of two distinct parent controls and I can’t think of how that’s possible. Generally, including the case of children, we can represent such as an F# list or, potentially, especially if we thought we were going to deal with infinite paths, a sequence as a list/sequence can be empty or have one or more elements.

• Note however, that both children and parents explicitly only refer to adjacent connected nodes - a child is not the same as a grandchild and a
parent is not the same as a grandparent. Therefore, we have to adopt the “usual” concept of descendants for all children and their children recursively and ancestors for parents and all their parents recursively. If children were represented by a list, say of the unique node identifiers, then descendants would comprise of a rather complex data type! For example...

- The children of \((17,4)\) are \([ (18,6); (19,5); (20,5); (21,5) ] \).
- The descendants of \((17,4)\) are \([ (18,6); (19,5); (20,5); (21,5), [(22,6), [(23,7); (24,7)]] \).

You can see this can quickly become very convoluted and thus subject to typing errors! In fact, this is the biggest problem in attempting to “manually” populate a tree.

There are many, many more things that can be said or writ regarding trees but, from here-on in, let’s restrict ourselves to the “simplistic” requirement of modelling a WPF control hierarchy.

**A Type to represent a Control Node**

Let’s firstly consider how we want to represent a node. We know that we “should” have a unique identifier for a node and it would also be useful if we explicitly identified the \(x:\text{Name}\) of the control “behind” the node. However, bear in mind that it is not a requirement that controls must be named so the possibility exists that the node may have no name. Thirdly, let us associate the underlying WPF framework element with the node - this will permit us to manipulate the control, assign event handlers and so forth. For these requirements, let’s use an F# record type - add the following onto the end of your `BamlReader.fsx` script...

```fsharp
type ControlNode =
    { Id : Id
      Name : string option
      Element : FrameworkElement }

static member Assign (element : FrameworkElement) =
    { Id = Id.New
      Name =
        if String.IsNullOrEmpty(element.Name)
        then None
        else Some(element.Name)
      Element = element}
```

| Id of System.Guid |

**static member Empty = Id(System.Guid.Empty)**

**static member New = Id(System.Guid.NewGuid())**

**static member tryAssign (text : string) =**

```fsharp
match System.Guid.TryParse(text) with
| (true,id) -> Id(id), true
| _ -> Id.Empty, false
```

The first thing to note is that we’ve apparently “concatenated” two type declarations by using the **and** keyword! You can do this “generally” but usually you’ll only see this kind of construction when you are defining two or more inter-related types; the type **ControlNode** relies on the type **Id**. We could alternatively declare **Id** separately before **ControlNode**. You should also note that the use of the **and** keyword is not restricted to the “joining” of types but can also be used for the “joining” of inter-related functions - even recursive ones.

I’m “splitting” the declaration of our node modelling type from a type to model the WPF control hierarchy, since later one may choose to extend functionality by introducing further members to the **ControlNode** type.

**A Recursive Union to model the Control Tree**

I’m not sure I can adequately explain this concept in writing and most of the trivial F# samples “out there” won’t help much in this regard either. As with most things like this in F# it’s far easier to have some “decent” exemplary code “thrown at you” and then run it and try to figure out therefrom what is going on - this is where that suggested beer might help! Well, firstly, the declaration of a single-case union to model our required control tree is a one-liner so you won’t get much out of this...

```fsharp
type ControlTree =
| ControlTree of ControlNode*seq<ControlTree>
```

All you do get out of this is that the union case has an underlying type that is a tuple whose first element is a **ControlNode** instance and whose second element is another **ControlTree**! That is, a **ControlNode** tupled with another **ControlTree** - *ad infinitum*, literally. Therefore, like an infinite sequence, we have an infinitely self-referencing value type. It’s important to note in our type that the second tuple element is a sequence of control trees - in most “common” examples you won’t see this. We need a sequence since a control node may have multiple children and, otherwise, we would be restricting ourselves to a “chain” since any node can then only have a single child.
In and of itself it provides a very elegant mechanism for modelling a tree - if it works! How do we know if it works? Well, we have to populate an instance of a `ControlTree` with, for example, our WPF window hosted Silverlight user control. I am not even going to think of doing this manually - that is a quagmire of confusion - rather, let's add a static `Populate` member thus...

```plaintext
static member Populate (win : Window) =
  let rec logicalTree (element : FrameworkElement) =
    seq { for child in LogicalTreeHelper.GetChildren(element)
      .OfType<FrameworkElement>() do
      yield ControlTree
      ( ControlNode.Assign child, logicalTree <| child )
    }
    ControlTree
    ( ControlNode.Assign win, (logicalTree win) )
```

It's pretty trite to say "that's all there is to it"; getting "here" takes much head scratching and "playing around" - so reap the benefit and simply try to understand how this code works: First of all, I'm insisting that a control tree is populated from a `Window` instance - that is the argument type expected by the member. This is an "artificial" restriction and you could populate the tree "from anywhere". The member contains two expressions...

1. A recursive sequence comprehension in the function `logicalTree`.
2. The generation of the `Populate` member output as the instantiation of a new `ControlTree` whose (top-most) `ControlNode` is the parsed window instance and a `ControlTree` as the output of an invocation of the recursive sequence comprehension - `logicalTree`.

Figure 99: Swenson's FsEye view of our ControlTree hierarchy with the second tuple element, being a recursed ControlTree, expanded to "pick" up the child control tree sequences in turn. FsEye will display, at this detail level, the "correct" runtime types for these controls.
To be honest, I initially arrived at the recursive `logicalTree` via instinct and some experimentation; it would take a far better man than me to tell you exactly what’s going on “under the covers”. From my perspective, other than the maxim “if it works, leave it alone”, the sequence yields (and note that it’s not a `yield` - as for our previous recursive function) a `ControlTree` instance comprising a child control and the recursive invocation of `logicalTree` to do the same for any and all child controls of this control. Ultimately, the proof of the pudding is in the eating; add the following invocation to your script and execute it:

```fshARP
let tree = ControlTree.Populate slWin
```

In FsEye then expand the second tuple element for each `ControlNode` successively - you should see what’s depicted in Figure 99 - you can follow, all the way down to the Silverlight `TextBlock` and `Button`, what the `ControlNode` tuple element (the first element), actually is. At this level, that is, for the `TextBlock` and `Button`, you’ll see that the second element, the node’s `ControlTree`, is an empty sequence. It’s pretty clear from the tree view that the `ControlNode` for the `Grid` control has two child controls whereas the other controls only have one.

In terms of “layout”, the tree view is showing the “root” window and all of the descendant controls in the manner depicted in Figure 97.

Now that we have a tree and a function to populate it from a WPF window, we should provide some union members for the selection of nodes, parents, children, ancestors and descendants. I’m not too sure about tree traversal members for our immediate purposes in anything other than describing a path that can be used to include ancestors or descendants. In any event, this is just a starting point; it creates a recursive structure with uniquely identifiable nodes but it suffers some “drawbacks” and constraints that we shall address in detail. This will greatly complicate the `Populate` member and require it to be more flexible in the definition of a type to model a tree however, in going through that process, you should then be able to identify the precise steps and methodology being undertaken in the recursive population of a tree type. Apart from anything else our Silverlight user control is trivial and whilst it may be enough to demonstrate the concept, we need a “bigger” more complex WPF window with a wider variety of controls present thereupon to full demonstrate the tree type. Let’s
address this first...

A Rich WPF Sample Window

Being inherently lazy, I don’t particularly want to code a significant WPF application, especially in C#, to demonstrate and extend our WPF handling capabilities. Consequently, I found the Microsoft “WPF Controls Gallery Sample” on the MSDN documentation. The download is an executable SampleViewerLite.exe and, when run, it prompts you for a target location to unzips the content - I selected U:\Learning\F#\Code\Viewer. Then open Blend and then open the project file sdkxamlbrowser.csproj in the csharp folder of your extraction location. This will invoke a one-way upgrade to Visual Studio 2015 that you should allow to complete.

Subsequent to the auto-upgrade, which should complete without error, open the Application.xaml file; it contains the code, at the top, StartupUri = "Scene1.xaml" so we know that Scene1.xaml is what will be “run” when we execute the project. Now do a File ↝ Save all to save the new solution generated by Visual Studio to your csharp directory. You can now run the project to see what it produces. What you’ll see is a window entitled XAMLPad Express that will demonstrate a wide number of WPF controls and drawing elements such as a Canvas.

What’s in Scene1.xaml? Firstly, in Blend, in the Designer/Xaml view for the source, you’ll see that the top-level Xaml element is a Page - not a Window nor a User Control. Well, we’ve shown we can “host” a Page control via F# but you’ll also notice an x:Class attribute for the Page and the code for that is in the C# file Scene1.xaml.cs. Opening that C# source may fill you with despair - bear in mind we’re looking at learning F# not C#. There’s a significant amount of code that deals with event handlers and a nasty looking C# method called ParseCurrentBuffer().

Remember, if we want to run something like this from F# we have to delete the x:Class Xaml attribute and, if we do that, then we’ll have to code everything that’s in Scene1.xaml.cs in F# and I’m buggered if I’m going to do that! There’s an easier way - rather than have a “host window” in F# we can put one in C# so the application “starts” a Window that just contains the Scene1 Xaml. This is actually, even if you don’t know C#, a very simple process. As follows...

1. Right click the project node for sdkxamlbrowser in the solution explorer and add a new WPF Window named Main.xaml.
2. Open Main.xaml and in the Xaml editor...
   i. You’ll see that the x:Class attribute value is .Main - change it to SdkXamlBrowser.Main where SdkXamlBrowser is the same namespace as is specified in Scene1.xaml.cs.
   ii. You’ll see a blue squiggly for the declaration of xmlns:local="clr
ENUMERATING THE CONTROL HIERARCHY

-namespace:" for the Window element; this is a warning because no namespace is specified; in fact, if you look at the project properties, you'll see there is no default namespace specified. Just delete this declaration.

3. Now, in the Main.xaml.cs source, because the project has no default namespace, you'll see there is a namespace declaration but no namespace is given. Just type SdkXamlBrowser after the namespace keyword.

4. Edit Application.xaml and change the StartupUri to Main.xaml.

You should now be able to run the project and you'll see, this time, a blank Window entitled Main. That's "stage 1" complete, now...

1. Edit Scene1.xaml and change the Page element to a UserControl element; just over-type the opening element declaration from Page to UserControl.

2. Delete the WindowTitle="XAMLPad Express" attribute of the "changed" UserControl.

3. Re-edit the Main.xaml source and change the attributes for Title, Width and Height for the Window element in the Xaml to Title="XAMLPad Express" Height="768" Width="1024".

4. Rebuild the project.

Having rebuilt the project if you look at the Assets window you will now notice a new entry in the Project category for the UserControl Scene1. Finally;

1. Drag and drop the Asset Scene1 onto the Main window as a child of its bounding Grid control.

2. In the Xaml you'll see...

   i. A new namespace has been declared - xmlns:local="clr-namespace:SdkXamlBrowser" which will "pick up" the user control.

   ii. As a child of the grid, that a number of default attributes have been assigned – delete them all. Just leave the element declaration as <local:Scene1/>.

3. Edit the Scene1.xaml.cs C# source code and find the default public constructor public Scene1() { } - change it so it reads: public Scene1()
   { InitializeComponent(); }

Now re-run your project and you'll see the XAMLPad Express "window" almost as it originally was as a Page element. Now that it runs, delete the x:Class declaration in the Xaml for the Main.xaml Window element and make the Build Action property for the Main.xaml.cs source None so that we can now use the project from F#.
Let’s return to our BamlReader.fsx script - add the following declaration - change the target assembly path to suit your location of the SdkXamlBrowser WPF Control Gallery Sample executable...

```
let ctlSample,isLoaded3 =
    WpfLibrary.tryAssign(@"U:\LearningF\#\Code\Viewer\cslib\...
    rp\bin\Debug\SdkXamlBrowser.exe")
```

Run that in FSI then, in FsEye, expand the ctlSample node - you should see what is depicted in Figure 101. An attempt to evaluate the BAML resources has thrown an exception - the type of exception is irrelevant - what matters is that an exception occurred. The reason for this is actually very straightforward; although we have “converted” Main.xaml to our “Slimline WPF”, we have not done so for the UserControl Scene1. Any attempt to access WPF elements programmatically where an x:Class attribute is declared in the source XAML will result in an exception.

To this point, we have tried to create, in BamlResources, a sequence of the manifest WPF repository of g.resources and this is clearly not a good idea; we need to be specific - to parse a single manifest g.resources for a specific element only upon demand via a named resource, rather than attempt to do so for every named resource in g.resources. We need to change the WpfLibrary declaration. Now, BamlReader.fsx is becoming somewhat “messy” so let’s create a new script file called Enumeration.0.fsx. In Enumeration.0.fsx I’ve added the following from BamlReader.fsx...

1. The FsEye instantiation.
2. All of the open and #r statements.
3. The BamlResourceName type - unchanged.
4. The WpfLibrary type - but I have removed the members BamlResources and ParseBaml.
5. The ControlNode and Id types.
6. The ControlTree type.
7. The ctlSample,isLoaded3 declaration.
Then, add the following to your `Enumeration.0.fsx` script...

```fsharp
let ctlrr =
    new ResourceReader
    ( ctlSample.Value.GetManifestResourceStream
tuml (ctlSample.ShortName + ".g.resources") )
    |> Seq.cast<DictionaryEntry>
    |> Seq.pick(
        fun de ->
            if de.Key.ToString().ToLowerInvariant().StartsWith("main")
            then Some(de.Value :?> UnmanagedMemoryStream )
            else None)
    |> Seq.cast<DictionaryEntry>
let ctlWin = XamlReader.Load(new Baml2006Reader(ctlrr))
let ctls = ControlTree.Populate ctlWin
```

Here we are explicitly only “pulling” the `g.resources` dictionary entry for the resource whose lower cased name begins with `main` - as in our `Main.xaml` which, when optimised becomes `main.baml`. We then load `Main` as a `Window` and attempt to populate the control tree. When you execute this FSI won’t show you much...

```fsharp
val ctlWin : Window = System.Windows.Window
val ctls : ControlTree =
    ControlTree ({Id = Id
                  bdd85c3f-1482-41cc-b5d9-6883d3641852;
                  Name = null;
                  Element = System.Windows.Window;},<seq>)
```

This is one of the “problems” about using a sequence. If you look at the `ctls` node in FsEye though, you will “see” the tree and you can expand the tree view by looking at `Item1` and `Item2` - the `ControlNode` and `Seq<ControlTree>` at least once or twice anyway! If you expand `Item2` - the control tree sequence you will, eventually (depending upon the capacity and system resources of your machine) overwhelm FsEye and thence FSI. I have to reset FSI after trying to expand the second `Item2` on my box - you may have more luck.

### A Recursive List versus a Sequence

There’s not much point in having a tree structure as a sequence if you can’t traverse it - we should try a list so that the tree nodes are evaluated at declaration rather than “on-demand” - as for a lazy value type such as a sequence.
Given such a change I want to undertake another "major" implementation - I would like, instead of using our GUID based Id, to uniquely reference a tree node - a control, to use the tree "co-ordinate" tuple (see "Modelling a Tree" on page 564); this is a unique positional identifier comprising of an "index" and a "level" for a node within a tree.

As we know that we should now not attempt to also load a sequence of BAML resources let's simplify the code a bit so we can focus on the problem of generating a tree; I have copied the content of Enumeration.0.fsx into Enumeration.1.fsx and made the following changes...

1. I have retained the FsEye instantiation and all of the open and #r statements.
2. I have deleted the BamlResourceName type as this was largely used for dealing with resources names in a sequence.
3. I have, for the WpfLibrary declaration...
   i. Removed the member ResourceManager for the time being.
   ii. I have added the following member...

```fsharp
member µ.Window forName =
    let bamlStream =
        new ResourceReader
            (µ.Value.GetManifestResourceStream
                (µ.ShortName + ".g.resources"))
        |> Seq.cast<DictionaryEntry>
        |> Seq.pick(
            fun de ->
                if de.Key.ToString().ToLowerInvariant().StartsWith(forName) then
                    Some(de.Value :?> UnmanagedMemoryStream )
                else None)
    try
        XamlReader.Load(new Baml2006Reader(bamlStream)) :?> Window
    with _ ->
        failwithf "%s cannot be loaded as a BAML optimised Window" forName

4. I have added the following two expressions following the WpfLibrary declaration...

```
let ctlWin = ctlLib.Window "main"

5. I have deleted all subsequent code.

If you execute all of that, we’ll then have a “valid” Window instance, ctlWin, to work with. I now want to declare a “co-ordinate” as a single case union similarly to how we declared Scale and PixelDimension. Consider the following…

```fsharp
type IntCoord =
    | IntCoord of uint32*uint32

static member Origin =
    IntCoord(0u,0u)

static member tryAssign (location : 'a*'a) =
    match UInt32.TryParse (fst location).ToString() with
    | (true,x) ->
        match UInt32.TryParse (snd location).ToString() with
        | (true,y) -> IntCoord(x,y), true
        | _ -> IntCoord.Origin, false
    | _ -> IntCoord.Origin, false

member µ.X =
    match µ with
    | IntCoord(x,_ ) -> x

member µ.Y =
    match µ with
    | IntCoord( ,y) -> y
```

The IntCoord single case union has, as its underlying type, a tuple of unsigned 32-bit integers (designated as `uint32` with a suffix modifier of `u` - for example, `32u`) - these can have a value between 0 and $2^{32}-1$, inclusive. I want to define the Origin of this coordinate system to be at `(0u,0u)`. I’m also implementing a simple tryAssign member whose argument is an un-typed tuple of two elements. To get the first coordinate tuple element we’ll use the member X and the second, the member Y.

One of the main reasons for using this coordinate as a node’s unique identifier is that I want to be able to simply visually depict the tree structure. Let’s now declare a modified ControlNode type that uses IntCoord...
The first change is in the first field - it’s no longer our previous Id but now an instance of our IntCoord single case union - we call this field the Location for the node. Now, for the Assign member, I’m additionally taking two (implicitly u \| int32) arguments named index that will represent our X coordinate and level that will represent our Y coordinate. With this pair, we can use our IntCoord .tryAssign static member.

Finally, in representing a control node, I want a string that will easily depict the “level” of the node and use the level in a way that will demonstrate that the tree is hierarchical - composed of ancestors and descendants. For this I’ll use as many hyphens as indicated by the value of the level - the Y coordinate of the node, suffixed by a + sign - as in a tree view control. I append to that the actual Location of the node represented as a string coordinate pair and thence either the controls’ Name, if it exists, or the short name of the controls’ type.

We can now re-define our ControlTree thus…

Our problem is now in the Populate member wherein we must determine the appropriate X and Y coordinates (the index and level, respectively) for each node. Doing this should allow you to see precisely how the recursive population algorithm “works” - so we can eruditely expand upon it later. The code is as follows…
There are, essentially, three expressions...

1. A recursive function, `logicalTree` that accepts a `FrameworkElement`, node index and level as `uint32` arguments. The output is a `List<ControlTree>`.

2. A value type named `root` to represent the members’ argument of `rootElement` as a `ControlNode` that is to be placed at the Origin of a control tree.

3. An implicit value type of `ControlTree` as the member output that, for its first tuple element is the root node and for its second tuple element, invokes the recursive `logicalTree` function to evaluate the `ControlTree` instances “hanging off” the root element.

Let's consider the recursive function in more detail; it's a list comprehension for which...

- We declare a value type named `children` that comprises an `IEnumerable` of a tuple of three elements - a `FrameworkElement` and two `uint32` values. In its evaluation we use the WPF Logical Tree Helper to return all of the children for the function's argument as this “sequence”. For each element in this sequence, we evaluate...
  - The node index as the functions `index` argument (which is initialised with zero) plus one to account for the fact that the “parent” assigned index number “is already taken” so we need the “next” index number and, added to this, is the element number of the `IEnumerable` that is yielded by the Linq `Select` through the lambda expressions’ anonymous function argument of `idx`. This Linq `Select` thus ensures
that children are numbered in a monotonically increasing sequence 
of integers greater than the `index` parsed as the functions `index` 
argument.

- For the `level`; this only needs to increment for each “group” of chil-
dren - it is a number that is assigned equally to all the `IEnumerable` 
child elements although, as for the `index`, we do have to add one to it 
since the function parsed `level` is “occupied” by the control against 
which we are executing the WPF Logical Tree Helper `GetChildren` 
method.

- For the `IEnumerable/sequence yielded in children`, we then iterate 
through each element such that...

  - For each child control of the parent against which the child is evalu-
ated through `GetChildren`, it’s `index` and `level` (these have the prefix 
child), we yield the tuple of two elements...
    1. A `ControlNode` instance against the enumerated child element, 
its `index` and `level`.
    2. A re-invocation of the `logicalTree` recursive function using the 
child element, its `index` and `level` in order to determine all of 
the children of the child item.

There are two components to the complexity herein...

1. Our recursive type is actually a list of control trees so we can’t do recur-
sion as we have before outside of a list (or a sequence) - we have to do 
recursion as part of an iteration (a `for` loop) within a comprehension.

2. This type of recursion is inherently, as far as I can make out, not tail-
recursive; it will eat up your system resources in generating the output 
list (or sequence) but we cannot, as we have intimated from FsEye, use a 
sequence - in fact, this will become more evident when we attempt, in a 
moment, to encode a member to “display” our tree.

We could “throw anything we want” at the population as long as it “looks” Ok, 
however, we should really encode a member that allows us to visualise the tree 
by recursion though the nodes to verify we have “what we expect”. For this, we 
use the following member...

```csharp
member µ. Depict =
  let output = new Text. StringBuilder() 
  let rec traverse (node : ControlNode) (branch : ) = 
    ... List<ControlTree>() = 
      output.AppendLine(node.ToString()) |> ignore 
    for item in branch do
```

> Web Sample
match item with
  | ControlTree(leaf,twig) ->
    traverse leaf twig
match μ with
  | ControlTree(root,stem) ->
    traverse root stem
output.ToString()

Depict uses a StringBuilder to each line of which we append the ControlNode ToString override that we coded; this will highlight the “level” of the node and thereby show the varying “indentation” of the output. Now, in order to “traverse” the tree we must once again use recursion. Furthermore, as we have a ControlNode*List<ControlTree> rather than a simple “chain” such as in ControlNode*ControlTree, we have to iterate through each nodes’ children recursively. Then, the recursive function traverse, given a “decomposed” tuple of ControlNode and List<ControlTree>, will append a line to our string builder of the “current” ControlNode (named node) and then iterate through the current List<ControlTree> named branch. In this iteration we once again “decompose” the control tree tuple into the ControlNode named leaf and the List<ControlTree> called twig and re-invoke the recursive function traverse for our leaf and twig.

To “initiate” the Depict property we match the “root” of our control tree by decomposing it into the tuple elements named root and stem - these are then used to kick-off our recursive traverse. Finally, we just output the string builder contents as a string.

As they say, the proof of the pudding is in the eating - so add the following expression to your script then execute it all...

```fsharp
let cts = ControlTree.Populate ctlWin
let str = cts.Depict
```

FSI will show you some significant output for the value of cts that you can scroll through but you may find it rather confusing with all the brackets! Also, it truncates the str output to the first couple of lines so, just type str;; in the FSI output window and press Enter and FSI should show you the following extract...

```fsharp
> str;;
val it : string =
  "(0,0) Type: Window
  ++ (1,1) Type: Grid
  ----+ (2,2) Type: Scene1
  -----+ (3,3) Type: Grid
  ------+ (4,4) Name: DocumentRoot
  ------++ (5,5) Type: TextBlock
```

---

*Learning Visual F# 4.0: Foundations I*  ◆  978-1537049960  ◆  Copyright © 2016, Chris M. Shattock
Now, this looks very promising! In fact, in Figure 102, I’ve expanded the control hierarchy for the `Scene1.xaml` source. You’d think, at first glance, that there are some discrepancies between what our Depict shows you and what the Objects and Timeline window shows - for example, consider the control named `cc` towards the bottom of the hierarchy. In the Objects and Timeline window, it is shown as a single, childless entity with immediate foregoing and following siblings of `Rectangle` instances. However, our Depict output is...

This indicates that whilst `cc` at (15,9) is “surrounded” by `Rectangle` siblings at (14,8) and (16,9), it also has one child, a `Grid` at (16,10) that, in turn has a child `Button` at (17,11). There is no evidence of these in the source Xaml however, if you inspect the `Scene1.xaml.cs` source, you will find in the `ParseCurrentBuffer()` method the statement (line #53) `cc.Children.Add((UIElement)content)` - so the code-behind initialisation is actually adding controls to `cc` - which is picked up by our Depict even before the window is rendered on-screen.

I’m reasonably happy that our “coordinates” are valid. More to the point, the way we’ve evaluated them during the recursive population of the tree fits in...
perfectly with the “convention” of how such index numbers are evaluated from bottom left-most to top right-most so there is no more work that needs to be done there!

I would warn you, if you were still going to try to use FsEye to expand all the Item2 elements in our tree you’re still going to get into trouble since it uses an IEnumerable GetEnumerator that is lazy and it will still have the thread issues with WPF objects.

Now, in keeping with the title of this sub-section, what happens if we revert from using a list to a sequence? This is easy enough to check; in Enumeration1.fsx, make the following changes after resetting your FSI session…

1. Change the underlying type of the ControlTree declaration from a tuple that has a List<ControlTree> as its second element to seq<ControlTree>.

2. In the Populate member of ControlTree, change the list bounding of the recursive function to a sequence so it now reads as (also, don’t forget to change the comprehensions terminating ] to }):

```
  let children =  
  seq { let children = 
```

3. In the Depict member of ControlTree, for the recursive traverse function declaration, change the second argument type to seq<ControlTree>.

Now re-run the entire script - it should complete without error and the Depict output should be the same as before. I’m still not convinced, however and I’m going to continue using a list until such time as I’ve finished all of the “other” code for our tree.

Asynchronously Evaluating the Node Index

You may regard this as a slight diversion and we will be dealing with asynchronous processing in more detail later. Until I arrived at the Population member of the last sub-section, evaluating a monotonically increasing index was “problematical” - it forced one to fully understand the recursive process to ensure that the index was being incremented “properly”. One can use a different methodology to evaluate the index and. Although it’s couched in an asynchronous methodology, the evaluation is invoked synchronously.

At this stage just bear in mind that whilst asynchronous processing may appear “parallel” it is not, technically, “parallelisation”; in parallelisation code is explicitly “branched” into distinct processors - CPU’s. Now, if you only have one CPU this isn’t going to help you much, but if you have more than one CPU or a dual or quad core CPU then you can make use of parallelisation since each
CPU maintains a distinct set of resources and threads upon which it executes its code - a set of “thread pools”. An asynchronous process becomes “parallelised” since the thread it’s allocated may be on a thread pool from any one of the available CPU’s/cores. However, no matter how many threads any system can marshal, only one thread can use the execution mode of the CPU or a core at a time. What asynchronicity gives you is that a process on one thread can “hand off” tasks to another thread so it can “get on” and continue doing “some other” processing whilst awaiting a “result” (if required) from a spawned threads’ process.

Now, the fact of asynchronicity is not required by me for this task; what we make use of is that this “model” we use for evaluating an index uses a scheduler that is inherently designed to support asynchronous processes. The model we’ll use is known as a Mailbox Processor; the processor implements an “agent” which is used to process “messages” as asynchronous tasks - often referred to as asynchronous computations. This is part of the Messages and Agents Pattern of F# in which we declare an instance of an FSharp.Core.Control namespace’s class MailboxProcessor that implements a dedicated queue for the reception of messages and of their processing and, where required, the posting of a reply to a message. The MailboxProcessor can handle its queue both synchronously and asynchronously.

MailboxProcessor is a sealed class so there isn’t much you can do with it outside of its pre-defined methods; it contain members such as, for example, Start, PostAndReply (which is synchronous) and PostAndAsyncReply for an asynchronous reply. I’m going to define a queue that maintains a counter with a Next method that starts with the value 0 and every time Next is invoked it increments the counter by one, returns that value and then waits for another message. The point of using the mailbox processor this way is that since requests come in on the queue they are strictly “ordered” so one cannot accidentally concurrently invoke Next and return the “same” value. This is the raison d’être of the queue - it disavows concurrent message handling requests and deals with requests on its queue strictly in order - that “order” being first in, first out.

That said, I’ve copied Enumeration.1.fsx into Enumeration.2.fsx and, apart from the control tree’s Populate method that we’ll deal with in detail shortly, the changes I’ve made are as follows...

1. I’ve changed the name of the type IntCoord to Position. I’ve changed its member name X to Index and Y to Level.

2. In the ControlNode type, I’ve change the type of the field Location to Position.

Now, for our mailbox processor I want to classify the types of message that it
can handle via a discriminated union. At the moment, I only need two message
types so I’m adding the following union to the declaration of the type ControlTree...

and

```fsharp
NodeIndexMessage =
| Release
| Next of AsyncReplyChannel<uint32>
```

The case Release has no underlying type - I’m just going to use this message
to stop the mailbox processor so its resources can be released. I’m going to use
a synchronous PostAndReply for the Next message - the reply being the next
unsigned integer value to be assigned to a node as its index. Now, even though
I make use a synchronous call, the reply will come back on an asynchronous
“channel” and in order to effect this I must give the Next message an underlying
type of AsyncReplyChannel<uint32> where, as you can see with the generic
type argument, I am restricting the reply content to be an unsigned integer.
Message processing is all about the “type” of a message and its return type - if
any.

Given the foregoing, I am now going to replace the Populate member with the
following code...

```fsharp
static member Populate rootElement =
    let indexCounter =
        MailboxProcessor<NodeIndexMessage>.Start
            ( fun inbox ->
                let rec loop n =
                    async {
                        let! message = inbox.Receive()
                        match message with
                        | Release -> return ()
                        | Next(reply) ->
                            reply.Reply(n)
                            return! loop (n + 1u) }
                loop 1u)

    let rec logicalTree (element : FrameworkElement) level =
        let children =
            LogicalTreeHelper.GetChildren(element)
                .OfType<FrameworkElement>()
                .Select( fun elem _ ->
                    elem,
                    indexCounter.PostAndReply
                        ( fun reply -> Next(reply) ),
                )
```

Learning Visual F# 4.0: Foundations I  ◆  978-1537049960  ◆  Copyright © 2016, Chris M. Shattock
For `indexCounter`; when we instantiate the mailbox processor we use the generic type argument of `NodeIndexMessage` - this is the union that we declared that encapsulates what messages we want to handle on the mailbox processors’ queue. With this generic type argument, the mailbox processor will not handle messages of any other type. Now, we then specify as the argument of `Start`, a lambda expression that encapsulates an anonymous function which comprises the entire logic of the queue processor. Our queue processor, therefore, does the following...

- The anonymous function will receive as its sole argument the “in-box” of the mailbox processor - this is a handle into the queue into which incoming messages arrive. The function supports two expressions - a recursive `loop` and its initialisation - as detailed in the following...
  - We define a recursive function named `loop` that takes as its single argument an unsigned integer - we don’t specify this explicitly but the
The compiler can impute this type for the value type argument named \( n \). Within \texttt{loop} we define an \texttt{async\{} \texttt{wrapper} that quantifies a task that is to be undertaken asynchronously.

An asynchronous task is scheduled to execute on the “next available” thread. When such a thread is available the wrapped task is executed by that thread then “control” returns to the invoker of the wrapper. The “owner” of an asynchronous wrapper can spawn many, many “concurrently” active threads - each dedicated to the sole purpose of executing its assigned task.

The use of an asynchronous wrapper requires us to use asynchronous variations of commands - such as \texttt{let} since these commands are “imperative” - they are executed in the control flow immediately and synchronously. However, we cannot tell with an asynchronous task, \textit{when} that command is to be executed - the asynchronous task has been removed from the synchronous flow of control that is expected via imperative commands such as \texttt{let}. Furthermore, asynchronous tasks execute on “their own” thread and have no implicit knowledge of the imperative, synchronous thread that spawned them - and vice versa.

For example, the asynchronous equivalent of the imperative \texttt{let} is \texttt{let;}!; I regard the ! as meaning \textit{loiter with intent} - the \textit{intent} is to execute the \texttt{let} command, the \textit{loitering} is due to the fact that since the control flow is not necessarily synchronous, so it has to wait until such time as it is \textit{able} to undertake the execution; for a \texttt{let} assignment, the object of the assignment may not be synchronously “available” so it must wait until it is available.

For the asynchronous task...

- We use \texttt{let;}! to declare a value type named \texttt{message} that is assigned as the next item off the \texttt{inbox} queue via the \texttt{inbox.Receive()} invocation. \texttt{message} has a type of \texttt{NodeIndexMessage} since that’s the type this mailbox processor is set-up to handle. This is a fine example of loitering with intent; we can’t “guarantee” that there is, at any time, anything in the \texttt{inbox} queue against which \texttt{message} can be instantiated using a \texttt{let} - so we must use \texttt{let;}! to state that one must wait until such time as a message becomes available to be processed on the queue.

- Given that \texttt{message} has been assigned then processing on a spawned, asynchronous thread can continue; it does so by executing a pattern match against \texttt{message} to determine the message type and thereby perform the appropriate task. We have two types...
  - \textbf{Release} - for which we just issue the expression \texttt{return ()}. \texttt{return} is an imperative command so it is executed \textit{immediately} - there will be no loitering.
The purpose of \texttt{return} is to terminate not just the asynchronous thread that it’s running on but also to instruct the owner of the asynchronous wrapper to terminate all other asynchronous tasks it is responsible for. \texttt{return} by itself may be catastrophic - it’s a forced shut-down of all spawned threads and the owner’s controlling process of same, yet you may wish to be able to manage the consequences of such an action. You may wish, for example, to manually dispose of resources, to signal an error or status message, to determine what process/branch you should undertake next on the owning imperative, synchronous thread. Consequently, \texttt{return} is always issued with an argument that can be any function. The argument is processed following the shut-down of the owning process. For this mailbox processor, we don’t really care to do anything - we will just allow the “system” to clean up and dispose of what it may/can. Therefore, the argument we specify is \texttt{()} - that is, do nothing! In our case the \texttt{return} \texttt{()} signals the completion of the \texttt{MailboxProcessor.Start} lambda expressions anonymous function - the mailbox queue is thus closed and disposed of.

- \textbf{Next} - in which case we have a message that requires us to handle our counter and deal with its associated underlying type that we’ve named \texttt{reply} - it being of \texttt{type AsyncReplyChannel<uint32>}: The process is...
  - Invoke the \texttt{Reply} method of our \texttt{reply} instance (of an asynchronous reply channel) and give it an argument of the current \texttt{loop} counter - \texttt{n}.
  - Use a \texttt{return}! to signal completion of the asynchronous task; unlike \texttt{return}, this does not signal the termination of all sibling tasks - it just signals that it’s finished undertaking whatever task was required to be executed by it on its thread.

Like \texttt{return}, we specify an argument to \texttt{return}! that instructs the “owner” upon what to do next after the thread has been released. In this case, we specify that we want to re-invoke our recursive \texttt{loop} function but this time give it an argument of \texttt{n + 1u} - that is, we increment our counter by one.

- We can now initialise the \texttt{loop} recursive function and associated asynchronous mailbox processor by invoking the \texttt{loop} function with an initial value of \texttt{1u}.

Having started our mailbox counter we can now use it to assign an \texttt{index} to
a node as a “guaranteed” monotonically increasing positive integer starting with 1 - we can be assured that because of the way the mailbox processor works we’ll never get a duplicate index value. Since we’re using indexCounter to assign the index there is no need for our recursive logicalTree to now accept the index as an argument - the level is sufficient alongside the framework element itself. Thus, you can see in the Linq Select, the invocation of our mailbox processor, indexCounter using the synchronous PostAndReply method.

Now, PostAndReply, like most messaging methods that require a reply, specifies an argument that is a lambda expression that defines, via an anonymous function that takes an AsyncReplyChannel instance as its argument (this channel is “assigned to you” by the system) what message you want to be queued. In this case, we just invoke the Next message type and give as its argument our system-assigned asynchronous reply channel, reply.

Finally, I don’t want to “immediately” output the control tree that’s evaluated, since I’d like the opportunity to cleanly terminate the mailbox processor; hence, we “save” the recursively evaluated control tree in the value type output, then we can issue the IndexCounter Release message that synchronously terminates the mailbox processor - after which we output the result of the members’ computation - the value type output.

Having gone through the above then run the entire script - you should find that it completes without error and shows you the Depict output correctly and no differently than before.

Control Tree List or Sequence - revisited

In my testing I became very suspicious of the “hang” that was evident in some cases when using FsEye and this kind of thing is not easy to “tie-down” - especially if we introduce asynchronous processing tasks. Now I can solve this mystery; when we specify List<ControlTree> in Enumeration.1.fsx and Enumeration.2.fsx (which uses the mailbox processor) then we can complete execution without a problem. In Enumeration.1.fsx we could also complete when using a seq<ControlTree>. Now, make the same list to sequence changes we specified on page 582 to the script Enumeration.2.fsx.

On my system, this doesn’t quite “hang” but I got tired of waiting for it to produce any output when depicting the tree. I now, therefore, think the issue might be in mixing a seq<ControlTree> definition with another asynchronous process - such as a mailbox processor (or, I’ll bet, the mechanism that FsEye may use to evaluate what members of a value type it can). I would therefore suggest the following “rule”…

Video

160. Using a Mailbox Processor to evaluate the Index - Parts I through III.
If you want to use a tree structure based upon potentially infinite sequences of child controls then don’t include in your code any asynchronous processes. You can use counters within the population process as we originally did for evaluation of the node index. If you need to include asynchronous processes then use a tree structure based upon a (non-lazy) list of child controls.

It may be possible to use a sequence with asynchronous evaluation if the entire tree can be built asynchronously but I don’t immediately, obviously see how this can be done given the inter-relation between nodes - certainly not in “one-pass” in the way we’ve been doing it to date. If you think you have to populate the tree in more than one pass then you have a problem and that is immutability: Our structures are immutable - we cannot add nodes although, based upon the underlying types, we may be able to update fields of a node structure or record type. Since our types are immutable I don’t have an issue using a non-negative monotonically increasing integer as a unique index since the entire domain of the tree is constructed in one pass, sequentially. It’s common in SQL and the like, when persisting a hierarchy to “leave enough space” to permit the insertion of nodes - then you also have to handle deletions and updates - let alone how fragmented the indexes for your hierarchies in SQL can get. This is not an issue we have with F# immutable trees but, if you need mutability, you may have to resort to using “pure” .NET types and a union is not one of those!

In our final refinements, following of control enumeration, we’ll stick with a list of child controls in the tree and continue to use our mailbox processor to evaluate the index.

Refining our Control Tree

The basic principles of our tree are now in place but there are a number of potential issues and enhancements I’d like to address, primarily the fact of storing the framework element as part of the tree node - the ControlNode record type. I’m mindful of the fact that I’m using a list, for the moment, rather than a sequence for the control trees of child nodes and this will have an impact upon system resources. However, in order to aid in verification of node selection I would like to maintain a set of control names and a set of the encapsulated control types - at their short name as a string. Furthermore, I’d like to validate a Position as in referring to an existing node in a tree. In order to support such validation I will need to maintain the tree structure using lists rather than sequences for the child control trees.

Traversing a tree is “hard enough” without having to “drag along” the framework elements themselves. In fact, what we really want from a tree is simply its
structure and a unique identifier for the node within a tree for which we can use our Position single case union. The “essential” output of a tree, to my mind, demonstrated in what we have in the Depict member that enables us to visually quantify a trees’ structure and, for that, all we need is the node position, its name (if it exists) and the short name of its type. I would however, also like to add the position of each nodes parent.

Now, bear in mind here that our WPF oriented tree structure implies the existence only of a single parent per node - except for the root element, so I won’t deal with the possibility of permitting multiple parents; let’s put it this way - our tree wouldn’t support a record of your genealogy unless you were a creature that reproduced asexually!

Much like in a database, since we have a unique identifier for a tree node, we can use that fact to refer to a “lookup table” of referenced framework elements. Such a “table” can be “flat” as there is no need to incorporate the hierarchy as this would be defined in the three structure via the unique key of node position. We can maintain such a flat representation of framework elements in a list - in fact, we’ve already done so in looking at the recursion of the WPF Logical Tree Helper output before we tackled the subject of representing the content hierarchically.

More than the content of the node record type there is also the question of traversal of the tree and there are a number of methods we should support, such as evaluating a node’s children, parent, ancestors and descendants. We should also provide a method that can return a selection of nodes based upon the control type - for example, we might use the output of such to assign an event handler to every button in a tree. Tree traversal is a matter than can be approached in two ways:

1. We can use the characteristics of our Position typed unique key for a node. The nature of its assignment should permit us to “easily” select ancestors or descendants based upon the numeric values of the index and level.

2. We can use the tree structure to recursively walk the tree to pick out a “branch” of ancestors.

One method will be simpler than the other - but we’ll “demonstrate” both but, before doing so, we’re going to have to look, very closely, at the Position type.

All of the foregoing requires some fundamental changes in our methods of encapsulating a tree so we’ll go through the process systematically. Firstly, you’ll need to create a new script file, Enumeration.3.fsx and copy into it, from Enumeration.2.fsx, the FS Eye instantiation, all of the open and #r statements, the declaration of the WpfLibrary type and the declarations for ctlLib, isLoad, ed and ctlWin. From our Program.fs source and the Choices.fs source from our saTriology.Forms project, I’ve also copied the following...
let ([|ValidString|InvalidString|]) = function
| value when System.String.IsNullOrWhiteSpace value ->
  InvalidString
| _ -> ValidString

let IsValidString = function
| ValidString -> true
| InvalidString -> false

let inline (<> v (x,y) = v > x && v < y
let inline (<> v (x,y) = v > x && v <= y
let inline (=> v (x,y) = v >= x && v < y
let inline (=> v (x,y) = v >= x && v <= y

Let’s firstly consider some necessary changes to our Position type...

Refining our Position Key

Let’s copy the Position type from Enumeration.2.fsx across to Enumeration.3.fsx as the following - note that I’ve changed the static keyword Origin name to Root and added a ToString override...

type Position =
| Position of uint32*uint32

  static member Root =
  Position(0u,0u)

  static member tryAssign (location : 'a*'a) =
  match
  UInt32.TryParse
  <| (fst location).ToString() with
  | (true,x) ->
    match
    UInt32.TryParse
    <| (snd location).ToString() with
    | (true,y) -> Position(x,y), true
    | _ -> Position.Origin, false
  | _ -> Position.Origin, false

  member µ.Index =
  match µ with
  | Position(x,_) -> x
member μ.Level =
  match μ with
    | Position(_,y) -> y

override μ.ToString() =
  match μ with
    | Position(i,l) -> sprintf "(%u,%u)" i l

This is very simplistic and, as such, it contains a fatal flaw in using the location to traverse a tree for determining ancestors and descendants. To make this more obvious refer to Figure 103 in which I’ve depicted the structure of our control tree for the WPF Controls Sample Gallery and then, execute your script including the definition of the Position type. Now, in your script add these sample Position declarations…

| let p1 = Position(5u,5u) |
| let p2 = Position(6u,3u) |
| let p3 = Position(9u,7u) |
| let p4 = Position(19u,7u) |

Given these sample positional instances and Position.Root, let FSI evaluate these expressions…

<table>
<thead>
<tr>
<th>#</th>
<th>Proposition</th>
<th>FSI Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p1 &lt; Position.Root</td>
<td>val it : bool = false</td>
</tr>
<tr>
<td>2</td>
<td>p1 &gt; Position.Root</td>
<td>val it : bool = true</td>
</tr>
<tr>
<td>3</td>
<td>p2 &gt; p1</td>
<td>val it : bool = true</td>
</tr>
<tr>
<td>4</td>
<td>p3 &gt; p1</td>
<td>val it : bool = true</td>
</tr>
<tr>
<td>5</td>
<td>p3 &lt; p4</td>
<td>val it : bool = true</td>
</tr>
</tbody>
</table>

I would agree with the propositions of #1, #2 and #4 but I definitely do not agree with the propositions that p2 > p1 or p3 < p4. The reason that I don’t agree is that no account has been taken of the level of the position tuple. In the tree structure the pairs (5u,5u) and (6u,5u) are siblings - as are (9u,7u) and (19u,7u). Being siblings they are not, clearly "equal" to each other and, technically neither of each sibling pair is “greater” or “less” than the other sibling.

**Structural Equality & Comparison**

When we use operators like =, < and >, F# undertakes, by default, equality and comparison checks upon the basis of the structure of the type being referenced.
The structure being referenced in this case is a tuple of unsigned integers and no account is being taken of the tuple elements individually. F# will use a “hash” against a type instance by default to compare two different instances of a type - the hash is a numeric value that is the result of some computation that can be assigned uniquely against any instance of the type. For a hash value one might, for example use the checksum of the string representation of a type. In the FSharp.Core.Operators namespace there is a function called hash that takes an obj as its argument and yields an integer and this will be used to determine if one instance is less than or greater than or equal to another instance. Add the following to your script then run it in FSI:

```fsharp
FSharp.Core.Operators.hash p1
FSharp.Core.Operators.hash p2
```

![Figure 103: A depiction of the WPF Control Samples Gallery tree structure.](image)

You’ll see that the hash of p1 is 329500589 whereas for p2 it is 329500590. Technically, it’s therefore correct to state that p2 > p1.
Custom Equality & Comparison

For the Position type, we clearly need to be more “specific” than just using the “default” equality and comparison operators of =, < and > - we need to override this behaviour. The first thing to note about Position is that it is actually the Index, the first element of the position tuple, that provides uniqueness - the Level is “useful”, but doesn’t bring anything to the uniqueness party.

Do the following in your script...

- Add the attribute pair [CustomEquality;CustomComparison] to the declaration of the Position type. With the semi-colon delimiter, this is just the “shorthand” method of specifying a number of attributes.
- Add the following to the Position type declaration - two overrides and an interface declaration...

```fsharp
override x.Equals(y) =
    match y with
    | :? Position as y ->
        x.Index = y.Index && x.Level = y.Level
    | _ -> false

override x.GetHashCode() = int x.Index

interface System.IComparable with
    member x.CompareTo(y) = // Sort ordering
        match y with
        | :? Position as y ->
            if x.Index < y.Index then -1
            elif x.Index = y.Index then 0
            elif x.Index > y.Index then 1
            else failwithf "Cannot compare %s to %s"
                          (y.ToString()) (x.ToString())
        | _ ->
            failwithf "The type %s cannot be compared to the type %s"
                          y.GetType().Name ... Position
                          <| y.GetType().Name
```

With the Equals override, we’re offering a method of establishing, if required, the equality of the two Position tuple instances upon the basis of both the Index and Level. We undertake a pattern match against the type of the y argument and, if a Position instance, we’ll undertake an equality check based upon both the Index and Level being equal. We’re not interested in doing an
equality check where \( y \) is any other type so we just output \textit{false}.

For the \textit{hash} of a \texttt{Position} instance we don’t need to use anything other than the \texttt{Index} value; the way this is assigned, especially via the mailbox processor, will ensure it provides a unique value that can be used for hashing so we just output the \texttt{Index} value converted to an \texttt{int} (from an unsigned integer).

Now, for comparison; we have to declare that our type supports the \texttt{IComparable} interface and that “within” that interface we have one public member named \texttt{CompareTo}. This is similar to the \texttt{Equals} override but here explicitly declared as a \texttt{member} rather than an \texttt{override}. Its argument usage is as for \texttt{Equals} and we do a pattern match against the type of the argument \( y \) - as for the \texttt{Equals} \texttt{override}. The “trick” in comparison is in the output of the \texttt{CompareTo} method…

- If \texttt{CompareTo} output is a negative integer then \( x < y \).
- If \texttt{CompareTo} output is zero then \( x = y \).
- If \texttt{CompareTo} output is a positive integer then \( x > y \).

Because of this simplistic comparison output requirement, I cannot implement, herein, any check on the \texttt{Level} of a \texttt{Position} instance - I am only using the \texttt{Index} to check for \( =, < \) and \( > \). I cannot for example, encode…

\[
\begin{align*}
  \text{if } x.\text{Index} < y.\text{Index} \text{ \&\& } x.\text{Level} < y.\text{Level} \text{ then } -1 \\
  \text{if } x.\text{Index} = y.\text{Index} \text{ \&\& } x.\text{Level} = y.\text{Level} \text{ then } 0 \\
  \text{if } x.\text{Index} > y.\text{Index} \text{ \&\& } x.\text{Level} > y.\text{Level} \text{ then } 1 
\end{align*}
\]

If you try this, the code will fall over in a heap at runtime; if you try, for example \((5u,5u) < (6u,5u)\) then none of the above conditions when using both \texttt{Index} and \texttt{Level} are \texttt{true} so we’ll end up on the \texttt{failwith}.

\section*{Operator Overloads & New Operators}

We could try an operator overload as in…

\[
\begin{align*}
  \text{static member } (>) (x : \texttt{Position}, y : \texttt{Position}) &= \\
  x.\text{Index} > y.\text{Index} \text{ \&\& } x.\text{Level} > y.\text{Level} 
\end{align*}
\]

You will see a blue squiggly under the \( > \) operator; the F# compiler message for this warning is “\textit{The name (>) should not be used as a member name. To define comparison semantics for a type, implement the System.IComparable interface}”. Well, we’ve already tried that so it appears we can’t use this implementation of it to take account of siblings.
What we have to do is define a new operator and our choice of symbols for this is limited to combinations of !, %, &, +, −, /, <, =, >, @, ^, | and ~. So I’m going to define two new type-specific operators to take account of siblings...

```fsharp
static member (>^) (x : Position, y : Position) =
    x.Index > y.Index && x.Level > y.Level

static member (<^) (x : Position, y : Position) =
    x.Index < y.Index && x.Level < y.Level
```

Having added that to your script we can now check the following propositions...

<table>
<thead>
<tr>
<th>#</th>
<th>Proposition</th>
<th>FSI Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p2 &gt; p1</td>
<td>val it : bool = true</td>
</tr>
<tr>
<td>2</td>
<td>p2 &gt;^ p1</td>
<td>val it : bool = false</td>
</tr>
<tr>
<td>3</td>
<td>p3 &lt; p4</td>
<td>val it : bool = true</td>
</tr>
<tr>
<td>4</td>
<td>p3 &lt;^ p4</td>
<td>val it : bool = false</td>
</tr>
</tbody>
</table>

We now have a pair of operators that we can use for “strict” node comparison that takes account of siblings.

**A Composite WPF Map & Tree**

We now have the tools necessary in order to construct a type that can be used to fully characterise a WPF framework element - be it a window, page, user control or, even, “anything else”. The type will expose the hierarchical structure of the framework element and allow its traversal in any direction, to any depth. It permits selection of a framework element within the hierarchy by control type, position and, where such exist, the name of a control. Via it’s incorporated map one can then access any underlying framework element in the hierarchy in order to, for example, assign event handlers etc.

This code is in the *Enumeration.3.fsx* script. Let’s start with a top-down view of what we want to achieve; Figure

![Figure 104: Layout of the structure of our proposed Control Graph type comprising of a Control Map and Control Tree.](image-url)
104 depicts a collapsed view of our “master” type named Graph, an F# record type whose fields express the inter-related types that we’ll use to characterise a WPF root framework element and all its descendants...

Map A ControlMap that is used as a “flattened” view, via an F# map, of the roots descendants keyed upon their location (in the accompanying tree structure) with a value of the actual underlying WPF entity.

Tree A ControlTree that is the hierarchical structural representation of the distribution of the root elements descendants.

Span A tuple of Position instances; this is to be used for validation insofar as the first tuple element is the Position of the root element and the second tuple Position is the roots descendant node that has the highest positional Index value. The root need not be at Position.Zero - for instance the Graph instance may be that of a subset - a branch from a traversal of a “parent” Graph instance.

Names A set of strings being the collation of all of the tree nodes’ Name property (as a framework element) where such exists.

Types A set of strings being the collation of all of the tree nodes’ GetType().Name property (as a framework element).

I’ve added the following members to the Graph record type...

LocationValidator Used to validate a Position when used as an argument.

NameValidator Used to validate a controls’ name as an argument.

Populate Used to populate a Graph instance based upon a parsed root element as a FrameworkElement.

Children Used to extract all of the children of a parsed node that is specified via the Selector type.

Descendants Used to extract all of the descendants of a parsed node that is specified via the Selector type.

Parent - The parent node of a parsed node that is specified via the Selector type.

Ancestors Used to extract all of the ancestors of a parsed node that is specified via the Selector type.

ControlsOfType For the selection of a sequence of controls where their type matches that of the specified generic type argument - which selection can optionally occur from a parsed node or, by default, the root element.
**PickOne** For the selection of a single node as a framework element of the type specified via the generic type argument. The “other” argument is of the same type used in the selection of Children, Descendants etc.

In order to encapsulate and “standardise” arguments to our members I introduce the Selector union by which a user can specify an argument via...

- Either the Location of type Position that is used to qualify an argument of a Graph member as being a node Position instance, or,
- The Name, of type string, that is used to qualify an argument of a Graph member as being the Name of a framework element in the tree.

We use the alias type of ControlMap defined as an F# Map<Position, Frame workElement>. We also define our recursive type, ControlTree, a single case union of underlying type ControlNode*List<ControlTree> with the members...

**Depict** Used to visually depict the tree’s hierarchical structure.

**GetNode** Used to return a single ControlNode instance for a specified node Position.

**GetBranch** Outputs a segment of the underlying tree structure from the specified node location - it has an output type of ControlTree option. Consequently, the root element of the output may not have a location of Position.Zero.

We then have our type that encapsulates a node in the tree hierarchy, ControlNode - being a record type with the following fields...

- **Location** The Position tuple encapsulating the nodes’ Index and Level.
- **ParentLocation** The Position of this node instances parent node.
- **Name** The Name of the nodes’ underlying framework element - if such exists.
- **Type** The type Name of the nodes’ underlying framework element.

Finally, we have NodeIndexMessage that exposes via a union the two message types of Release and Next that are used in the population of a tree for the evaluation of the Index value of a node.

**The Graph Populate Member**

There are, therefore, a number of significant changes to how we want to characterise our control enumeration. Perhaps the best point to start is with the Populate member for the Graph type. Our modified member is as follows...
Refining our Control Tree

```csharp
static member Populate (rootElement : FrameworkElement) =
    let indexCounter =
        MailboxProcessor<NodeIndexMessage>.Start
            ( fun inbox ->
                let rec loop n =
                    async {
                        let! message = inbox.Receive()
                        match message with
                            | Release -> return ()
                            | Next(reply) ->
                                reply.Reply(n)
                                return! loop (n + 1u)
                    }
                loop 1u )

    let controlMap =
        new Concurrent.ConcurrentDictionary<Position,Framewo...
        ( seq {
            yield
            new
                ... Generic.KeyValuePair<Position,Framewo...
                        ( Position.Root,rootElement )
        }
    )

    let rec logicalTree (element : FrameworkElement) index level =
        [ let children =
            LogicalTreeHelper.GetChildren(element)
                .OfType<FrameworkElement>()
                .Select
                    ( fun elem _ ->
                    elem,
                    (indexCounter.PostAndReply(fun reply ->
                        ... Next(reply)))),
                    (level + 1u) )
            for child,childIndex,childLevel in children do
                let ctl =
                    ControlNode.Assign
                        (Position(childIndex, childLevel))
                        (Some <| Position(index,level))
                    child
controlMap.AddOrUpdate
                    (ctl.Location, child, fun _ _ -> child) |>
                ... ignore
```
yield
ControlTree
( ctl, logicalTree child childIndex ... childLevel ) ]

let output =
ControlTree
( ControlNode.Assign
  Position.Root
  None
  rootElement,
  logicalTree rootElement 0u 0u )

indexCounter.Post(Release)

{ Map =
  controlMap.Select( fun de -> de.Key,de.Value)
  |> Map.ofSeq
  Tree = output
  Names = controlMap.Values
  |> Seq.map(fun de -> de.Name)
  |> Seq.filter(fun elem -> IsValidString elem)
  |> Set.ofSeq
  Types = controlMap.Values
  |> Seq.map(fun de -> de.GetType().Name)
  |> Set.ofSeq }

There are, essentially, four expressions followed by, as before, a call to the indexCounter mailbox processor to terminate processing via a Release message, then the generation of the member output as an instance of a Graph record type. These expressions perform the following processes...

1. We define indexCounter as a mailbox processor - as before; there is no change to this code nor to the NodeIndexMessage union.

2. We define a concurrent dictionary; a ConcurrentDictionary "represents a thread-safe collection of key/value pairs that can be accessed by multiple threads concurrently". We don’t really need the dictionary to be concurrent but we do need a dictionary, since such is mutable and we’ll be sequentially appending controls to it as framework elements keyed upon the evaluated Index and Level. It’s this dictionary from which our F# map for the type instance of ControlMap is derived as an immutable value type. Note that in instantiating the concurrent dictionary we initialise it with a key/value pair representing the root element of our Graph.

3. We next have our recursive logicalTree function. We’ve left in the prior
Refining our Control Tree

changes to support using our `indexCounter` mailbox processor to evaluate the node index. A change here is that we’ve added the field `ParentLocation` to the `ControlNode` record type - consequently, in assigning `ctl` via the `ControlNode.Assign` call, we also specify the parent location as `Position(index, level)` - where the child `index` and `level` are child `dIndex` and child `level`; `level` comes from the `logicalTree` argument and we’ve now also re-included `index` as a function argument so we can pass it between recursive calls.

Another change is that before the `yield!` that re-invokes `logicalTree` for each child, we use the `ConcurrentDictionary.AddOrUpdate` method to update the dictionary. For `AddOrUpdate` the arguments are...

i. The key value that, in this instance is our child framework elements evaluated `Position`.

ii. The value that, in this instance, is our child framework element.

iii. An anonymous function that handles an update process in case the concurrent dictionary already contains the key specified as the first argument. The anonymous function receives two arguments that I’ve “ignored” by using the underscore but, in general, these are, in turn, the “existing” key and value in the concurrent dictionary. Our function simply outputs the current `child` to be saved as the value for the existing key.

The recursive function therefore concurrently updates the dictionary from which we can derive the `ControlMap`.

4. We next declare the value type named `output` that will be used as the point where we invoke our recursive function and save its output to be used subsequently for the member output. We yield a `ControlTree` where the `ControlNode` is the root element and the `List<ControlTree>` is evaluated by the call to the `logicalTree` recursive function.

As to the member output, here we output a Graph record type by assigning each of the fields in turn, for which...

- For the `Map` field you can see that we use a Linq `Select` to produce a key, value tuple that can then be transformed into an F# map using `Map.ofSeq`.

- Given our concurrent dictionary we can also, easily use it to evaluate the `Span` - the first element of the tuple always being the `Position.Root` but the second is the last key value saved into the concurrent dictionary for which we use the Linq `Max` function.

---

8 When you populate the tree the root will always be at `(0, 0)`. It may not be if one subsequently chooses to select a branch of the “primary” tree from a node that is not at the root of the “primary” tree.
• For the Names we simply use the dictionary values piped through a sequence map to just extract the control Name property, then a sequence filter since we’re not interested in controls that don’t have a Name assigned and thence a conversion from a sequence to a set.

• For Types we use a similar approach as for Names - just that we’re pulling out the controls type as a string - the GetType().Name in the sequence map.

Normalising Graph Member Arguments

For all of the “selection” oriented members such as Children, Descendants etc. in the Graph type we should allow a user to specify, as the node against which such selection will occur, either a node Location or a node Name. To govern such flexibility (in C# we’d use member overloads) we’ll use a union - the Selector union with the cases of Location of Position and Name of string. Consequently, wherever you see arg as a member argument, it will have a type of Selector so it could be either a Position or a string - for Location or Name respectively.

I would like to perform validation against the argument case for which I code two private members. Let’s consider the Location case first for which the member is...

```fsharp
member private µ.LocationValidator = function
  | Position(_,_) as position ->
    if position =><= (fst µ.Span, snd µ.Span)
    then
      match µ.Map.ContainsKey position with
        | true -> Some position
        | false -> None
    else None
```

The implied argument using the function keyword is a Selector case of Position; we pattern match this against a Position into the “local” value type named position. We then verify that the parsed location lies within the Span of the graph instance. Now, this is “imperfect” it tells us only that the individual index and level are “valid” insofar as the location potentially lies within the tree but it does not tell us that the referenced position actually exists as a node Location. For that, we can then check that the parsed position is present in our map as a key - in which case the parsed position is valid. We could just use the Map.ContainsKey but I’ve left both checks in place for demonstrative purposes. You should also note that for the first check, position =><= (fst µ.Span, snd µ.Span), there is a logical flaw;
Position implements an IComparable interface which coerces comparison for the \(\Rightarrow\) operator to the code specified in our CompareTo member - and that does not actually perform any check against the level - just the index.

For validation against a controls’ Name we use...

```fsharp
member private μ.NameValidator = function
  | value ->
    if μ.Names.Contains value
    then Some value
    else None
```

This validation is trivial given our F# set of Names as a field of the Graph type - we only have to use the set Contains method to determine validity.

**Tree Traversal**

Before we can consider the selection-oriented methods of the Graph type we should consider the members of ControlTree that allow selection of a node and a branch - the Depict member is unchanged so we need not consider it further. Let’s firstly consider the GetNode member that will yield a ControlNo de option - I specify option since selection may not find the node requested as an argument that is of type Position. The code is as follows...

```fsharp
member μ.GetNode (position : Position) =
  let mutable notFound = true
  let rec traverse tree =
    seq {
      match tree with
      | ControlTree(currentNode, branch) ->
        if notFound then
          if currentNode.Location = position
          then
            notFound <- false
            yield ! currentNode
          for item in branch do yield! traverse item
        ...
      ...
      match μ with
      | ControlTree(root, tree) ->
        if root.Location = position
        then seq { yield root }
        else seq { for item in tree do yield! traverse item }
        ...
      |>
        Seq.tryHead
```

I’m going to use a mutable, Boolean switch name notFound, with an initial value of true, to signify whether or not I’ve found the required target as specified...
by the position argument - this will “limit” the number of yield expressions evaluated as part of a sequence comprehension.

The recursive function named traverse takes a ControlTree instance as its argument, named tree. Recursion occurs within a sequence comprehension. Essentially, within the comprehension...

- Use a pattern match to decompose the ControlTree instance into two value types named currentNode and branch. With these value types...
  - If the target is not yet found (so notFound = true)...
    - If the Location of the current node equals the position specified argument (note, again, this will use our Position IComparable interface member CompareTo but, in this case, it is “acceptable” to just perform the compare on the unique Index and to disregard the Level) then...
      - Set notFound to false - we’ve located our target node.
      - Yield the currentNode as the result of the find.
    - We need to re-invoke our recursive traverse function for each child element of the branch that is parsed out of the ControlTree pattern match against the function argument, tree.

Finally, we need to “kick-off” our recursive search for the target node with Location of position - the member argument. To do so, we “decompose” the union case into a root ControlNode and its List<ControlTree> as the value of tree. We first check if the root ControlNode is actually the target and, if so, we just yield a sequence of the root node. Otherwise, we produce a sequence that invokes traverse for each ControlTree in tree. In either case, since we only “want” one node, we’ll use a Seq.TryHead to pull out the target node so it will be presented as an option type.

Getting a “branch” of the tree from a given node is very similar - rather than yielding just a ControlNode we want to yield that ControlNode and its List<ControlTree> - which, as a tuple element is a “sub-tree”, a ControlTree whose root element is the target node. The member is as follows...

```fsharp
member μ.GetBranch (position : Position) =
    let mutable notFound = true
    let rec traverse tree =
        seq { match tree with
            | ControlTree(currentNode, branch) ->
                if notFound then
                    if currentNode.Location = position then
```
notFound <- false
    yield currentNode, branch
    for item in branch do yield! traverse item

match (match µ with
       | ControlTree(root, tree) ->
         if root.Location = position
            then seq { yield root, tree }
            else seq { for item in tree do yield! }$
       | traverse item }
       |> Seq.tryHead )

We again use a mutable notFound Boolean switch and a recursive traverse function; the difference is that the “intermediate” sequence yields the tuple of currentNode, branch. Now, it may appear odd that we’re using a sequence when the control tree tuple’s second element is a List<ControlTree> but F# takes care of the “switch” between sequence and list for us “transparently”. In addition, we don’t have to recurse the entire tree - as soon as we’ve found the target node, as specified in the member argument, the branch of the tree “hanging off” that node is entirely “contained” within the tuples’ second element.

Once again, we “kick off” traversal with a decomposition pattern match against the root of the tree and take a Seq.tryHead from the traverse output but now, additionally, we’ll coerce the member output as a ControlTree option.

A Nodes Children and Descendants

The primary reason for normalising our argument for these and other members of Graph is not only to undertake a validation process but also was so that, given a control name, we can yield its Position from our Map field. Consequently, “under the covers”, selection is always via a node’s Position. Given this, there’s a piece of code I put at the start of every member that uses this arg of type Selector - as follows...

```
let nodePosition =
    match arg with
    | Location(value) -> µ.LocationValidator value
    | Name(value) ->
      match µ.NameValidator value with
```
I want to assign the value type `nodePosition` as the output of validating the argument; to do so, there is a pattern match to first establish whether the argument is the `Selector` case `Location` or `Name` and, for each case, the validator is invoked for the arguments’ case value. If the argument is of the case `Name` we then use a Linq `Single` function against our `Map` (which is an F# map of `<Position,FramworkElement>`). Linq `Single` would raise an exception if the element sought via the lambda expression of its argument cannot be found – however, we have already verified the `Name` that is being used as the argument as valid, so the Linq `Single` should always work without error. We use the `Key` property against the output of the Linq `Single` to get the named node’s `Position`. Because the `arg` argument may be invalid then `nodePosition` has the output type of `Position option` so we would subsequently have to check that its value is not `None`, but `Some location`.

This validation code needs to be invoked for each member of `Graph` that uses the `arg` argument so, what we should be doing is declaring a function that performs this task rather than repeating the code in each member. We’ll do this by adding the private member as follows…

```fsharp
member private µ.ValidateArgument = function

| Location(value) -> µ.LocationValidator value
| Name(value) ->
  match µ.NameValidator value with
  | Some(name) ->
    Some (<| µ.Map.Single (fun de -> de.Value.Name = name).Key
  | None -> None

Given that, the code for the `Children` member is as follows…

```
Refining our Control Tree

match target with
| ControlTree(_,tree) ->
  let strip branch = seq {
    match branch with
    | ControlTree(node,_) -> yield node
  seq { for item in tree do yield! strip item }
    |> Seq.map (fun node -> µ.Map.[node.Location])
...

Should the argument fail validation we just output an empty sequence, otherwise, the node’s location being the value of position...

- Perform a pattern match against the output of a GetBranch for the tree using the position as the starting point...
  - If no branch is yielded then output an empty sequence
  - If Some ControlTree is returned in the value of target then...
    - Do a pattern match decomposition of the ControlTree to yield the List<ControlTree> tuple element as the value of tree, for which...
      - We declare a function called strip that takes a List<ControlTree> named branch as an argument. The strip function does another pattern match decomposition of the tuple to just yield the first element - a ControlNode and yield that to a sequence. This doesn't need to be recursive, as we only want the children rather than the descendants of the target.
      - Declare a sequence via comprehension such that for every ControlTree in the tree list, we invoke the strip function. We use yield! to flatten the sequence into a sequence of ControlNode elements rather than a sequence of sequences of ControlNode.
    - The output of the pattern match decomposition of target, that is a sequence of ControlNode elements, is piped into a Seq.map such that, for each ControlNode element, we use our Map to yield the FrameworkElement value for that ControlNode Location.

For the Descendants member it’s a very similar process - but we do need to recursively invoke the strip function to pick up all of the children of the “first” children. The code is as follows...

```csharp
member µ.Descendants arg =
  match µ.ValidateArgument arg with
  | None -> Seq.empty
```
| Some(position) ->  
  match μ.Tree.GetBranch position with  
  | None -> Seq.empty  
  | Some(target) ->  
    match target with  
    | ControlTree(_,tree) ->  
      let rec strip branch =  
        seq { match branch with  
          | ControlTree(node,tree) ->  
            yield node  
            for item in tree do yield! strip item  
        }  
        ... item }  
      seq { for item in tree do yield! strip item }  
    |> Seq.map (fun node ->  
          ... μ.Map.[node.Location])

The only differences are...

1. We’ve used the rec keyword to make strip a recursive function.
2. In using a pattern match decomposition of branch we also “pull out” the List<ControlTree> associated with the child node in the value type named tree.
3. We still yield the node but we then add an expression to recursively invoke strip for each ControlTree in the list tree. Again, we use a yield! to “flatten” the resultant sequence.

A Nodes Parent and Ancestors

I’m going to use a “different” methodology from that of using GetBranch as for the Descendants but first, as it’s trivial, let’s deal with the Parent member. Recall, that in the Populate member I’ve now added the parent location to update our new ControlNode field ParentLocation of type Position option - it’s an option since the root won’t have a parent. The code for the Parent member is as follows...

```
member µ.Parent arg =  
  match µ.ValidateArgument arg with  
  | None -> None  
  | Some(position) ->  
    match µ.Tree.GetNode position with  
    | None -> None  
    | Some(target) ->  
      match target.ParentLocation with  
      | None -> None
```
Refining our Control Tree

After argument validation, if we have Some(position) we just use that position to invoke GetNode - that will output a ControlNode option. If the node exists, of Some(target), then we'll use our Map to yield an option of the FrameworkElement value based upon the instances ParentLocation.

Now, with regard to Ancestors, I'm going to use our Map key values with the <^ operator that we defined for the Position type. This operator is "strict" insofar as it will exclude siblings in its usage, whereas using < would include siblings - and siblings cannot be regarded as ancestors! The code is as follows...

Selecting a Sequence of Typed Controls

This member exists since it may be useful in assigning event handlers to collections of particularly typed controls - such as buttons. The easiest way to both "filter" the controls by type and coerce the output typing of the member is via the specification of a generic type argument. It may also be the case that one would only want a collection of the controls on "one branch of the tree" rather than base the selection as all descendants of the root element - consequently, we'll allow an optional argument that will also use the Selector type. This code is as follows...
Since we've made the argument optional - as in `?arg`, we can't immediately invoke our `ValidateArgument` member - we must first determine whether the argument was specified and, if not, or if validation fails, then we'll use `Position.Root`.

Given our "starting point", we'll then use a Linq `Where` filter and choose all `Map` key/value pairs where the `Index` (this is not a "strict" compare as in `>^` since we want to include siblings) is greater than or equal to the specified `position` argument and the `FrameworkElement` is of the same type as that specified in the generic type argument. The filtered collection is then passed through a Linq `Select` to just yield the framework element value and then, for the member output, the result is coerced into a sequence of the type specified in the generic type argument.

Picking a Single Node

This is the final member to consider and, the simplest. The code is as follows...

```fsharp
member µ.PickOne<'TControl> arg =
    match µ.ValidateArgument arg with
    | None -> None
    | Some(position) ->
        tryUnbox<'TControl> µ.Map.[position]
```

The member takes a generic type argument for coercion of the output type as for the `Controls.OfType` member. For this member the argument, `arg`, is required - it is again of type `Selector`. Given successful validation of the argument, we then select the `FrameworkElement` value from `Map` where the key matches the arguments' position. The output is type as a `'TControl option` to cover the possibility that the argument fails verification.

Testing Node Selection

In `Enumeration.3.fsx` I have entered a number of examples of usage so you may see how to invoke members; they are...
let n1 = ctlGraph.Tree.GetNode <| Position(23u,8u)
let n2 = ctlGraph.Tree.GetNode <| Position(43u,8u)
let n3 = ctlGraph.Tree.GetNode <| Root
let subtree1 = ctlGraph.Tree.GetBranch <| Position(23u,8u)
let subtree2 = ctlGraph.Tree.GetBranch <| Position(13u,9u)
let subtree3 = ctlGraph.Tree.GetBranch <| Position(9u,7u)
let str1 = subtree1.Value.Depict
let str2 = subtree2.Value.Depict
let str3 = subtree3.Value.Depict
let d1 = ctlGraph.Descendants <| Selector.Location(Position(13u,9u)) |> List.ofSeq
let d2 = ctlGraph.Descendants <| Selector.Location(Position(9u,7u)) |> List.ofSeq
let c2 = ctlGraph.Children <| Selector.Location(Position(9u,7u)) |> List.ofSeq
let cts1 = ctlGraph.ControlsOfType<Controls.Button>()
let cts2 = ctlGraph.ControlsOfType<Shapes.Rectangle>()
let cts3 = ctlGraph.ControlsOfType<Controls.Grid>()
let cts4 = ctlGraph.ControlsOfType<Window>()
let pnt1 = ctlGraph.Parent <| Selector.Location(Position(13u,9u))
let d3 = ctlGraph.Descendants <| Selector.Name("Details") |> List.ofSeq
let c3 = ctlGraph.Children <| Selector.Name("Details") |> List.ofSeq
let pnt2 = ctlGraph.Parent <| Selector.Name("Details")
let a1 = ctlGraph.Ancestors <| Selector.Name("Details") |> List.ofSeq
let a2 = ctlGraph.Ancestors <| Selector.Location(Position(13u,9u)) |> List.ofSeq
let po1 = ctlGraph.PickOne<Controls.TextBlock>
                  (Selector.Location(Position(14u,10u)))
let po2 = ctlGraph.PickOne<Controls.TextBox>(Selector.Name("TextBox1"))
For a number of these I’ve piped sequence output into a list so results can be seen immediately in the FSI output window. It’s “easier” to view this output in FsEye, as below, but bear in mind if you try to expand a FrameworkElement FsEye will have trouble enumerating all its members.

Figure 105: Swenson’s FsEye showing a number of our WPF Graph type’s sample outputs.