HoGent

Faculteit Bedrijf en Organisatie

Ensuring code quality through diagnostic analyzers on the Roslyn platform

Jeroen Vannevel

Thesis submitted in partial fulfillment of the requirements for the degree of bachelor applied computer science

Promotor:
Marc Van Asselberg
Co-promotor:
Schabse Laks

Instelling:

Academiejaar: 2015-2016

1e examenperiode

Faculteit Bedrijf en Organisatie

Ensuring code quality through diagnostic analyzers on the Roslyn platform

Jeroen Vannevel

Thesis submitted in partial fulfillment of the requirements for the degree of bachelor applied computer science

Promotor:
Marc Van Asselberg
Co-promotor:
Schabse Laks

Instelling:

Academiejaar: 2015-2016

1e examenperiode

Abstract

Foreword

Contents

Chapter 1

Introduction

1.1 What is Roslyn?

Roslyn is a city in Washington, a former coal mining town and the filming location for multiple movies. It is also said that it was in this place, drinking one of Roslyn's beers, that a project manager on the Managed Languages team over at Microsoft came up with the name of their new compiler: Roslyn.(Wischik 2015) A compiler is a program that translates code from one language to another. In this case, it translates C# code to Microsoft Intermediate Language (MSIL) – Microsoft Intermediate Language.

1.2 What can Roslyn do?

There are three major aspects to source code that can be analyzed: the syntax, the semantic model and the data flow. The syntax, which is really just a collection of tokens that make up our source code, is the very basic level: no interpretation is done, we just make sure that the code that is written is valid syntactically. These validations are performed by confirming that the source code adheres to the rules specified in the C# Language Specification[1]. Note, however, that there is no analyzing done of the actual meaning of these symbols thus far – we only determined that the textual representation is correct. The next level facilitates this aspect: it interprets the semantic model. This means that so-called 'symbols' are created for each syntactical construct and represent a meaning. For example the following code will be interpreted as a class declaration (through the type `ClassDeclarationSyntax`).

```
1  public class MyClass { }
```
Listing 1.1: Interpretation of a class declaration

---

[1]https://msdn.microsoft.com/en-us/library/ms228593.aspx

While in this code, the `MyClass` will be interpreted as an `IdentifierNameSyntax`:

```
1 var obj = new MyClass();
```
<div align="center">Listing 1.2: Interpretation of an object instantiations</div>

This shows us that the semantic model knows the meaning of each syntactic construct regarding its context. This allows for more extensive analysis since we can now also analyze the relationship between specific symbols. Last but not least: the third aspect of source code that we can analyze is its data flow in a specific region. The 'region' here refers to either a method body or a field initializer. This tells us whether or a variable is read from/written to inside or outside the specified region, what the local variables in this region are, etc. This is a step above "simple" semantic analysis: rather than interpreting the meaning of a symbol in the code, we can now analyze how certain symbols are interacted with without actually executing the code itself. For example consider the following method:

```
1     public void MyMethod()
2     {
3         int x = 5;
4         string y;
5     }
```
<div align="center">Listing 1.3: A method with definite assignments</div>

Analyzing this method's data flow and looking at the `DataFlowAnalysis.AlwaysAssigned` property, would tell us that only 1 local here would be always assigned. However if we now alter the above code to this:

```
1     public void MyMethod()
2     {
3         int x = 5;
4         string y;
5         if(true)
6             y = "My string";
7     }
```
<div align="center">Listing 1.4: A method with conditional assignments</div>

We now receive "2" as a result. This is an extremely powerful (albeit currently fairly limited) feature that allows us to actually interpret a code snippet's data flow without executing it.

1.3   Why was Roslyn built?

Now that we know what Roslyn does, it is important to realize what gap is being closed. After all, something has to be special about Roslyn if it is to receive such widespread attention.

Roslyn introduced two very big changes to the community: the C# compiler's language changed and it has now become open-source. Prior to Roslyn, the C# compiler was like a black box: everybody had access to the language specification so you could know what the rules of language were but the actual process of verifying your code against these rules was out of your reach. By open-sourcing Roslyn and placing it on CodePlex[2] and afterwards Github[3], this box was suddenly opened and everybody could look at the internals of how a high-end compiler is written.

This brings us to the other big change: because Roslyn is written in C# rather than C++, the C# compiler is now able to bootstrap itself. This means that Roslyn can compile itself in C# to compile the next versions of the Roslyn compiler! The process is simple: Roslyn is compiled using the old C++ compiler after which that newly created compiler re-compiles the Roslyn code base and suddenly you have a C# compiler that is written in C#. An added benefit of rewriting the compiler is the fact you can "discard" the older one.

With the change from C++ to C# the compiler switched from an unmanaged language to a managed one. In essence this means that in a managed language like C#, a special service called the "Garbage Collector" (GC) will take care of allocating and freeing memory. Without going too deep into the GC's working and purpose, the Roslyn team has found that this switch allowed them to focus more on improving the algorithms and less on taking care of memory issues. This resulted in a keystroke response time of under 80ms with the 98th percentile even lowering as far as 40ms − a "significant improvement over the C++ version".(Wischik 2015)

It should be noted here that Roslyn is also VB.NET's new compiler and as such, VB.NET-specific parts are written in VB.NET. You will also notice that every unit test that applies to both C# and VB.NET is duplicated in both languages. In this paper, however, we will focus on the C# aspects.

---

[2]https://roslyn.codeplex.com/
[3]https://github.com/dotnet/roslyn/

1.4 Positivity of an open compiler

Opening the compiler to the public brought a lot of good things, not in the least external contributions! Only 6 days after the Roslyn source was released, the first PR (Pull Request) was already integrated in the system!

Figure 1.1: Announcement of the first PR by Kevin Pilch-Bisson[4]

There are many more benefits to open-sourcing:

- The amount of scrutiny the code goes through is greater now that the entire developer community has access to it.

- Developers themselves can now more easily learn how compilers work in a language they're familiar with. Writing compilers has become more and more a "far-from-my-bed-show" with the rise of high-level languages – this might (re-)introduce some interest into such low-level concepts.

- It also facilitates advanced users by allowing them to inspect the bare bones of the compiler when something unexpected happens. Compilers are not bug free[5][6] and implementations change over time[7] which may lead to small but distinct differences in behaviour. An example that comes to mind is laid out in this stackoverflow question titled "Why does Roslyn crash when trying to rewrite this lambda?"[8] where a user noticed that identical code compiles differently on the new compiler. Merely hours later, another user had identified the issue, discovered the offending commit in the Roslyn repository and submitted a Pull Request (PR) with a fix.

- Last but not least it simply provides a very robust and efficient parser that external parties now won't have to develop themselves. This doesn't necessarily mean that those tools should switch to the Roslyn platform but they might

---

[4]https://twitter.com/pilchie/status/453968834408763392
[5]http://stackoverflow.com/a/28820861/1864167
[6]http://stackoverflow.com/q/33694904/1864167
[7]http://stackoverflow.com/a/30991931/1864167
[8]http://stackoverflow.com/a/34085687/1864167

incorporate some ideas of it in their own parser (as is the case with ReSharper, a popular code-refactoring tool).(Gorohovsky 2014)

1.5   Positivity of compiler APIs

Separately, providing APIs to your compiler brings benefits with it as well. Because of the broad support of APIs for all kinds of aspects of a compilation's lifetime, a very convenient service is created where external parties can easily hook into and create their own diagnostics using the information that comes straight from the compiler. This encourages new tools aimed at the platform since the hassle of writing your own parser is removed – developers can go straight to implementing business logic.

On top of that, the "native" support means that there is a seamless integration with Visual Studio, Microsoft's own Integrated Development Environment (IDE)[9]. This integration allows you to use the results of your diagnostics to be displayed and used directly inside of Visual Studio rather than needing to build a third-party tool or a plugin.
This is the essence of why Roslyn is regarded as "Compiler as a Service".

1.6   Introduction to the paper

In this paper we will take a look at some services that are exposed through the roslyn platform: writing your own Analyzer, SyntaxWalker and SyntaxRewriter. Each of these provide a useful way of analyzing and/or modifying source code without having to bother with any of the overhead that typically comes with it. There are more of these services available like Code Fixes and SymbolVisitors or specific analysis approaches like analyzing data flow and control flow but these are not discussed here.

After the practical examples we will take a look at Roslyn's implementation and answer questions such as "What are the phases it goes through?", "What are these syntax trees and nodes exactly?" and "How do red-green trees fit into all this exactly?".

We will proceed by taking a detailed look at performance considerations inside the code base. "Performance" in the sense of memory-impact and execution time through coding techniques – not actual algorithms.

In conclusion there is also a section that highlights some security considerations in Roslyn and the general architecture of it.

---

[9]IDE: Integrated Development Environment, the editor in which you modify source files

Chapter 2

Implementing a Diagnostic

As a way to demonstrate the usefulness of Roslyn, a specific usecase will be investigated. You might be familiar with `string.Format()` but if not: it's a way of creating a formatted string by using placeholders and corresponding arguments to interpolate certain portions of it.

For example the code in listing 2.1 will display "Hello John!". As you can tell, the placeholder (which is an integer delimited by angle brackets) will correspond to the argument you pass in afterwards.

```
1 | string.Format("Hello {0}!", "John");
```
Listing 2.1: Example usage of string.Format

One scenario where this can go wrong if you have a placeholder inside your format but no corresponding argument as is demonstrated in listing 2.2. This would make substitution impossible and throw a `FormatException`[1] at runtime. This immediately highlights the danger: C# is generally a statically typed language so we are used to some degree of confidence as soon as the code compiles. A formatting string, however, is not evaluated until that formatting is actually requested through the application's flow.

```
1 | string.Format("Hello {0}! My name is {1}", "John");
```
Listing 2.2: Example usage of string.Format with a missing argument

In this section we will create an analyzer that scans our solution for occurrences where `string.Format` is used with more placeholders than it has arguments. There are some restrictions that apply: certain scenarios are not supported because it's impossible or not feasible to calculate it at compile-time. Additionally, some edge cases have been omitted but will be integrated at a later stage. The characteristics:

---

[1]https://msdn.microsoft.com/en-us/library/system.string.format(v=vs.110)
.aspx#Format_Exceptions

- The format must be a constant. This can be in the form of a string literal, a constant variable, a compile-time constant concatenation, etc

- Interpolated strings that specify the format are ignored for now

- If the arguments are passed in using an array and the `object[]` overload is used, only inline-initialized arrays are considered

- The analyzer does not work for just `string.Format` but for every method that has a 'format' parameter and an `object` or `object[]` parameter. This allows us to also support `Console.WriteLine` for example.

- Supports overloads that take a `CultureInfo` like `string.Format(CultureInfo, string, object[])`

The full code can be found in !!REFERENCE ATTACHMENT!!.

2.1   Getting started

There are a few requirements in order to get started with writing your own diagnostic. Once you have these installed you can continue by creating a new solution using the "Diagnostic with Code Fix (NuGet + VSIX)" template which will get you started with a class library for your analyzers, a testing project to write unit tests and a Vsix project to install it as an extension.

- Visual Studio 2015[2]

- The .NET Compiler Platform SDK [3]

2.2   Implementing the diagnostic

There are a few ways to implement this: you could keep it easy and find all formats, normalize them by removing format specifiers for each placeholder and replacing the arguments with example strings and then calling `string.Format` inside your analyzer and see if it throws any exception. This would certainly be a valid approach but hardly a good demonstration of what Roslyn can do. Instead, we will analyze the syntax tree, use the semantic model to get info about its nodes and parse the format ourselves.

The general layout of an analyzer is straightforward and consists of the following aspects:

---

[2]`https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx`
[3]`https://visualstudiogallery.msdn.microsoft.com/2ddb7240-5249-4c8c-969e-5d05823bcb89`

- Metadata such as supported languages, severity, category, message and title

- Registration of what should trigger the analyzer

- Implementation of the actual analyzer

The two most important ones (that the user will get into contact with) are the severity and the message. The severity determines whether it will be a warning, an error, info or something that only shows up when you look at the context-actions for a specific line of code ('hidden'). The message is the message the user sees in his 'Error List' window and is essential to tell the user what's wrong.

This metadata is exposed to the underlying architecture by means of the `SupportedDiagnostics` property as we can see in listing 2.3. Specifying which languages are supported has to be done separately through an annotation (listing 2.4).

```
 1  internal static DiagnosticDescriptor Rule =>
 2    new DiagnosticDescriptor(
 3        "MyUniqueId",
 4        @"This guards against using a placeholder
 5          without corresponding argument",
 6        "A string.Format() call has too few arguments",
 7        "My own category",
 8        DiagnosticSeverity.Error,
 9        enabledByDefault: true);
10  public override ImmutableArray<DiagnosticDescriptor>
11    SupportedDiagnostics => ImmutableArray.Create(Rule);
```

Listing 2.3: Configuring metadata for your analyzer

```
 1  [DiagnosticAnalyzer(LanguageNames.CSharp)]
 2  public class MyAnalyzer : DiagnosticAnalyzer
```

Listing 2.4: Configuring supported languages

Once this is taken care of we have to decide what should trigger our analyzer. Depending on what you, choose your analyzer can be fired during the parsing stage or something at a higher level. We will fire ours each time it comes across an invocation but there are many ways to approach it: you could also register your analyzer when it comes across a string literal or an argument list. Each of these three options require you to verify that you're dealing with a `string.Format` call so it's up to you to decide what path to take. Keep in mind though that firing an analyzer is not free of charge so you should strive for a configuration that has as little false-positive invocations as possible.

In order to setup this configuration you have to override the `Initialize` method, register the kind of action you're interested in and pass along the actual analyzer implementation followed by a `params` list of accepted firing conditions (listing 2.5).

```
1  public override void Initialize(AnalysisContext context)
2  {
3    context.RegisterSyntaxNodeAction(
4      AnalyzeNode,
5      SyntaxKind.InvocationExpression);
6  }
```

Listing 2.5: Registering your analyzer's firing condition

Once this is done the real work starts: implementing the actual analysis. As we've just defined `AnalyzeNode` as our handler, we have to define a method that corresponds to that. Depending on what level you register your analyzer you will get a different parameter representing the options you have at that point in time. Since we use `RegisterSyntaxNodeAction` we will receive a `SyntaxNodeAnalysisContext` parameter (listing 2.6) containing things like the node that triggered our inspection and the semantic model of the tree it is connected to. If instead you would register your handler on compilation start (`context.RegisterCompilationStartAction`), you would receive information about the compilation.

```
1  void AnalyzeNode(SyntaxNodeAnalysisContext context)
2  {
3  }
```

Listing 2.6: Defining the analyzing method

The first thing we do is cast our node (which we receive as a general `SyntaxNode`) to the appropriate type. Considering our analyzer is only triggered when it comes across an invocation expression, we can safely assume the node is of type `InvocationExpressionSyntax` (listing 2.7). Additionally we verify that there is a list of arguments defined to make sure we won't encounter any `NullReferenceException`

```
1  var invocation =
2    context.Node as InvocationExpressionSyntax;
3  if (invocation?.ArgumentList == null)
4  {
5    return;
6  }
```

Listing 2.7: Casting the node to the expected type

The approach taken in this implementation can be summarized like this:

- Verify there is a parameter called 'format' in the called method (listing 2.8)

- Verify there is either a single `object[]` parameter or all the remaining parameters are `object` (listing 2.8)

- Verify the format string is a compile-time constant (listing 2.9)

- Extract the formatting arguments (listing 2.10)

- In case there is only 1 argument of type array, verify it is a simple inline initialization without referring to a method or retrieving it from another variable (listing 2.11)

- Get all the placeholders from the formatting string (listing 2.12)

- Verify the highest placeholder is greater than the amount of formatting arguments passed in (listing 2.13)

If and only if all of these conditions have been met, the analyzer will report a diagnostic on the location of the formatting string. We can see all of these steps reflected in the implementation below.

```
1  var invokedMethod = context.SemanticModel
2                             .GetSymbolInfo(invocation);
3  var methodSymbol = invokedMethod.Symbol as IMethodSymbol;
4
5  var formatParam =
6   methodSymbol?.Parameters
7                 .FirstOrDefault(x => x.Name == "format");
8  if (formatParam == null)
9  {
10    return;
11 }
12 var formatIndex = formatParam.Ordinal;
13 var formatParams = methodSymbol.Parameters
14                             .Skip(formatIndex + 1)
15                             .ToArray();
16
17 var hasObjectArray =
18  formatParams.Length == 1 &&
19  formatParams.All(x =>
20   x.Type.Kind == SymbolKind.ArrayType &&
21   ((IArrayTypeSymbol)x.Type).ElementType
22                             .SpecialType ==
23                             SpecialType.System_Object);
24
25 var hasObject = formatParameters.All(x =>
26   x.Type.SpecialType == SpecialType.System_Object);
27
28 if (!(hasObject || hasObjectArray))
29 {
30    return;
31 }
```

Listing 2.8: Verifying the method definition expects a format and object arguments

```
1  var formatExpression = invocation.ArgumentList
2                                   .Arguments[formatIndex]
3                                   .Expression;
4  var format = context.SemanticModel
5                      .GetConstantValue(formatExpression);
6  if (!format.HasValue)
7  {
8    return;
9  }
```

Listing 2.9: Verifying the format is a compile-time constant

```
1  var formatArgs = invocation.ArgumentList
2                             .Arguments
3                             .Skip(formatIndex + 1)
4                             .ToArray();
5  var amountOfFormatArguments = formatArguments.Length;
```

Listing 2.10: Extracting the formatting arguments

```
 1  if (formatArguments.Length == 1)
 2  {
 3    var argumentType =
 4      context.SemanticModel
 5            .GetTypeInfo(formatArgs[0].Expression);
 6    if (argumentType.Type == null)
 7    {
 8      return;
 9    }
10
11    if (argumentType.Type.TypeKind == TypeKind.Array)
12    {
13      var methodInvocation =
14       formatArgs[0].DescendantNodes()
15                    .OfType<InvocationExpressionSyntax>()
16                    .FirstOrDefault();
17      if (methodInvocation != null)
18      {
19        return;
20      }
21
22      var inlineArray =
23       formatArgs[0].DescendantNodes()
24                    .OfType<InitializerExpressionSyntax>()
25                    .FirstOrDefault();
26      if (inlineArray != null)
27      {
28        amountOfFormatArguments = inlineArray.Expressions
29                                             .Count;
30        goto placeholderVerification;
31      }
32
33      if (hasObjectArray)
34      {
35        return;
36      }
37    }
38  }
```

Listing 2.11: Verifying single array arguments are inline initialized

As you can see in listing 2.12 there are references to a helper class that aids us in extracting the placeholders and removing any potential formatting from them. These are defined in `PlaceholderHelpers.cs` and merely consist of a few regular expressions[a] that extract those placeholders from our formatting string, keeping potential escaping characters in mind.

```
placeholderVerification:
var placeholders =
  helper.GetPlaceholders((string) formatString.Value)
        .Cast<Match>()
        .Select(x => x.Value)
        .Select(helper.GetPlaceholderIndex)
        .Select(int.Parse)
        .ToList();

if (!placeholders.Any())
{
   return;
}
```

Listing 2.12: Extracting placeholders from the format string

```
var highestPlaceholder = placeholders.Max();
if (highestPlaceholder + 1 > amountOfFormatArguments)
{
   context.ReportDiagnostic(
    Diagnostic.Create(
     Rule,
     formatExpression.GetLocation()));
}
```

Listing 2.13: Comparing the highest placeholder index with the number of arguments

---

[a]Regular expression: a sequence of characters that define a search pattern

2.3   Difficulties when implementing an analyzer

Writing a diagnostic introduces a different kind of problem domain than what you typically encounter: rather than modeling your software according to a physical real-world idea, you have absolutely no idea in what kind of environment your code will be used. All you know is that it will contain code that tends to follow the C# language specification (C#LS) – "tends" because more often than not, source code is not in a C#LS-compliant state (which in itself poses an extra complication).

This means we have a double-edged sword: on one hand we have a very detailed manual that tells us the exact rules of what is possible. On the other hand this is a document of 527 pages (C#LS v5.0) with often very niche scenarios detailed in it. It is up to the developer to decide which scenarios his analyzer supports but that implies a very strong knowledge of how the language works in itself. For this reason it should be no surprise that writing diagnostics that are truly robust is a really hard task to perform. This introduces a secondary problem: diagnostics should have a very, very low false-positive rate. The more false-positives it produces, the more it will annoy the user and subsequently the more likely it is people will turn off the diagnostic altogether.

Another important aspect to keep in mind is that the code is often in an invalid state. When you think about the expression `myVar.MyMethod()` you see the end-result which an sich is compliant with the language specification. However before that state was reached you also had `m`, `my`, `myV` and so on. This means that your analyzers have to handle situations with invalid code gracefully. Roslyn is a very smart compiler and is able to parse a syntax tree even when there are errors – often this means you have half-constructed nodes in the sense that some of its properties are `null`. For this reason you will often find a lot of `null` checks in analyzers: using a "soft" cast with as allows us to hesitantly check if some node has been constructed already. Likewise we should `null`-check every nullable property we access because there is no guarantee that, for example, `InvocationExpression.ArgumentList` has a value since the user might not yet have typed the opening braces for those arguments.

2.4 Testing an analyzer

Luckily you have useful tools at your disposal in order to increase your confidence in the analyzer's implementation. We've already mentioned before that Roslyn is not just a compiler: it's an entire service package that comes with the compiler. One of those services is the ability to create in-memory solutions (technically: 'workspaces') complete with projects, documents and, yes, syntax trees. In fact, the API is so helpful here that the essence consists of the following two statements:

```
1  var comp = project.GetCompilationAsync().Result;
2  var diagnostics =
3    comp.WithAnalyzers(ImmutableArray.Create(analyzer))
4        .GetAnalyzerDiagnosticsAsync()
5        .Result;
```

Listing 2.14: Analyzing an in-memory project

After constructing a basic setup you pass your source code under test (which is just a `string`) to the in-memory project. Afterwards you add the analyzer(s) you wanted to unleash on your code snippet, look at possible diagnostic results and you're done: you now know if your analyzer had any result. Lucky for us we don't have to implement any of this: the solution template that ships with Visual Studio 2015 includes a test project containing helper classes that provide these things for us. An example of such a unit test can be found in listing 2.15.

```
1  [TestMethod]
2  public void MyAnalyzerTest()
3  {
4      var original = @"
5  using System;
6
7  namespace ConsoleApplication1
8  {
9      class MyClass
10     {
11         void Method(string input)
12         {
13             Console.WriteLine(""{0}{1}{2}"",
14                 new object[] { ""arg"", ""arg2"" });
15         }
16     }
17 }";
18     VerifyDiagnostic(
19         original,
20         MyAnalyzer.Rule.MessageFormat.ToString());
21 }
```

Listing 2.15: Unit test verifying our analyzer works with Console.WriteLine and object array arguments

Perhaps the most useful way to find out what scenarios you're lacking is by executing your analyzer on a relatively large number of open-source solutions. In the end, the community as a whole will probably encounter more distinct usages than you on your own. Finding open-source repositories is easily done on github or any other repository service so there is an abundance of scenarios to work with. This also serves multiple purposes:

- The open-source repositories help you reinforce your confidence in a working analyzer if it doesn't crash on any of them

- Using external repositories greatly expands the number of scenarios your analyzer has been exposed to thus increasing the chance of finding a scenario that you haven't accounted for

- Through testing your analyzer you will also find mistakes in open-source repositories so it provides a way to pro-actively contribute to the community

Chapter 3

Implementing a SyntaxWalker

In the previous chapter we've explored how an analyzer can analyze any given piece of code. It is perfectly suited when you want to perform a certain action for specific kinds of nodes but sometimes you just want to do something with every single node (and possibly tokens or trivia as well). In such a scenario an analyzer is less well suited but luckily there are other helpers available – in this case the **SyntaxWalker**. A SyntaxWalker does exactly what its name indicates: it traverses ("visits") the entire syntax tree and allows you to define an action for every node that's being visited. Alternatively it also provides a large amount of methods that can be overridden which allow you to handle certain visits differently on a node-type basis.

As a way of demonstrating this usage, we will implement a walker that will visit all methods, properties and classes present in a certain tree. Upon visiting, it will display these members with a few of their characteristics which will result in something that resembles a textual version of a Unified Modeling Language (UML) diagram: access modifiers are replaced with +, # and − to represent `public`, `internal` and `protected`, and `private` respectively. It will also display the methods in the format `name(argumentType, argumentType) : returnType`.

One particular use case SyntaxWalkers can be very useful for is when you want to extract metadata information from your code base: take for example a UML representation or as a way of generating documentation for your APIs.

Getting started with your own walker is very straightforward: all you have to do is extend the `CSharpSyntaxWalker` class (listing 3.1) and walking through it starting from a particular position – typically the root (listing 3.2).

```
1  public class MySyntaxWalker : CSharpSyntaxWalker
2  {
3  }
```

Listing 3.1: Getting started with your own SyntaxWalker

```
1  const string source = @"
2    public class MyOuterClass
3    {
4        public void FirstMethod(int x, int y) { }
5        private void SecondMethod(string s) { }
6
7        public int Property1 { get; set }
8        internal StringBuilder Property2 { get; set }
9
10       private class MyInnerClass
11       {
12           protected int InnerMethod(Func<int, int> f) {
                 return 42; }
13       }
14   }
15 ";
16 var tree = CSharpSyntaxTree.ParseText(source);
17 new MySyntaxWalker().Visit(tree.GetRoot());
```

Listing 3.2: Walking through the syntax tree

We're using a few helper methods that will extract the access modifier for us (listing 3.3) and that handle the writing to the console (listing 3.4). If no access modifier was found, we'll return a default of –. This is not entirely how C# would interpret it but it allows us to retain the focus on the SyntaxWalker's core concept.

Notice how in listing 3.4 we pass in an `Action` as third parameter. In order to walk through the tree you have to call the base implementation of your visitor – otherwise it will stop there and no further nodes will be visited. Passing in an `Action` allows us to defer this step to our helper since it basically acts like a function pointer: the call is not being executed just by passing it along but it will be when you execute the `Action` itself.

```
 1  private string GetAccessLevel(SyntaxTokenList modifiers)
 2  {
 3      var levels = new[] { SyntaxKind.PublicKeyword,
            SyntaxKind.ProtectedKeyword, SyntaxKind.
            InternalKeyword, SyntaxKind.PrivateKeyword };
 4      var foundModifier = modifiers.FirstOrDefault(modifier =>
            levels.Any(level => modifier.Kind() == level));
 5      return foundModifier.Kind() == SyntaxKind.PublicKeyword
 6      ? "+"
 7      : foundModifier.Kind() == SyntaxKind.InternalKeyword ||
            foundModifier.Kind() == SyntaxKind.ProtectedKeyword
 8        ? "#"
 9        : "-";
10  }
```

Listing 3.3: Extracting the access modifier

```
 1  private void Write(string text, string accessLevel, Action
       visitBase)
 2  {
 3      _indentLevel++;
 4      var indentation = new string(' ', _indentLevel *
          _spacesPerLevel);
 5      Console.WriteLine($"{indentation}{accessLevel} {text}");
 6      visitBase();
 7    _indentLevel--;
 8  }
```
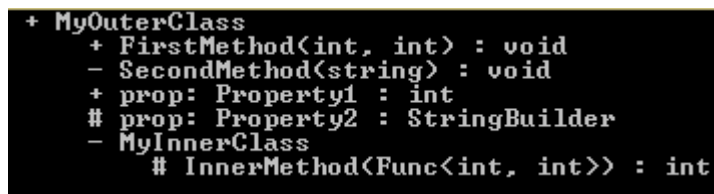
Listing 3.4: Writing text to console

In order to write out our class declarations we have to provide an implementation for the hook that handles class declarations. Each hook receives one argument: the node that is being visited – this allows us to further inspect that section of the syntax tree and get more information from underlying nodes. In listing 3.5 you can see how classes, methods and properties are being handled. Note that there is no distinction between an outer and an inner class: on a syntactic level there is absolutely no difference between the two. That immediately highlights the difference between a `SyntaxWalker` and a `SymbolWalker`: the former works at the syntax level while the latter is already at the binding phase (more on that later in section 5.1.

```
1  public override void VisitClassDeclaration(
       ClassDeclarationSyntax node)
2  {
3       Write(node.Identifier.ValueText, GetAccessLevel(node.
           Modifiers), () => base.VisitClassDeclaration(node));
4  }
5
6  public override void VisitMethodDeclaration(
       MethodDeclarationSyntax node)
7  {
8    var identifier = node.Identifier.ValueText;
9    var parameters = string.Join(", ", node.ParameterList.
         Parameters.Select(x => x.Type));
10   var returnType = node.ReturnType;
11
12   Write($"{identifier}({parameters}) : {returnType}",
         GetAccessLevel(node.Modifiers), () => base.
         VisitMethodDeclaration(node));
13  }
14
15 public override void VisitPropertyDeclaration(
       PropertyDeclarationSyntax node)
16 {
17   var identifier = node.Identifier.ValueText;
18   Write($"prop: {identifier} : {node.Type}", GetAccessLevel(
         node.Modifiers), () => base.VisitPropertyDeclaration(
         node));
19 }
```

Listing 3.5: Handling the visits

The result of our walker can be seen in figure 3.1

Figure 3.1: Displaying member layout with a SyntaxWalker

Chapter 4

Implementing a SyntaxRewriter

A **SyntaxRewriter** is very similar to the SyntaxWalker we saw in chapter 3 and provides us with a clean Application Programming Interface (API) which we can use to, as the name gave away, rewrite certain parts of the syntax tree. Here, too, we are working on the syntactic level so we can't make use of the 'smarter' semantic model and accompanying symbols. However, the scenario we will look at suffices perfectly with just syntactic knowledge so there's no problem here.

A common dispute[1] between C# programmers is whether to use an underscore as a prefix for private fields. This makes a great scenario for us to test a SyntaxRewriter: we will change all `private` fields to use an underscore as prefix. Again we will ignore the scenario where no access modifier is defined and just pretend that we don't handle those to keep focus on the rewriter itself.

Just like the SyntaxWalker, we have to start by parsing our source code as a syntax tree. Once we have that, we do the same thing as we did before: retrieve the root node and start traversing your tree from there. After all defined transformations have been applied, we will format the newly generated tree and display it (listing 4.1). Important to note: the tree we pass in and get back are *not* the same: everything is immutable in Roslyn (more on that in section 5.2.1).

---

[1] `http://stackoverflow.com/q/4540146/1864167`

```
1  const string source = @"
2    public class MyOuterClass
3    {
4        private int _myField;
5        private string anotherField;
6        private double x, _y;
7
8
9        private class MyInnerClass
10        {
11            private int _innerField;
12            public int anotherInnerField;
13        }
14    }
15  ";
16  var tree = CSharpSyntaxTree.ParseText(source);
17  var rewriter = new MyRewriter().Visit(tree.GetRoot());
18  var formattedTree = Formatter.Format(rewriter, Formatter.
       Annotation, new AdhocWorkspace());
19  Console.WriteLine(formattedTree.ToFullString());
```

Listing 4.1: Getting started with a SyntaxRewriter

When you think of a field declaration it (typically) consists of several aspects. Take the following line for example:

```
1  private int x, y = 10;
```

Listing 4.2: Example of a FieldDeclaration

This in its entirety is a `FieldDeclarationSyntax`. Looking at the next level, we can find a `VariableDeclarationSyntax`:

```
1  int x, y = 10
```

Listing 4.3: Extracted VariableDeclaration

One level further, we can finally find the `VariableDeclaratorSyntax` (listing 4.4). It's this declarator that will contain the information we're looking for: the identifier. This example also immediately shows a tricky aspect to our rewriter: multiple variables can be defined in one field declaration so this is something we have to support.

```
1  x           // First declarator
2  y = 10      // Second declarator
```

Listing 4.4: Extracted VariableDeclarators

The importance of realizing everything is immutable really shows itself in listing 4.5. Here we override the `VisitFieldDeclaration` method which is triggered every

time the rewriter comes across a field. We can also see that the method returns a `SyntaxNode` which will be the `FieldDeclarationSyntax` we construct with the new identifier.

When we look at the implementation of this method we can distinguish a few steps:

- Create a container to store declarators

- Get all identifiers from our field declaration

- Create a new identifier for each `private` identifier that does not start with an underscore

- Create a new declarator for each newly created identifier

- Create a new declaration based on the (potentially) new declarators

- Create a new field based on this new declaration and add the `Formatter` annotation to it so we can properly format the changed nodes afterwards

Immediately it's clear that we will have to create a lot of new objects to represent our changed syntax nodes. This is a result of the public Roslyn API being immutable but luckily there are a lot of useful factory methods available in `SyntaxFactory` to create those elements for us. This also represents the key consideration when manipulating a syntax tree: when a node is changed, every ancestor node will also be changed up until the highest level which can be a simple root node or possibly a `Document` or `Solution`. In our rewriter we should only replace until the level that was visited: a field declaration. Internally, the rewriter will then do the additional replacing by itself.
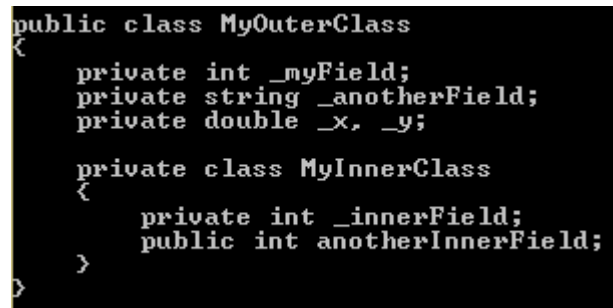
```
1  public override SyntaxNode VisitFieldDeclaration (
       FieldDeclarationSyntax node)
2  {
3    var newDeclarators =
4      new List < VariableDeclaratorSyntax >();
5
6    foreach (var declarator in node.Declaration.Variables)
7    {
8      var currentIdentifier = declarator.Identifier;
9      if (!currentIdentifier.ValueText.StartsWith("_") &&
10         node.Modifiers.Any(
11           x => x.Kind() == SyntaxKind.PrivateKeyword))
12     {
13       var newIdentifier =
14         SyntaxFactory.Identifier(
15           currentIdentifier.LeadingTrivia ,
16           "_" + currentIdentifier.ValueText ,
17           currentIdentifier.TrailingTrivia);
18
19       var newDeclarator =
20         declarator.WithIdentifier(newIdentifier);
21
22       newDeclarators.Add(newDeclarator);
23     }
24     else
25     {
26       newDeclarators.Add(declarator);
27     }
28   }
29
30   var newDeclaration =
31     SyntaxFactory.VariableDeclaration(
32       node.Declaration.Type ,
33       SyntaxFactory.SeparatedList(newDeclarators));
34
35   return
36     SyntaxFactory.FieldDeclaration(
37       node.AttributeLists ,
38       node.Modifiers ,
39       newDeclaration)
40     .WithAdditionalAnnotations(Formatter.Annotation);
41 }
```

Listing 4.5: Implementing the SyntaxRewriter

The resulting source code after our transformation can be seen in figure 4.1.

There are a lot of interesting use cases this opens up. Think for example about a build system where every user specifies their preferred naming conventions and when checking in and checking out source code, it will automatically transform it to the requested representation. This would eliminate the eternal discussion about coding styles entirely! Large refactorings are made easier by this as well: if you change from one convention or idiom to another, creating a SyntaxRewriter might take away a lot of the pain that typically comes with something like that.

```
public class MyOuterClass
{
    private int _myField;
    private string _anotherField;
    private double _x, _y;

    private class MyInnerClass
    {
        private int _innerField;
        public int anotherInnerField;
    }
}
```

Figure 4.1: Renaming fields with a SyntaxRewriter

Chapter 5

Roslyn internals

5.1 Compiler phases

A compiler is a complicated application and consists of many different services. Some of these are meant to help developers write their own tools (which is what this paper is about) while some of those services make up the internal tooling. During the compilation process we can identify 4 phases[1] where we go from the original source code to the eventual compilation into MSIL. One important remark to make here is that not everything happens from start to finish together: symbols are created on-demand and as such, some symbols might never be materialized.

5.1.1 Parsing phase

This is the first phase and receives the source code. Here, the source is tokenized into syntactical constructs according to the C# Language Specification. This process is performed by two important classes: `Lexer.cs` and `LanguageParser.cs`. Both services their function overlaps largely but the lexer works at a lower level than the parser: while the lexer can give meaning to certain symbols (not to be confused with the symbols in section 5.1.2) such as operators, the parser is able to combine multiple tokens and construct larger syntactical constructs such as expressions.(Studencki 2010)

---

[1]`https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview#exposing-the-compiler-apis`

The eventual result of the parsing phase will be a syntax tree which also means that every part of the source code is parsed in this step. This phase encompasses the concepts of lexical analysis (understanding individual words) and syntactic analysis (understanding sentences). It is in this phase that syntax errors will be caught and no further analysis on those nodes is required. The parser is able to handle invalid code through two approaches: (Ebeid 2013)

- If it can make a guess as to what token should be expected, a missing token will be inserted and have its `IsMissing` property set

- Otherwise it can also skip tokens and add them as trivia until it finds a location where parsing can continue

5.1.2 Declaration phase

Once the syntax tree has been constructed in the parsing phase, the declaration phase will add meaning to all the nodes. Imagine going from a blank syntax tree to a syntax tree that uses syntax highlighting: that happens here. Like the parsing phase, this step is executed for all tokens in the syntax tree and has as result a look-up table with the created symbols[2]. In the next phase, these constructed symbols can be used further. This phase is together with the next one a part of the semantic analysis concept. This step will perform more complicated analysis by verifying the parsed symbols adhere to the C# Language Specification.

5.1.3 Binding phase

The binding phase will, on-demand, bind symbols to identifiers in text. This is also part of the semantic analysis and results in the eventual semantic analysis. As a developer you will most often interact with this through the `semanticModel.GetSymbolInfo()` (and related) methods where we can retrieve the results. The previously mentioned method will allow us to retrieve a symbol from a given syntax node – which is what's created in the declaration phase.

5.1.4 Emit phase

Lastly, the emit phase will bundle everything in a module, wrap it in an assembly and emit to an outputstream together with debug information (Program Database (PDB) files), XML documentation, etc. One such a result that can come from this is a class library on disk.

---

[2]`http://stackoverflow.com/a/10183455/1864167`

5.2 Type structure

At its core, Roslyn creates tree representations from any given piece of code. Whether this concerns a single statement or an entire class doesn't matter – in the end, everything is reduced to a tree data structure. We can distinguish 2 different kinds of elements that make up the tree and a 3rd element that augments additional information.

5.2.1 Syntax Tree

The syntax tree in itself is the starting point from which a lot of aspects start or end: in a CodeFix you can end up replacing the original tree with the modified one while it might as well be the starting point with which you interpret plain text as something you can inspect.

Creating a syntax tree can be as easy as parsing plain text:

```
1 var tree = CSharpSyntaxTree.ParseText(@"
2 public class Sample
3 {
4 }");
```

which will result in the tree as shown in figure 5.1
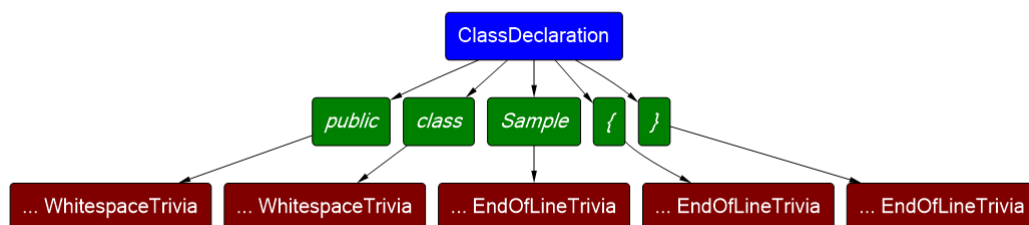
Figure 5.1: Syntax tree representation of a class definition

A syntax treerepresents at the highest level a single top-level type (class or struct) with all its containing members – fields, methods and nested types. Multiple syntax trees can be grouped in a compilation. It is however not a requirement to represent a type when parsing a syntax tree: any kind of statement or expression can be parsed and represented in a syntax tree. In the end, every syntax node represents a tree of itself. This idea of trees comprised of other trees is what enables interesting performance optimizations (more on that later).

Syntax trees have a few very important properties:

- Full fidelity – The guarantee that every character, whitespace, comment, line ending, etc from the source code is represented in the syntax tree.

- Round-trippable – Following from the above: translating the source code to a syntax tree and back results in exactly the same code.

- Code as written, not as executed – When you use the `async` and `await` keywords it will represent these keywords in the syntax tree and not the state machine that the compiler generates from it.

- immutable – Every syntax tree is immutable. This allows for thread-safe operations since everything handles its own "copy" of the syntax tree anyway.

5.2.2 Syntax Node

A syntax node is an element of the tree that is always the parent to other nodes or to tokens (also known as 'non-terminal') and represents an expression or a statement.

Syntax nodes are defined in the Syntax.xml[3] file which generates classes that represent the relationship between all kinds of syntax nodes.

5.2.3 Syntax Token

Syntax tokens are what make up the individual, small pieces of the code. Whereas syntax nodes mainly represent the syntactic constructs of the language, syntax tokens are the building blocks that these syntactic constructs consist of. A syntax token can be a keyword, identifier, literal and punctuation and only exists as a leaf in the tree: it will never be the parent of another node or token.

Contrary to syntax nodes, syntax tokens are not represented by separate classes. Instead, a single struct exists which can be differentiated based on its `Kind` property.

5.2.4 Syntax Trivia

The last aspect of the tree is the syntax trivia. This is slightly misleading because it's not actually a part of the tree: it is stored as a property on a syntax token. This stems from the idea that syntax tokens could not have child nodes and as such it is implemented by adding it to a token's `LeadingTrivia` and `TrailingTrivia` properties.

---

[3]`https://github.com/dotnet/roslyn/blob/master/src/Compilers/CSharp/Portable/Syntax/Syntax.xml`

Syntax trivia is the rest of the source code with no actual semantic meaning: comments, whitespace, newlines, etc. Important to note are preprocessing directives: these are also considered trivia.

It is important to represent the trivia inside the syntax tree in order to ensure full-fidelity: if we wouldn't, we would have a differently layed out source code after translating our source to a syntax tree and back.

5.3 Red-Green trees

5.3.1 The problem

We already know that we can use the syntax tree to traverse a node's descendants and we have also seen that we can look at a node's ancestors. Additionally we also know that every node is immutable: once created it cannot be changed. This introduces an interesting dilemma: how do you create a node that has knowledge of its parent *and* its children at the moment it is created? As we've just established: we can't create child- or parent-nodes afterwards and then modify the node since that would violate the immutability principle.

Another consideration stemming from this is performance related: node objects will never be able to be re-used because you can't change certain aspects such as the parent it refers to (again coming from that immutability requirement).

5.3.2 The solution

The solution to these problems came in the form of the so-called red-green trees. The basic idea is straightforward: Roslyn will create two trees, a red and a green one. The green nodes will contain "vague" information such as their width instead of their start- and end-position and they won't keep track of their parent. This tree is built from the bottom-up: this means that when a node is built, it knows about its children. When an edit occurs two things happen: a green node is created alongside a trivial red node (a node without a position or parent). The key here is that this red node also contains a reference to the underlying green node[4] which will allow us to construct a red node at *some point in time* using this green node's data. The emphasis is very important here: a red node is materialized lazily – this means that the red tree itself is only created when it's actually requested. For example when you browse through code in the Visual Studio IDE, it will create these red trees for each part of the code that the user can actually see and which needs highlighting. Code that the user cannot see doesn't need to be highlighted and as such does not need to be materialized into a red tree.

---

[4]http://source.roslyn.codeplex.com/#Microsoft.CodeAnalysis/Syntax/SyntaxNode.cs,41

When the red tree is built, Roslyn works with a top-down approach. This allows us to discover the exact location of every node by starting from position 0 and incrementing the position by the width of each subsequent node which we store in every red node's underlying green node. We also know the underlying green node's children because we built that green node from the bottom-up. This provides us with all the information we need to create a new red node at a certain position with a given parent and the necessary contextual information. Indeed, when we look at the constructor of a `SyntaxNode` (which is what we consider a 'red node') we can see it has the following definition:

```
1  internal SyntaxNode(GreenNode green,
2                      SyntaxNode parent,
3                      int position)
```

Listing 5.1: Constructing a (red) SyntaxNode

This perfectly displays the principle we talked about of creating a red node based on its parent, the position (which is calculated from the cumulative width of elements preceding it) and the associated green node.

A performance consideration worth mentioning here is the fact that generation of the red tree is cancelled when an edit occurs. Any analysis performed until that moment is outdated (or is potentially outdated) so there is no point in finishing it. The red tree at this point is discarded and a new green tree is generated based on the newly typed characters. In reality this green tree will be resolved into a red tree *almost* immediately – there is a slight delay between typing and generation of the red tree so there won't be a constant stream of newly created and aborted trees while the user is typing rapidly.

The colours "red" and "green" have no particular meaning – they were simply the two colours used to describe the idea on the Roslyn team's blackboard when designing this approach.(Lippert 2012)

5.4 Performance considerations

Performance is an integral aspect of any application and as such Roslyn doesn't escape from it either. It's not just about compiling an assembly as fast as possible: that's just one of the use cases in which Roslyn is used. Think about other scenarios such as getting quick intellisense[5] or fluent syntax highlighting of text as you scroll but also affects other performance aspects such as the memory impact.

In this section we will look at a handful of techniques the Roslyn team uses to optimize the platform. These are general approaches that often apply their ideas in several areas of the codebase, sometimes through a common resource. This should indicate that these are often optimizations at a high level rather than very specific single-use optimizations.

5.4.1 Concurrency

> A concurrent system is one where a computation can make progress without waiting for all other computations to complete – where more than one computation can make progress at "the same time"
>
> *Abraham Silberschatz*
> *Operating System Concepts 9th edition*

When you think of performance, concurrency is often one of the first aspects that come to mind. As such, it also takes a prominent place in Roslyn's architecture. We have already established that one of the characteristics of a syntax tree is its immutability – the inability to make changes to it after it is constructed. This is done with concurrency in mind: if we create a new tree for every concurrent operation, we have the guarantee that changes from operation X does not affect the tree that is being manipulated by operation Y. There are several more areas of concurrency though:

Source parsing

Every file containing source code is parsed independently of any other files. The file is parsed sequentially (from top to bottom) and multiple files can be parsed at once.(Sadov 2014)

---

[5]Intellisense: context-aware hints as you type

Symbol binding

When identifiers are bound to symbols (see Chapter 5.1.3) there is no real need to do this sequentially: in the end you basically just lookup the symbol in a table according to an identifier. One remark we have to add here though is the fact that a type's base members should also be bound when that type is being bound.

Consider the following example:

```
 1  class BaseType
 2  {
 3    public virtual void Method() { }
 4    public void BaseMethod() { }
 5  }
 6
 7  class SubType : BaseType
 8  {
 9    public override void Method()
10    {
11      base.BaseMethod();
12    }
13  }
14
15  class AnotherType
16  {
17    public void Method() { }
18  }
19
20  class StartUp
21  {
22    public static void Main(string[] args)
23    {
24      BaseType a = new SubType();
25      a.Method();
26      AnotherType b = new AnotherType();
27      b.Method();
28    }
29  }
```

Listing 5.2: Why type hierarchy matters for binding

When we bind `SubType` we have to bind `BaseType` as well because, as is indicated, there might be a semantical dependency: we have to know, for example, whether that `override` keyword is appropriate there. This is not a problem when there is no explicit base type involved: we know the base type is just `System.Object` which is likely already bound and as such there are no other types we have to investigate. For

38

this reason we can bind unrelated symbols in no specific order. We can say there is "implicit partial ordering".(Sadov 2014)

Method body compilation

Method bodies are bound and emitted on a type-by-type basis and in lexicographical order (based on the alphabet). This order is important: when compiling identical code multiple types it should emit the same result every time. Sometimes compiling a method creates an additional structure such as a state machine (as is the case with async/await and iterator blocks). This idea of a 'deterministic build' is looked at more closely in section 5.5.1.

Important to note is that method bodies are not emitted if there were any declaration errors. If that is the case, only binding is done for those aspects that can help in diagnosing the issue.(Sadov 2014)

5.4.2   Small nodes

Another performance-oriented aspect is the so-called "memory-footprint"[6]. When an application is executed, it uses memory to temporarily store data that makes up the program's workflow. This memory is (unfortunately) not unlimited: every device only has a certain amount available. The more memory is used, the less memory becomes available for other tasks and the sooner you run out. C# and VB.NET are managed languages which means the allocating and freeing of memory is done for the developer. The freeing of memory is done through the "Garbage Collector" (GC), a specialized service that will free unused memory locations when it deems it to be necessary. Discussing the GC is outside of the scope of this paper but it is important to realize that the more memory we use, the more the GC will have to step in and free up some of the unused memory locations so we can re-allocate these.(Todorov 2013)

Knowing that, we can take a look at the next performance aspect. As we have seen, syntax nodes are some of the most elemental constructs in our Abstract Syntax Tree (AST)[7]. Considering the sheer amount of nodes that a large applications exists of, it would pay off to keep these as small as possible. Doing so would reduce the amount of memory allocated which in turn would reduce the amount of garbage collections. A garbage collection is a relatively expensive operation since you basically put a hold on all active threads (except for the GC thread) so the GC can do its work.(Botelho 2014)

For this reason the Roslyn team decided to store certain information related to syntax nodes in a place that is not associated with that specific syntax node's object.(Sadov 2014) Metadata like diagnostic info and annotations are the two prime examples of this. In any given code base, most syntax nodes won't have any diagnostic information attached to them nor do they have annotations. This means that providing a field on every single syntax node object to store this information would be a waste of memory: in the end, a field with a `null` value will still use some memory due to its pointer. For a 32-bit process this will be a 4-bit pointer while a 64-bit process will use an 8-bit pointer[8]. Multiply this by all the relevant syntax nodes and you reach a sizable amount of wasted memory space.

What Roslyn does instead is store the diagnostics inside the syntax node's associated syntax tree and when the diagnostic information is actually requested, it will look them up inside that tree.

---

[6]Memory footprint: The amount of memory software uses when running
[7]AST: Abstract Syntax Tree
[8]https://msdn.microsoft.com/en-us/library/system.intptr.size(v=vs.110).aspx

This becomes clear when we take a look at the execution path in the code:

```
1  public new IEnumerable<Diagnostic> GetDiagnostics()
2  {
3     return this.SyntaxTree.GetDiagnostics(this);
4  }
```
Listing 5.3: CSharpSyntaxNode.GetDiagnostics

which passes it on to the following chain of calls inside the syntax tree:

```
1  GetDiagnostics(node.Green, node.Position);
2  EnumerateDiagnostics(greenNode, position);
3  new SyntaxTreeDiagnosticEnumerator(this, node, position);
```
Listing 5.4: CSharpSyntaxTree.GetDiagnostics

We can see a new specific enumerator is created which will traverse the syntax tree's nodes and return any diagnostics it finds. That is not the end, however. Eventually we still have to look up the diagnostic information and we know it's not stored inside the syntax node object. We follow the execution path again and now it becomes clear:

```
1  node.GetDiagnostics(); // node is a GreenNode
```
Listing 5.5: SyntaxTreeDiagnosticEnumerator.MoveNext

```
1  if (s_diagnosticsTable.TryGetValue(this, out diags))
```
Listing 5.6: GreenNode.GetDiagnostics

and eventually we reach a statically defined look-up table where we connect given syntax nodes with their respective diagnostic information:

```
1  private static readonly
2    ConditionalWeakTable<GreenNode, DiagnosticInfo[]>
3      s_diagnosticsTable =
4        new ConditionalWeakTable< GreenNode,
5                                  DiagnosticInfo[]>();
```
Listing 5.7: GreenNode.s_diagnosticsTable

This approach comes with the caveat that when there *are* diagnostics or annotations, retrieving them will be more expensive compared to just accessing a field. A trade-off had to be made here between either permanent extra memory-usage or occasionally extra look-up time and apparently the latter was decided to be the best option.

41

5.4.3 Object re-use

We've already highlighted the importance of memory impact in section 5.4.2 by making the nodes as small as possible. There is another approach we can take to this by making *duplicate* nodes take absolutely no space: we simply re-use existing objects. In this section we will mimic the experiments done by Robin Sedlaczek(Sedlaczek 2015) based on an explanation by Vladimir Sadov(Sadov 2014). In the end we will have shown how we reach a red-tree façade as shown in image 5.2 while having an underlying structure as shown in image 5.3.
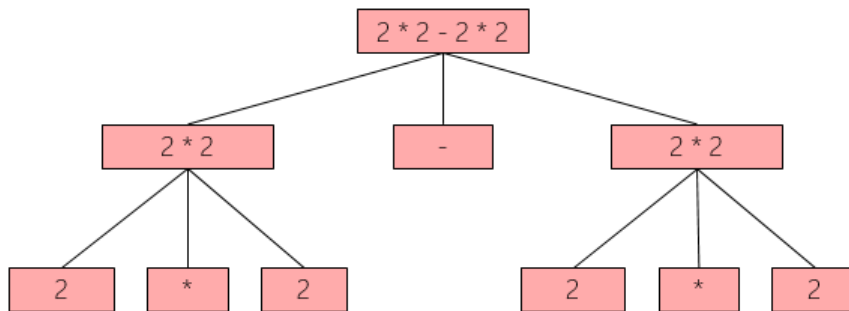
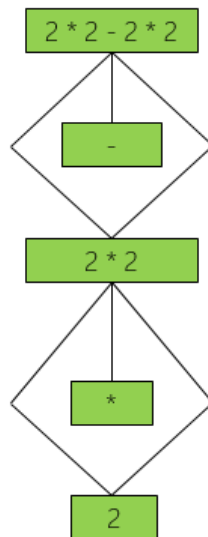Figure 5.2: Red-tree façade ©Robin Sedlaczek

Figure 5.3: Underlying green tree nodes ©Robin Sedlaczek

Re-using nodes

The first aspect we will look at is re-using our green nodes. As we established in section 5.3.2, green nodes have "vague" information: instead of a specific location in the syntax tree they instead contain the width of themselves. This is good because if it would have a specific location, we would (almost) never be able to re-use it: two distinct nodes in a single tree will be located at a different position. However by using the width instead we *can* consider two similar nodes as identical because we merely look at their width which is the same for both.

We can demonstrate this in the following example where we look at the node that represents 2 + 2.

```
1  int x = 2 + 2; // Width: 5; Position: 9
2  int y = 2 + 2; // Width: 5; Position: 23
```
Listing 5.8: Re-using nodes based on width vs position

Following out of this and the fact that green nodes don't contain any other uniquely identifying information such as a parent, we can conclude that a single green node can represent multiple identical subtrees. In order to see how this is implemented we have to take a look at the SyntaxNodeCache[9].

There are three main conclusions that we can take away from this implementation:

- Limited cache size

  Only a limited amount of nodes is stored in this cache. A trade-off has to be made between execution time and memory impact and the Roslyn team has decided to use a cache size of 65536 items (1 « 16) as evidenced here:

```
1  private const int CacheSizeBits = 16;
2  private const int CacheSize = 1 << CacheSizeBits;
3  private const int CacheMask = CacheSize - 1;
```
Listing 5.9: Definition of the SyntaxNodeCache's size

- First In, First out

  When a node is added, a hash key will be calculated. As is typical with hash implementations, this should strive to reach a uniform distribution across the data structure (an array in this case) for optimal performance. Based on this hash key and the mask, the to-be-cached node will be inserted at a certain location of the array. If an entry already exists at this location it will be overwritten, hence the First In, First Out (FIFO) (First In, First Out) principle.

---

[9]https://github.com/dotnet/roslyn/blob/b908b05b41d3adc3b5e81f8cf2d0055c13e4a1f6/
src/Compilers/CSharp/Portable/Syntax/InternalSyntax/SyntaxNodeCache.cs

```
1 var idx = hash & CacheMask;
2 s_cache[idx] = new Entry(hash, node);
```

Listing 5.10: Inserting a node using FIFO

- Up to 3 children

The last important aspect of this cache is the fact that every node can only have up to three children. If it has any more, the chance of a cache miss is too big(Sadov 2014) so these are not allowed in the first place. This is enforced by making sure there are only overloads available for 1, 2 and 3 nodes.

```
 1 private static bool CanBeCached(GreenNode child1)
 2 {
 3    return  child1 == null || child1.IsCacheable;
 4 }
 5
 6 private static bool CanBeCached(GreenNode child1,
 7                                GreenNode child2)
 8 {
 9    return  CanBeCached(child1) &&
10            CanBeCached(child2);
11 }
12
13 private static bool CanBeCached(GreenNode child1,
14                                GreenNode child2,
15                                GreenNode child3)
16 {
17    return  CanBeCached(child1) &&
18            CanBeCached(child2) &&
19            CanBeCached(child3);
20 }
```

Listing 5.11: Caching up to three children

Re-using tokens

Tokens are also cached but take a slightly different approach to doing so. When the source text is being parsed, the `QuickScanner` looks inside the `LexerCache` which maintains caches for both trivia and tokens.

```
1  if (state == QuickScanState.Done)
2  {
3    // this is a good token!
4    var token = _cache.LookupToken(
5        TextWindow.CharacterWindow,
6        TextWindow.LexemeRelativeStart,
7        i - TextWindow.LexemeRelativeStart,
8        hashCode,
9        _createQuickTokenFunction);
10   return token;
11 }
```

Listing 5.12: QuickScanner.QuickScanSyntaxToken

Inside the `LexerCache` the caches for trivia and tokens are stored by way of a `TextKeyedCache` implementation. A `TextKeyedCache` maintains two levels of caching: a local one and a shared one, each with their own characteristics.
L1 Cache:

- Small cache size (2^{11} items)

- Fast

- Thread-unsafe

- Local to the parsing session

L2 Cache:

- Larger cache size (2^{16} items)

- Slower

- Thread-safe

- Shared between all parsing sessions

When an item is searched for in the cache, the `TextKeyedCache` will first attempt to find it in the local cache (`_localTable`) and if it wasn't found there, look in the shared cache (`s_sharedTable`).

```
1  internal T FindItem(char[] chars ,
2                      int start ,
3                      int len ,
4                      int hashCode)
5  {
6    var idx = LocalIdxFromHash(hashCode);
7    var text = _localTable[idx].Text;
8
9    if (text != null && arr[idx].HashCode == hashCode)
10   {
11     if (StringTable.TextEquals( text ,
12                                 chars ,
13                                 start ,
14                                 len))
15     {
16       // Return from local cache
17     }
18   }
19
20   SharedEntryValue e = FindSharedEntry( chars ,
21                                         start ,
22                                         len ,
23                                         hashCode);
24   if (e != null)
25   {
26     // Return from shared cache
27   }
28 }
```

Listing 5.13: TextKeyedCache.FindItem

Measurements

In order to visualize the impact of this approach, we will look at three scenarios of syntax tree parsing and see how they affect the heap in terms of object allocation.

- **Scenario 1:** Parsing two identical syntax trees

  Expectation: When the first tree is created, there will be a relatively large allocation which comes from the green nodes, trivial red nodes and the tokens. Since almost all nodes and tokens should be re-used, we expect only very minimal overhead that comes from creating the second syntax tree object itself.

  Demonstration:

```
1  internal class Experiment1
2  {
3    public const string tree1 = @"
4  using System;
5  using System.Text;
6
7  class MyClass
8  {
9      void MyMethod()
10     {
11         int result = 2 + 2;
12         Console.WriteLine(result);
13     }
14 }";
15   private static void Main(string[] args)
16   {
17     var obj1 = CSharpSyntaxTree.ParseText(tree1);
18     var obj2 = CSharpSyntaxTree.ParseText(tree1);
19   }
20 }
```

Listing 5.14: Parsing two identical syntax trees

Results:

	Time	Objects (Diff)		Heap Size (Diff)	
1	0.08s	1 260	(n/a)	127,25 KB	(n/a)
2	0.24s	2 910	(+1 650 ↑)	2 450,95 KB	(+2 323,70 KB ↑)
3	0.24s	2 923	(+13 ↑)	2 451,42 KB	(+0,47 KB ↑)

Figure 5.4: Memory impact comparison when parsing two identical syntax trees

47

Object Type	Count Diff. ▼	Size Diff. (Bytes)
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxToken+SyntaxTokenWithTri...	+918	+22 032
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxToken	+304	+4 864
Roslyn.Utilities.TextKeyedCache+SharedEntryValue<Microsoft.CodeAnalysis.CSharp.Synt...	+24	+384
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxTrivia	+13	+344
RuntimeType	+5	+140
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxToken+SyntaxIdentifier	+5	+100
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.IdentifierNameSyntax	+5	+100
StringBuilder	+3	+380
Roslyn.Utilities.StringTable	+3	+49 248
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxList+WithTwoChildren	+2	+48
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxToken+SyntaxIdentifierWith...	+2	+48
Roslyn.Utilities.TextKeyedCache+SharedEntryValue<Microsoft.CodeAnalysis.CSharp.Synt...	+2	+32
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxToken+SyntaxTokenWithVal...	+2	+64
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.UsingDirectiveSyntax	+2	+72
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.PredefinedTypeSyntax	+2	+40
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.LiteralExpressionSyntax	+2	+40

Figure 5.5: Memory impact of the first parsing session with identical trees

Object Type	Count Diff.	Size Diff. (Bytes)
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.UsingDirectiveSyntax	+2	+72
HandleCollector+HandleType	+1	+28
Microsoft.CodeAnalysis.Text.StringText	+1	+32
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxToken+SyntaxToken...	+1	+24
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxList+WithTwoChildren	+1	+24
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxList+WithManyChildren	+1	+40
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.BlockSyntax	+1	+28
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.MethodDeclarationSyntax	+1	+60
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.ClassDeclarationSyntax	+1	+60
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.CompilationUnitSyntax	+1	+36
Microsoft.CodeAnalysis.CSharp.Syntax.CompilationUnitSyntax	+1	+40
Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree+ParsedSyntaxTree	+1	+68

Figure 5.6: Memory impact of the second parsing session with identical trees

48

Conclusion: The first parsing session creates a lot of different objects while the next session only creates a relative tiny amount of new objects. An interesting aspect to note here is the kind of objects we see created in that first run (see figure 5.5): every token is cached when that first parsing session starts which causes a lot of initial allocations. When we look further down the list of allocated objects we also see caches, the parser, specific object pools, factories and a lot more. However when we now look at the second parsing session and see how it affects the heap we see none of this: there are a few new nodes representing the broad lines of the tree but no new tokens, identifiers, parsers, caches, etc have been allocated.

- **Scenario 2:** Parsing two slightly different syntax trees

  Expectation: Corresponding to the previous experiment, we expect a lot of allocations in the first parsing session. The second parsing session should display a slightly higher amount of newly allocated objects than in the first experiment but it should still be relatively little considering most of the allocations came from tree-independent work like populating caches and creating parsers.

Demonstration:

```
internal class Experiment2
{
  public const string tree1 = @"
using System;
using System.Text;

class MyClass
{
    void MyMethod()
    {
        int result = 2 + 2;
        Console.WriteLine(result);
    }
}";

  public const string tree2 = @"
using System;
using System.Text;

class MyClass
{
    private string myString = ""hello"";

    void MyMethod()
    {
        int result = 2 + 2;
        Console.WriteLine(result);
    }
}";

  private static void Main(string[] args)
  {
    var obj1 = CSharpSyntaxTree.ParseText(tree1);
    var obj2 = CSharpSyntaxTree.ParseText(tree2);
  }
}
```

Listing 5.15: Parsing two slightly different syntax trees

Results:

	Time	Objects (Diff)		Heap Size (Diff)	
1	0.08s	1 261	(n/a)	127,66 KB	(n/a)
2	0.23s	2 911	(+1 650 ↑)	2 451,35 KB	(+2 323,70 KB ↑)
➤ 3	0.23s	2 950	(+39 ↑)	2 452,47 KB	(+1,12 KB ↑)

Figure 5.7: Memory impact comparison when parsing two slightly different syntax trees

Object Type	Count Diff.	Size Diff. (Bytes)
String	+9	+276
Roslyn.Utilities.TextKeyedCache+SharedEntryValue<Microsoft.CodeAnalysis.CSharp.Synt...	+4	+64
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxToken+SyntaxTokenWithTri...	+3	+72
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxList+WithTwoChildren	+3	+72
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.UsingDirectiveSyntax	+2	+72
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.PredefinedTypeSyntax	+2	+40
HandleCollector+HandleType	+1	+28
Microsoft.CodeAnalysis.Text.StringText	+1	+32
Microsoft.CodeAnalysis.ArrayElement<Microsoft.CodeAnalysis.CSharp.Syntax.InternalSy...	+1	+20
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxToken+SyntaxIdentifierWith...	+1	+24
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.LiteralExpressionSyntax	+1	+20
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.EqualsValueClauseSyntax	+1	+24
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.VariableDeclaratorSyntax	+1	+28
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.VariableDeclarationSyntax	+1	+24
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxList+WithManyChildren	+1	+20
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.BlockSyntax	+1	+28
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.MethodDeclarationSyntax	+1	+60
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.ClassDeclarationSyntax	+1	+60
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.CompilationUnitSyntax	+1	+36
Microsoft.CodeAnalysis.CSharp.Syntax.CompilationUnitSyntax	+1	+40
Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree+ParsedSyntaxTree	+1	+68
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.SyntaxToken+SyntaxTokenWithVal...	+1	+32
Microsoft.CodeAnalysis.CSharp.Syntax.InternalSyntax.FieldDeclarationSyntax	+1	+32

Figure 5.8: Memory impact of the second parsing session with slightly different syntax trees

Conclusion: As expected there are indeed more additional allocations in experiment 2 compared to experiment 1 (39 vs 13 respectively) which is still dwarfed by the initial allocation of 1650 objects. What we can see in figure 5.8 are the additional string allocations and the syntax tree that makes up a field declaration (`FieldDeclarationSyntax`, `VariableDeclarationSyntax`, `VariableDeclaratorSyntax`, and so on).

51

- **Scenario 3:** Iterating through the nodes of a tree

Expectation: We know that red nodes are materialized lazily: they have a trivial structure when the green node is constructed and get expanded to full-fledged red nodes only when this is requested. By retrieving all the descendent nodes from the root of the tree, we expect a lot of extra allocations (relative to previous experiments). If we look at previous results we can tell they are all green nodes because the objects are contained in the `InternalSyntax` namespace. By materializing, we should find allocations from objects based in the `Syntax` namespace – the red nodes that are exposed through APIs.

Demonstration:

```
 1  internal class Experiment3
 2  {
 3      public const string tree1 = @"
 4  using System;
 5  using System.Text;
 6
 7  class MyClass
 8  {
 9      void MyMethod()
10      {
11          int result = 2 + 2;
12          Console.WriteLine(result);
13      }
14  }";
15      private static void Main(string[] args)
16    {
17      var obj1 = CSharpSyntaxTree.ParseText(tree1);
18      var objects = obj1.GetRoot()
19                        .DescendantNodesAndSelf()
20                        .ToArray();
21    }
22  }
```

Listing 5.16: Iterating through the nodes of a tree

Results:

	Time	Objects (Diff)		Heap Size (Diff)	
1	0.08s	1 260	(n/a)	127,25 KB	(n/a)
2	0.23s	2 910	(+1 650 ↑)	2 450,95 KB	(+2 323,70 KB ↑)
➤ 3	0.33s	2 950	(+40 ↑)	2 452,31 KB	(+1,37 KB ↑)

Figure 5.9: Memory impact comparison when materializing red nodes

Object Type	Count Diff.	Size Diff. (Bytes)
Microsoft.CodeAnalysis.CSharp.Syntax.IdentifierNameSyntax	+6	+144
Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax	+2	+64
Microsoft.CodeAnalysis.CSharp.Syntax.PredefinedTypeSyntax	+2	+48
WeakReference<Microsoft.CodeAnalysis.SyntaxNode>	+2	+24
Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax	+2	+48
Int32[]	+1	+20
HandleCollector+HandleType	+1	+28
Microsoft.CodeAnalysis.SyntaxNode+<>c	+1	+12
Microsoft.CodeAnalysis.SyntaxNode+ChildSyntaxListEnumeratorStack+<>c	+1	+12
Roslyn.Utilities.ObjectPool+Factory<Microsoft.CodeAnalysis.ChildSyntaxList+Enumerato...	+1	+32
Roslyn.Utilities.ObjectPool<Microsoft.CodeAnalysis.ChildSyntaxList+Enumerator[]>	+1	+20
Roslyn.Utilities.ObjectPool+Element<Microsoft.CodeAnalysis.ChildSyntaxList+Enumerat...	+1	+72
Microsoft.CodeAnalysis.SyntaxNode[]	+1	+124
Microsoft.CodeAnalysis.ChildSyntaxList+Enumerator[]	+1	+204
Microsoft.CodeAnalysis.CSharp.Syntax.SyntaxList+WithTwoChildren	+1	+32
Microsoft.CodeAnalysis.CSharp.Syntax.QualifiedNameSyntax	+1	+32
Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax	+1	+44
Microsoft.CodeAnalysis.CSharp.Syntax.MethodDeclarationSyntax	+1	+56
Microsoft.CodeAnalysis.CSharp.Syntax.ParameterListSyntax	+1	+28
Microsoft.CodeAnalysis.CSharp.Syntax.BlockSyntax	+1	+28
Microsoft.CodeAnalysis.CSharp.Syntax.SyntaxList+WithManyWeakChildren	+1	+32
Microsoft.CodeAnalysis.ArrayElement<WeakReference<Microsoft.CodeAnalysis.SyntaxN...	+1	+20
Microsoft.CodeAnalysis.CSharp.Syntax.LocalDeclarationStatementSyntax	+1	+28
Microsoft.CodeAnalysis.CSharp.Syntax.VariableDeclarationSyntax	+1	+32
Microsoft.CodeAnalysis.CSharp.Syntax.VariableDeclaratorSyntax	+1	+32
Microsoft.CodeAnalysis.CSharp.Syntax.EqualsValueClauseSyntax	+1	+28

Figure 5.10: Memory impact when materializing red nodes

Conclusion: We can see that a significant amount of allocations has occurred by iterating through our syntax tree. These objects are almost all located in the `Syntax` namespace which confirms our expectation that we're dealing with red nodes. It's interesting to notice that every type that existed in our green tree now created the corresponding type in the red tree. We also see the `WeakReference` in play which we will discuss in section 5.4.4.

5.4.4   Weak references

> A weak reference, simply put, is a reference that isn't strong enough to force an object to remain in memory. Weak references allow you to leverage the Garbage Collector's ability to determine reachability for you, so you don't have to do it yourself.
>
> *Ethan Nicholas*
> *Understanding Weak References*[10]

As we talked about before in section 5.4.2, C# is a managed language that manages its memory usage with the GC. In essence it boils down to this: if an object has no references it is eligible for Garbage Collection and the GC will mark it as such so it can be cleaned up in the next iteration. You might have guessed by the title of the section that the above explanation should actually specify it as *strong references* rather than just *references*. A "strong reference" is the default reference type we typically use when referencing another object.

A `WeakReference` is a construct that allows us to reference an object but if the GC decides to clean up that reference it can do so without a problem. You notice from this description that it is perfectly suited for a caching mechanism: that's basically the same result as when we create a cache with a certain size – the size is supposed to prevent us from reaching memory issues.

In section 5.3.2 we already talked shortly about syntax highlighting in the IDE: scrolling through the editor will cause the syntax tree to become materialized which means creating new objects to represent those syntactic constructs. However once a part of the tree leaves the window of the user there is no point in keeping it materialized anymore since the user can't see it anyway. At this point you want those unnecessary materialized nodes to be garbage collected but there is one major problem: nodes are referenced by their parent and its children. If we would use traditional strong references we would never be able to collect any of these objects.

One major area in which this technique is used is when it comes to a method body. Method bodies are in essence a group of statements. While it can be very useful to have information such as the method's definition available while programming, the exact contents of that method its implementation are less important – this makes it a perfect target to use weak references.

This is implemented in the `SyntaxList.WithManyWeakChildren` type. It works very straightforward: by storing an array with as type a `WeakReference<SyntaxNode>`

---

[10]http://weblogs.java.net/blog/enicholas/archive/2006/05/understanding_w.html

we are already done: now these syntax nodes will be able to be GC'd if the GC marks them and no strong references exist to those particular nodes.

```
private readonly
   ArrayElement<WeakReference<SyntaxNode>>[] _children;
```
Listing 5.17: SyntaxList.WithManyWeakChildren._children

In the base type SyntaxList.WithManyChildrenBase we can see how the creation of the red node is delegated:

```
internal override SyntaxNode CreateRed(SyntaxNode parent,
                                       int position)
{
  var p = parent;
  if (p != null && p is CSharp.Syntax.BlockSyntax)
  {
    var gp = p.Parent;
    if (gp != null &&
        (gp is CSharp.Syntax.MemberDeclarationSyntax ||
         gp is CSharp.Syntax.AccessorDeclarationSyntax))
    {
      return new
        SyntaxList.WithManyWeakChildren(this,
                                        parent,
                                        position);
    }
  }

  if (this.SlotCount > 1 && HasNodeTokenPattern())
  {
    return new
      SyntaxList.SeparatedWithManyChildren(this,
                                           parent,
                                           position);
  }
  else
  {
    return new
      SyntaxList.WithManyChildren(this,
                                  parent,
                                  position);
  }
}
```
Listing 5.18: SyntaxList.WithManyWeakChildrenBase.CreateRed

5.4.5 Object pooling

In chapter 5.4.3 we've discussed a specific approach at minimizing the amount of objects created. In this section we will take a look at a more general implementation of this idea. While nodes and tokens are at the very core of the Roslyn code base, they are not the only objects we will create.

When creating a lexer, one of the major tasks you're performing is reading source code and creating strings. If you would intermittently concatenate strings, this would increase memory pressure by a lot: every concatenation using + creates a new string but you might only be interested in the result[11]. However, creating a new `StringBuilder` object every time you want to concatenate strings is an expensive operation as well: since a lot of string concatenation is done, a lot of distinct `StringBuilder` instances will be created and you again have the issue as described before. The solution here is to take pooling one step further: every type can be pooled through the `ObjectPool<T>` class.(Warren 2014) This provides a generic implementation of the object pooling pattern and will be used through intermediate classes.

An accurate description of this class' workings can be found in its documentation[12]:

> Generic implementation of object pooling pattern with predefined pool size limit. The main purpose is that limited number of frequently used objects can be kept in the pool for further recycling.
>
> Notes:
>
> 1) it is not the goal to keep all returned objects. Pool is not meant for storage. If there is no space in the pool, extra returned objects will be dropped.
>
> 2) it is implied that if object was obtained from a pool, the caller will return it back in a relatively short time. Keeping checked out objects for long durations is ok, but reduces usefulness of pooling. Just new up your own.
>
> Not returning objects to the pool in not detrimental to the pool's work, but is a bad practice. Rationale: If there is no intent for reusing the object, do not use pool - just use "new".

An interesting addition here is the implementation of `SharedPools`. As the name indicates, this is a class that manages the sharing of object pools. There are two ways to use this class:

---

[11]https://msdn.microsoft.com/en-us/library/2839d5h5(v=vs.110).aspx
[12]`https://github.com/dotnet/roslyn/blob/b908b05b41d3adc3b5e81f8cf2d0055c13e4a1f6/src/Compilers/Core/SharedCollections/ObjectPool%601.cs`

- As a single, standalone pool
  This does not share the pool but instead just makes it easier to create your object pools. For this purpose you can choose between a small pool (20 elements) and a big pool (100 elements). For example in the case of a small pool you call

```
public static ObjectPool<T> Default<T>()
                             where T : class,
                                       new()
{
  return DefaultNormalPool<T>.Instance;
}
```

Listing 5.19: SharedPools.Default<T>

This method, in turn, will lazily create a new pool the first time a pool of that type is requested. This pool will defer to the beforementioned `ObjectPool` implementation:

```
private static class DefaultNormalPool<T>
                             where T : class,
                                       new()
{
  public static readonly ObjectPool<T> Instance =
    new ObjectPool<T>(() => new T(), 20);
}
```

Listing 5.20: SharedPools.DefaultNormalPool<T>

- As a shared pool

  Taking the entire idea another step further, you can also share the pools themselves. This is useful to share data inside the pools: if in one parsing session you create a string "using" then it would be nice to re-use that in another parsing session. This follows the same idea of the statically shared `TextKeyedCache` pool as we talked about in section 5.4.3 but expands this usage across all types due to its generic nature. Here again we see two separate usages:

  As a pre-defined pool

  Inside the `SharedPools` we have a few pre-defined pools for general use cases. One of those is `StringIgnoreCaseHashSet` which provides a hashset pool that compares its elements case-insensitively.

```
1  public static readonly
2    ObjectPool<HashSet<string>> StringIgnoreCaseHashSet =
3    new ObjectPool<HashSet<string>>(
4      () => new HashSet<string>(
5                StringComparer.OrdinalIgnoreCase), 20);
```

Listing 5.21: SharedPools.StringIgnoreCaseHashSet

As a pre-defined type

There are also specific types that provide some type-specific behaviour. An example of this is the `StringBuilderPool`:

```
1  internal static class StringBuilderPool
2  {
3    public static StringBuilder Allocate()
4    {
5      return SharedPools.Default<StringBuilder>()
6                     .AllocateAndClear();
7    }
8
9    public static void Free(StringBuilder builder)
10   {
11     SharedPools.Default<StringBuilder>()
12               .ClearAndFree(builder);
13   }
14
15   public static string ReturnAndFree(
16                         StringBuilder builder)
17   {
18     SharedPools.Default<StringBuilder>()
19               .ForgetTrackedObject(builder);
20     return builder.ToString();
21   }
22 }
```

Listing 5.22: StringBuilderPool

5.4.6 Specialized collections

Collections in the .NET framework are heavily optimized by themselves already but it is done so in a general manner: in the end, the .NET framework will be used by many different people so optimizations have to be weighed against a lot of different scenarios. By creating your own specialized collections, you have the possibility to cut corners that might not be applicable in your domain's scenario. In its crudest form: if you have a `null` check inside the collection's operations but you know for sure that that can never be `null` then that is an instruction you can skip. Another important reason to use your own custom collections is to formalize the specific behaviour of your domain that belongs to a collection.

IdentifierCollection

An example of the latter can be found in the form the `IdentifierCollection`. This class acts as a wrapper around a `Dictionary` that maps a certain key to one or more alternative spellings of that key. While in essence nothing more than a mapper between key and value, it provides more intuitive methods to interact with such as `AddIdentifier`, `AddInitialSpelling` and `AddAdditionalSpelling`. In this case, we can see a performance consideration in the way it stores the alternative identifiers: the backing `Dictionary` is actually mapping a `string` to an `object` instead of an array as one would expect. The reason this is done is to avoid having to allocate an array when there is just one alternative identifier – now we can just define it as a `string` without the need for an array.

We can see this consideration reflected when we try to add a new spelling to a given key:

```csharp
private void AddAdditionalSpelling(string identifier,
                                   object value)
{
  // Had a mapping for it.  It will either map to a single
  // spelling, or to a set of spellings.
  var strValue = value as string;
  if (strValue != null)
  {
    if (!string.Equals(identifier,
                       strValue,
                       StringComparison.Ordinal))
    {
      // We now have two spellings. Create a collection
      // for that and map the name to it.
      _map[identifier] = new HashSet<string>
        { identifier, strValue };
    }
  }
  else
  {
    // We have multiple spellings already.
    var spellings = (HashSet<string>)value;

    // Note: the set will prevent duplicates.
    spellings.Add(identifier);
  }
}
```

Listing 5.23: IdentifierCollection.AddAdditionalSpelling

Notice also the note on line 20: smart usage of existing collections reduces the complexity of our own code.

CachingDictionary

Another implementation is the `CachingDictionary`. As its name indicates, it represents an in-memory caching mechanism based on the `Dictionary` collection. There is a very interesting implementation detail here: the backing dictionary is a `ConcurrentDictionary` meaning its exposed methods are thread-safe. However, when the backing dictionary is full, it actually swaps that out for a normal `Dictionary` to allow for less-expensive read-operations without the thread-safety overhead.

```
1  private static bool IsNotFullyPopulatedMap
2      (IDictionary<TKey,  ImmutableArray<TElement>> map)
3  {
4    return
5      map == null ||
6      map is
7       ConcurrentDictionary<TKey, ImmutableArray<TElement>>;
8  }
```

Listing 5.24: CachingDictionary.IsNotFullyPopulatedMap

5.4.7 LINQ

> The code regions which take the most execution time are called *hot spots*. The hot spots are the best places to tune and optimize because a little bit of effort on making a hot spot faster can have a big pay-off. It's not worth spending engineering effort on a bit of code that executes only once or doesn't take much time.
>
> *Paul J. Drongowski*
> *PERF tutorial: Finding execution hot spots*[13]

An explicit performance consideration mentioned in the "Contributing Code" guidelines on the project's Github page[14] explicitly mentions Language Integrated Query (LINQ)[15]:

> Avoid allocations in compiler hot paths:
>
> - Avoid LINQ.
> - Avoid using `foreach` over collections that do not have a `struct` enumerator.
> - Consider using an object pool. There are many usages of object pools in the compiler to see an example.

It's already clear that the main idea here is minimizing allocations made – an overarching theme which we have also discussed in previous sections. In this section we will focus on LINQ and some of the performance implications that come with it. The two other bullet points also have to do with memory allocation: both concern memory allocation on the heap which will eventually trigger a Garbage Collector.

The concern here is that LINQ adds an unacceptable overhead in the team's eyes. This overhead is inherent to LINQ: you make a trade-off between memory and performance impact and instead end up with much more readable code.

As means of a simple test, we'll take a look at a snippet of code and its LINQ alternative and compare their memory impact. We'll create two memory snapshots

---

[13]http://sandsoftwaresound.net/perf/perf-tutorial-hot-spots/
[14]https://roslyn.codeplex.com/wikipage?title=How%20to%20Contribute
[15]LINQ: Language-Integrated Query, syntax features to query data

Object Type	Count Diff.	Size Diff. (Bytes)
Enumerable+WhereListIterator<Int32>	+1	+44
Func<Int32, Boolean>	+1	+32
HandleCollector+HandleType	+1	+28

Figure 5.11: Memory impact when comparing the LINQ query to the baseline

and see how many extra objects have been created for each implementation. It should obviously be noted that not all LINQ statements are equally wasteful and in some scenarios LINQ can even make sure there is less memory impact.

The code we use to test can be found in listing 5.25 and adds 3 items from one list to another based on a certain condition. One approach uses a LINQ query while the other uses a `foreach` loop which is basically the unrolled equivalent of the former. The heap snapshots in figures 5.11 and 5.12 give use a clear view of what objects are being allocated along the way: we can see that constructing the LINQ query creates two new objects of types `Enumerable+WhereListIterator<Int32>` and `Func<Int32, Boolean>`, representing the `Where` structure and the where condition respectively. Additionally there is an object `HandleCollector+HandleType` which is overhead created by the GC.

When we take a look at the next snapshot we can see that after Garbage Collecting and running our `foreach` loop, we now notice that we only create the overhead object and that the `WhereListIterator` has been Garbage Collected. You might wonder what happened to the `Func` – in the end, that condition isn't referenced anywhere anymore and as such it should be eligible for collection. The optimization at play here is that the compiler caches lambdas if they adhere to certain characteristics such as not using a local variable[16]. By using this lambda in our code we now have an object that will live for a long time and, by definition, increase memory pressure. Using the caching technique keeps this memory pressure limited but knowing that there are at least one or two object allocations for the simplest of LINQ queries makes it easy to see why they should be avoided in hot paths.

Object Type	Count Diff. ▼	Size Diff. (Bytes)
HandleCollector+HandleType	+1	+28
Enumerable+WhereListIterator<Int32>	-1	-44

Figure 5.12: Memory impact when comparing the loop to the LINQ query

```
1  static void Main(string[] args)
```

---

[16]http://stackoverflow.com/a/6280711/1864167

```
 2  {
 3          // Setting baseline snapshot
 4          var list1 = new List<int> {4862, 6541, 7841};
 5          var list2 = new List<int>(list1.Count);
 6          var list3 = new List<int>(list1.Count);
 7
 8          // First snapshot: LINQ usage
 9          list2.AddRange(list1.Where(item => item > 5000 &&
10                                              item < 7000));
11
12          // Second snapshot: foreach-loop
13          foreach (var item in list1)
14          {
15              if (item > 5000 && item < 7000)
16              {
17                  list3.Add(item);
18              }
19          }
20  }
```

Listing 5.25: Comparing LINQ memory impact

5.5   Security Considerations

In this last section we will take a look at some security considerations surrounding Roslyn. It is important to realize that the code base by itself is very hard to secure: there are several algorithms in it that have exponential or polynomial time – executing a Denial of Service (DoS)[17] is trivial since all you have to do is pass in the appropriate data. For this reason it is very hard to secure a compiler server(Gocke 2015) since legitimate input and malicious input are virtually impossible to distinguish. However, there are some areas where we can identify a (shared) responsibility for developers in the way they utilize the Roslyn platform.

---

[17]DoS: Denial of Service by flooding the application with work

5.5.1 Deterministic builds

> With the number of dependencies present in large software projects, there is no way any amount of global surveillance, network censorship, machine isolation, or firewalling can sufficiently protect the software development process of widely deployed software projects in order to prevent scenarios where malware sneaks into a development dependency through an exploit in combination with code injection, and makes its way into the build process of software that is critical to the function of the world economy.
>
> *Mike Perry*
> *Deterministic Builds Part One:*
> *Cyberwar and Global Compromise*[18]

When it comes to security, one of the main guarantees you're interested is that the the file you have locally is the same as what has been audited by the vendor and/or community. A popular way to do this is by having the vendor/community create a hash of the application and store it publicly. Afterwards the user can download the binaries or source files (which can be anything from a small plugin to a full-fledged application), generate a hash from those files using the same algorithm and compare the results. Ideally, these two hashcodes should be the same. If they are not the same, there are three possible causes:

- The remote hash is outdated

- The source files have been replaced (maliciously)

- The application is not deterministic

In this section we will talk about the third option. It should be noted that at the time of writing deterministic builds have not been entirely implemented yet in the Roslyn code base but if the tracking issue #372[19] is any indication, there is only a limited amount of work left to be done.

---

[18]https://blog.torproject.org/blog/deterministic-builds-part-one-cyberwar-and-global-compromise
[19]https://github.com/dotnet/roslyn/issues/372

What makes a build deterministic?

> "deterministic builds" – packages which are byte-for-byte identical no matter who actually builds them, or what hardware they use.(Perry 2013)

This definition immediately highlights the key idea behind deterministic builds: when a developer builds the Roslyn solution on his local machine, he should receive an identical result as if it had happened on Microsoft's build-servers. This is a harder task than it looks: it means that your code base can't contain any non-deterministic behaviour. While this might seem like an obvious conclusion, it implies that every single aspect of the code base that has an effect on its output must be implicitly or explicitly the same every time the code is built.

As an example we can look at Roslyn issue #223[20] which specifies that in a certain scenario, anonymous types are emitted in a different order than expected. Looking closer at the diagnosis presented by the team we can see that the culprit seems to be the enumerating of dictionary entries.

Looking at the exact fix in PR #948[21] we see that the change consisted switching from a non-deterministic iteration to a deterministic one. When you iterate the `Values` property of a `Dictionary<K, V>` you do so in an unspecified order as we can read from the documentation:

> The order of the values in the `Dictionary<TKey, TValue>.ValueCollection` is unspecified, but it is the same order as the associated keys in the `Dictionary<TKey, TValue>.KeyCollection` returned by the `Keys` property.[22]

Indeed, the solution was to switch from an undeterministic iteration (listing 5.26) to a deterministic one by ordering the collection first (listing 5.27).

```
1 | foreach (var template in anonymousTypes.Values)
```

Listing 5.26: Undeterministic iteration of values

```
1 | foreach (var template in from kv in anonymousTypes
2 |                          orderby kv.Key
3 |                          select kv.Value)
```

Listing 5.27: Deterministic iteration of values

---

[20]https://github.com/dotnet/roslyn/issues/223
[21]https://github.com/dotnet/roslyn/pull/948/files
[22]https://msdn.microsoft.com/en-us/library/ekcfxy3x.aspx

66

Other benefits to deterministic builds

The security aspect is a major reason to provide this characteristic to your code base but there are others:

- Reduces surprises by not relying on unspecified behaviour

- Reduces I/O load by being able to skip a specific binary in certain situations if a binary with that exact hash is already stored[23]

- Reduces unit testtest execution time through the ability of re-using cached test results if the tested binary has not changed[23]

5.5.2   External analyzers

Perhaps the most important security risk at all is the fact that the developer is adding external code to his development environment. There are no restrictions to what an analyzer can do: it could upload your source code to a remote server, it could search through your file system, it could download malware[24] and hide it on your system, etc. This is not an uncommon risk however: people install extensions in their software all the time and Visual Studio is no exception to that: think about browser extensions or Node Package Manager (npm) packages[25] for example.

An interesting attack vector here is that it's not something we're always aware of. For example you could be browsing a repository on Github when you suddenly decide to contribute so you fork the project and make alterations locally. What you didn't know, however, was that the project contained an analyzer which looked through your personal files, found a folder called 'my_company_repositories' and uploaded your proprietary code to locations unknown. At no point were you aware this was happening because analyzers are somewhat hidden in the sense that they're considered references (figure 5.13) but contrary to 'traditional' references, these are executed when the application isn't: as soon as you open a solution with a malicious analyzer, it has complete access to your system.

---

[23]https://www.chromium.org/developers/testing/isolated-testing/
deterministic-builds
[24]Malware: malicious software
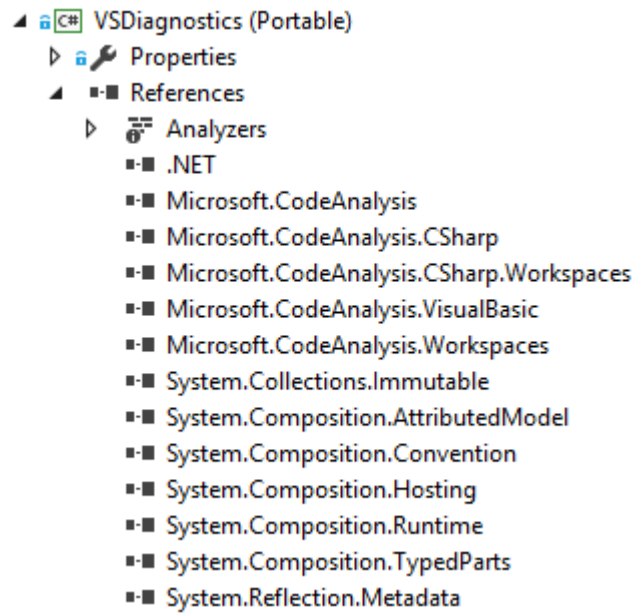[25]npm: Node Package Manager – Javascript-based scripts for the NodeJS platform

Figure 5.13: Analyzers amongst the project dependencies

That being said, in the end this is the same risk as with every other piece of software you download. Caution is advised anytime you download something from the internet and it's important to be aware of where things might be hidden.

Bibliography

Botelho, Levi (2014). *[C#] How does the garbage collector work?* URL: `http://www.levibotelho.com/development/how-does-the-garbage-collector-work/` (visited on 12/06/2015).

Ebeid, El Sayed (2013). *Overview of Microsoft Roslyn – The first compiler-as-service product*. URL: `http://ebeid-soliman.blogspot.be/2013/08/overview-of-microsoft-roslyn-first.html` (visited on 12/29/2015).

Gocke, Andy (2015). *Conversation on Gitter with Andy Gocke on Roslyn's threat models*. URL: `https://gitter.im/dotnet/roslyn` (visited on 12/28/2015).

Gorohovsky, Jura (2014). *ReSharper and Roslyn: Q&A*. URL: `https://blog.jetbrains.com/dotnet/2014/04/10/resharper-and-roslyn-qa/` (visited on 12/03/2015).

Lippert, Eric (2012). *Persistence, Facades and Roslyn's Red-Green Trees*. URL: `http://blogs.msdn.com/b/ericlippert/archive/2012/06/08/persistence-facades-and-roslyn-s-red-green-trees.aspx` (visited on 12/03/2015).

Perry, Mike (2013). *Deterministic Builds Part One: Cyberwar and Global Compromise*. URL: `https://blog.torproject.org/blog/deterministic-builds-part-one-cyberwar-and-global-compromise` (visited on 12/21/2015).

Sadov, Vladimir (2014). *Roslyn's performance*. URL: `https://roslyn.codeplex.com/discussions/541953` (visited on 12/05/2015).

Sedlaczek, Robin (2015). *Inside the .NET compiler platform – performance considerations during syntax analysis*. URL: `http://robinsedlaczek.com/2015/04/29/inside-the-net-compiler-platform-performance-considerations-during-syntax-analysis-speakroslyn/` (visited on 12/08/2015).

Studencki, Michal (2010). *Lexers vs parsers*. URL: `http://stackoverflow.com/a/3614928/1864167` (visited on 12/29/2015).

Todorov, Tsvetomir Y. (2013). *Understanding .NET Garbage Collection*. URL: `http://www.telerik.com/blogs/understanding-net-garbage-collection` (visited on 12/06/2015).

Warren, Matt (2014). *Roslyn code base – performance lessons (part 2)*. URL: `http://mattwarren.org/2014/06/10/roslyn-code-base-performance-lessons-part-2/` (visited on 12/16/2015).

Wischik, Lucian (2015). *Designing C# 7*. URL: `http://www.infoq.com/presentations/` `design-c-sharp-7` (visited on 12/19/2015).

Glossary

API Application Programming Interface. 26, 28

AST Abstract Syntax Tree. 40

bootstrap A self-starting process, a compiler that compiles itself. 7

cache miss A lookup in the cache that yields no result. 44

compile-time constant A variable of which its value is guaranteed known when the compilation process starts. 11

compiler A program that translates code from one language to another. 5, 7–9, 31, 62, 63

concatenation Combining two strings together. 56

data flow The execution path of a piece of code. 6, 9

DoS Denial of Service. 64

emit Writing data to an output stream. 39, 66

FIFO First In, First Out. 43

fork Creating a copy of a repository. 67

Garbage Collector A service which allocates and frees memory. 7, 40, 54, 55, 62, 63

IDE Integrated Development Environment. 9, 35, 54

immutable An object that, once constructed, can no longer be modified. 26–28, 34, 35, 37

intellisense Context-aware hints as you type. 37

invocation Executing a code path by calling a function. 12, 13

LINQ Language Integrated Query. 62, 63

malware Malicious software. 65, 67

managed language A language which offsets memory management from the user to a Garbage Collector. 5, 7, 40, 54

memory-footprint The amount of memory software uses when running. 40

metadata Information about the data or action. 12, 22, 40

MSIL Microsoft Intermediate Language. 5, 31

npm Node Package Manager. 67

open-source Source code that is freely available and which can be modified by the public. 7, 21

parser A service that parses source code into syntactical constructs. 9, 31

PDB Program Database. 32

PR Pull Request. 8

semantic model A model of the meaning attributed to symbols. 5, 11, 26

solution A solution is a set of code files and other resources that are used to build an application. 10, 11, 20, 21

syntax A collection of plain-text tokens that make up source code. 5, 24, 54

syntax tree A data structure which represents the hierarchy between syntactical constructs. 9, 11, 19, 20, 22, 24, 26, 28, 32–35, 37, 40, 41, 43, 47, 51, 53, 54

thread-safe Safety from threads interfering with each other on a shared resource. 34, 45, 61

UML Unified Modeling Language. 22

unit test A self-contained automated test which has no interaction with external systems and tests a software execution path. 7, 11, 20, 67

List of Figures

List of Tables