universität innsbruck

University of Innsbruck
Innsbruck, Austria

Department of Computer Science

Improving code quality using the Roslyn compiler API

Bachelor Thesis

Robin Knoll

robin.knoll@student.uibk.ac.at

Innsbruck, 10 December 2021

Supervisor: assoz. Prof. Dr. Michael Felderer

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Unterschrift

Abstract

Code quality is one of the main concerns in software development to keep large projects maintainable. Code needs to comply with certain standards that are defined specifically in a team. To enforce these rules, static code analysis is often used. It is usually done either by a third party tool or through code review by a team member.
The .NET Compiler Platform, also called *Roslyn*, makes it possible to write powerful code analyzers that are integrated into the C# compiler. Using this technique, rules are applied while writing code and suitable improvements can be suggested and implemented using code generation the moment mistakes happen. The code analyzer can also define rule transgressions as compiler errors, so that these rules have to be followed for the project to build. The code is only reviewed when no more rules are violated and team members can focus on other aspects in their code review.
In this paper, the usefulness of the Roslyn compiler API for improving code quality is evaluated. This is accomplished by the example of creating multiple analyzers for C# projects. The analyzers will cover different aspects such as analyzing naming schemas of classes or improving exception handling.

Contents

List of Figures

1 Introduction

1.1 Motivation

Maintainability is one of the most important aspects of software development. Large code bases need to be sensibly structured and code needs to be written in a certain way so that developers can quickly understand and become acquainted with projects. Code quality can be interpreted as the measure of how readable and maintainable code is [5]. To guarantee a high level of code quality, developers need to be held accountable as it is often easier and quicker to ignore best practices and documentation while writing code.

Static code analysis using tools or code reviews is the main way of assessing the quality of code after it has been written. Third-party tools can be expensive and licensed on a per-developer basis, which often motivates teams to rely on manual code reviews. These can take a lot of time, and changes require the author to refamiliarize and rethink their code a second time, taking up a lot of resources.

Roslyn provides developers with the capability to write their own static code analysis tools. An API is exposed so that information from all stages of the compiler pipeline can be accessed [7]. The analysis is executed while writing code. Developers see their mistakes light up in the same moment a sequence of code is written, making code fixes much faster and easier. Additionally, automatic code fixes can be implemented, which makes the process of fixing faulty code even simpler. Analyzers are not specific to the developer, but can be added to a project simply by adding a NuGet package to its dependencies. The diagnostics are shown to every developer opening the project from there on out.

Roslyn also allows developers to quickly test out a theory they might have about a potential improvement to the quality of their code. As simple analyzers are developed without too much effort, these theories can anecdotally be tried out on existing projects and it quickly becomes visible which parts of a code base could profit from refactoring.

1.2 Solution approach and contribution

This thesis is structured into the following sections. First, a general overview of the structure and purpose of compilers is given. Then, a more accurate look will be taken specifically at the *Roslyn* compiler and its workflow will be explained. Additionally, multiple analyzers will be visited and implemented, their purpose explained and their

development documented.

The ideas for these analyzer have been formed together with developers at *World-Direct*[1]. Many ideas for useful analyzers have come up, but we settled on three immediately useful diagnostic analyzer that will be implemented and looked at in more detail. All code will be publically available at `https://github.com/knollsen/csharp_analyzers`. In each analyzer's project, NuGet packages of the analyzer can be found as well.

[1] `https://world-direct.at`

2 Compilers

Compilers are software with the purpose of translating programs from one language into another. In most cases, the target language is a machine language so that the program can be executed on a specific hardware architecture. However, a compiler might also translate from one higher level language to another according to Aho et al. [6].

The main advantage of compiled languages in contrast to interpreted ones is their higher speed of execution. Interpreted languages on the other hand can provide a higher level of error traceability and diagnostics. Also, they are often shorter than programs with the same purpose written in a compiled language.

As a hybrid between these types of languages, just-in-time compilers have been created. These languages are first compiled to bytecode, which is then at run time translated to machine code by the just-in-time compiler. These languages combine the advantages of both of the previously mentioned types of languages.

As this paper's purpose is to explore the ways in which a compiler can help with code analysis and code quality, we need to establish some terms and explain the main stages or phases of compilers as described in Aho et al.[6]. A representation can be found in figure 2.1.

2.1 Lexical, Syntax and Semantic analyzer

Beginning with source files, the lexical analyzer takes in the textual character stream that is contained in a file. White spaces and comments are removed. The remaining characters are grouped into lexemes. Lexemes are sequences of characters that match a pattern in the source language. The lexemes are output in the form of tokens. These tokens are tuples containing the name of the lexeme and a reference to an entry in the symbol table. This way, variable references can be recognized. When a new variable is introduced, its name is saved in the symbol table and a future reference to that variable will point to the same entry.

Another purpose of the lexical analyzer is to associate compile time errors with their corresponding line number by counting the number of new-line characters that appear. It may also take over the role of a macro-processor, if the source language supports such constructs.

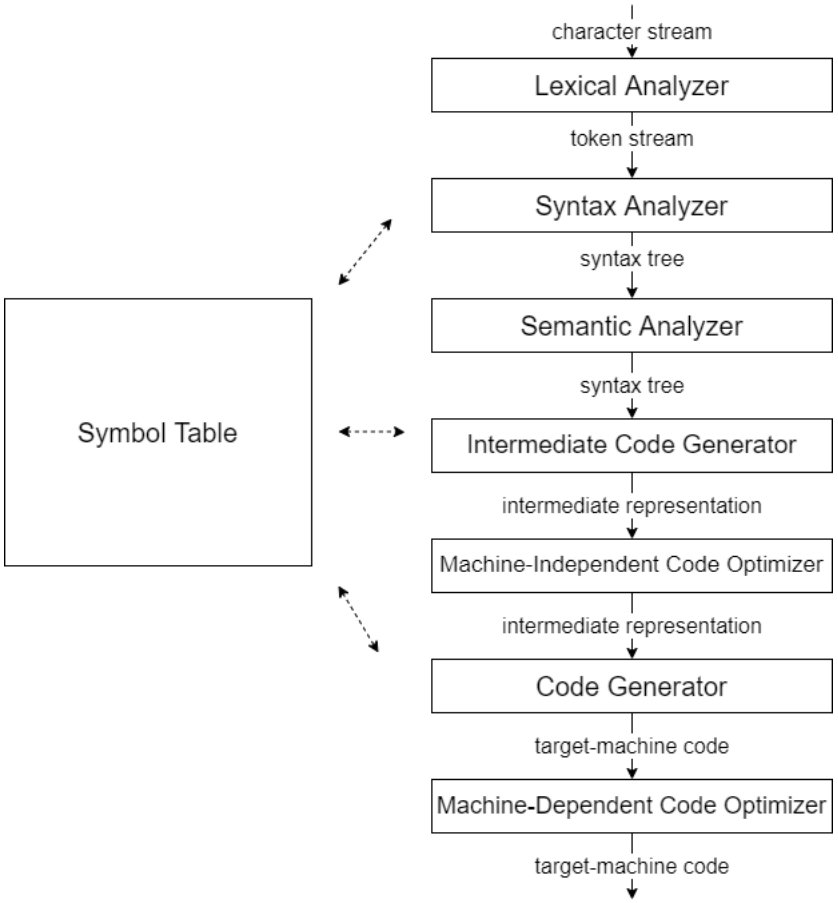

Figure 2.1: The phases of a compiler

The syntax analyzer uses the stream of tokens to create a syntax tree corresponding to the defined syntax of the source language. A language is made up of complex rules describing the structure of valid programs. These rules can be defined as context-free grammars or Backus-Naur-Form notation. The syntax analyzer needs to be able to report syntax errors in a meaningful way. Commonly occurring errors might also be recovered from automatically.

The semantic analyzer takes the syntax tree created by the syntax analyzer and checks it for semantic consistency. An example of this would be to check for correct and valid typing. It also adds coercions to allow for certain operations. For example, an arithmetic operation on a floating point number and an integer first requires the integer to be coerced to a float.

2.2 Intermediate code generator

The intermediate code generator takes the checked syntax tree and creates an intermediate representation. This intermediate representation might look similar to assembly code. An example is three-address code, where every operation is transformed to the shape of $x = y\ op\ z$. This also includes static checking, syntactic checks that could not be done in the parser. For instance, in C, a *break* must be contained either in a *while*, *for* or *switch* statement.

This intermediate code is improved by the machine-independent code optimizer. It only uses optimizations that work everywhere. To fulfill this purpose, mainly data-flow analysis is used. Unnecessary instructions are removed, inefficient sequences replaced by faster operations. An example is removing the at run time float conversion of a constant integer, as can be seen below.

$$x = converttofloat(5) * 3.2 \longrightarrow x = 5.0 * 3.2$$

All described stages up to this point are also called the front-end of the compiler. These stages are specific to the source language and produce intermediate code. Now begins the back-end of the compiler, which takes the intermediate code and translates it into the target language. The front-end and back-end can be swapped out for another if it uses the same intermediate language, which can be very useful for portability.

2.3 Code generator and architecture specific optimizer

The code generator creates target language code from the intermediate language. This is a complex step, as there are multiple translations possible for each code sequence. Because this is an undecidable problem for a complete program, heuristics need to be employed.

After this, optimizations on an architecture specific level can be executed. This might include vectorization or other forms of parallelism.

3 Roslyn: The .NET Compiler Platform

3.1 Structure

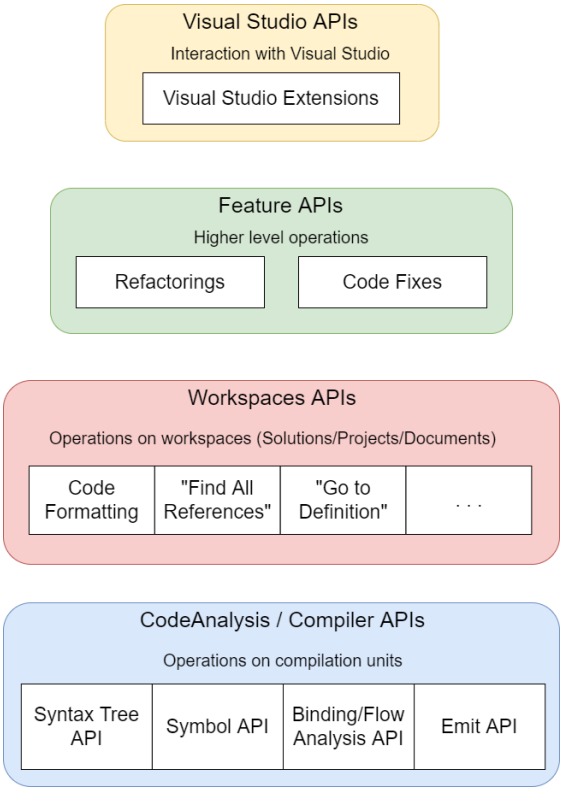

Figure 3.1: Layers of *Roslyn*'s APIs

A compiler's primary function is to compile source code into runtime-specific binaries containing machine code. The .NET compiler platform, code named *Roslyn*, provides developers with some additional functionality. *Roslyn* is a compiler for .NET programming languages, i.e. C# and Visual Basic [2]. It gives its users the ability to access the internal structure of the compilation process and integrate their own tools [7]. Functionality for analysis or code generation can easily be added using an API. To be able to access this information, developers need to know about the specific layers of the *Roslyn* compiler.

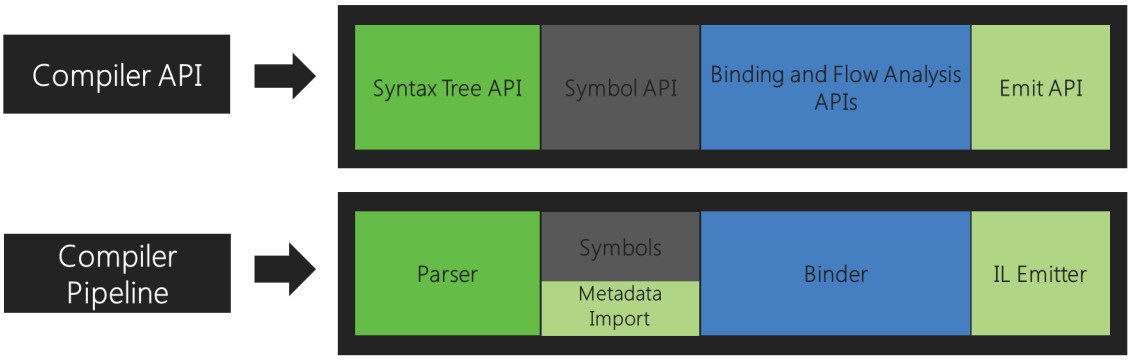

Figure 3.2: *Roslyn*'s *Compiler* layer[3]

There are 4 layers to *Roslyn*'s APIs, as visible in fig. 3.1. Each layer provides public and mostly documented APIs to interact with it. The base layer is the *CodeAnalysis* or *Compiler* layer. With it, users can gain access to data and compilation units of the different phases of the compilation process like syntax trees, symbol tables, semantic information and intermediate language byte codes. An abstracted compiler pipeline with the corresponding APIs can be seen in fig. 3.2.

The *Workspaces* layer unites related source files to projects and solutions. With it, operations on multiple documents can be completed. The *Features* layer provides an API for integrating IDE features such as finding possible references, code fixes, code completion or access to *IntelliSense*. Finally, the so called *Visual Studio* layer is exclusive to the IDE Visual Studio and can be used to interact with its interface.

The APIs giving access to these layers can be used to expand and round off the development experience of teams. Developers can write diagnostic analyzers and code fixes specifically useful to their team, making *Roslyn* a powerful tool in software development.

3.2 Diagnostic analyzers

Analyzers are mostly built on top of the *CodeAnalysis* layer of *Roslyn*. Developers can register actions for specific building blocks of the compilation. Examples of such building blocks would be a symbol, a code block, a syntax node or a whole compilation. Whenever *Roslyn* compiles one of these units, the registered action is called and can report a diagnostic.

The action gets the context of the code unit of interest as an argument. The information this context includes depends on the chosen unit of code and will be discussed in more detail later.

Analyzers can either be stateless or stateful, depending on whether they need to re-

tain information about the code across multiple calls and actions. Stateless analyzers do not maintain any state across their calls. This keeps writing them simpler, but more complex analysis is not possible. Stateful analyzers define and access a mutable state that is initialized at a registered point in the compilation process. They are more difficult to write and need to be correctly memory-managed, but can make more powerful analysis possible.

In case developers are sure that they do not need a certain diagnostic or the analyzer is wrong about a certain code sequence, diagnostics can be manually disabled on a per-section, per-file and per-project scope. However, the developer needs to explicitly state a reason for the deactivation so that other team members can review the action.

3.3 Development with Roslyn

3.3.1 Analyzer Solutions

Analyzers can easily be created by using the template provided by Visual Studio. The generated solution will include multiple projects. The main project with the same name as the solution will contain the diagnostic analyzer. Additionally, there will be a test project containing unit tests, a package project containing the configuration on how to pack the analyzer for deployment, a code fix project for optionally implementing a fix for the analyzed diagnostic and finally a VSIX project. The VSIX project is the default startup project and contains the configuration for a Visual Studio instance for trying out the analyzer.

3.3.2 Development of Diagnostic Analyzers

Fig. 3.3 shows the structure of a diagnostic analyzer. Classes containing a diagnostic analyzer need to be attributed with *DiagnosticAnalyzer*. The attribute contains as a parameter the language or languages that the analyzer is developed for. The class additionally inherits from the abstract class *DiagnosticAnalyzer*. The property *ImmutableArray<DiagnosticDescriptor>SupportedDiagnostics* and the method *Initialize(AnalysisContext context)* need to be implemented.

The *SupportedDiagnostics* array contains *DiagnosticsDescriptors* of the diagnostics that are implemented in this class. A *DiagnosticsDescriptor* includes the identification string, title, message, severity and some additional information of a diagnostic.

In the *Initialize* method, a context of type *AnalysisContext* is given. Using this argument, some configuration can be done, like enabling concurrent execution of analyzers and analysis for generated code. More importantly, all analysis actions need to be registered here. The context provides multiple methods to register for different layers of the compilation process. Some examples include *RegisterSyntaxTreeAction*, *RegisterSyntaxNodeAction* and *RegisterSymbolAction*.

```
[DiagnosticAnalyzer(LanguageNames.CSharp)]
public class TemplateAnalyzer : DiagnosticAnalyzer
{
    public const string DiagnosticId = "TemplateAnalyzer";

    private static readonly string Title = "This is the title of the diagnostic.";
    private static readonly string MessageFormat = "This is the message a diagnostic would show.";

    private const string Category = "Naming";

    private static readonly DiagnosticDescriptor Rule = new DiagnosticDescriptor(DiagnosticId,
        Title, MessageFormat, Category, DiagnosticSeverity.Warning, isEnabledByDefault: true);

    public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics => ImmutableArray.Create(item: Rule);

    public override void Initialize(AnalysisContext context)
    {
        context.ConfigureGeneratedCodeAnalysis(GeneratedCodeAnalysisFlags.None);
        context.EnableConcurrentExecution();

        context.RegisterSymbolAction(AnalyzeSymbol, params symbolKinds: SymbolKind.NamedType);
    }

    private static void AnalyzeSymbol(SymbolAnalysisContext context)
    {
        context.ReportDiagnostic(Diagnostic.Create(descriptor: Rule, location: context.Symbol.Locations.First()));
    }
}
```

Figure 3.3: Basic structure of a Diagnostic Analyzer

These methods are used with an action as a parameter that will be called when a section of code corresponding to the layer described in the method name has been compiled [7]. All of the mentioned registration methods also have counterparts for stateful analyzers. Additionally, for syntax node and symbol actions, a *SyntaxKind* or *SymbolKind* can be passed to only register for these types of nodes or symbols. These include syntax nodes such as invocation expressions or symbols such as a method. Examples on how to correctly use these methods can be found in the following chapters, where some analyzers are described in more detail.

Analyzer development can become relatively complex and it is important to know helpful tools and resources. Tools like the *RoslynQuoter*[1] or the *SyntaxViewer* integrated into Visual Studio can be very useful to get an understanding of what building blocks syntax trees are made of. The *RoslynQuoter* is mainly useful when developing code fixes, but can also help for analyzer development.

A challenging part of development can be the usage of more specific parts and features of the *Roslyn* APIs as the documentation can at times be sparse. One such challenge may be the management of dependencies, in cases where the analyzer needs to use another

[1] `https://roslynquoter.azurewebsites.net/`

```
[ExportCodeFixProvider(LanguageNames.CSharp, Name = nameof(TemplateCodeFixProvider)), Shared]
public class TemplateCodeFixProvider : CodeFixProvider
{
    public sealed override ImmutableArray<string> FixableDiagnosticIds
        => ImmutableArray.Create( item: TemplateAnalyzer.DiagnosticId);

    public sealed override FixAllProvider GetFixAllProvider()
    {
        return WellKnownFixAllProviders.BatchFixer;
    }

    public sealed override async Task RegisterCodeFixesAsync(CodeFixContext context)
    {
        var diagnostic = context.Diagnostics.First();

        context.RegisterCodeFix(
            CodeAction.Create(
                title: CodeFixResources.CodeFixTitle,
                createChangedDocument: c :CancellationToken => MakeChangesToDocumentAsync(context.Document, c),
                equivalenceKey: nameof(CodeFixResources.CodeFixTitle)),
            diagnostic);
    }

    private async Task<Document> MakeChangesToDocumentAsync(Document document, CancellationToken cancellationToken)
    {
        return document;
    }
}
```

Figure 3.4: Basic structure of a Code Fix

package to work. Another may be the use of a configuration file in the target project. A more elaborate explanation and the solutions to both of these challenges can be found in chapter 5. A useful technique for transmitting custom information about a diagnostic from the diagnostic analyzer to a potential code fix provider is used in chapter 6.

3.3.3 Development of Code Fixes

Optionally, an analyzer solution can contain a project implementing a code fix. The basic structure of such a code fix is presented in fig. 3.4. The class containing the fix needs to inherit from the abstract class *CodeFixProvider* and have the attribute *ExportCodeFixProvider* which includes the targeted language as an argument. The code fix class is required to override the property *ImmutableArray<string>FixableDiagnosticsIds* containing the identification strings of the diagnostics this class can fix.

Overriding the method *RegisterCodeFixesAsync(CodeFixContext context)* is mandatory as well. In this method, a check should be performed to see if the diagnostic given in the context can be improved by this code fix. If it can, a code action is registered on the context so that the user can see the code fix in the list of possible actions on the diagnostic.

The source code is represented in the immutable class *Document*. If the user calls the

code action, a new *Document* needs to be generated where the problem is fixed. There are a few classes provided to help with this, such as the *SyntaxFactory* and *SyntaxGenerator*. Using these classes, a new syntax tree can be generated, with nodes in the tree added, changed or removed. The new *Document* is then returned and the changes will be applied in the user's solution.

The previously mentioned tools *SyntaxViewer* and *RoslynQuoter* are of great use for code fix development as well. The *RoslynQuoter*[2] takes a code snippet from its user and shows a collection of function calls using the *Roslyn* API's *SyntaxFactory* utility class with which the given snippet can be generated.

3.3.4 Unit Testing

Most *Roslyn* analyzers will contain a unit test project for using test-driven development or simply for testing the implementation. Unit tests can be implemented using predefined utility classes for assertions that use *Microsoft*'s *MSTest* under the hood. Using these classes, diagnostic analyzers can be tested by defining specific diagnostics that are expected to show up when compiling certain code snippets. Code fixes are tested by defining one snippet where the associated diagnostic is present and another illustrating what the code should look like after the code fix. After applying the code fix, both snippets need to be the same for the test to succeed.

One simple technique to get suitable and valid code snippets for unit testing is to open a second project and write code that the analyzer would be applied to. This way, there is confidence that the code is valid and can be used in the unit test. Sadly, in these unit test snippets imports from .NET Core libraries are not supported. Because of this, any interfaces, attributes or abstract classes from external libraries need to be simulated in the unit test.

Test-driven development techniques prove very useful when developing diagnostic analyzers and code fixes. Being able to use a debugger and set break points while executing different test scenarios is very important as otherwise there is no other easy way to quickly debug analyzer code.

[2] `https://roslynquoter.azurewebsites.net/`

4 Authorize Analyzer

4.1 Functionality

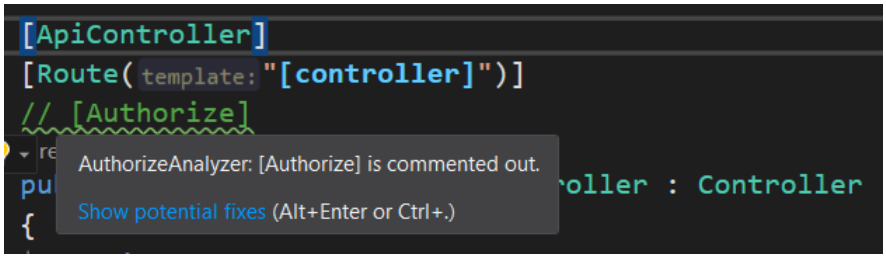

Figure 4.1: The Authorize Analyzer showing a warning

This analyzer was the first project completed for this paper and served as an introduction to the process of writing code analyzers for *Roslyn*. It has a very simple purpose in finding [Authorize] attributes in C# code and reporting a warning if the attribute is commented out. A visual representation of the analyzer in action can be seen in fig. 4.1.

In C#, attributes can add metadata to classes, methods or entire projects [1]. They work in a similar way as annotations in Java. In .NET projects, web controllers provide HTTP endpoints [4]. The controller can be secured with authentication by adding the C# attribute [Authorize]. The attribute can either be placed on the class to secure all endpoints defined in the controller or on a specific endpoint. On endpoints and controllers with the attribute, the authentication scheme that is configured as default will be used to authenticate the controller or endpoint. In development, this attribute is often commented out so that the developer does not need to authenticate the application each time they want to use an endpoint. However, the attribute is sometimes still left commented out in release builds, which can pose a problematic security risk.

To prevent this from happening, this analyzer will report a warning to show the developer that a commented out [Authorize] attribute is most likely a mistake.

4.2 Implementation

The complete implementation of this analyzer can be found on `https://github.com/knollsen/csharp_analyzers`. The core of the implementation is shown in figure 4.2.

```
1  public override void Initialize(AnalysisContext context)
2  {
3      context.ConfigureGeneratedCodeAnalysis(GeneratedCodeAnalysisFlags.Analyze)
4      context.EnableConcurrentExecution();
5
6      context.RegisterSyntaxTreeAction(HandleSyntaxTree);
7  }
8
9  private void HandleSyntaxTree(SyntaxTreeAnalysisContext context)
10 {
11     SyntaxNode root =
           context.Tree.GetCompilationUnitRoot(context.CancellationToken);
12
13     var commentNodes = root.DescendantTrivia().Where(x =>
           x.IsKind(SyntaxKind.SingleLineCommentTrivia)).ToList();
14
15     if (!commentNodes.Any())
16     {
17         return;
18     }
19
20     foreach (var node in commentNodes)
21     {
22         var commentText = node.ToString().TrimStart('/');
23
24         if (!commentText.TrimStart().StartsWith("[Authorize]"))
25         {
26             continue;
27         }
28
29         var diagnostic = Diagnostic.Create(Rule,
               node.GetLocation());
30         context.ReportDiagnostic(diagnostic);
31     }
32 }
```

Figure 4.2: Core implementation of the Authorize analyzer

In the overridden *Initialize* method, we register an action for whenever the parsing of a code document is completed. When this happens, the *HandleSyntaxTree* method (line 9) is called.

In the *HandleSyntaxTree* method, the root of the compilation unit is retrieved. It is of type *CompilationUnitSyntax*. From this root node, all descendant trivia is fetched. The trivia consists of all whitespaces and comments in the document. The trivia is then filtered so only single line comments are left behind (line 13).

If there are any single line comments, we remove slash characters and whitespaces. Then, each of them is scanned for the [Authorize] attribute. If the attribute is present, we report a diagnostic (line 29). The configuration for the diagnostic including id, name and severity are saved in the property *Rule*. To report the diagnostic, we also have to pass the location where the diagnostic is supposed to show up. The location consists of line and column number and can be easily retrieved from the trivia node.

4.3 Code fix

The goal of this code fix is to replace the commented out [Authorize] attribute with a valid one. Intuitively, this seems like a quite simple change, but the code fix turned out more complex than expected. This is on the one hand because there are a lot of checks to be performed so that the correct action is taken and on the other hand because we want to change the type of the node. A comment belongs to trivia, while an attribute is a syntax node.

In the code fix provider's *RegisterCodeFixesAsync* method (fig. 4.3), we try to fetch the *SyntaxNode* that the comment with [Authorize] in it is assigned to. We do this because trivia is assigned to the following syntax node on parsing. If the node is not found, we return as a safety measure. Safe guards such as this have to be implemented as the compiler will call this method for every diagnostic in our code fix provider's *FixableDiagnosticIds* property and a third party analyzer may be using the same identification string as our analyzer. Should this happen, our code fix provider will be called for code sections that it can not fix and might behave unexpectedly.

The *GetCommentAsync* method can be seen in fig. 4.4. We obtain the diagnostic we are trying to fix as a parameter, which simplifies the process of looking for the correct syntax node. First, we try to find the trivia node, our comment, in the location that the diagnostic was shown in. Then, we try to find the syntax node the compiler assigned our comment as trivia to. We only want to find nodes that are class declarations or method declarations (line 19) as these are the only types of syntax nodes that can be attributes with [Authorize]. The trivia might also have been assigned to a child token of our syntax node, so we also have to check these for the comment (line 27). If either the comment or the assigned syntax node can not be found, we return null so that no code

```
1  public override async Task RegisterCodeFixesAsync(CodeFixContext
       context)
2  {
3      var diagnostic = context.Diagnostics.First();
4
5      var nodeAndTrivia = await
           this.GetCommentAsync(context.Document, diagnostic,
           context.CancellationToken);
6
7      if (nodeAndTrivia == null)
8      {
9          return;
10     }
11
12     // Register a code action that will invoke the fix.
13     context.RegisterCodeFix(
14         CodeAction.Create(
15             title: CodeFixResources.CodeFixTitle,
16             createChangedDocument: c =>
                   UncommentAsync(context.Document,
                   nodeAndTrivia.Value.Item1,
                   nodeAndTrivia.Value.Item2, c),
17             equivalenceKey:
                   nameof(CodeFixResources.CodeFixTitle)),
18         diagnostic);
19 }
```

Figure 4.3: Registration of the Authorize code fix

```
1  private async Task<(SyntaxNode, SyntaxTrivia)?>
       GetCommentAsync(Document document, Diagnostic diagnostic,
       CancellationToken ct)
2  {
3      var tree = await
           document.GetSyntaxTreeAsync(ct).ConfigureAwait(false);
4
5      var trivia = (await tree.GetRootAsync(ct))
6          .DescendantTrivia()
7          .SingleOrDefault(t =>
               t.IsKind(SyntaxKind.SingleLineCommentTrivia) &&
               t.GetLocation() == diagnostic.Location);
8
9      if (trivia == new SyntaxTrivia())
10     {
11         return null;
12     }
13
14     var syntaxNodes = (await
           tree.GetRootAsync(ct)).DescendantNodesAndSelf();
15
16     var node = syntaxNodes
17         .SingleOrDefault(n =>
18         {
19             if (!(n is ClassDeclarationSyntax || n is
                   MethodDeclarationSyntax))
20             {
21                 return false;
22             }
23             if (n.GetLeadingTrivia().Contains(trivia))
24             {
25                 return true;
26             }
27             if (n.ChildTokens().Any(x =>
                   x.LeadingTrivia.Contains(trivia)))
28             {
29                 return true;
30             }
31             return false;
32         });
33
34     if (node == null)
35     {
36         return null;
37     }
38
39     return (node, trivia);
40 }
```

Figure 4.4: Fetching of the correct node in the Authorize code fix

fix will mistakenly be shown. Otherwise, we return the syntax and trivia node as a tuple.

Finally, the method *UncommentAsync* seen in fig. 4.5 makes the necessary changes to the code. In case the comment is assigned directly to our syntax node (line 6), we create a new *SyntaxNode* containing the [Authorize] attribute and remove the comment from the leading trivia of our syntax node. If the comment is assigned to a child token of the syntax node, we retrieve the correct token (line 14). Then we create a new token from it without the comment inside its leading trivia. Then, we replace the old token with our newly generated one. We then create a new attribute with the *SyntaxFactory* utility class. Using the *SyntaxGenerator*, we insert the new attribute before our new syntax node. Finally, having generated a new syntax node that we want to swap with our old one, we use the *SyntaxGenerator* to generate a new syntax tree with our desired node replaced. Then, we return a new document with the new syntax tree.

4.4 Testing

The unit tests for this analyzer are there to test edge cases and make sure the analyzer shows diagnostics in the correct locations. Additionally, the code fix needs to be tested so that the correct comment is removed and the attribute is inserted correctly. For this purpose, code snippets can be defined as seen in fig. 4.6. In the unit test described here, we have a class definition with a commented out [Authorize] attribute that we expect the diagnostic analyzer to generate a warning for. Our second code snippet shows the same class declaration after we apply the code fix. For our test assertion, we first define the diagnostic expected to show including its location (line 31). Then, we use our test utility classes to assert that the diagnostic is shown, and afterwards, if the code fix is applied, that our initial code snippet now looks like our fixed class declaration. If there are any differences whatsoever, the unit test will fail.

The other unit tests for this analyzer try out different cases of interest. They can be found in the analyzer repository as well[1]. Additional attributes are placed around the comment containing the [Authorize] attribute to make sure the comment is still correctly replaced. The code fix is used on a commented out attribute placed on a method to ensure that it works for methods as well. Finally, a combination of the above mentioned use cases is tested to try out the code fix on multiple instances of the diagnostic, placing the commented out attribute on both the web controller itself as well as on a method inside it.

[1] `https://github.com/knollsen/csharp_analyzers/blob/main/AuthorizeAnalyzer/AuthorizeAnalyzer/AuthorizeAnalyzer.Test/AuthorizeAnalyzerUnitTests.cs`

```
1  private async Task<Document> UncommentAsync(Document document,
       SyntaxNode node, SyntaxTrivia trivia, CancellationToken ct)
2  {
3      var nodeLeadingTrivia = node.GetLeadingTrivia();
4
5      SyntaxNode nodeWithCommentRemoved;
6      if (nodeLeadingTrivia.Contains(trivia))
7      {
8          var newLeadingTrivia =
               nodeLeadingTrivia.Remove(trivia).NormalizeWhitespace();
9
10         nodeWithCommentRemoved =
               node.WithLeadingTrivia(newLeadingTrivia);
11     }
12     else
13     {
14         var token = node.ChildTokens().SingleOrDefault(x =>
               x.LeadingTrivia.Contains(trivia));
15
16         var tokenLeadingTrivia = token.LeadingTrivia;
17         var tokenLeadingTriviaWithoutComment =
               tokenLeadingTrivia.Remove(trivia);
18
19         var newToken =
               token.WithLeadingTrivia(tokenLeadingTriviaWithoutComment);
20
21         nodeWithCommentRemoved = node.ReplaceToken(token,
               newToken);
22     }
23
24     var gen = SyntaxGenerator.GetGenerator(document);
25
26     var authNode =
           SyntaxFactory.Attribute(SyntaxFactory.IdentifierName("Authorize"));
27
28     var finalNode = gen.InsertAttributes(nodeWithCommentRemoved,
           0, authNode);
29
30     var oldRoot = await
           document.GetSyntaxRootAsync(ct).ConfigureAwait(false);
31
32     var newRoot = gen.ReplaceNode(oldRoot, node, finalNode);
33
34     return document.WithSyntaxRoot(newRoot);
35 }
```

Figure 4.5: Swapping the comment for an attribute in the Authorize code fix

```
[TestMethod]
public async Task CodeFixUncommentsAuthorizeAttribute()
{
    var test = @"
using System;

namespace ConsoleApplication1
{
    // this is a comment
    // [Authorize]
    class Test
    {
    }

    class AuthorizeAttribute : Attribute {}
}";
            var fix = @"
using System;

namespace ConsoleApplication1
{
    // this is a comment
    [Authorize]
    class Test
    {
    }

    class AuthorizeAttribute : Attribute {}
}";

    var expected =
       VerifyCS.Diagnostic(AuthorizeAnalyzer.DiagnosticId)
                .WithSpan(7, 5, 7, 19);
    await VerifyCS.VerifyCodeFixAsync(test, expected, fix);
}
```

Figure 4.6: Example of a unit test for the Authorize code fix

5 Naming Analyzer

5.1 Functionality

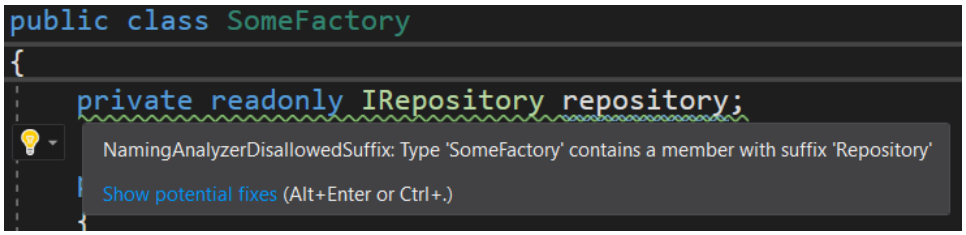

Figure 5.1: Example of a class containing a member with a disallowed suffix

This analyzer's purpose is to make sure that design patterns are used correctly. Developers who are not comfortable with certain patterns stand to benefit from instant feedback when they use these pattern incorrectly. By putting constraints on the fields and properties of classes that are part of a specific pattern, this is made possible. For example, when using the repository and factory pattern, it is not allowed to have a repository as a member of a factory. An illustration of what the diagnostic analyzer would show in this case can be seen in fig. 5.1. The analyzer does not use the name of the field or property for this check, but the name of the type. This way, the issue is not simply resolved by changing the name of the variable.

Additionally, it has the functionality to ban certain terms in class names. This may be useful for some teams that do not want to use class names with overloaded meaning like *Manager* or *Helper*. In these cases, the analyzer will even show the diagnostic as a compiler error, like in fig. 5.2.

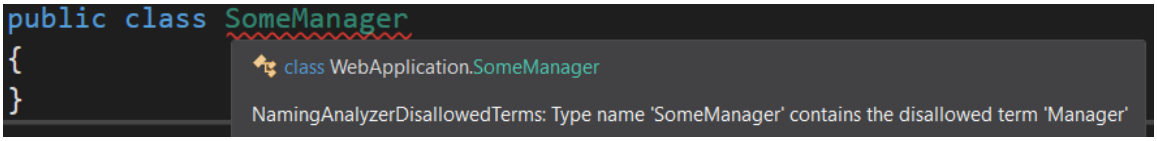

Figure 5.2: Example of a class name containing a disallowed term

For this analyzer, the user has to be able to do some amount of configuration. Not every team using it will have the same naming conventions and repacking the analyzer with a new configuration is tedious and impractical. For this reason, the analyzer will look for a file called *namingConfig.json* in the solution it is used in. This JSON file is

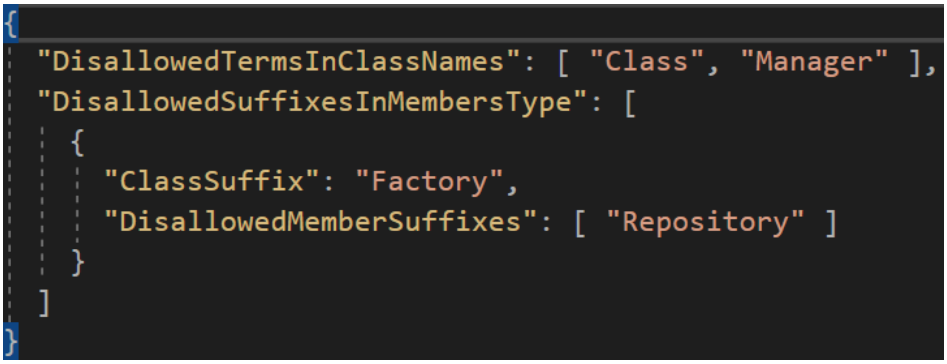

Figure 5.3: Configuration file for the Naming analyzer

structured in a way so that it is easily extendable. An example of a valid configuration file is visualized in fig. 5.3.

The file contains an array of terms that are not allowed to be used in class names. It can be found with the key *DisallowedTermsInClassNames*. The key *DisallowedSuffixesInMembersType* contains an array of objects, where each one stands for a design pattern. The object contains a *ClassSuffix*, the suffix of the classes this rule applies to, and an array of suffixes that are not allowed to appear in its members' types. In the example file shown, this means that classes ending with *Factory* cannot contain any fields or properties of types ending with *Repository*.

Because a fix for one of these diagnostics requires the developer to rethink the decisions they made when naming the affected classes or implementing the design pattern, there is no code fix provided.

5.2 Implementation

The analyzer is registered as a symbol action specifically for named types, meaning that it is only called when a named type like a class or an interface is compiled.

The first challenge of implementing this diagnostic analyzer is how to read and manage the configuration file. The solution is shown in fig. 5.4. There are two ways in which we can get the content of the file. *Roslyn* provides an API to get any files designated as *C# Analyzer Additional files* in Visual Studio. The access is simple, as shown in line 8, and with one line of code we can get the file's content.

However, the user of the analyzer has to specifically designate the file as such which might lead to some confusion. An instruction on how to correctly configure the file can

```
1  private void AnalyzeNamedType(SymbolAnalysisContext context)
2  {
3      // read config
4      if (this.Config == null)
5      {
6          // can either come from the context's additional files
               (preferable)
7          string configContent;
8          if (context.Options.AdditionalFiles.Any(file =>
               Path.GetFileName(file.Path).Equals("namingConfig.json")))
9          {
10             configContent =
                   context.Options.AdditionalFiles.First(file =>
11                  Path.GetFileName(file.Path).Equals("namingConfig.json"))
12                  .GetText().ToString();
13         }
14         // or directly from the file system
15         else
16         {
17             var currentFiles = Directory.GetFiles(".");
18
19             var configFile = currentFiles.SingleOrDefault(x =>
                   x.EndsWith("namingConfig.json"));
20
21             // report a warning and return if we can't find the
                   file
22             if (configFile == null)
23             {
24                 context.ReportDiagnostic(Diagnostic.Create(MissingFileRule,
                       Location.None));
25                 return;
26             }
27
28             configContent = File.ReadAllText(configFile);
29         }
30         ...
```

Figure 5.4: Reading the configuration file for the Naming analyzer

be found on the analyzer's GitHub page[1]. Additionally, in unit tests there is no access to the *Additional Files* API.

Because of these problems, a second way of reading the content of the configuration file is implemented. In the second implementation from line 17 to line 28 in fig. 5.4, the configuration is taken directly from the file system. If we cannot find the file, a diagnostic is reported to the user to suggest adding the configuration. To make this work with unit tests, we simply have to include a *namingConfig.json* file in the unit test project.

```
1     ...
2
3     NamingConfig config = null;
4     try
5     {
6         config =
              JsonConvert.DeserializeObject<NamingConfig>(configContent);
7     }
8     catch (JsonReaderException)
9     {
10        // ignore this exception, as it is handled below because
              config == null
11    }
12
13
14    // if the config file does not contain the correct structure,
          return as well
15    if (config?.DisallowedSuffixesInMembersType == null ||
          config?.DisallowedTermsInClassNames == null)
16    {
17        context.ReportDiagnostic(Diagnostic.Create(MissingFileRule,
              Location.None));
18        return;
19    }
20
21    this.Config = config;
22 }
23 ...
```

Figure 5.5: Parsing the configuration file for the Naming analyzer

Independent of which method to read the file is used, once its content is retrieved, it is parsed using the *Newtonsoft.Json*[2] library's *JsonConvert* class, seen in line 6 of fig. 5.5. This is a well supported and extremely popular library for .NET projects that can

[1] https://github.com/knollsen/csharp_analyzers/blob/main/NamingAnalyzer/README.md
[2] https://www.newtonsoft.com/json

```
1  ...
2
3  var namedType = context.Symbol;
4
5  // check for disallowed terms in class names
6  foreach (var disallowedTerm in
       this.Config.DisallowedTermsInClassNames)
7  {
8      if (namedType.Name.Contains(disallowedTerm))
9      {
10         context.ReportDiagnostic(Diagnostic.Create(DisallowedTermsRule,
               namedType.Locations.First(), namedType.Name,
               disallowedTerm));
11
12         break;
13     }
14 }
15
16 ...
```

Figure 5.6: Checking type names for disallowed terms in the Naming analyzer

perform actions related to the JSON format in a performant manner. We try to parse the JSON to a class containing structurally matching properties. In case the parsing does not work, we report a diagnostic as well to show that the format of the JSON needs to be changed. If we are able to parse a valid configuration, we save it in a property of the analyzer so that the configuration does not have to read each time the method is called. However, because this is a stateless analyzer, it will still be read more than once.

This provides us with a second challenge on how to include the *Newtonsoft.Json* library in a package with our analyzer. Microsoft sadly does not provide any documentation on this problem. However, the *Roslyn* community is able to provide a well explained solution[3]. It works by reconfigurating the way the analyzer is packed to a NuGet package. First, all NuGet dependencies of the project are scanned, then all .NET Standard packages are removed as they have to be present in the user's project anyway. Finally, the *.dll* files of all remaining dependencies are copied inside the NuGet package of the analyzer. Using this method, the analyzer will only use the included binary files and is not reliant on the user installing additional NuGets which might lead to dependency problems if they want to use a different version of the same library.

After successfully parsing the configuration, we can now perform the required checks for the type the analyzer has been called for. First, we check its name for disallowed

[3] `https://www.meziantou.net/packaging-a-roslyn-analyzer-with-nuget-dependencies.htm`, Accessed: 04.12.2021

terms. This can be done simply by taking the name of the type and iterating over all terms specified in our configuration. If the name contains a disallowed term, we report a diagnostic informing the developer of this transgression, as seen in fig. 5.6.

Fig. 5.7 shows the implementation of checking the members of a class for disallowed types. For this purpose, we need to retrieve the corresponding syntax node as currently we only have the symbol of the class. *Roslyn* allows us to do this in a simple way by giving us its declaration (line 3). We make sure that the syntax node is of the correct type *ClassDeclarationSyntax* and check if the suffix of our class in included in the analyzer configuration (line 11). If that is the case, we retrieve the disallowed member suffixes and the members of the class. Then, we iterate over each member. We are only interested in fields and properties, but methods, constructors and other definitions are also part of a class's members. Because of this, we perform a check on the type of the member (lines 20 and 33). In case we are dealing with a field or property, we search for its type name in the configuration to see if there are any matching suffixes. If we find any, we report a diagnostic (lines 27 and 39).

5.3 Testing

Unit tests for this diagnostic analyzer like the one included in fig. 5.8 are not as extensive as those for other analyzers because there is no code fix. Nevertheless, the analyzer is still tested for correctly reading and parsing the configuration file and if it works on abstract classes as well. The test in fig. 5.8 makes sure that the diagnostic is shown for both fields and properties. The configuration file for this test disallows *Factories* to contain a *Repository*. For the test to succeed, a diagnostic has to be shown on both members.

Of course, the diagnostic shown for invalid terms in class names is tested as well. For example, we test if the disallowed term can be at any position in the class name.

To test the reading and parsing of configuration files, there are multiple test projects for this analyzer. The configuration file has to be unique per project, so additionally there is one unit test project where the configuration is missing and one where the configuration is in an invalid format.

```
...

var syntaxNode =
    namedType.DeclaringSyntaxReferences.First().GetSyntax();

// for class declarations, check for members with disallowed
    suffixes
if (this.Config.DisallowedSuffixesInMembersType != null &&
    syntaxNode is ClassDeclarationSyntax classDeclaration)
{
    var name = classDeclaration.Identifier.Text;

    // check if there is a configuration for this class
    if (this.Config.DisallowedSuffixesInMembersType.Any(x =>
        name.EndsWith(x.ClassSuffix)))
    {
        var config =
            this.Config.DisallowedSuffixesInMembersType.First(x =>
            name.EndsWith(x.ClassSuffix));

        var members = classDeclaration.Members;

        // check if a member is of a type with a disallowed suffix
        foreach (var member in members)
        {
            if (member is PropertyDeclarationSyntax
                propertyDeclaration)
            {
                var disallowedSuffix =
                    config.DisallowedMemberSuffixes.FirstOrDefault(x
                        =>
                        propertyDeclaration.Type.ToString().EndsWith(x));
                if (disallowedSuffix != null)
                {
                    context.ReportDiagnostic(
                        Diagnostic.Create(DisallowedSuffixRule,
                            member.GetLocation(), namedType.Name,
                                disallowedSuffix));
                }
            }

            if (member is FieldDeclarationSyntax fieldDeclaration)
            {
                var disallowedSuffix =
                    config.DisallowedMemberSuffixes.FirstOrDefault(x
                    =>
                    fieldDeclaration.Declaration.Type.ToString().EndsWith(x));
                if (disallowedSuffix != null)
                {
                    context.ReportDiagnostic(
                        Diagnostic.Create(DisallowedSuffixRule,
                            member.GetLocation(), namedType.Name,
                                disallowedSuffix));
                }
            }
        }
    }
}
```


Figure 5.7: Checking type members in the Naming analyzer

```
1  [TestMethod]
2  public async Task DisallowedMemberSuffixesShowDiagnostic()
3  {
4      var test = @"
5  using System;
6
7  namespace ConsoleApplication1
8  {
9      class TestFactory
10     {
11         private IRepository repository2 { get; }
12         private readonly IRepository repository;
13     }
14
15     interface IRepository {}
16 }";
17     var propertyDiagnostic =
           VerifyCS.Diagnostic(NamingAnalyzer.DisallowedSuffixDiagnosticId)
18         .WithSpan(8, 9, 8, 49).WithArguments("TestFactory",
               "IRepository");
19
20     var fieldDiagnostic =
           VerifyCS.Diagnostic(NamingAnalyzer.DisallowedSuffixDiagnosticId)
21         .WithSpan(9, 9, 9, 49).WithArguments("TestFactory",
               "IRepository");
22
23     await VerifyCS.VerifyAnalyzerAsync(test, propertyDiagnostic,
           fieldDiagnostic);
24 }
```

Figure 5.8: Example of a unit test for the Naming analyzer

6 Exception Analyzer

6.1 Functionality

Figure 6.1: Example of the Exception Analyzer's functionality

C# does not include support for compile-time checking of exceptions. In Java[1], this is possible using the *throws* keyword when defining a method. If a method definition contains this keyword in its signature, callers of this method need to catch all exception types after the *throws* keyword. This diagnostic analyzer will try to emulate this feature using C#'s XML comments.

Microsoft recommends documenting possible thrown exceptions of methods[2] using XML documentation. The <*exception*> tag can appear multiple times in the documentation and contains an exception type and a description of when this exception will be thrown. We can take advantage of this and implement a diagnostic analyzer that requires a method's usage to catch any exception that method has documented. An example of what the analyzer is capable of is shown in fig. 6.1. A code fix for this issue

[1] https://docs.oracle.com/javase/specs/jls/se7/html/jls-11.html#jls-11.2
[2] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/xmldoc/recommended-tags#exception

is implemented as well, where the method call will be wrapped in a try-catch block for the documented exception.

6.2 Implementation

```
1  private static void AnalyzeInvocation(SyntaxNodeAnalysisContext
       context)
2  {
3      var invocation = (InvocationExpressionSyntax)context.Node;
4      var semanticModel = context.SemanticModel;
5
6      var method =
           semanticModel.GetSymbolInfo(invocation.Expression);
7
8      if (method.Equals(default))
9      {
10         return;
11     }
12
13     var documentation =
           method.Symbol?.GetDocumentationCommentXml();
14
15     if (string.IsNullOrEmpty(documentation))
16     {
17         return;
18     }
19
20     var documentedExceptions =
           GetExceptionsFromXmlComment(documentation).ToList();
21
22     if (!documentedExceptions.Any())
23     {
24         return;
25     }
26
27 ...
```

Figure 6.2: Retrieving the documented exceptions of a method

The method *AnalyzeInvocation* is registered to be called whenever a syntax node of type *InvocationExpression* is compiled. An invocation in this context is the call of a method. Fig. 6.2 shows how the documented exceptions of the called method are retrieved. For this purpose, we use the semantic model of our compilation to access the symbol API and get the *SymbolInfo* of the called method (line 6). This struct allows us to retrieve the XML comments on the method's definition (line 13). Using a

self implemented helper method *GetExceptionsFromXmlComment*, we can collect all the documented exceptions. This helper method simply parses the XML documentation and looks for <*exception*> tags, then returns a list of the names of all the exception types in the tags. Again, at every step of the process checks for null and empty values are implemented to make sure the analyzer does not crash or behave unexpectedly.

Fig. 6.3 shows how the analysis is continued. In line 3, we try to find out if our method call is inside a try-catch block. We do this by searching through the syntax node's ancestors for a *try* statement. In case we find one, we access its *catch*es and collect the types of the caught exceptions. We now iterate over all documented exceptions and for each one, retrieve its type symbol, as before we only had its name. Then, we check if the exception is caught by iterating over all caught exceptions and examine if any of them are equal to the documented exception or if the caught exception is a base type of the documented one.

For this purpose, we again use a helper function *InheritsFrom* that takes two type symbols and goes through the first argument's base types to see if any of them is equal to the second argument. In case the documented exception is not caught or there is no try-catch block at all, we report a diagnostic at the invocation's location.

A special part of this analyzer is that it uses *properties* on a diagnostic. This is custom data than can be passed to the diagnostic and can then be read by a code fix provider. The data must be in the shape of a *ImmutableDictionary*<*string,string*>, but it still has potential to be very useful so that the code fix provider does not have to redo the computations done in the diagnostic analyzer. In this case, we pass the name of the uncaught exception as a property. The next section will describe how this data is used in the code fix provider.

```
...

var tryCatch = (TryStatementSyntax)invocation
    .FirstAncestorOrSelf<SyntaxNode>(n =>
       n.IsKind(SyntaxKind.TryStatement));
var caughtExceptions = tryCatch?.Catches.Select
    (x => x.Declaration.Type).ToList();

foreach (var documentedException in documentedExceptions)
{
    var documentedExceptionSymbol =
       context.Compilation.GetTypeByMetadataName(documentedException);

    if (documentedExceptionSymbol == null)
    {
        continue;
    }

    var caught = false;
    // check if documentedException inherits from any
       caughtException
    foreach (var caughtException in caughtExceptions ?? new
       List<TypeSyntax>())
    {
        var caughtSymbolTypeInfo =
           semanticModel.GetTypeInfo(caughtException);
        var caughtSymbol =
           context.Compilation.GetTypeByMetadataName(caughtSymbolTypeInfo
            .ConvertedType.ToString());
        if (caughtSymbol != null &&
           InheritsFrom(documentedExceptionSymbol, caughtSymbol))
        {
            caught = true;
            break;
        }
    }

    if (!caught)
    {
        // give documented exception type as property
        var properties = new Dictionary<string, string> { {
           PropertiesExceptionTypeKey, documentedException }
           }.ToImmutableDictionary();

        context.ReportDiagnostic(Diagnostic.Create(ReferenceRule,
           invocation.GetLocation(), properties,
           invocation.ToString(), documentedException));
    }
}
```

Figure 6.3: Checking the method call for caught exceptions

6.3 Code Fix

```
1  private static async Task<Document> AddTryCatchAsync(Document
       document, InvocationExpressionSyntax invocation,
       StatementSyntax statement, string exceptionName,
       CancellationToken ct)
2  {
3      var tryStatement =
           (TryStatementSyntax)invocation.Ancestors().FirstOrDefault(n
           => n.IsKind(SyntaxKind.TryStatement));
4
5      TypeSyntax exceptionIdentifier;
6      if (exceptionName.Contains('.'))
7      {
8          var parts = exceptionName.Split('.');
9          exceptionIdentifier = SyntaxFactory.QualifiedName(
10             SyntaxFactory.IdentifierName(parts.First()),
11             SyntaxFactory.ParseToken("."),
12             SyntaxFactory.IdentifierName(parts.Last()));
13     }
14     else
15     {
16         exceptionIdentifier =
               SyntaxFactory.IdentifierName(exceptionName);
17     }
18
19 ...
```

Figure 6.4: Retrieving the try-catch around the method call

For the code fix, in addition to the invocation expression that we operated on in the diagnostic analzyer, we need the statement that it is a part of. The reason for this is that the invocation may for example be part of a variable declaration statement like *var x = MethodCall()*. Only *MethodCall()* is the invocation, but for adding a try-catch, the whole statement needs to be put inside the block. This retrieval is not shown here as it simply involves iterating over the invocation's ancestors and taking the first node that is of type *StatementSyntax*.

Using the invocation, its surrounding statement and the name of the exception that is not caught, we can call the method *AddTryCatchAsync*, illustrated in fig. 6.4. First, we want to find out if our invocation is inside a try-catch block as that changes the way we handle our codefix. For this, we again iterate over our invocation's ancestors to see if any of them is a *TryStatement* (line 3). We will use this information later in the fix. The second preparation step for the code fix is constructing our exception type. As we only have the name of the exception as an argument, we need to create the *TypeSyntax* using

a *SyntaxFactory*. The type can either be a *QualifiedName* or a simple *IdentifierName*, depending on whether the namespace is a part of the name. We can simply differentiate this by scanning the exception name for a period character and then constructing it in the correct way depending on that.

```
1  if (tryStatement == null)
2  {
3      var block = (BlockSyntax) SyntaxFactory.ParseStatement("{\n"
           + statement.GetText() + "}");
4
5      // generate try catch around node
6      var tryCatch = SyntaxFactory.TryStatement(
7          SyntaxFactory.SingletonList(SyntaxFactory.CatchClause()
8              .WithDeclaration(
9                  SyntaxFactory.CatchDeclaration(exceptionIdentifier))))
10         .WithBlock(block);
11
12     var oldRoot = await
           document.GetSyntaxRootAsync(ct).ConfigureAwait(false);
13     var newRoot = oldRoot.ReplaceNode(statement,
           tryCatch).NormalizeWhitespace();
14
15     return document.WithSyntaxRoot(newRoot);
16 }
17 else
18 {
19     var newTryStatement =
           tryStatement.AddCatches(SyntaxFactory.CatchClause()
20          .WithDeclaration(SyntaxFactory.CatchDeclaration(exceptionIdentifier)))
21
22     var oldRoot = await
           document.GetSyntaxRootAsync(ct).ConfigureAwait(false);
23
24     var newRoot = oldRoot.ReplaceNode(tryStatement,
           newTryStatement).NormalizeWhitespace();
25
26     return document.WithSyntaxRoot(newRoot);
27 }
```

Figure 6.5: Adding the try-catch around the method call

Continuing in fig. 6.5, we now check if our invocation is inside a try-catch block. The easier case, shown in line 19 and onwards, is the one where we are inside a catch block. Here, we simply have to add another catch using the very useful extension method *AddCatches*. Then, we replace the old try-catch with our newly constructed one (line 24) and return an updated document.

```
1  [TestMethod]
2  public async Task UnknownExceptionsAreIgnored()
3  {
4      var test = @"
5  using System;
6  public class SomeClass
7  {
8      public void Execute()
9      {
10         ThrowsException();
11     }
12     /// <summary>
13     /// Will throw an exception.
14     /// </summary>
15     /// <exception cref=""SomeNoneExistingException"">Will throw
           this exception.</exception>
16     private static void ThrowsException()
17     {
18         throw new ArgumentException();
19     }
20 }";
21     await VerifyCS.VerifyAnalyzerAsync(test);
22 }
```

Figure 6.6: Example of a unit test for the Exception Analyzer

The alternative involves creating a new try-catch block and putting our invocation inside it. For this purpose, we first create a new block with our statement inside it (line 3). Then we use the *SyntaxFactory* again to create the *TryStatement* with a single *CatchDeclaration* catching our defined exception. For these calls to the *SyntaxFactory*, the *RoslynQuoter*[3] was very useful. It shows exactly which methods should be used to get the desired code sequence. After replacing our invocation with the new try-catch, we return the new document.

6.4 Testing

There are in total 12 unit tests for this analyzer and its code fix, one of them illustrated in fig. 6.6. The test shown here tries out a case where the exception type can not be found in the compilation. The diagnostic analyzer should be able to handle this problem and simply ignore the exception. The assertion used here makes sure that no diagnostic is shown and the analyzer does not error out while executing. Tests like this are important as developers tend to make mistakes and a stable diagnostic analyzer

[3] https://roslynquoter.azurewebsites.net/

needs to be able to handle erroneous inputs.

Some other unit tests of interest try out cases where multiple exceptions are documented and none of them or only one are caught. Also, We confirm that catching the base type of an exception is sufficient for the analyzer and try out the code fix in several situations, both where the try-catch block already exists and just needs to be extended and where it has to be newly created.

Analyzer and code fix are tested for static and non-static, public and private methods to ensure the accessibility level and whether it is a static or an instance method does not impact the functionality at all. The correct location and message arguments are also part of all assertions.

7 Conclusion and Future Work

The goal of this project was to evaluate if *Roslyn* provides a suitable API for C# developers to write their own code analysis tools. For this purpose, three diagnostic analyzers for different objectives were implemented. The ideas for them were worked out with developers at World-Direct[1].

In preparation, the structure and functionality of compilers in general and in particular of *Roslyn*, the .NET Compiler Platform, were examined and explained. Understanding the levels of APIs and different data structures used in *Roslyn* takes some time, but there are helpful tools and documentation papers that speed up this process.

After the initial familiarization, the three previously mentioned diagnostic analyzers were realized. Their implementation was documented and explained. Additionally, for two of them a code fix was implemented and documented.

The first analyzer by the name of Authorize Analyzer has a very simple purpose and was used as an introduction to programming with *Roslyn*. It scans comments in code for a specific attribute and reports a warning if it finds that attribute. It provides a code fix to uncomment that attribute.

The second analyzer called Naming Analyzer takes a look at class names and structures. It can be used to enforce correct usage of design patterns and with it, certain terms can be banned from occurring in type names.

Lastly, the Exception Analyzer emulates Java's compile-time exception checking. If a method explicitly states that an exception can be thrown in its XML documentation, a call to that method has to handle the exception.

The analyzers are all open source and available for free use[2]. In the future, they will also be available in the official NuGet package gallery[3].

In terms of potential improvements, the analyzers could all be extended with more features if needed. For example, the Exception Analyzer could support rethrowing exceptions if they are documented in the calling method's XML documentation. The Authorize Analyzer could support a configuration file and look for other attributes other than only [Authorize]. The Naming Analyzer might review class structures of design patterns more intricately so that even less mistakes are made. It could also be implemented as a stateful analyzer as that might improve performance if the configuration can be cached.

[1] `https://world-direct.at`
[2] `https://github.com/knollsen/csharp_analyzers`
[3] `https://www.nuget.org/profiles/Knollsen`

Additionally, there are some ideas for new analyzers like one that enforces a specific way of initializing fields and properties either in the constructor or in the member declaration. One already partly developed analyzer idea takes care of project structure by checking import paths. More analyzers will be developed by World-Direct in the future.

To conclude, the familiarization with the *Roslyn* compiler's API was a success. Working and useful analyzers were implemented, are now in active use and openly available. Techniques for dealing with more difficult features were discovered and utilized.

Bibliography

[1] Bill Wagner et al. Attributes (C#). https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/, 15.09.2021. Accessed: 04.12.2021.

[2] Bill Wagner et al. The .NET Compiler Platform SDK. https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/, 15.09.2021. Accessed: 04.12.2021.

[3] Bill Wagner et al. Understand the .NET Compiler Platform SDK model. https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/compiler-api-model, 15.09.2021. Accessed: 04.12.2021.

[4] Stephen Walther et al. ASP.NET MVC Controller Overview (C#). https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/aspnet-mvc-controllers-overview-cs, 19.02.2020. Accessed: 04.12.2021.

[5] Diomidis Spinellis. *Code Quality: The Open Source Perspective (Effective Software Development Series)*. Addison-Wesley Professional, 2006.

[6] Alfred V. Aho; Monica S. Lam; Ravi Sethi; Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2007.

[7] Manish Vasani. *Roslyn Cookbook: Compiler as a Service, Code Analysis, Code Quality and more*. Packt Publishing, 2017.