Meditation on Adding INotifyPropertyChanged to RoslynDom

I was faced with the dual realities of making a broad change to RoslynDom and the fact RoslyDom is supposed to empower exactly this kind of change.

So, obviously, it was time to go off the high dive.

The immediate change: implement a change mechanism, probably INotifyPropertyChanged, but can’t this be flexible?

I’m going to cut straight to the punch line in this post, and explain the process in another. RoslynDom needed several changes along the way, so if you tried to do this, it wasn’t you, really, it was me.

For this problem, the major difference between this approach and an approach using the .NET Compiler Platform, Roslyn trees is that the Compiler Platform trees are immutable (for very good reason) and RoslynDom is mutable (for very good reason). You can load the RoslynDom trees, alter them to your heart’s content, then write them back out as new syntax trees or files. So, the difference is that you just make the changes with RoslynDom.

# Create the test to find the files and make the change

By habit, I do these kinds of changes in tests. It means I have more ongoing stuff happening during testing, and when it annoys me, I’ll either disable the test, or disable writing the files.

If you think I’m going to put the altered changes on top of the originals, *are you crazy?* I’m actually not insane. I am won’t overwrite the RoslynDom source code in place because it is not a good idea for code generators to overwrite themselves, unless you really enjoy reverting your source code.

I’ll start by defining some directories. Since I’m basically a chicken, I designed the reusable part of the system assuming you aren’t overwriting you’re original code. The output directory will contain a copy of the subdirectories.

private string inputDirectory = @"..\..\..\RoslynDom";

private string outputDirectory = @"..\..\Walkthrough2\_Updated\_Files";

private string[] subDirectories = new string[] { @"BasesAndBaseHelpers", @"Implementations", @"StatementImplementations" };

The test itself is very simple, just calling the reusable part of the system:

[TestMethod]

public void Walkthroughs\_2\_update\_notify\_property\_changed()

{

DoUpdate(GetFilePairs("\*.cs",

inputDirectory, outputDirectory, subDirectories),

GetRDomClasses, AddINotifyPropertyChanged);

}

The DoUpdate method is part of the sample and takes a set of file pairs – just input and output file names – and two delegates or lambdas. The first delegate gets the items within the file to change, and the second makes the changes. The DoUpdate method is part of the sample because there are many ways to get to an input file name, an output file name, select and change delegates. The actual work is done in the factory by the passed delegates:

private bool DoUpdate<TChange>(

IEnumerable<Tuple<string, string>> filePairs,

Func<IRoot, IEnumerable<TChange>> getItemsToChange,

Func<TChange, bool> changeItems)

where TChange : IDom

{

var didAnythingChange = false;

foreach (var pair in filePairs)

{

if (RDomCSharp.Factory.DoUpdateOnFile(pair.Item1, pair.Item2, getItemsToChange, changeItems))

{ didAnythingChange = true; }

}

return didAnythingChange;

}

# Find the items in the file

Each file may have one, many or no items that require change. In this case, I want to change the classes that have some particular base classes. That’s the first delegate passed to DoUpdate or DoUpdateOnFile:

private static IEnumerable<IClass> GetRDomClasses(IRoot root)

{

return root.RootClasses

.Where(x => x.BaseType != null

&& ((x.BaseType.Name == "RDomBaseType"

&& x.BaseType.TypeArguments.Count() == 1)

|| (x.BaseType.Name.StartsWith("RDomBase")

&& x.BaseType.TypeArguments.Count() == 2)));

}

Your code will have a different mechanism to select items to change – I’m looking at the base type and number of generic type arguments. You might be checking an attribute or another approach.

If no items are found, no action is performed.

# Make the change

Within each class I find the appropriate properties, which are public auto-properties. I thencycle through them making changes:

private bool AddINotifyPropertyChanged(IClass cl)

{

var notifyingProps = cl.Properties

.Where(x => x.CanSet && x.CanGet

&& !x.SetAccessor.Statements.Any()

&& !x.GetAccessor.Statements.Any()

&& x.AccessModifier == AccessModifier.Public);

foreach (var prop in notifyingProps)

{

UpdateProperty(prop);

}

return true;

}

The update property method creates a backing field, adds it to the parent, and updates the getter and setter:

private void UpdateProperty(IProperty prop)

{

// Add the field without further checks because the programmer will find and resolve

// things like naming collisions

var parent = prop.Parent as ITypeMemberContainer;

var fieldName = StringUtilities.CamelCase(prop.Name);

var field = new RDomField(fieldName, prop.ReturnType, declaredAccessModifier: AccessModifier.Private);

FixWhitespace(field, prop);

field.Whitespace2Set.Add(new Whitespace2(prop.Whitespace2Set.First().Copy()));

parent.MembersAll.InsertOrMoveBefore(prop, field);

UpdatePropertyGet(prop, fieldName);

UpdatePropertySet(prop, fieldName);

}

Updating the property getter just requires creating a return statement for the backing field:

private void UpdatePropertyGet(IProperty prop, string fieldName)

{

var retExpression = RDomCSharp.Factory.ParseExpression(fieldName);

var statement = new RDomReturnStatement(retExpression, true);

prop.GetAccessor.StatementsAll.AddOrMove(statement);

prop.GetAccessor.EnsureNewLineAfter();

}

Updating the property setter just requires creating an invocation statement for the backing field. For a number or reasons, RoslynDom currently creates expressions by parsing the corresponding string (RoslynDom always relies on the great parsing in the .NET Compile Platform, Roslyn under the hood):

private void UpdatePropertySet(IProperty prop, string fieldName)

{

var expression = RDomCSharp.Factory.ParseExpression(string.Format("SetProperty(ref {0}, value)", fieldName));

var statement = new RDomInvocationStatement(expression, true);

prop.SetAccessor.StatementsAll.AddOrMove(statement);

prop.GetAccessor.EnsureNewLineAfter();

}

That’s it. That’s all there is to updating files to use INotifyPropertyChanged from the RoslynDom perspective. From a completeness perspective, I’ll also walk through how GetFilePairs finds files in the sample, changes to the RoslynDom base class to allow this very simple NotifyPropertyChanged implementation in the Setter, and how the factory handles changes and writing out of files, in case you want to do this by hand.

# Find the files

This sample collects a set of input and output file pairs from the passed directories using the GetFilePairs method. It doesn’t have much to do with RoslynDom, but is included for completeness:

private IEnumerable<Tuple<string, string>> GetFilePairs(string pattern,

string inputDirectory, string outputDirectory, params string[] subDirectories)

{

IEnumerable<Tuple<string, string>> filePairs = new List<Tuple<string, string>>();

foreach (var subDir in subDirectories)

{

filePairs = filePairs.Union(

Directory

.GetFiles(Path.Combine(inputDirectory, subDir), pattern)

.Select(x => Tuple.Create(x, Path.Combine(outputDirectory, subDir, Path.GetFileName(x)))));

}

return filePairs;

}

The resulting file pair is the path from the current directory to each input and output file name.

# Alter the base class

One of the big enhancements I’m working on for RoslynDom is for it to understand what changes have been made to it. This is very important it accessing functionality in the underlying syntax and symbolic trees.

I really don’t want to commit to INotifyPropertyChanged. It works, but it sometimes feels so creaky to use events for the simplest notification. So, I decided to isolate notification into the base class. Honestly, this isn’t enough and I’ll be back to work here. This approach doesn’t offer you a simple way to insert a different notification. But I want to postpone that problem, and just implement INotifyPropertyChanged in an isolated manner in the base class. Here it is:

public abstract class RDomBase : IDom, INotifyPropertyChanged

{

// Lots of other stuff

protected void SetProperty<T>(ref T field, T value, [CallerMemberName] string name = "")

{

if (!EqualityComparer<T>.Default.Equals(field, value))

{

field = value;

// TODO: Update to null conditionals in C# 6

var handler = PropertyChanged;

if (handler != null)

{

handler(this, new PropertyChangedEventArgs(name));

}

}

}

}

By putting this in the base class, the derived classes need very little change. This code replaces an auto-property for all editable properties:

private string \_name;

public string Name

{

get { return \_name; }

set { SetProperty(ref \_name, value); }

}

# Step 3: Alter the code

The first step is to identify the properties that are candidates for change:

private bool NotifyPropertyChange(IRoot root)

{

IEnumerable<IClass> classes = GetClasses(root);

if (!classes.Any()) return false;

foreach (var cl in classes)

{ AddINotifyPropertyChanged(cl); }

return true;

}

The GetClasses method, predicatably, filters the

Updating each one is very easy with RoslynDom:

private void UpdateProperty(IProperty prop)

{

// Add the field without further checks because the programmer will find and resolve

// things like naming collisions

var parent = prop.Parent as ITypeMemberContainer;

var fieldName = StringUtilities.CamelCase(prop.Name);

var field = new RDomField(fieldName, prop.ReturnType, declaredAccessModifier: AccessModifier.Private);

FixWhitespace(field, prop);

field.Whitespace2Set.Add(new Whitespace2(prop.Whitespace2Set.First().Copy()));

parent.MembersAll.InsertOrMoveBefore(prop, field);

UpdatePropertyGet(prop, fieldName);

UpdatePropertySet(prop, fieldName);

}

For each property, I create a field for the backing field, fix the whitespace, and alter the get and set properties. RoslynDom still needs another generation of whitespace management – you should not need to do any whitespace changes for a change this small – but currently you do:

private void FixWhitespace(RDomField field, IProperty prop)

{

// TODO: This is rather detailed because of featuresnot yet in the whitespace system

var leading = prop.Whitespace2Set[LanguageElement.Public].LeadingWhitespace;

field.Whitespace2Set[LanguageElement.Private] = new Whitespace2(LanguageElement.Private, leading, " ", "");

}

So far, nothing is too hard, but what about changing the code? Is that hard?

The property get needs to create a return statement, which needs an expression. RoslynDom currently uses The .NET Compiler Platform, Roslyn for expressions, because it’s expression parsing is so powerful:

private void UpdatePropertyGet(IProperty prop, string fieldName)

{

var retExpression = RDomCSharp.Factory.ParseExpression(fieldName);

var statement = new RDomReturnStatement(retExpression, true);

prop.GetAccessor.StatementsAll.AddOrMove(statement);

prop.GetAccessor.EnsureNewLineAfter();

}

The property set requires invoking the base class method. Again, created from a string representation of the expression.

private void UpdatePropertySet(IProperty prop, string fieldName)

{

var expression = RDomCSharp.Factory.ParseExpression(string.Format("SetProperty(ref {0}, value)", fieldName));

var statement = new RDomInvocationStatement(expression, true);

prop.SetAccessor.StatementsAll.AddOrMove(statement);

prop.GetAccessor.EnsureNewLineAfter();

}

That’s it. The entire code to implement INotifyPropertyChanged. I will continue to work on making it easier, particularly with whitespace, but that’s not too bad.

# But it’s ugly!

Right, at the moment, when you load the code you should reformat with Ctl-K, D. Just wanted you to remember me via my initials.

# But this didn’t even come close to working in the previous version!

That’s true. All of the RoslynDom classes needed updating to add a constructor that took the editable fields.

# Manage loops