Walkthrough: 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 and create a general purpose mechanism for making changes with RoslynDom. An especially nice thing about this change is that it’s one I’m willing to (or someone else will) work out for Roslyn, allowing a comparison approach. Some of what RolsynDom can do (like code generation from scratch) is crazy hard with Roslyn.

The source code is in the Walkthrough\_2\_CodeChanges.cs file. I want to just highlight the fun parts here.

public void Update\_notify\_property\_changed()

{

var filePairs = UpdateUtilities.GetFilePairs("\*.cs", inputDirectory, outputDirectory + "\_A", subDirectories);

UpdateUtilities.DoUpdateOnFiles(filePairs,

root => root.Descendants

.OfType<IProperty>()

.Where(x => IsInRDomClass(x)),

new UpdateNotifyPropertyChanged());

}

The first step is to get a list of filename pairs (as Tuple<string, string>), which I don’t think is all that interesting.

I pass this, a lambda selection expression, and an implementation of an interface to the DoUpdateOnFiles method. This method, predictably, loops through files and makes changes.

If my implementation of UpdateNotifyPropertyChanged suits you, all you have to do is provide input and output and sub directories, and any filtering of the properties to change.

# Behind the curtain

Understanding the code behind this will help you decide whether this change fits your needs, modify it if it doesn’t, and let you better understand RoslynDom in general.

The interface is:

public interface IUpdateRefactoring<T>

where T : IDom

{

bool MakeChange(T prop);

bool NeedsChange(T prop);

}

There are two selection mechanism, because I anticipate that you will have selection criteria (my selection IsInRDomClass) and the implementation has a common selection criteria. For the UpdateNotifyPropertyChanged class, the NeedsChange method is:

public bool NeedsChange(IProperty prop)

{

if (prop.DeclaredAccessModifier != AccessModifier.Public) return false;

if (!(prop.CanGet && prop.CanSet)) return false;

if (prop.GetAccessor.Statements.Count() != 1) return true;

if (prop.SetAccessor.Statements.Count() != 1) return true;

var returnStatement = prop.GetAccessor.Statements.First() as IReturnStatement;

if (returnStatement == null) return true;

var fieldName = returnStatement.Return.Expression;

var setStatement = prop.SetAccessor.Statements.First() as IInvocationStatement;

var expected = string.Format("SetProperty(ref {0}, value)", fieldName);

if (setStatement.Invocation.Expression.NormalizeWhitespace() != expected.NormalizeWhitespace()) return true;

return false;

}

The first two tests determine if the change is appropriate and the rest of the test determine if the change has already been made. This is a first stab at this, I plan to spend more problem on the question of whether existing code matches an anticipated change.

# Making the change

The change involves adding the field, and modifying the getter and setter:

public bool MakeChange(IProperty prop)

{

string fieldName = AddFieldForProperty(prop);

UpdatePropertyGet(prop, fieldName);

UpdatePropertySet(prop, fieldName);

return true;

}

Adding the field involves finding the parent, creating the field with the appropriate name, doing some whitespace fix-up, and inserting the field into the parent. I’m adding it just before the property.

private static string AddFieldForProperty(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 = (prop.Name.StartsWith("\_") ? "" : "\_") + 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);

return fieldName;

}

NOTE: I anticipate two changes to this pattern as RoslynDom evolves – you should not need to be concerned about whitespace here, and the mechanism for determining a standard name should be held in a context.

The getter update just creates the return expression and add it to the property accessor:

private static 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();

}

Similarly, updating the setter just requires creating a call to the base class method, which holds the implementation details.

# Other changes

I did a more complex change on constructors in RoslynDom. Initially the only constructors were those used when loading from SyntaxNodes. This isn’t appropriate for creating new items from scratch. I left the code in place in the Walkthrough\_2\_CodeChanges file.

I’m not writing this up, because it was not completely successful in this form, and I made a number of manual changes, rather than spending more time on a one-off rather unusual change.