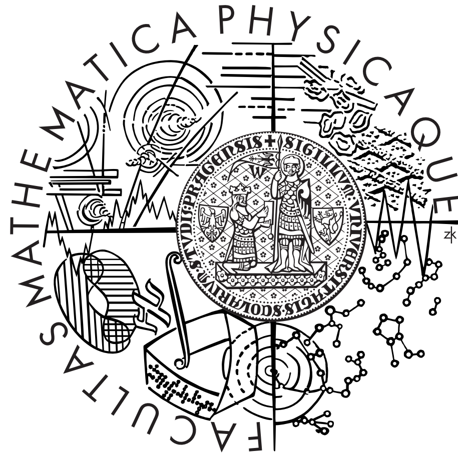
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Bc. Marek Linka

Visual Studio Refactoring and Code Style Management Toolset

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Pavel Ježek, Ph.D.

Study programme: Informatics

Specialization: ISS

Prague 2015

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.
I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In .... Date ....   Signature ....

Název práce: Sada Visual Studio nástrojů pro refaktoring a správu stylu kódu

Autor: Bc. Marek Linka

Katedra/Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D.

Abstrakt: Dodržování konzistentního stylu je nezbytné pro udržení spravovatelného zdrojového kódu. V době, kdy složitost softwarových řešení neustále roste, je tento požadavek důležitější než kdy dřív. Většina komerčně dostupných nástrojů pro zvýšení produktivity psaní kódu se ale zaměřuje více na refaktoring a podporu dodatečných technologií než na dodržování konzistentního stylu psaní. Rozhodli jsme se proto napravit tuto situaci tím, že naimplementujeme sadu nástrojů pro Visual Studio rozšiřitelnou pomocí zásuvných modulů zaměřenou na hledání a nápravu porušení stylistických pravidel v jazyku C#. Dokončením našeho záměru jsme vytvořili nástroj, který se hladce integruje s Visual Studiem a poskytuje uživatelům efektivní a intuitivní prostředky pro zlepšení spravovatelnosti jejich kódu.

Klíčová slova: Visual Studio, Refaktoring, Styl kódu, Roslyn

Title: Visual Studio Refactoring and Code Style Management Toolset

Author: Bc. Marek Linka

Department/Institute: Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek, Ph.D.

Abstract: Keeping a consistent coding style is an important part of having a maintainable code base. In times when software solutions become increasingly complicated this requirement is more important than ever. However, most commercially available coding productivity tools put a much bigger focus on refactoring and support of additional technologies than on maintaining consistent code style. We decided to remedy this situation by implementing a plugin-extensible toolset for Visual Studio focused on diagnosing and correcting code style violations in C# code bases. By completing our intent we created a tool that integrates seamlessly with Visual Studio and provides the user with effective and intuitive tools to improve the overall maintainability of their code base.

Keywords: Visual Studio, Refactoring, Code style, Roslyn

Dedication

To my father, for helping me get as far as I got. We miss you.

To my family, for supporting me in this endeavor all these years.

And finally, to my friends. Stay crazy!

Had to be me. Someone else might have gotten it wrong.
—Mordin Solus, Mass Effect 3

Acknowledgements

I would like to express my gratitude to all the people who helped me in the course of writing of this thesis. Whether by providing constructive criticism, technical expertise, or simply moral support, you all helped me make this big step forward.

A special thank you belongs to the person charged with supervising this thesis, Mgr. Pavel Ježek, Ph.D., whose feedback was invaluable and who helped transform this thesis from a loose set of ideas into a coherent whole I can be proud of.

To everyone, THANK YOU.

1. Introduction 1
1.1. Problem statement 3
1.2. Goals of this thesis 5
2. Analysis 6
2.1. Understanding source code 6
2.1.1. Parsing C# 8
2.2. Integrating with Visual Studio and Roslyn 10
2.3. Extensibility 12
2.4. Code transformations 16
2.5. Settings 18
2.6. Code navigation 22
3. Implementation 24
3.1. Layout overview 24
3.2. Code diagnostic implementation 27
3.3. Refactoring implementation 30
3.4. Platform abstraction 31
3.5. Settings composition 37
3.6. Settings 37
3.7. Visual Studio commands 38
3.8. Navigation pane 39
3.9. Main installation package 40
3.10. Code diagnostics package 40
3.11. Code refactorings package 41
3.12. Diagnostics unit tests 41
3.13. Refactorings unit tests 41
3.14. Platform services unit tests 42
4. Comparison with similar applications 43
4.1. JetBrains ReSharper + StyleCop 43
4.2. DevExpress CodeRush 43
4.3. Conclusion 44
5. Conclusion 45
5.1. Fulfillment of thesis goals 45
5.2. Future development 45
A. User Manual 47
A.1. Installation 47
A.2. Usage 48
A.3. Implemented code transformations 51
A.3.1. Diagnostics 51
A.3.2. Refactorings 53
B. Plugin Development Guide 54
B.1. Prerequisites 54
B.2. Developing a plugin 54
C. References 63
D. Content of the enclosed CD 64

1. Introduction

In real-world software development, software systems are usually developed in teams of multiple developers. This allows for more flexible workload decomposition and more effective development, but also leads to situations where programmers with vastly different backgrounds need to work together on a single solution.

These different backgrounds reflect, among other things, in the way programmers write and style their code – how they wrap lines of code, what brace layout they use, etc. These differences don't necessarily cause problems for the system being developed, but they can make cooperation in the team problematic. For most programmers, reading and understaing code written by someone else is one of the most difficult thing they might be asked to do. And if the situation is complicated by the fact that the code's author uses a significantly different code style, understanding such a code might become almost impossible.

Authors of most modern programming languages realize these issues and many languages ship with a set of recommended practices and guidelines for styling code. The main issue with these conventions is that there is a rather large gap between knowing that guidelines exist and actually following them. This difficulty comes from the fact that programmers are taught to be lazy (in certain aspects) and unless something forces them to acknowledge that their code is not following standards, they tend to postpone any corrections until “when I have the time”.

To overcome this problem, automated tools are necessary. If a programmer writes a line of code that breaks a stylistic rule and an automated check will immediately inform them of the problem, there is a good chance that the code will be fixed immediately. But watching for and reporting issues is just half of the story. To further increase the chance that the programmer will actually resolve the code issue, the tool should also provide an option to fix the problem automatically – to transform (or *refactor*) the problematic code so that it no longer violates the guidelines. Apart from resolving style issues, refactorings can also be used to automate certain repetitive and/or uninteresting tasks and save developers some time (renaming fields/methods/classes etc., creating new classes from usage etc.).

If we take a closer look at the Microsoft .NET family of languages, these come with a rather comprehensive set of rules and guidelines for writing consistent code [1]. These conventions are followed within Microsoft itself and present a stable set of stylistic rules for the most popular .NET language, C#. The rules contained within the specification deal with the most important aspects of writing code, from naming, through layout (braces, spacing, indentation, etc.), to documentation and several more.

Consider the following code. The method implemented in this example doesn't do anything interesting, but reading through the code is very difficult. The name of the method is generic and doesn't hint at the actual purpose of the method. Variable names are confusing (`t`, `T`, `t2`, etc.), typing is inconsistent (`var`, `Int32`), and the overall layout is chaotic (braces, line breaks). Fixing these issues manually would take a non-trivial amount of time.

```
public myspecialthingy Get(int x, string t, int a, bool t2)
{
  Int32 T = 456;
  var n = 28D;
  if (t2 == false) throw new ArgumentException();
      string y = new String();
  for (var m = 0; m <= a; m = m + 1) {
  y = y + t;
  }
  myspecialthingy tt = new myspecialthingy();
  tt.str = y;
  tt.t = n * a;
  return tt;
}
```

The next code example demonstrates the difference that proper styling can make. It performs exactly the same operation as the code above, but it's styled according to the C# guidelines. Variables and class have been renamed to better reflect their intended purpose. Implicit typing is used across the whole method and braces and line breaks are normalized. In the end, an object initializer is used to create a new instance of the CustomTuple class.

```
public CustomTuple GetTuple(
  string stringSeed,
  int copies,
  bool isValid)
{
  var numberSeed = 28D;
  if (!isValid)
  {
    throw new ArgumentException();
  }

  var stringValue = string.Empty;

  for (var index = 0; index <= copies; index++)
  {
    stringValue += stringSeed;
  }

  var result = new CustomTuple
               {
                  StringValue = stringValue,
                  Number = numberSeed * copies
               };

  return result;
}
```

In comparison to the first example, this code should be much easier to read and understand. Using an automated tool to validate and fix the original code would save a lot of time which could be spent on actual development.

Unfortunately, Visual Studio, as the most popular and advanced IDE for writing C# code, does not contain the ability to evaluate code against the aforementioned Microsoft guidelines. Developers either have to study the conventions and check their code as they are writing it, or look for a third-party tool to help with this task. StyleCop [2], a Visual Studio plugin, was one such tool. StyleCop used to be very popular with Visual Studio users and contained over a hundred and fifty various code style checks, ranging from simple (class must have a documentation comment) to complex (enforcing correct parentheses for logical expressions). Unfortunately, development of StyleCop was discontinued some time ago and no viable replacement has been found as of yet.

Situation on the refactoring side of things is slightly better. Even a freshly installed Visual Studio contains a set of basic refactorings. These include symbol renaming, symbol declaration, and several more. These are the most widely used refactorings, but they are far from covering even a half of the situation where (semi-)automated code transformations might come in handy. Developers who want to utilize more advanced forms of refactoring have to look for a third-party tool.

There are two widely used refactoring toolsets – JetBrains ReSharper [3] and DevExpress CodeRush [4]. These are both commercial products with a lot of features and can help with much more than refactoring C# code – they work with a variety of languages and frameworks (C#, Visual Basic.NET, XAML, and more). These tools are definitely much more powerful that the basic refactorigns that ship with Visual Studio, but they also have their own shortcomings, which this thesis will attempt to address.

1.1. Problem statement

First main problem we already mentioned in the introduction – Visual Studio does not contain the ability to monitor code for style violations, so developers haave to install third-party tools. And while StyleCop fit the requirements perfectly, its development has been cancelled and no new versions will be published. This is especially problematic, since at the time of writing of this thesis (2015), Microsoft is preparing to release Visual Studio 2015 – the latest StyleCop version will not support the new IDE at all.

Second issue we see with Visual Studio is the very limited number of refactorings available in it. This can be remedied by installing ReSharper or CodeRush, but both of these tools are very expensive, costing upwards of 150€. Considering the fact that since the release of Visual Studio Community Edition (early 2015), all the power of the VS IDE is now available to anyone for free, this price can discourage many programming beginners (often students).

There is one more problem with at least the ReSharper tool (but most likely with CodeRush also) – its huge set of features. ReSharper, besides refactoring C# code, performs a high number of other tasks. It can refactor XAML code and HTML, it can simplify working with resource files, it can work with JavaScript and TypeScript files and CSS. ReSharper also modifies the behavior of certain Visual Studio components, such as IntelliSense (smart autocompletion) and unit testing. Many of these features are only used rarely (such as TypeScript support) and the overloading of default behavior (for

example for IntelliSense) can cause more problems that it solves, because of potential bugs and performance drops.

All these features are installed together with the C# refactoring support and usually just occupy space and consume system resources, without providing any significant advantages. The whole toolset is therefore bloated, which leads to the final issue: performance.

Most users who installed ReSharper complain about degraded performance. This is best demonstrated on a simple scenario of opening a solution file. Since we wanted to see how much ReSharper affects the IDE performance, we measured the time required to open a relatively small-sized VS solution file with and without ReSharper running. We used the folowing experiment settings:

- CPU: Intel Core i7 3770K @ 3,50 GHz
- RAM: Kingston HyperX Beast @ 2133 MHz, 24 GB (23,8 GB usable)
- Storage: 2x Intel 530 SSD 120 GB (RAID1; hosting system, Visual Studio, and ReSharper) + 2x Seagate Barracuda 7200.12 500 GB (RAID0; hosting the tested solution)
- OS: Windows 8.1 x64, all the available updates installed, with paging file disabled
- Tested solution: 8 projects (console application, ASP.NET MVC application, Windows Service, several class libraries)

After opening the solution multiple times both with and without ReSharper, the following numbers were measured:

- Load time **without ReSharper**: **5 - 6 seconds**
- Load time **with ReSharper**: **15 - 17 seconds**

It is obvious that the performance impact of the ReSharper tool is significant, even on a rather powerful hardware. It is also not limited to solution load, but affects the performance of the whole IDE. One of the source of the performance hit is the use of a third-party source code parser. This component is used to scan and analyze the source code being edited, but since it does not use the Visual Studio compilation pipeline, it must perform its own parsing in addition to the parsing VS performs itself. This might easily lead to doubling the time necessary to analyze the source code.

It is also worth noting that both ReSharper and CodeRush support plugins, which allows their features to be expanded. This allows for users to implement features they need even if the core tool does not contain them, leading to improved flexibility.

There is one more, relatively independent, issue with Visual Studio we wish to address – navigation within the code base. For solutions consisting of only one or two projects, this is not a problem, since the number of places to look for a specific piece of code is rather low. For real-world systems consisting of ten or more sub-projects, orientation becomes much more difficult. Using the solution explorer integrated into the VS IDE requires a lot of mouse clicks and the integrated search ability is easier to access, but rather slow

on the actual search. Both these issues cause users to break their flow of work and costs time that could be spent elsewhere.

1.2. Goals of this thesis

The overall goal of this thesis is to develop a set of tools that would resolve the issues mentioned in the previous chapter. This goal can be further decomposed into the following areas of interest:

1. **Code style validation**

   Wathing for code style violations will be the cornerstone of the solution. The resulting application will be scanning the source code for problems and notify the user when a rule violation is detected.

   The core set of implemented style rules will be related to areas where proper styling has the greatest advantage – enforcing documentation comments and proper naming. If time permits, additional rules will be implemented, such as monitoring brace layout and member accessibility.

2. **Refactoring**

   Closely related to code style monitoring, (semi-)automated code refactorings will be implemented. These will allow for easy correction of detected stylistic problems and automate other often-performed tasks.

   In addition to automated fixes for the style violations mentioned above, refactorings for manipulating properties (e.g. auto-property to backing field), implementing popular concepts (INotifyPropertyChanged, DataContract), and typing (explicit to implicit typing and back). More popular refactorings will be implemented as time allows.

3. **Extensibility**

   Even though there will be refactorings and code style diagnostics available in the solution, this initial set will be far from comprehensive. The solution will come with the ability to extend the list of supported features by the way of plugins.

4. **Code navigation**

   Navigating within a code base should be as painless as possible, to minimize workflow disruption. This thesis will attempt to improve upon the existing VS capabilities in this area.

5. **Performance and IDE integration**

   The toolset proposed and implemented in this thesis will attempt to find a way to minimize its performance impact. At worst, the performance of our toolset should be equivalent to other currently available tools, but if at all possible, it should be better. The solution should also integrate with the Visual Studio IDE to match the ease of use of commercial alternatives.

2. Analysis

In order to meet the thesis goals, the toolset implemented as part of this thesis must have several basic capabilities around which the required features are built. Understanding source code is one such capability, because in order to perform a refactoring it's necessary to first know what is being transformed. Another basic capability leading to fulfillment of thesis goals is direct Visual Studio integration, which is required in order to provide the best possible usability, user experience, and performance. This thesis also aims to be extensible by plugins to allow easy introduction of new features, which influences the design of certain core features.

This chapter will cover these basic capabilities and discuss the main decisions affecting the design and implementation of this toolset, including certain important encountered issues. From this point onward, this analysis will be specific to the C# language, since it is both this thesis' target language and the language this thesis is implemented in.

2.1. Understanding source code

Source code for most programming languages is usually stored in the form of simple text files. Text files are easy to read and write, they can be compressed effectively, and have other advantages over, for example, binary files (files whose content is encoded in some kind of machine-readable-only binary format). But any source code file also has to follow a certain structure (*syntax*), to be considered valid. Without proper syntax, a source code file becomes just another text file that has no inherent meaning to the operating system and other software.

Therefore, any application attempting to transform a source code file (refactor it) must guarantee the resulting code will be syntactically valid – no developer would consider using a tool that is known to break the code it is used on. For this requirement to be met, the application must understand the syntax of the programming language it is designed for and follow it strictly. In other words, whatever the transformation input, the output must be syntactically correct for the given programming language.

However, there are situations where syntactic information is not enough – certain more advanced code transformations might require a deeper understanding of the source code. Imagine a method rename refactoring – a method might be used in source files different from the file that declares the method, in which case renaming it only at the place of declaration would break compilation. Additionally, there might be several *different* methods with the same name, so even performing a global textual search and replace would fail – it would rename even methods that were supposed to remain unchanged. Performing such rename operation safely requires a so called *semantic* model. Semantic model describes the meaning of the syntactic elements (classes, methods etc.), including their usage, across the whole code base. In the method rename example, semantic model would be used to find all the places where the particular method being renamed is used. The renaming operation would then happen at all the usage places, thus ensuring that the code will compile properly, but leaving the unrelated same-named methods unchanged.

It is important to note that this thesis also imposes one rather specific requirement on its language parser – the ability to write the performed changes back into source files. The reason for this requirement should be apparent: any refactoring performed by this thesis

must become part of the code base. Transforming code only in memory has no effect on the code base itself, therefore these changes need to be written back to files.

There are two main ways of reading and manipulating source code programmatically – text-based and graph-based. The text-based approach is simpler of the two and uses the fact that source code is basically just text. Using this approach, an application would read the source file character by character, construct words, sentences, and analyze the sentence structure.

Text-based parsers

Manipulating code with the help of a text-based parser would require the whole source code to be stored in memory as text. Any change to the code would then be executed by modifying the memory-stored text (e.g. inserting the text of a new statement at a particular place). This approach presents several problems. Since text data ("strings") cannot store additional information, finding the proper location for manipulations (insertions, deletions) would be difficult. Operations like "Rename method Foo to Bar" would need to scan the source code stored in memory to discover where the method is declared. The location of the method would be discarded after the method was found (or not found) since the text-based approach has no way of remembering it for later use.

This behavior is highly inefficient – every operation requires at least one scan of the source code text. Constructing a semantic model from strings would require the whole code base to be loaded into memory and multiple string searches to discover all the semantic links between the files.

Graph-based parsers

The graph-based way of parsing source code mitigates this performance issue by reading the source code file and constructing its in-memory representation as a graph. For most modern programming languages, these graphs take the form of a tree. Concepts such as classes and methods are described using vertices (tree nodes) and their position within the graph (tree) describes their position within the original source file. Strings are only used to store the information that requires a string form, such as identifiers (names).

Performing a rename operation (method "Foo" to "Bar") would then require the application to traverse the tree and search for the target method node, then changing its identifier. It should be obvious that traversing a tree stored in memory is usually faster than performing a text search on the whole souce file – navigating the tree using pointers/references to child nodes is much simpler in comparison to the repetitive string comparisons required by the text-based approach. The fact that the tree nodes can store additional information also allows for faster execution of more complex queries, such as "Give me the locations of all declared classes" – the locations could be pre-computed as the parser reads the source file, without the need to perform a text search.

There are two main types of trees used by currently used parsers and compilers – *abstract sytax trees* (ASTs) and *concrete syntax trees* (CSTs). The main difference between the two is in what information they store. ASTs only retain the information about "significant" code elements – things like classes, methods, statements etc. Information about the layout of the code – which includes whitespace, comments, braces, and more – is lost during the construction of an AST. Concrete syntax trees retain even this layout information and thus are much closer to representing the actual code file than the ASTs.

CSTs tend to have more complex structure because of this reason, but they allow for more detailed operations (e.g. “Add a comment to a statement”).

Graph-based parsers are also more efficient when constructing semantic models. Even though this operation still requires the whole code base to be loaded into memory, the tree object model lowers the space requirements and still allows for faster searches, so the queries necessary to construct the semantic model of a code base finish faster.

Writing changes to files

The text-based approach has its advantages in writing files – since a code file is stored in text form even in memory (including layout – braces, whitespace etc.), writing any changes just means the application must take the current text and save it into a file as is. This can be done easily and requires no additional steps.

Writing changes to files is more difficult with graph-based parsers. The whole code tree must first be turned into text (serialized), then the text written to a file. Depeneding on the size of the tree, this serialization operation might take a relatively long time because it requires every node to be turned into text. This means a lot of memory allocation to store the strings and a large number of string operations (e.g. concatenation). The distinction between ASTs and CSTs becomes important in this scenario as well. Reading a file into an AST and writing it back (even without any modifications) would lose information such as whitespace positions and comments. This is definitely not the desired behavior for any refactoring solution. CSTs don't suffer this information loss and are therefore better suited for use in refactoring solutions.

Conclusion

Since goal 5 of this thesis focuses on minimizing the performance overhead caused by this thesis, it was decided to use graph-based source code parsing. Even though text-based parsers are better at writing changes into files, the graph-based parsers are faster at parsing and manipulating code. It is also important to realize that changes are only written once at the end of a code transformation, while code might be analyzed and transformed dozens of times during a transformation. Graph-based parsers are therefore better suited for this thesis.

This thesis has C# as its target language, which means a graph-based C# parser was necessary. Writing such a parser from scratch would easily outgrow the scope of a master thesis, therefore it was decided a third-party component for parsing source code would be used. This approach would allow for better focus on implementing the desired features instead of worrying about the complexities of the programming language.

2.1.1. Parsing C#

There were several solutions for parsing C# code available at the time of writing of this thesis, so it was necessary to define a set of criteria by which to measure their respective usability. These are closely related to the thesis goals and cover things like language version support, parser capabilites, and performance.

1. **Language version**

   The C# language has first appeared in the year 2000 and has undergone significant modifications and upgrades. Current stable version of the language is C# 5.0, available since 2012, supported by the Visual Studio 2012 and 2013. C# 6.0 is currently in development and is planned to be released together with the next version of Visual Studio later this year (2015). As this thesis aims to provide support for currently available tools, the C# parser picked for the implementation needed to support at least the C# 5.0 language specification

2. **Capabilities**

   As discussed in the Section 2.1, “Understanding source code”, there are refactorings that require a semantic model of the whole code base to work properly. This creates another requirement on the parser of choice – ability to construct and work with semantic models. And since this thesis must be able to write code modifications back into source files, only solutions supporting this ability were considered, ideally with support for concrete syntax trees. Freely available and open-source parsers were given priority

3. **Resource consumption and performance**

   The last important factor in choosing a language parser was the amount of system resources it requires. Since every parser run outside of the main Visual Studio compilation pipeline subsitutes a performance degradation, the ability to integrate directly into the pipeline was considered a major advantage

Available tools

A list of available C# parsers was compiled, based mostly on popularity of the tool and its capabilities. After evaluating the candidates, three tools seemed suitable for use for this thesis' implementation: *NRefactory* [7], *C# Parser and CodeDOM* [6], and *Microsoft Roslyn* [8].

1. **NRefactory**

   One of the most popular code refactoring toolsets currently available is ReSharper. This toolset uses the *NRefactory* open-source parser to analyze and transform C# code. *NRefactory* supports C# 5.0, both syntactic and semantic analysis, and writing changes to files, but only works with abstract syntax trees. It contains facilities for preserving comments and other information, but special care must be taken in order to prevent information loss.

   NRefactory also ignores Visual Studio compilation and performs its own parsing runs *in addition* to any parsing Visual Studio does. This means higher system utilization and degraded analysis times.

2. **C# Parser and CodeDOM**

   C# Parser and CodeDOM by Inevitable Software is another powerful C# parser. It supports C# 5.0, performs both syntactic and semantic analysis, but uses ASTs. Comments and other non-significant entities can be preserved, but it requires special

effort. It is also closed-source and paid, which makes it unsuitable for the needs of this thesis.

3. **Microsoft Roslyn**

   Microsoft Roslyn is an open-source project developed by Microsoft to reimplement the aging C# and VB.NET compilers (originally written in C++) in managed code. Starting with Visual Studio 2015, Roslyn will become the main .NET compiler available in the IDE. In addition to tree-based syntactic and semantic analysis, Roslyn uses concrete syntax trees and also allows for byte-code emission (altough this capability is not necessary for this thesis).

   Roslyn also meets the requirement for direct integration with Visual Studio compilation pipeline – with Visual Studio 2015, Roslyn *will become* the compilation pipeline. Therefore any application using Roslyn to perform code analysis will require less parser runs, thus lessening the performance hit this analysis causes. Roslyn will also support the C# 6.0 language specification at the time of its release, making any applications using it prepared for the upgrade.

   However, Microsoft Roslyn has one major downside – it is currently in beta stage, with the final version due to be released together with Visual Studio 2015. This means that not all features are implemented yet and there have been several rounds of breaking changes in the APIs.

Conclusion

After considering the pros and cons of the available options, it was decided to use Microsoft Roslyn for this thesis. Roslyn fits the requirements set by the thesis goals and its beta stage is more than offset by the fact that it will allow for easy integration with Visual Studio and its coming support for C# 6.0.

2.2. Integrating with Visual Studio and Roslyn

In order to achieve its main goals, this thesis must be able to both integrate with Visual Studio and parse code using the Roslyn parser. There are several options available to achieve these goals and will be discussed below.

Visual Studio integration

Recent versions of Visual Studio (since at least Visual Studio 2010) support several different ways of integrating third-party plugins, the latest of which is the Visual Studio Packages (VSPs) framework. Older extensibility options, such as Visual Studio add-ins, still exist (mostly for backwards compatibility reasons), but VSPs are the latest standard and their use is recommended.

VSPs are a framework for implementing, packaging, and distributing Visual Studio plugins, but are also used by the Visual Studio IDE itself to provide basic services such as code highlighting, smart text editor, and IntelliSense. This clearly demonstrates that VSPs can be used to customize virtually any aspect of the IDE, a task not easily achieved using the older extensibility APIs, mainly becuase the older APIs don't cover the necessary features and capabilities.

To remain as future-proof as possible, it was decided to use the VSP framework for this thesis – this way this thesis remains open to potential expansion of features in the direction of IDE enhancements (e.g. smarter autocomplete) and the newer APIs make an overall better choice for this thesis. Backwards compatibility is also not an issue for this thesis, since Roslyn will only be available for Visual Studio 2015 and (presumably) later. 3. *Implementation* will cover more detail about the usage of VSPs in this thesis.

Accessing Roslyn

Once this thesis integrates with Visual Studio, the next required step is to harness the power of Roslyn to analyze code loaded within the IDE. This can be done in two different ways: by creating and using a distinct Roslyn instance only accessed by this thesis or by integrating with the Roslyn instance “living” within the IDE itself.

Creating a separate Roslyn instance within an application is rather simple – all it requires is to reference the necessary assemblies (freely available for download or built from source) and then use the APIs available within them. However, this approach has one major downside – a Roslyn instance created in this way allocates additional system resources and every parser run in such an instance happens *in addition* to any parsing Visual Studio performs internally. This means this thesis would lose most performance advantages resulting from the fact that Roslyn will become the main compiler in the next VS version and perform similarly to NRefactory and ReSharper.

If this approach was chosen, this thesis would have to monitor the Visual Studio code editor for changes (new code written, code deleted etc.) and parse the code every time a change happened. This would often result in two parsings of the same file (one be the IDE, one by this thesis), slowing down code analysis. This is obviously not ideal for this thesis. Nontheless, this way of using Roslyn is extremely easy to implement and for this reason it was used to implement the code navigation pane contained in this thesis. This feature will be covered later on.

Accessing the native Roslyn instance present within Visual Studio resolves the performance degradation. There would be no reason to allocate additional system resources and there would only be a single parser instance analyzing a particular code file. The downside here is that getting third-party code into the native Roslyn instance is more difficult and requires the knowledge of how VS handles its packages and how Roslyn expects the third-party code to behave. In short, code accessing the native Roslyn instance must be packaged using the VSP framework, with special configuration. In order for this code to be executed by the Roslyn instance, it also need to implement a specific set of base classes and interfaces. Discovering and configuring all of these details takes certain effort and it is not straightforward to do it manually.

Roslyn SDK

To lessen the aforementioned configuration difficulty, the Roslyn project comes with its own *software development kit (SDK)*. SDKs are generally described as sets of tools that allow or simplify development for a particular system or framework. In Roslyn, the SDK installs into Visual Studio and contains libraries and VS project templates that take care of most of the complexity necessary to reach the native Roslyn instance running in Visual Studio. These project templates create VS projects that are automatically configured to integrate with the native Roslyn instance and provide a simple skeleton code on which

to implement any Roslyn-enabled features (code diagnostics, refactorings) in a way that can be invoked by the native instance.

Achieving the same level of integration is certainly possible even without the use of the SDK, but every detail would have to be first researched and then configured manually. This would still provide virtually no advantage over the configuration created by the SDK's project templates, it would just cost more effort. Therefore it was decided to employ the Roslyn SDK to create the necessary projects and configurations, thus allowing the development to focus more on features and less on technicalities of the Roslyn and Visual Studio architectures.

The problem of portable classes

Using the Roslyn SDK project templates also presents several complications, foremost of which is that their output consists of Portable Class Libraries (PCLs). As the name suggests, portable class libraries in .NET are designed to allow execution on different platforms with different sets of hardware/software (such as Windows Desktop, Windows Modern, Windows Phone etc.). PCLs can only work with a limited subset of the .NET base class library and platform-specific APIs are not accessible to them at all. This means that Roslyn can (at least theoretically) run even on platforms different from Windows Desktop. This is certainly an interesting possibility, but one that is not required for implementation of this thesis – this thesis' toolset only runs within Visual Studio, which is only available for the full-featured Windows Desktop.

PCLs within an application behave a little like a virus – all code accessed from within a PCL must itself be portable, turning the containing assemblies into PCLs. This is especially troublesome for code that is accessed from a PCL (e.g. from a Roslyn-based refactoring) and still requires to access APIs that are not available from within a PCL. One such example is the global settings manager implemented by this thesis to manage and store user settings, making them accessible wherever they are needed. Settings are stored in files, therefore the manager must be able to read and write files and cannot be a portable class. On the other hand, the manager must be accessible from within Roslyn-powered code and therefore must be portable, leading to a contradiction.

There are several ways to overcome this issue, most of which rely on dependency injection (DI). To allow the portable bits of this thesis to interact with the non-portable bits, an abstraction layer is created between the two parts (this usually takes the form of an interface). Reusable non-portable services (e.g. the settings manager) are registered in a dependency injection container and whenever a service is required, it is retrieved from the container and accessed through the abstraction layer (interface), making it callable from a PCLs. This allows PCLs to call even non-portable services, since they don't reference the non-portable code directly.

More details about this approach and its actual implementation will be discussed in later chapters.

2.3. Extensibility

One of the main goals of this thesis is extensibility – the ability to expand the available capabilities using a plugin system. This goal stems from the fact that the core implementation of this thesis can only contain a limited number of code style diagnostics and refactorings and many potential users might find this core set lacking in features.

To accomplish this goal, a plugin architecture had to be designed. This includes areas such as plugin design, distribution, installation, management, and execution. After considering the requirements and specifics of this thesis (Roslyn, VS integration etc.), two approaches to plugin architecture seemed viable: an entirely custom plugin engine where all the aforementioned areas are managed by this thesis, or a system that delegates some of the tasks to the underlying IDE and only provides certain unifying services.

Custom architecture

Implementing an entirely custom plugin architecture means that literally every task related to plugins must be performed by the application itself. Questions such as "What do plugins look like?", "Where and how to install plugins?", "How to discover, load, and execute plugins?" would all have to be answered and their answers manually implemented.

Implementing a plugin architecture in this way means total freedom of design and also total control over the plugin capabilities, making implementation more straightforward and moslty free of external constraints. However, it also means that the architecture would have to be designed extremely well – once designed, modifying the interaction points between the main application and the plugins later on would be costly and could easily lead to breaking changes (situations where older plugins stop working with the newer plugin APIs). This would degrade the user experience and forcing plugin developers to constantly look out for problems. This would become especially problematic if the original design was flawed in some fundamental way – if it would be too slow, or too restrictive, or similar. Redesigning the architecture would in this case most likely lead to a total breakdown of any existing plugins.

Since this whole thesis and most of its plugins deal with transforming source code, there was also the question of how to let plugins access the Roslyn APIs. As discussed in Section 2.2, "Integrating with Visual Studio and Roslyn", this thesis uses the VS-native instance of Roslyn to achieve the best analysis performace possible. To provide the same performance for plugins would mean that even the plugin code would have to be executed by the native Roslyn instance. In the end, the main application would only be responsible for finding installed plugins, loading them, and then passing them to the Roslyn instance – effectively becoming a simple bridge between the plugins and Roslyn.

Building upon VS extensibility

Looking at the issues and complexities of the fully custom approach, it was deemed too complicated – especially given that the main application would only perform the "boring", boilerplate role of installing, finding, and loading plugins. It is necessary to realize here that all of these tasks are already implemented in Visual Studio itself. The VSP framework is capable of performing plugin installation and load. Visual Studio also has an intuitive way to view installed plugins, including update and uninstallation. In addition, the VSP framework is robust, tested by time, and standardized, making any potential breaking changes unlikely.

In combination with the Roslyn SDK (discussed in Section 2.2, "Integrating with Visual Studio and Roslyn") the VSP framework is also capable of installing packages that can access the native Roslyn instance, without the need for a "middle man" of this thesis.

Leaving the basics of the plugin architecture to the IDE itself thus makes a lot of sense, implementation-wise – all the necessary features are already implemented in a standardized and popular way, no need to “reinvent the wheel”. However, leaving things entirely to Visual Studio and its VSP framework would mean that any plugins would effetively be *Visual Studio plugins*, not plugins for this thesis. Still, there are areas where the standard VS extension framework falls short – developing solutions for these areas would bring added value to any Roslyn-based plugins, thus making integration with this thesis an asset.

After looking at what the VSP framework can and can't do, two important aspects of plugin development where Visual Studio extensibility doesn't provide suitable facilities were identified: settings management and user interface.

Most Visual Studio plugins are configurable and require a way of storing and accessing this configuration. Visual Studio allows plugins to integrate with its *Options* window but this integration is cumbersome to implement for the developer and cumbersome to use for the user – the *Options* window is often very cluttered. Many plugin authors therefore create their own settings windows, leading to situations where various “Settings” menus are strewn about the whole IDE, making the task of finding the proper one for a given plugin confusing.

Providing a unified way to manage settings (store, load, distribute), including an intuitive user interface, would mean that users would be able to manage all Roslyn-related settings from one place and in a consistent way. This would help lessen confusion originating from large number of independent plugins with different UI, leading to improved user experience.

It was therefore decided that this thesis will implement facilities that allow other VS plugins (even those not powered by Roslyn) to register their settings with this thesis and have them managed in a unified manner. This includes ability to show user interface for displaying and changing these third-party settings in a single window and in a way that is visually consistent across plugins and the core thesis, virtually erasing the distinction between core settings and plugin settings.

Platform services

Allowing third-party code to integrate with the core thesis also brings several issues. In order for a plugin architecture to be viable, development of plugins must be made as painless as possible, to encourage developers to actually write plugins. There is also the problem of inconsistency of behavior – in an application with several plugins there might arise a situation where two plugins perform similar tasks differently, leading to user confusion. Two examples of these issues follow:

1. **Complicated API – code emission**

   Emitting new code will be happening virtually in every refactoring (for obvious reasons) and lies at the very core of this thesis, which is why it should be as simple as possible for the plugin developers. Unfortunately, the Roslyn APIs responsible for code emission are rather cumbersome. Imagine a plugin needs to create a new field declaration as part of a refactoring:

   ```
   string username;
   ```

This is a simple enough piece of code, and it should be possible to emit is using a simple API call (pseudo-code):

```
FieldFactory.Create("string", "username");
```

This would be the ideal scenario – simple, with obvious intentions and result. However, as mentioned earlier (Section 2.1.1, “Parsing C#”), Roslyn uses concrete syntax trees to manipulate code and the APIs are designed in a way that supports all the edge-cases that exist within the C# language. This means that to emit a simple uninitialized field declaration from the example above, the plugin must perform the following call:

```
var declaration =
  SyntaxFactory.LocalDeclarationStatement(
    SyntaxFactory.VariableDeclaration(
      SyntaxFactory.IdentifierName("string"),
      SyntaxFactory.SeparatedList(
        new[] {
          SyntaxFactory.VariableDeclarator("username")
        })));
```

This code is far from intuitive and its author must be relatively familiar with the Roslyn APIs. This goes directly against the requirement that plugin development should be as simple as possible.

2. **Inconsistency of behavior – name casing analysis**

   As an example of inconsistent behavior, let's look at the issue of maintaining consistent member (class, property, method, field etc.) naming. Any programmer will readily agree that naming things is difficult, which is why the C# stylistic guide provides extensive rules for naming code elements. The core solution will of course implement stylistic checks against these guidelines, but many developers might find these rules unusable for their work and they might want to implement their own using plugins.

   Important aspect of consistent naming is casing – C# developers typically use variations of camelCase and PascalCase to name code elements. Therefore virtually every stylistic check that verifies naming must also check whether the casing is correct. This includes both the rules implemented in the core thesis and those implemented in plugins. But different implementations of this casing check might behave differently in certain situations, e.g. in the decision whether underscores (“_”) are considered valid name elements. If the core implementation differed from the plugin implementation in such cases, it could easily lead to confusion of the end user.

Solution

In order to resolve both the issue of inconsistency and complicated APIs, this thesis implements a platform abstraction layer. This layer serves as a repository for the most often-used application logic related to stylistic validation, code analysis, and code emission. It is available both to the core implementation and to plugin developers. This

layer concentrates all the often-used logic in one place, making sure that behavior of these methods is consistent no matter where they are called from. It also hides certain complexities of the Roslyn APIs behind a much simpler interface, reducing the amount of code necessary to perform certain frequently used actions.

However, because of the size of the Roslyn's API surface and its complexity, the abstraction layer could not possibly cover it all. Implementing such an abstraction would be redundant and it would either end up similar in complexity or removing a lot of features (keeping the API simple would require many edge-cases to be ignored). Instead, the implementation was decided to start with the methods most usable for the core features – more methods will be added as they are discovered.

2.4. Code transformations

Transforming source code is the cornerstone of this thesis. All the previously discussed topics are mainly just stepping stones that lead to an efficient and powerful way of performing these transformations. As mentioned in Section 2.1, "Understanding source code", Microsoft Roslyn will be used to facilitate the code analysis and manipulation, helped by the Roslyn SDK APIs for integrating with the native VS Roslyn instance. That still leaves the question of what code transformations will be available and what form they will take.

Roslyn-based transformations

Before deciding upon the set of code transformations to implement in this thesis, it was necessary to discover what is possible to implement with Roslyn. After working through the available APIs it was apparent that Roslyn recognized three main types of code operations: *code diagnostics*, *code fixes*, and *code refactorings*. These three categories differ in their capabilities and way of execution within the IDE and will be covered in the following paragraphs.

1. **Code diagnostics**

   Code diagnostics in Roslyn serve as a way to analyze the syntactic and/or semantic structure of a code file (or the whole code base), provide the user with visual clues as to any discovered problems, and possibly provide automated fixes – they are used to *diagnose* code issues (hence the name).

   Roslyn diagnostics can operate on both syntactic and semantic models and can access both at the same time, allowing for very complex and thorough code base analysis. Compared to code refactorings (discussed below), diagnostics differ mainly in the way they are invoked by the Roslyn instance – all code diagnostics are run with every VS parser run, analyzing the code on the fly. Whenever a source code file changes (signifficantly, not with every key press, for performance reasons), Roslyn runs code diagnostics on it.

   This means that for a given code file refactorings might be run tens of times a minute, depending on the way it is edited. This leads to an important factor for writing code diagnostics: their implementation should be as fast as possible, otherwise they might cause lag issues in the IDE. If a particular diagnostic took 10 seconds to complete, the user would be stuck for 10 seconds every time the diagnostics were run. This would hamper productivity and greatly frustrate the user.

Therefore even if code diagnostics can perform very complex syntactic and semantic checks, they should only be used to implement checks that are fast, have low system resource consumtion, and running them often makes sense. Code style analysis is a good example of such diagnostics, since checking code style usually doesn't require semantic information and it only works with a single source file at a time.

2. **Code fixes**

   Code diagnostics are used to analyze code and detect issues, but they don't have the ability to actually change the code – they can only read. To remedy a problem detected by a code diagnostic, code fixes are used.

   Every code diagnostic may declare one or more related code fixes. After all diagnostics are run and their results collected, these fixes are made available to the user through contextual icons at the edge of the code editor (this is also handled by Roslyn). The user might decide to perform some fixes, but no code fixes are performed automatically.

   Because of this manual execution, code fixes don't necessarily have to be extremely fast. Users should be well aware that they are invoking a potentially complex action and should be willing to wait for a little bit. However, given the fact that the original diagnostic is required to be fast, the analyzed problem can't be too complex (otherwise the diagnostic would be complex as well), which means that code fixes also tend to be rather fast.

   In this thesis, stylistic code diagnostics will be paired with code fixes providing the user with the ability to automatically fix discovered style violations.

3. **Code refactorings**

   Code refactorings are the last category of code manipulations Roslyn recognizes. Refactorings are very similar to a combination of code diagnostics and fixes – they implement both the code responsible for deciding whether it is possible to refactor a given piece of code and the code of the actual transformation.

   Refactorings have access to both syntactic and semantic models of the code base and therefore lack no ability compared to diagnostics. They can also transform code just like code fixes.

   The main difference from diagnostics is that refactorings are not performed automatically, but must be executed by the user. There is no visual indication that a certain piece of code is suitable for a refactoring – the user needs to ask the IDE to show the list of refactorings available for a specific piece of code. At this time, the analytical part of the available refactorings is executed and all refactorigns that are capable of transforming the given piece of code are displayed to the user. The user might subsequently choose to execute the transformative part of any of these refactorings.

   This manual execution mode means that refactorings are executed much less often than diagnostics and so neither the analysis nor the transformation needs to be instant.

This allows for more complex, even solution-wide transformations to be written as refactorings.

This thesis will use refactorings to implement transformations that don't need to be executed with every parser run and that the user doesn't want to apply to every instance where thay can be applied. For example, automatic implementation of a specific interface will be implemented as a Roslyn refactoring – the user will not want all classes to implement such an interface, only a small subset of them, therefore there is no need to run such a transformation with every parser run and display visual indicators about the transformation to the user. When the user decides that a piece of code is worth of refactoring, they will execute the transformation manually.

Core code transformations

This thesis will implement a set of core code transformations using the means of Roslyn code diagnostics, code fixes, and refactorings. However, it was necessary to decide what code transformations will be part of this core set. There are hundreds of potentially useful code transformations, but due to the time constraints and other factors it was not possible to implement them all.

After a brief analysis of past experiences with similar tools (namely StyleCop and ReSharper) it was decided that the core transformations will be picked by their usefulness and overall impact on code quality, with the addition of features deemed useful but missing from existing refactoring tools.

In case of code style diagnostics, this thesis focuses on partially replacing capabilities of the recently-discontinued StyleCop extension. StyleCop implemented over a hundred different code style checks, which is still a rather large number, so this thesis chose to implement checks for stylistic rules whose violation has the biggest impact on code quality. This mainly includes checks for documentation comments, brace layout, and naming. These areas of style are the most visible and have a great impact on the overall code readability and understanding.

Refactorings implemented in this thesis mainly contain transformations that work with properties (changes to/from auto-properties), implement certain popular base classes, interfaces, or other concepts (e.g. data contracts), and more. These are mostly, from personal experience, often-used tasks that come in handy in practice.

2.5. Settings

As discussed in Section 2.3, “Extensibility”, a unified settings manager should be implemented in this thesis, to allow the management of plugin settings in a consistent manner. It was therefore necessary to implement a way for plugins to access this settings manager. It was also necessary to design a proper way to store the settings.

Software is most often developed in teams of multiple people, which might lead to stylistic incosistencies. And even if all the team members used the same plugins for stylistic checks, their settings could differ, leading to inconsistent code style across the code base. To prevent this issue from occuring, the settings should ideally be synchronized between the various users and machines. This could be achieved in multiple ways, including some sort of entirely custom implementation, but this would again be just reinventing the wheel. Source code is traditionally stored within code versioning systems

whose primary objective is to make sure that all developers always have access to the latest version of the code. Versioning systems also serve to synchronize and distribute changes made by different programmers.

This is exactly the behavior that could be used to distribute and sychronize settings for this thesis. By making the settings part of the code base, this thesis can rely on the underlying versioning system to distribute them to all developers and make sure the settings are kept in sync. Storing settings within the code base also makes it possible to version the settings themselves and even track users changing them. This approach allows for easy answering of questions such as “What settings were in effect in at this time or code base version?”, should it become necessary.

Allowing the settings to become versioned with the code itself brings several requirements, first of which is that the settings must be stored in a file somewhere within the code base. This, by extension, means that the settings manager must be able to read and write files, requiring platform specific APIs for doing so. This is where the issue of portability discussed in Section 2.2, “Integrating with Visual Studio and Roslyn” becomes obvious.

Writing settings into files could be accomplished in two different ways: in text form or in binary form. Implementation-wise, this distinction is not very important for this thesis – writing both text and binary files is easily accomplished using the available APIs. However, the integration with versioning systems warrants a more thorough analysis of both options. Binary files have a theoretical advantage over text files in performance – reading and writing binary data is usually faster and leads to smaller files. Given the size of the settings files, though, saving a few kilobytes in a solution of tens or hundrends of megabytes of source code is not very noticable. Binary files are also harder to read and edit without the proper tools. Binary files would therefore prevent certain features of the versioning system, such as web preview or diff computation, from working properly.

Text-based settings file will be slightly larger than the binary files but they are more suitable for versioniong systems, mainly for change tracking reasons. Most versioning systems are able to compute diffs of text files, allowing for more thorough change tracking and analysis. In contrast, most versioning systems ignore binary files in diffs and simply treat them as text files. Though change tracking of settings files might not be an often-performed operation, it might come in handy from time to time. Text files are also much easier to edit manually, compared to binary files – any old text editor (e.g. notepad) can be used to view and modify them. The aforementioned web preview integrated into certain versioning systems would also work properly on text-based settings files.

In the end, it was decided to use text files to store settings information, mainly for their easy editability and suitability for versioning systems. However, there are many textual formats that could be used to store the settings, for example XML, JSON, INI files, in addition to an entirely custom implementation. All of these formats can be stored within a file and can be managed effectively by a versioning software, including diff computation.

After considering several text-based file formats for storing structured information (custom-build, INI files, JSON, and XML), it was decided that the implemented toolset will use XML for storing settings. This decision was made mostly because XML is easily read and written in .NET, does not require any additional libraries, and fulfills all requirements the implemented toolset might have.

Distributing settings and portability

Settgins, apart from being stored, also need to be distributed to the code that makes use of them. This thesis therefore implements a settings manager that takes care of both the storage and distribution of settings and is accessible from virtually every part of the core application and any potential plugins. This means that settings manager can be used from both non-portable *and portable* code (e.g. code transformations).

As mentioned in Section 2.2, “Integrating with Visual Studio and Roslyn”, portable code (refactorings, diagnostics etc.) may only interact with other portable code, which is a problem for this settings manager, since it must be able to read and write files, which is not supported from a portable library. To overcome this issue, this thesis uses the generally recommended solution of dependency injection (DI) and injects the non-portable code (hidden behind a portable interface) into the portable code.

There are several popular frameworks for performing DI in .NET but this thesis sticks with the tools available in BCL. Visual Studio itself contains a powerful dependency injection container based on Managed Extensibility Framework (MEF). MEF is basically .NETs way of implementing DI – objects can be designated as exports (providers on which other code can depend) and can also declare imports (dependencies that should be injected). When the dependency container is asked to get an instance of (*resolve*) a particular export, all its imports are recursively satisfied, returning an object that has all its dependencies instantiated “automatically”. Since MEF is a standard .NET way of performing DI and VS supports it, it was initially decided to use this way of DI to get around the PCL problem within this thesis.

To be able to use DI to inject non-portable services into portable classes, the non-portable code first had to be abstracted behind interfaces. These interfaces had to be contained in a portable library in order to be usable from portable code. The non-portable code then implemented these interfaces and was declared as a MEF export in order to be injected into objects that required it by the composition container. Next step was to make Visual Studio load this non-portable code into its composition container. This was done by configuring the VS package containing the exported code as a MEF component. MEF component VSPs are automatically included in Visual Studio's composition container at the time of load.

Next step was to actually import the non-portable code into the portable classes. The Roslyn SDK designates all VS packages created by it as MEF components, which ensures that any composable parts found within those packages can take part in dependency resolution. Classes requiring access to the non-portable code were annotated with imports to allow dependency injection. After compilation and debugging, the MEF composition container within Visual Studio managed to discover the imports and exports and satisfy them for both code fixes and refactorings, but not for code diagnostics. Figure 2.1 describes this injection process.

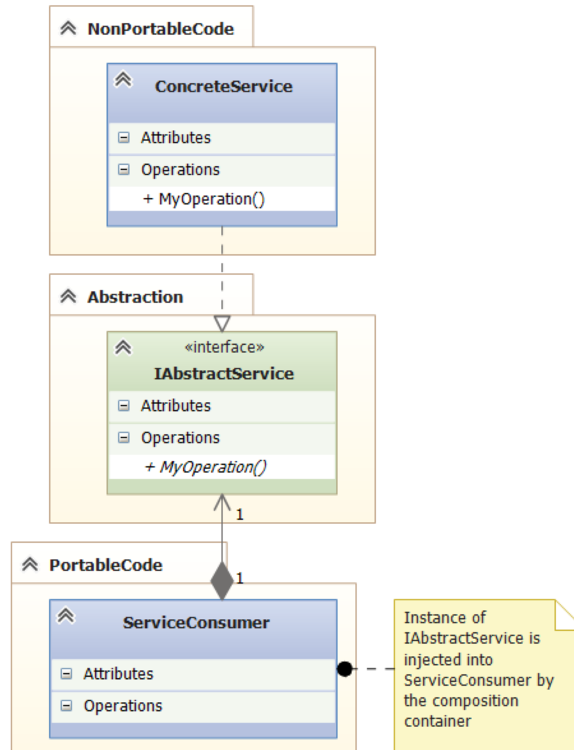


Figure 2.1. Injecting non-portable code into portable code through abstraction

After thorough investigation it was discovered that no matter the configuration, code diagnostics would **never** participate in dependency injection using the VS MEF container. After contacting the Roslyn developers, it was discovered that this behavior was intentional and stemmed from the fact that while both code fixes and refactorings are instantiated using the MEF container, diagnostics are instantiated using .NET reflection. The whole dicussion can be viewed at CodePlex [9]. The Roslyn developer answering the question acknowledged that this behavior might change in the future, but a different approach was necessary to make this thesis work as intended.

It was deemed unnecessary to rewrite the whole DI implementation because of this problem, since large part of the injection worked well using the MEF container. Instead, a very simple custom service container was implemented (in a portable assembly, to be usable from both portable and non-portable code) and the code diagnostics, instead of having the dependency injected by the composition container, requested the necessary code from this service container. This resolved the issue in all the necessary cases – MEF composition is used where possible and custom solution is employed where MEF composition is not available. Figure 2.2 describes the modified injection process for both portable and non-portable code.

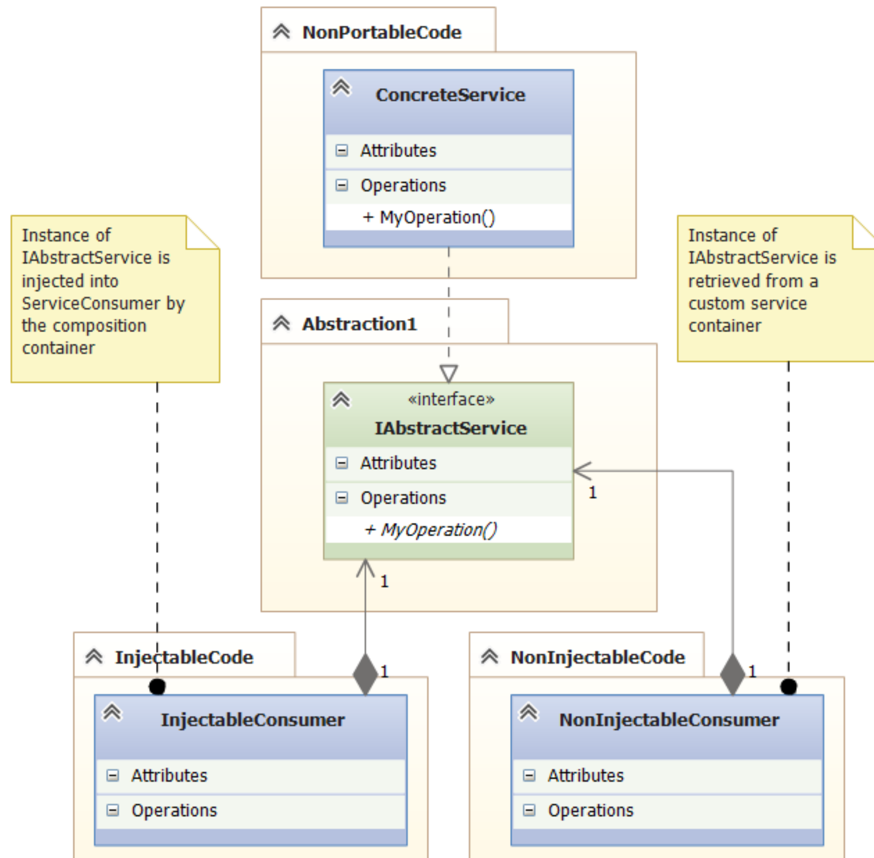


Figure 2.2. Work-around for non-conposable portable code

2.6. Code navigation

Improving code navigation is one of the primary goals of this thesis (goal 4). In Visual Studio, navigation within an average code base is often relatively difficult. This difficulty stems from the fact that a typical “real-world” code base might consist of scores or even hundreds of files and finding a particular piece of code in it must therefore rely on the tools provided by the IDE.

From experience, there are two different aspects of code navigation. One is the navigation between files contained within the code base, e.g. when the user wants to switch from editing file `MyTestClass.cs` and open file `Window.xaml`. Visual Studio provides its “Solution Explorer” to facilitate this “inter-file” navigation between code elements.

The second type of navigation is navigation within a single code file. Code files might (and often do) contain thousands of lines of code and declare scores of methods, properties, and other code elements. Trying to find a specific member (method, property etc.) becomes more difficult the longer the code file is. Additionaly, Visual Studio contains no native way of making this “intra-file” navigation easier for the user (apart from a standard text search, which is insufficient – e.g. there is no autocompletion). For this reason it was decided that the implementation of this thesis should contain a tool to simplify this kind of single file navigation.

There were two approaches considered to implementing this navigation tool. The first aproach was similar to ReSharper's “Navigate to member” feature – the tool would employ a keyboard shortcut that would display a special search box. This search box would autocomplete member names from the current source code file and would be able to navigate directly to the user-selected member. Figure 2.3 illustrates ReSharper's implementation of this approach, inlucing the autotomplete mechanism.

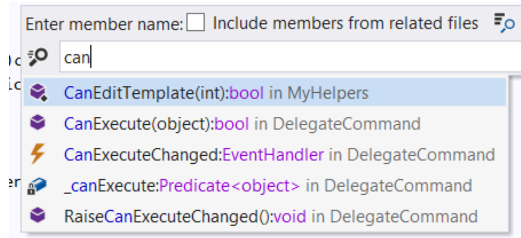


Figure 2.3. ReSharper's “Navigate to member” feature

This approach is definitely viable but has several drawbacks, greatest of which is the necessity for a keyboard shortcut. Visual Studio already contains lots of keyboard shortcuts so finding one that is both convenient to use and still unused by the IDE would be problematic. Having to memorize this shortcut also puts pressure on the user and could easily lead to a situation where this navigation feature would remain unused because of the necessary memorization of the shortcut.

The alternative solution would be to implement a “code map” of sorts. This map would extend the code editor with a special area that would display a condensed overview of the whole code file, showing the names of all the namespaces, classes, and other members in alphabetical order for easy orientation. Clicking a code member on the map would take the user directly to the member's declaration in the code file.

This approach eliminates the need for a keyboard shortcut as the map is part of the code editor itself. It also provides a good way to view a summary of the code file as it only displays member signatures. However, the drawback here is that the map occupies screen space, decreasing the amount of code that is visible to the user. To combat this issue the map needs to be collapsible – that way the user can hide the map when it's not needed but still keep it close at hand to make navigation in the code file a matter of a single click.

It is important to note that the code map **does not** integrate with the native Visual Studio's Roslyn instance. Instead, it references the Roslyn libraries and uses the syntax parsing APIs directly. This behavior is necessary because the native Roslyn instance only supports three types of integrations – code diagnostics, code fixes, and refactorings. There is no way for the code map to be implemented as one of these, therefore direct integration with the native instance is not possible.

3. Implementation

This chapter covers the overall organization of the application and describes the implementation of its components. The implementation work was done using Visual Studio 2015 Preview (version number 14.0.22310.1 DP), corresponding Visual Studio 2015 Preview SDK, and the Roslyn SDK (version 1.0.0.41031). The implemented code should work properly even with newer versions of the necessary tools but due to the preview stage of all the tools this is impossible to guarantee.

Due to its length, plugin development guide is not part of this chapter. Instead, it can be found in Appendix B, *Plugin Development Guide*.

The whole solution is covered by *Microsoft Public License* (Ms-PL) [11]. Creation and distribution of both free and commercial derivative works based on the current implementation is allowed, given that the creator of such works respects the conditions as stated in the license.

3.1. Layout overview

The implementation part of this thesis consists of 13 distinct projects organized into a Visual Studio solution. Seven of these projects contain the implementation of the core features (blue; *Diagnostics*, *Refactorings*, *PlatformServices*, *Settings*, *Settings.Composition*, *CodeMapPane*, and *Commands*), three contain unit tests for certain parts of the core implementation (green; *Diagnostics.Test*, *Refactorings.Test*, *PlatformServices.Test*), and three serve to package the solution output into Visual Studio Packages for ease of distribution and installation (orange; *Package*, *Diagnostics.Vsix*, *Refactorings.Vsix*). Figure 3.1 summarizes the top-level organization of the solution:

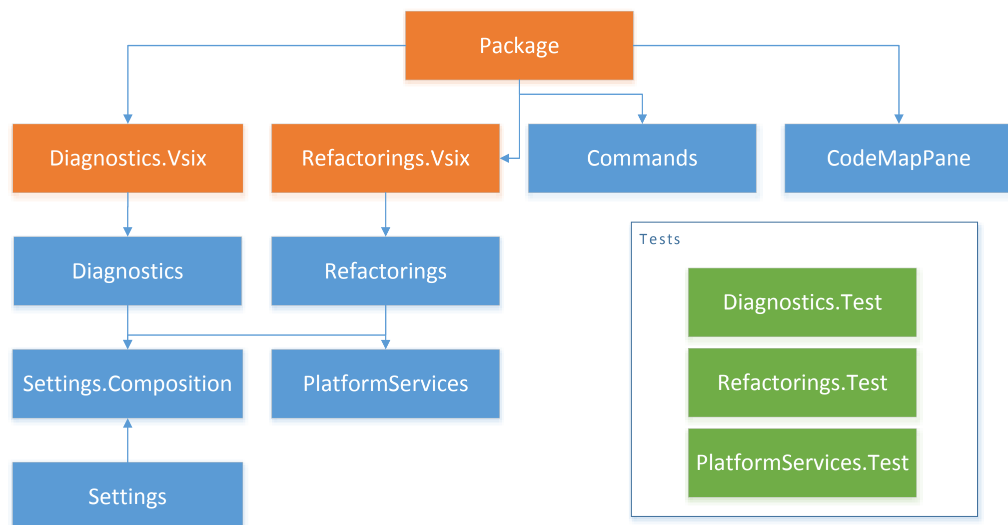


Figure 3.1. Top-level solution overview – an arrow from project A to project B signifies that A uses objects from B (possibly transitively)

Brief overview of these project follow (the projects will be discussed in-depth after this overview):

1. **Main installation package** (*ProjectLuna.2015.Package*)

   The main distribution and installation package which, upon build, contains all the other packages necessary to install the solution output as a Visual Studio plugin.

2. **Code diagnostics package** (*ProjectLuna.2015.Diagnostics.Vsix*)

   The package containing the Roslyn-based diagnostics implemented within this thesis, making them ready to install into the VS IDE and integrate with the native Roslyn instance.

3. **Code refactorings package** (*ProjectLuna.2015.Refactorings.Vsix*)

   The package containing the Roslyn-based code refactorings implemented within this thesis, making them ready to install into the VS IDE and integrate with the native Roslyn instance.

4. **Code diagnostics implementation project** (*ProjectLuna.2015.Diagnostics*)

   The project containing the implementation of the Roslyn-based code diagnostics and fixes. This is a portable class library (PCL) project.

5. **Code refactorings implementation project** (*ProjectLuna.2015.Refactorings*)

   The project containing the implementation of the Roslyn-based refactorings. This is a portable class library (PCL) project.

6. **Unit test project for code diagnostics** (*ProjectLuna.2015.Diagnostics.Test*)

   Contains unit tests covering certain aspects of code diagnostics and fixes. The tests make use of the *PlatformServices* project. These tests are only used in development and are not part of the final ditribution package.

7. **Unit test project for code refactorings** (*ProjectLuna.2015.Refactorings.Test*)

   Contains unit tests covering certain aspects of code refactorings. The tests make use of the *PlatformServices* project. These tests are only used in development and are not part of the final ditribution package.

8. **Platform services project** (*ProjectLuna.2015.PlatformServices*)

   Implements the platform abstraction layer discussed in Section 2.3, “Extensibility” and is referenced by both the *Diagnostics* and *Refactorings* projects, as well as the various unit test projects.

9. **Unit test project for platform service** (*ProjectLuna.2015.PlatformServices.Test*)

   Contains unit tests covering certain parts of the platform abstraction implementation. These tests are only used in development and are not part of the final distribution package.

10. **Settings base classes project** (*ProjectLuna.2015.Settings.Composition*)

    Contains basic classes related to settings management that should be accessible from both portable and non-portable code. This is a portable class library project referenced by both the *Diagnostics* and *Refactorings* projects, as well as the *Settings* projects.

11. **Settings manager project** (*ProjectLuna.2015.Settings*)

    Implements the settings manager discussed in Section 2.5, “Settings”, including the necessary UI components. This is a non-portable project.

12. **Code map project** (*ProjectLuna.2015.CodeMapPane*)

    Implements the code navigation pane that extends the code editor, discussed in Section 2.6, “Code navigation”.

13. **Visual Studio commands package** (*ProjectLuna.2015.Commands*)

    Contains code necessary to extend the VS IDE's user interface to allow the user to interact with the rest of the implementation.

Please note that the names of all the projects included in the solution start with “ProjectLuna.2015.” (named after the code name of the project chosen by the author: “Project Luna”). The following text only refers to the respective projects by the remainder of the name, for brevity's sake.

Solution operation

The toolset implemented in this thesis works by integrating with Visual Studio 2015. The main distribution package (*Package*) is responsible for the integration process itself. Once the toolset is installed, there are two components that perform the user-visible actions of analyzing and transforming source code. These are implemented in the *Diagnostics* and *Refactorings* projects, respectively. The code implemented in these two projects integrates with the Microsoft Roslyn instance running within the host IDE and, when executed by this instance, provides the user with new code diagnostics, fixes, and refactorings in the code editor. The implemented diagnostics are run automatically as the user edits the source code. Code fixes and refactorings are executed manually when the user chooses to perform an action using the provided user interface.

The code transformations implemented in these two projects also rely on the platform abstraction layer implemented in the *PlatformServices* project which includes helper functions that simplify code analysis, manipulation, and emission. Several of the implemented code transformations also require access to the settings manager and therefore make use of the *Settings* and *Settings.Composition* projects as well. These two projects are responsible for loading, storing, and distributing the toolset's settings.

The *Commands* project is also related to the settings manager as it extends the host IDE's user interface with the menu items necessary to bring up the settings window where the user can view and edit settings for the toolset. The *CodeMapPane* projects implements the code navigation pane that extends the code editor to simplify navigation within code files. This project stands relatively separate from the rest as it does not make use of any other project in the solution.

There are three unit test projects present in the solution (*Diagnostics.Test*, *Refactorings.Test*, and *PlatformServices.Test*) that are only used during development and are not part of the actual distribution and operation of the toolset. The VSIX projects (*Diagnostics.Vsix* and *Refactorings.Vsix*) are necessary to properly package and configure the contained assemblies (code diagnostics and refactorings, respectively) to allow their execution by the Roslyn instance.

3.2. Code diagnostic implementation

The project *Diagnostics* implements the core set of code style diagnostics and fixes. This is a Portable Class Library (PCL) project. As mentioned in Section 2.2, “Integrating with Visual Studio and Roslyn”, the Roslyn SDK was used to create and configure this project, including its portability. The project references other parts of the solution, including the platform services and settings manager. There are 24 diagnostics with code fixes implemented in this project.

Diagnostics

All the implemented code diagnostics inherit from the `DiagnosticAnalyzer` class (provided by the Roslyn libraries). This makes it possible for the diagnostics to be executed by the native Visual Studio Roslyn instance. Figure 3.2 illustrates this inheritance hierarchy:

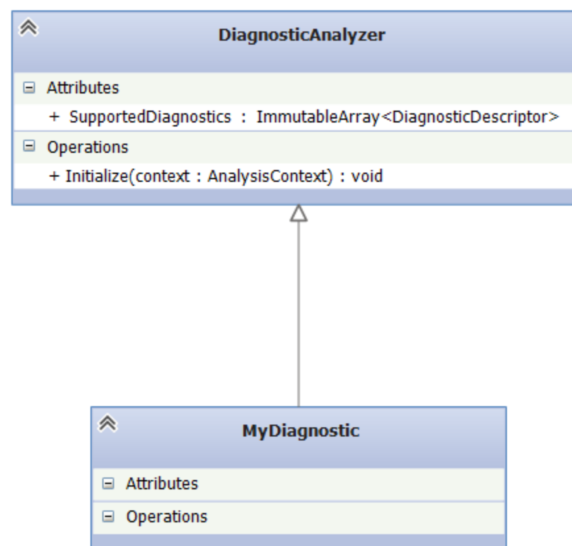


Figure 3.2. Code diagnostic inheritance

The actual diagnostic implementation overrides the declared base class methods. The `SupportedDiagnostics` property describes the diagnostic(s) to the Roslyn API. It is possible for a single diagnotic class to diagnose more than one issue, but the diagnostics implemented in this thesis only diagnose a single code issue.

As an example, let's look at a dignostic that analyzes whether a field is declared as `private` and if not, reports a warning (this is an actual diagnostic implemented in this thesis). The `SupportedDiagnostics` property for this diagnostic first creates a diagnostic descriptor, then returns it:

```
var descriptor = new DiagnosticDescriptor(
  DiagnosticId,
  Description,
  MessageFormat,
  Category,
  DiagnosticSeverity.Warning,
  true);
return ImmutableArray.Create(Rule);
```

The `DiagnosticId`, `Description` etc. are fields containing the necessary values.

After the Roslyn instance discovers that there is a code diagnostic with a description returned by the `SupportedDiagnostics` property, it initializes the diagnostic itself by calling the `Initialize` method. This method is used to tell the analysis engine what kind of code elements is the diagnostic interested in (syntax nodes, semantic model, comments etc.). Most implemented diagnostics work with specific symbol types (such as field symbols). This requirement is signalled to the Roslyn runtime by using the provided *`context`* parameter and its methods `Register*`. The diagnostic implementation calls one or more of these `Register*` methods and provides a method callback that performs the actual code analysis.

In the example diagnostic above, the registration is rather simple: the diagnostic is only interested in field declarations, therefore its `Initialize` method uses the `RegisterSymbolAction` method:

```
context.RegisterSymbolAction(
  AnalyzeSymbol,
  SymbolKind.Field);
```

This call tells the Roslyn instance that whenever a field symbol is encountered during the code analysis, it should call the `AnalyzeSymbol` method on it. The `AnalyzeSymbol` method is the core of the actual code diagnostic and performs the code analysis itself. It receives an argument containing information about the analyzed piece of code, analyzes it, and if an issue is detected, reports this fact back to the Roslyn instance:

```
void AnalyzeSymbol(SymbolAnalysisContext c)
{
  // analysis here
  var diag = (...) // create a diagnostic report
  c.ReportDiagnostic(diag);
}
```

Diagnosed issues reported by code diagnostics are subsequently collected by the Roslyn instance and appropriate user interface is displayed to the user. This mostly consists of highlighting the problematic piece of code and allowing the user to execute a code fix (if one is avaiable for the diagnosed issue).

All the implemented code diagnostics have the same general shape enforced by the `DiagnosticAnalyzer` class inheritance.

Implementation of code diagnostics was also slightly complicated by the composition issue discussed in Section 2.5, “Settings”. Because diagnostics are currently instantiated

by the IDE using reflection, they cannot participate in dependency injection, which prevents them from getting a settings manager reference from the dependency injection container. Therefore a diagnostic required to access user settings currently retrieves the settings manager service from a special service container. This container will be covered in more detail in Section 3.5, “Settings composition”. Code diagnostics also make use of the platform abstraction and helper methods implemented in the *PlatformServices* project.

Code diagnostics implemented within the *Diagnostics* project are all located in the `ProjectLuna.Diagnostics.Diagnostics` namespace and are separated into child namespaces according to their area of interest.

Code fixes

Roslyn-based code fixes implemented for this toolset, similarly to diagnostics, inherit from a specific Roslyn-provided class and are also located in the *Diagnostics* project. For code fixes, this class is called `CodeFixProvider` and the inheritance hierarchy is illustrated in Figure 3.3.

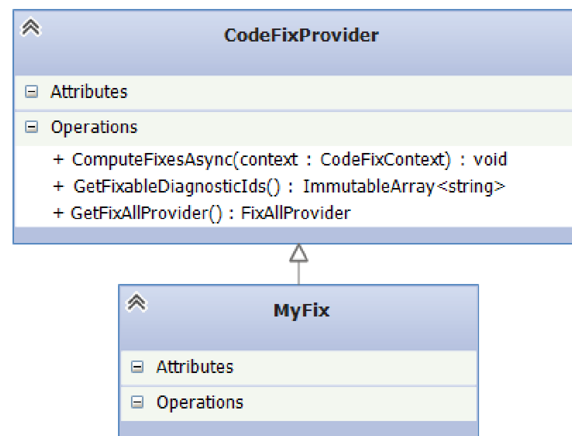


Figure 3.3. Code fix inheritance

The actual code fix implementation overrides the base class members to allow the Roslyn instance to discover and execute the fix. Fixes are only executed when the user explicitely selects to run a code fix in response to an issue previously reported by a code diagnostic.

Code diagnostics may and may not have an associated code fix implemented. To determine what code fix (if any) applies to a given code diagnostic, the Roslyn instance calls the `GetFixableDiagnosticIds`. This method might return an array of IDs in case the fix applies to several different diagnostics, but the code fixes implemented as part of this thesis only ever apply to a single one.

As an example, imagine a code fix that applies to the code diagnostic example mentioned above (check whether a field is declared `private`). This code fix takes a field with access modifier other than `private` and turns this access modifier into `private`. To tell the Roslyn instance executing the code analysis that this code fix applies to the a forementioned diagnostic, the `GetFixableDiagnosticIds` has a body like this:

```
return ImmutableArray.Create("FieldDiagnosticId");
```

If this diagnostic exists and it evaluates a piece of code as having issues, the IDE automatically discovers that there is a code fix for this issue and allows the user to execute this fix.

If the user wants to see the fixes available for the diagnosed issue, the Roslyn instance calls the `ComputeFixesAsync` on the fix, passing information about the diagnosed issue as a parameter. This method is responsible for analyzing the diagnosed issue and informing the Roslyn instance about the code transformation that fixes the issue. This information also includes the method callback that should be executed when the fix is actually applied. The execution of a code fix ends when this callback return control back to the Roslyn runtime.

The `CodeFixProvider` also declares a `GetFixAllProvider` method. This method is called when the user decides to “Fix all” diagnosed issues. However, the code fixes implemented in this thesis do not implement a custom FixAll provider and instead use fallback to a default FixAll implementation provided by the Roslyn libraries.

Some implemented code fixes require access to the settings manager implemented in the toolset. Because code fixes compose properly using the Visual Studio MEF dependency injection container (discussed in Section 2.5, “Settings”), these code fixes are decorated with the `ExportCodeFixProviderAttribute` attribute. This attribute marks the class as MEF-composable and ensures that any required dependencies will be injected by the DI container. This attribute is also used to specify the target language for the fix, which is C# for this thesis. Example:

```
[ExportCodeFixProvider(
  FieldAccessiblityAnalyzer.DiagnosticId,
  LanguageNames.CSharp)]
```

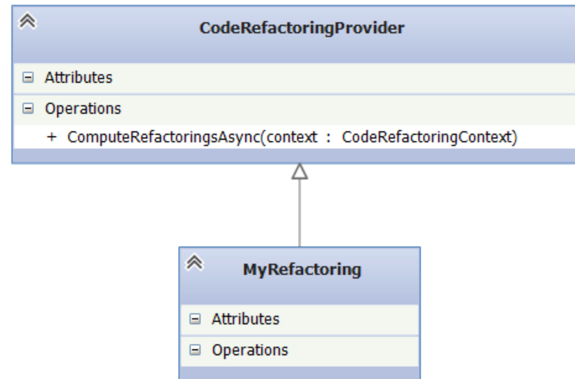
All code fixes implemented as part of this thesis use to same basic concepts of overriding the `GetFixableDiagnosticIds` and `ComputeFixesAsync` methods, they only differ in their implementation of these methods. They also make use of the platform abstraction and helper methods implemented in the *PlatformServices* project.

Code fixes implemented in the *Diagnostics* project are located in the `ProjectLuna.Diagnostics.Fixes` namespace and are separated into child namespaces according to their area of interest.

3.3. Refactoring implementation

Code refactorings implemented for the toolset discussed in this thesis are located in the *Refactorings* project. Just like the diagnostics project, the refactoring project was created and configured by the Roslyn SDK and is a portable project. It also references the platform services and setttings manager projects. This project implements 16 different refactorings in total.

Similarly to code diagnostics and fixes, all implemented refactorings inherit from a specific Roslyn-provided base class, `CodeRefactoringProvider`:



Code refactoring inheritance

The actual implemented refactoring overrides the base `ComputeRefactoringsAsync` method. When the user decides to refactor a piece of code, the Visual Studio IDE runs all the available refactorings against this piece of code by invoking their `ComputeRefactoringsAsync` method and providing the code in question in the *`context`* parameter.

The method's implementation then takes the provided parameter and analyzes the code. When a possible refactoring is available for the given piece of source code, the method reports this fact to the Roslyn instance executing the refactoring. This includes the callbackmethod that should be executed to perform the refactoring itself.

The Roslyn runtime collects results for all the available refactorings and displays them to the user in the form of a contextual icon with a dropdown menu. When the user decides to apply the refactoring, the callback method is executed, performing the actual code transformation. Execution of a refactoring ends when this callback method returns.

All refactorings implemented as part of this thesis follow the same execution workflow, they only differ in the implementation of the `ComputeRefactoringsAsync` method.

To allow for MEF-based composition and dependency injection, the code refactorings classes are decorated with the `ExportCodeRefactoringProviderAttribute` attribute. This allows the code refactoring to import the settings manager using DI, similarly to code fixes – refactorings compose properly using the MEF-based composition of Visual Studio. Refactorings also make use of the platform abstraction and helper methods implemented in the *PlatformServices* project.

3.4. Platform abstraction

The *PlatformServices* project implements the plarform abstraction discussed mainly in Section 2.3, “Extensibility”. Code implemented here contains mostly various useful helper methods for manipulating code elements, facilities for easy code emission, and methods that simplify testing. The helpers are used mostly from code diagnostics and refactorings (projects *Diagnostics* and *Refactorings*).

Helper methods

The helper methods for manipulating code contain a varied set of operations, from analyzing string casing to custom code to insert members into classes. These methods were implemented over time as the need for them was discovered. There are several areas covered by these helper methods:

- **Documentation comments** (`Comments.cs`)

  Documentation comments are improtant part of a readable code base and the toolset implemented in this thesis puts great emphasis on enforcing them. Helpers in this file focus on detecting and adding documentation comments to code elements without them. The implementation makes use of the Roslyn APIs to both check whether a code element has a documentation comment and for adding new comments. The helper methods don't simply create an empty skeleton comments but try to fill the content also. Instead of

  ```
  /// <summary>
  ///
  /// </summary>
  ```

  the helper methods also create the summary description (example for a constructor documentation comment):

  ```
  /// <summary>
  /// Initializes a new instance
  /// of the <see cref="MyClass"/> class.
  /// </summary>
  ```

  The actual content depends on the type of code element being commented and is usually based on the code element's name.

- **Miscellaneous** (`Helpers.cs`)

  This source file contains helper methods with no explicit categorization. There are currently two public methods implemented in this file – one for analyzing symbol inheritance (`IsDerivedFrom`) and one for analyzing the type of exception thrown by a `throw` statement (`GetExceptionType`). The first method uses Roslyn's Symbol API to walk the inheritance hierarchy and matching the target type name. The second method uses the Syntax Node API to find the exception declaration and return the exception's type name.

- **Member positioning** (`Positioning.cs`)

  Positions of members within structs and classes should follow certain basic guidelines to improve code readability. The `FindPosition` method returns the appropriate position for a code element. The order of precedence of code elements used in this method is as follows: fields, enums, constructors, event fields, events, properties, methods, classes, structs.

- **Member injector** (MemberInjector.cs)

  The member injector implementation (InsertDeclaration method) serves to insert new code elements into classes and structs at their appropriate positions. It uses the member positioning helper methods to find the proper position for the element being inserted and inserts it at that position using the Syntax Node API.

- **Naming** (Naming.cs)

  There are several helper methods for analyzing and manipulating code element names implemented in this source file. They mostly deal with proper casing (PascalCase and camelCase) and are used for stylistic validation in code style diagnostics. However, to manipulate casing, names must first be split into words they consist of. Example:

  Names _my_class and MyClass both consist of the same words "my" and "class" but their form is different. The helper methods implemented in Naming.cs attept to parse the actual words out of the original string. This parsing is done by iterating over the original string and splitting it in meaningless characters (e.g. "_") and character case change (e.g. when the "C" is encountered in MyClass).

  Numbers within strings are not considered word breaking and are evaluated as part of the previous word: MyClass3 is parsed into "My" and "Class3".

  After the original string is partitioned into words, the case-manipulating methods then work on these words by changing the case of the words: PascalCase to MyClass and camelCase to myClass.

  For the full list of implemented methods and details about their operation see the programmer's documentation.

- **Property manipulation** (Properties.cs)

  This file implements the code necessary to analyze declared properties (IsAutoProperty, HasGetter, HasSetter, FindGetter, FindSetter) and transform them (ExpandAutoProperty). These methods are implemented using Roslyn's Syntax Node API and work by analyzing the property declaration syntax tree.

  These methods are used from code diagnostics, fixes, and refactorings.

- **Inserting usings** (UsingDeclarations.cs)

  Inserting using directives (not to be confused with using statements) is performed using the Syntax Node API. Usings already present in the code file are collected and new using directives are added.

  The InsertUsings method also automatically puts the usings *inside* the namespace declaration they belong to and sorts them by name, as recommended by the C# code style guidelines.

This list of helper methods implemented in the platform abstraction reflects the current state of implmentation. Future work might see more methods added to the list as the need for them arises.

Most of the public methods contained in the abstraction layer are implemented as *extension* methods. This makes their invocation much more convenient and mimics the behavior of certain Roslyn APIs. Consider the folowing code:

```
if (property.IsAutoProperty())
{
  ...
}
```

This way of invocation is more fluent and its meaning slightly more obvious than the standard static invocation:

```
if (Properties.IsAutoProperty(property))
{
  ...
}
```

This thesis uses extension methods wherever possible, to make the code using these methods more readable.

Code emission

The platform services project implements extensive facilities for simplifying code emission. These facilities are located in the `ProjectLuna.PlatformServices.Synthesis` namespace.

Emission of all the standard code elements is supported: namesapces, classes, structs, fields, properties, and attributes. It is also possible to emit `if`, `using`, and `foreach` statements. The API implemented in this project is not comprehensive–it focuses mainly of performing the basic code emission tasks and there might be situations where a desired code cannot be emitted by it (e.g. emitting `throw` statements is not supported). In such a case the standard Roslyn APIs are used.

It is also important to note that the code emission API was designed and implemented relatively late in the implementation process, therefore not many code transformations use it. It was designed more for use from plugins.

Code emission using the platform services APIs starts with the `Luna` class and uses fluent form.

As an example, the following code emits a simple field declaration:

```
var field =
  Luna.Field("test")
    .Public()
    .Type("int")
    .ToNode();
```

The resulting code element from such a call would look like this:

```
public int test;
```

The fluency of the API is achieved by passing a *context* between the API calls. This context accumulates information about the requested operations and the actual code element is only created at the end of the call chain, when all the necessary information is known. In the example above, a `FieldContext` class is instantiated by the call to `Luna.Field(...)`. The `FieldContext` contains methods that manipulate its state, such as the `Public` method, which modifies the context's state to indicate that the resulting field should be public. The final `ToNode` call then takes the state of the context and constructs a new `FieldDeclarationSyntax` node that can be further consumed by Roslyn.

Emission of every code element is achieved using the same approach, but the capabilities of different emission contexts might differ, depending on the code element they emit. It is also worth noting that the constructors of the emission context classes are `internal` to force users to use the `Luna` class to access the emission API. This requirement is in place mainly to prevent users from potentially misuing the emission API.

Testing facilities

The abstraction layer also implements classes and methods that help with testing of code diagnostics, fixes, and refactorings. These were implemented in response to this toolset's need for debugging and verifying the behavior of the implemented code transformations. The testing facilities are located in the `ProjectLuna.PlatformServices.Testing` namespace.

The `AnalyzerTestBase` class is used to simplify instantiation of the code analytic and fix that are being tested. Both analytics and fixes tend to have long class names (e.g. `CommentsMustBeginWithWhitespaceAnalyzer`) and writing `new MyAwesomeLongAnalyzerName` became bothersome fast. The `AnalyzerTestBase` contains two properties that instantiate the diagnostic and code fix respectively and then hold the instances for later use. Instead of calling

```
var analyzer =
  new CommentsMustBeginWithWhitespaceAnalyzer();
```

it is possible to use the folloing code:

```
this.Analyzer
```

Both the `Analyzer` and `Fix` properties are lazy, meaning they are initialized at their first usage, not with the class constructor. All the test classes in the *Diagnostics.Test* project inherit from this class.

The `CodeAnalyzerTesting` class implements extension methods that make testing code diagnostics easier. There are methods that simplify test project creation (test project means a “code base” against which the diagnostic is run and tested), verification of diagnostic result, application of code fixes, and verification of the fix. Using the helper methods implemented in this class, unit tests for code diagnostics are written like this:

```
const string Code = "(SOURCE CODE)";
Code.Project()
  .VerifyDiagnosticsCount(this.Analyzer, 1)
  .Doc(0)
  .Fixed(this.Analyzer, this.Fix)
  .Verify(
      document => { (VERIFICATION) });
```

This code again resembles a fluent syntax, this time achieved by using extension methods on existing Roslyn classes (such as `Document`). It is worth noting that not all the diagnostic tests are written in this form. This is caused by the fact that some tests are older than the testing API and were therefore written using only the base Roslyn facilities. The most interesting parts in this call chain are execution of a code diagnostic and application of a code fix.

To achieve the desired effect, the implementation mimics the behavior of the Roslyn code analysis engine – at the beginning, diagnostics are computed over the provided code file using the `AnalyzerDriver` Roslyn class. Results (diagnosed issues) are collected and ordered by their position in the code file (this is performed by the `GetDiagnostics` method).

After diagnostics are computed, corresponding fixes are applied. The API creates a `CodeFixContext` and passes it to the provided code fix class. Results of the code fix operation are accumulated and applied to the code file being analyzed. This is performed by the `Fixed` method.

Helpers that simplify testing of refactorings follow very similar concepts. The `RefactoringTestBase` class fulfills the same role for refactorings as the `AnalyzerTestBase` does for code diagnostics – it simplifies instantiation and improves readability by hiding the long class names behind a property called `Refactoring`.

The `RefactoringTesting` is analogous in purpose to the `CodeAnalyzerTesting` class. It makes writing unit tests for refactorings easier by providing extension methods that allow writing of the following test code:

```
const string Code = "(SOURCE CODE)";
Code.Doc()
    .Context(node => (SELECTOR))
    .GetRefactorings(this.Refactoring)
    .VerifyActionCount(1)
    .Fix()
    .Verify(solution => (VERIFICATION));
```

The tested refactoring is executed by the test in a way similar to what Roslyn does in Visual Studio. A refactoring context is created in the `Context` method. The tested refactoring is then executed against this context and the results are collected. The `VerifyActionCount` method verifies that the number of available refactorings corresponds to what is expected. Subsequently, the refactoring is applied to the code file.

This is where is was necessary to slightly bend the rules of the Roslyn APIs – the method required to access the post-refactoring code file (`CodeAction.GetChangedSolutionAsync` method) is declared `internal` in the Roslyn version this thesis uses, preventing the test framework from actually seeing the result of the refactoring. The `Fix` method therefore uses reflection to invoke the necessary methods, bypassing the `internal` access modifier entirely. This is a very fragile concept and it might easily break with the next version of Roslyn, but it works for now. Alternative solution will be implemented in case of problems.

After the fix is applied, verification of the refactored code occurs in the `Verify` method.

3.5. Settings composition

The *Settings.Composition* is a Portable Class Library storing the interfaces and classes necessary to access the settings manager. Its portability allows the implemented code to be accessed from both code transformations (which are portable) and the settings manager (which is non-portable).

The `ISettingsProvider` interface is used to abstract the actual non-portable settings manager for use in portable classes. Since this interface is declared in a portable library, even portable code can use it. By using the dependency injection process described in Section 2.5, “Settings”, the non-portable settings manager implementing this interface is injected into the portable classes as a dependency. This effectively bypasses the limitation that portable code cannot call non-portable code.

The `ServiceContainer` class serves as a replacement for dependency injection where DI is not available (code diagnostics). It is a very simple container class that stores references to “services”. The settings manager registers itself in this container when settings are first loaded and code diagnostics that require access to this settings manager retrieve the settings manager instance from it. The `ServiceContainer` is portable and static and all services registered within are singletons.

3.6. Settings

The *Settings* project implements the core of the settings manager. This project is non-portable and builds upon the *Settings.Composition* project by using its classes and interfaces.

Settings are stored as XML files and accessed using LINQ-to-XML technology. The `SettingsSubsystem` class is reponsible for loading and storing the settings files. Settings files are also implemented as hierarchical – the manager walks the folder structure from root to the currently loaded solution folder and looks for settings files. The discovered files are then merged into a single XML document from the inside out – the settings “furthest” from the solution folder have the lowest priority, the settings “closest” to it override any settings that were declared in the files further up the folder tree.

Settings are internally divided into “sections”, which serve to group related settings and provide a more granular ways to access them. Sections are identified by GUIDs to minimize the chance of collision.

The `SettingsProvider` class is responsible for reading the loaded XML settings file and parsing the settings into an object model. It also implements the

ISettingsProvider from the *Settings.Composition* project. This class is marked as a MEF export and participates in dependency injection, hidden behind the interface. Instances of this class are injected into portable classes that require access to the settings manager and are also registered in the ServiceContainer class mentioned above. By hiding behind a portable interface this class serves to distribute settings to where they are needed.

The ISettingsEditor interface is used to identify classes that serve as settings editors. Settings editors are discovered using MEF composition by the Settings window and displayed to the user. The core toolset implements this interface in the GlobalEditor class which serves as an editor for the core implementation's settings.

3.7. Visual Studio commands

The *Commands* project is responsible for extending the Visual Studio user interface with elements that allow the user to interact with the implemented toolset. This currently means providing a single menu item to open the settings window. The constructor of the main class is called when a solution is loaded in the IDE and is therefore also responsible for initializing and release of the settings manager.

This project was created using the Visual Studio Package project template with support for commands, which generated a basic skeleton code for adding new commands to the Visual Studio IDE.

There are three important aspects to this project: command definition, command registration, and the settings window.

Command definition

Command definitions specify the visual aspect of commands – where the IDE should display them and how the commands look (text, tooltip, icon etc.). Commands definitions are created using XML and are located in the ProjectLuna.2015.Commands.vsct file.

Visual Studio commands are organized into groups, therefore the VSCT file defines a new command group:

```
<Group
  guid="guidProjectLuna_2015_CommandsCmdSet"
  id="MyMenuGroup"
  priority="0x0600">
  <Parent
    guid="guidSHLMainMenu"
    id="IDM_VS_MENU_TOOLS"/>
</Group>
```

The definition specifies the group's GUID, ID, and priority within the parent menu. By providing the *Parent* element, it is also requested that the group be displayed as part of the “Tools” Visual Studio menu. The “Settings” menu item is defined as a button in the aforementioned group.

Command registration

Defining the necessary XML elements allows Visual Studio to discover the commands but they don't define the menu's behavior when activated (clicked). This is perfromed in the `ProjectLuna.2015.CommandsPackage.cs` file, in the `Initialize` method by retrieving the IDE's command service and registering the menu item with it.

A method callback is specified in this registration and is invoked when the user activates the command. This method constucts and displays the settings window used to view and edit settings for the implemented toolset and its plugins.

Settings window

This window is used to manage the available settings both for the core toolset and third-party plugins. This window is displayed to the user by clicking a command registered into the Tools menu (discussed above).

The window is a standard WPF window that uses bindings and templates to render its user interface (the standard MVVM pattern is employed). The main feature of the window is the ability to discover all the individual settings pages belonging to the core implementation and plugins and display them. This is performed by using the Visual Studio's composition service:

```
var componentModel =
  Package.GetGlobalService(typeof(SComponentModel))
  as IComponentModel;
this.editors =
  componentModel.GetExtensions<ISettingsEditor>();
```

The discovered `ISettingsEditor` instances are then stored in a backing field of the `Editors` property and displayed in the window's user interface through binding and templating.

This window's `Save` button uses the settings manager API to save the current settings by calling the `SettingsSubsystem.SaveSettings` method.

3.8. Navigation pane

The *CodeMapPane* project implements the code navigation pane that extends the code editor with the ability to easily navigate between code members by clicking. This is a Visual Studio Package project (VSIX) that integrates with the Visual Studio code editor using the VSP API.

This integration is achieved by implementing a custom editor adornment. This adornment is capable of drawing WPF controls over the code editor surface, displaying the navigation pane. The adornment is instantiated by a custom factory class implementing the `IWpfTextViewCreationListener`. This interface's `TextViewCreated` method is called by the IDE whenever a code editor window is being opened. The factory creates a new code map pane and attaches it to the code editor in this method.

The `CodeMapPane` class represents the adornment itself. This class is responsible for instantiating the WPF user control containing the code map's user interface and drawing

it over the code editor. This includes any potential repositioning resulting from window resize and similar events.

The `NavigationPane` user control implements the user interface for the code pane that is displayed to the user. It's a standard WPF user control that uses MVVM (model-view-viewmodel) pattern to display information and interact with the “business logic” of the code map pane. The actual logic responsible for scanning the source code file for changes is implemented in the `NavigationPaneViewModel` class.

This class is instantiated by the `CodeMapPane` class and passed to the user control as its `DataContext`. It implements the `INotifyPropertyChanged` which allows it to inform the view (user control) that a change has happenend in one of its properties, enforcing the view to redraw its user interface to reflect the change.

The viewmodel also hooks the `Changed` event of the text buffer belonging to the code editor the code map is attached to. This allows the code map to react to changes in the edited source code by calling the Roslyn APIs and parsing the content of the file. Signifficant syntax nodes (namespaces, classes, properties etc.) are selected from the file, transformed from the relatively heavy Roslyn objects into lightweight object model, and stored in a property. Upon change, this property notifies the user control that its value has changed and the user interface is redrawn to reflect the modified code.

It is important to note that the code map does not use the native Roslyn instance present in Visual Studio to parse the code, instead it references the Roslyn libraries and calls the syntax parser directly (this behavior is explained in more depth in Section 2.6, “Code navigation”):

```
var tree = CSharpSyntaxTree.ParseText(...);
```

The analysis of code members is then performed on the resulting syntax tree.

3.9. Main installation package

The main installation package is a Visual Studio Package project used to bundle all the other components of the implementation into a single installation file. This package file can subsequently be used to install the implemented toolset into a supported version of Visual Studio.

This project does not contain any code, everything is configured using the package's manifest file (`source.extension.vsixmanifest`) located in the project root.

The manifest file declares six “assets” (an asset is basically an external component that should be included in the package)–four VS packages containing the implementation components (diagnostics, refactorings, commands, and code map) and two MEF component declarations (settings manager and base classes). This MEF declaration is important to ensure that the components participate in dependency injection using the Visual Studio MEF framework (discussed in Section 2.2, “Integrating with Visual Studio and Roslyn” and Section 2.5, “Settings”).

3.10. Code diagnostics package

Code diagnostics package is a Visual Studio Package project responsible for installing the code diagnostics and fixes. This project, just like the main package, does not contain

any code. It's manifest file (`source.extension.vsixmanifest`) is configured to include the output of the *Diagnostics* project both as an asset and as a MEF component to ensure its participation in dependency injection.

This package is not installed directly, its part of the main installation package which is also responsible for installing it.

3.11. Code refactorings package

Code refactoring package is a Visual Studio Package project responsible for installing the implemented code refactorings. This project, just like the main package, does not contain any code. It's manifest file (`source.extension.vsixmanifest`) is configured to include the output of the *Refactorings* project both as an asset and as a MEF component to ensure its participation in dependency injection.

This package is not installed directly, its part of the main installation package which is also responsible for installing it.

3.12. Diagnostics unit tests

The project *Diagnostics.Test* contains unit tests for the implemented diagnostics. These tests were used during development to make sure the code behaves as expected without the need to run a debugging instance of Visual Studio. This project references both the platform abstraction project and the settings manager.

The tests are written for the MS Test testing framework which is a native part of Visual Studio and they make use of custom-built testing facilities implemented in the platform abstraction layer. This mainly means that the unit tests classes inherit from the `AnalyzerTestBase` class which simplifies access to the tested code diagnostic and fix and several helper methods that simplify creation of a test code base, executing the diagnostic, and applying the code fix for analysis (these are discussed in detail in Section 3.4, "Platform abstraction").

The tests implemented in this project are far from comprehensive. Due to time constraints it was not possible to dedicate enough time to test development to allow for optimal test coverage of the implemented code. Therefore the tests mainly illustrate the concept of testing code diagnostics and fixes and the use of the custom testing facilities and serve as a base for future work.

3.13. Refactorings unit tests

Unit tests for the implemented code refactorings are located in the *Refactorings.Test* project. This project also references the platform abstraction project to access the custom-built testing facilities implemented in it.

Refactoring unit tests are written for the MS Test testing framework and make extensinve use of custom testing facilites implemented in the platform abstraction later. All refactoring tests inherit from the `RefactoringTestBase` class which simplifies the process of instantiating the tested refactoring class.

The tests themselves then work by first creating test code base from strings, running the refactorings on this test code base, and verifying the analysis result. If the analysis result

is correct, refactoring transformation is applied and the result of this transformation is verified. APIs implemented in the platform abstraction layer are used for all of these steps and will are discusses more in Section 3.4, “Platform abstraction”.

Time constraints did not allow to cover all the implemented refactoring code with tests but the existing tests serve as a viable proof of concept and base for future work.

3.14. Platform services unit tests

The *PlatformServices.Test* project contains unit tests targeting the *PlatformServices* project. The implemented tests are writtent for the MS Test testing framework and focus on testing the code emission API. The tests rely only on the testing facilities provided by the test framework (`Assert.IsNotNull` etc.), there are no custom testing facilities used in these tests.

Time constraints did not allow for extensive test coverage of the *PlatformServices* project and so this project remains open for future work.

4. Comparison with similar applications

This chapter discusses application functionally similar to the toolset implemented as part of this thesis, how they differ from it, and how this thesis addresses their shortcomings.

4.1. JetBrains ReSharper + StyleCop

ReSharper by JetBrains is probably the most widely-used “productivity tool” for Visual Studio on the planet. It implements a large number of advanced refactorings and code transformations and supports additional languages and frameworks, such as XAML, Visual Basic.NET, and TypeScript. ReSharper also supports extensibility using plugins.

However, these extensive features come at a price: ReSharper is rather expensive (costing approximately 150€) and its many features consume a lot of system resources, slowing down the IDE noticeably.

How this thesis differs

The toolset implemented in this thesis is nowhere near ReSharper, feature-wise. The number of implemented code transformations is very small in comparison and this toolset only support the C# language.

However, ReSharper contains very limited abilities when it comes to validating code style. This feature has been long supported by installing the StyleCop extension (which integrated with ReSharper), but since StyleCop has been discontinued, this is no longer an option. The code style validations currently implemented in this thesis focus on the most important aspects of code style, but this could easily be extended to rival (or even surpass) StyleCop's abilities.

This toolset also shows better performance characteristics than ReSharper. Solution load with the toolset installed feels approximately as fast as it was before the toolset's installation. Code diagnostics are running smoothly and don't hinder the user's workflow, there is virtually no lag involved. Comapred to ReSharper, the toolset's operation feels much smoother.

The last point of comparison is price – this toolset, if it ever becomes publicly available, builds upon open-source technologies and frameworks and would be also available as an open-source project, making it freely available to anyone. This makes the toolset much more accessible to users.

4.2. DevExpress CodeRush

CodeRush is a direct competitor to ReSharper, with similar features, except CodeRush is more focused on manipulating C# and VB.NET code, there is no support for XAML, HTML, JavaScript or other technologies that ReSharper supports.

However, CodeRush's main focus is code refactoring, there is very little ability to check or improve code style. On this front, CodeRush looks even less capable than ReSharper

because StyleCop was never available as a CodeRush plugin. CodeRush's performance seems to be slightly better than ReSharper's.

CodeRush's price-tag is even heavier than ReSharper's – at $250 (or 234€) for a single license only professional software developers might even consider purchasing it.

How this thesis differs

CodeRush implements a much more comprehensive set of refactorings than the implemented toolset. However, CodeRush's lack of code style checking capabilities (such as monitoring documentation comments) still makes the implemented toolset useful, even used id tandem with CodeRush.

Performance-wise, usage of Roslyn and smaller size give the implemented toolset a noticeable advantage. Since this toolset does not require any code base preprocessing (CodeRush does), the solution load times are virtually unaffected by the toolset's installation.

4.3. Conclusion

After evaluating the two most popular coding productivity tools available for Visual Studio, it is obvious that they both offer a much broader spectrum of features than what is implemented in this thesis. However, both of these tools cost a rather large amount of money and have noticable performance impact on the host IDE. Their features also lack the focus on code style validation.

The implemented solution offers a smaller set of features, but these features are focused mainly on managing code style and providing several other code transformation that the paid tools omit. This implementation's integration with the Roslyn technology makes it unique among the evaluated tools and translates to lessened performance degradation.

Overall, the implemented toolset's price (free), focus on areas where existing tools fall short (code style), and minimal performance overhead make this tool a viable addition to other, more established code refactoring and productivity tools.

5. Conclusion

5.1. Fulfillment of thesis goals

This thesis' original aim was to implement a toolset that would provide code style validation, refactoring, and improved code navigation for C# code. This toolset was supposed to integrate seamlesly with Visual Studio without hindering performance and allow its abilities to be extended using third-party plugins.

After evaluating the final implementation result and comparing it with similar applications, it is safe to say that the goals have been met to an acceptable degree. The implemented toolset allows powerful C# code analysis and manipulation using a new technology directly from Microsoft and implements 24 code style checks and 16 refactorings. The implemented code map enables the user to easily navigate among various member declarations in a file with a single click and responds in timely fashion to changes in the code file.

The Visual Studio integration has been achieved and its performance impact is very low, compared to alternative tools. The plugin architecture, while partially relying on Visual Studio's native capabilities, allows third-party plugins to make use of implemented services and integrate with the toolset's settings manager.

5.2. Future development

The implementation of this thesis' toolset fulfills the thesis goals, but is far from finished. There are several areas where future work should focus in order to improve capabilities and bring the implementation closed to a publicly releasable state:

- **Code style rules and refactorings**

  The currently implemented code style validation rules only cover a small part of what the C# code style guide proposes. This coverage should be improved by implementing additional rules and related fixes. New refactorings could also be introduced to help improve productivity.

- **Improvements to platform services**

  The platform service library should also be extended to cover a larger portion of the Roslyn APIs, thus simplifying their usage. Areas such as manipulating method bodies and arguments, analyzing LINQ expressions, and support for C# 6.0 features should be implemented.

- **User interface**

  The user interface for the settings manager and code pane is usable, but it could be improved upon to provide better experience for the suer. This includes support for different color schemes (or “themes”) supported by Visual Studio and introduction of icons and images instead of text to make orientation easier.

- **Stability improvements and bugfixes**

  While the current implementation tries to be as defensive as possible when it comes to various edge cases in the source code file, it is difficult to predict all the scenarios where a code manipulation might fail. Future work should invest more effort into testing and verification to provide the best possible stability of the toolset.

Appendix A. User Manual

This chapter covers the installation and basic operation of the implemented toolset from the end user's point of view.

A.1. Installation

System requirements

- Operating system: Windows 7 SP1 or newer, both x86 and x64 based
- .NET Framework version: 4.6 preview
- Hard drive space: 25 MB
- Additional software: Visual Studio 2015 Preview or newer

Installation process

The Visual Studio Refactoring and Code Style Management Toolset is distributed in the form of a Visual Studio Package file. This package is called `ProjectLuna.2015.Package.vsix` and supports automatic installation into Visual Studio 2015 Preview or newer. Unfortunately, the VSIX package only supports installation into the main Visual Studio settings hive, which has Roslyn disabled, rendering the whole toolset inoperable. This behavior should be corrected with the release of the final version of VS 2015, for which the VSIX package should behave correctly.

To install the toolset into the "Roslyn" settings hive which has Roslyn enabled by default, follow these steps:

1. Make sure that Visual Studio has been run with the necessary root suffix at least once. This is necessary because it creates the folder structure required by the next steps of the installation process. If you are not sure, simply run the following command line command (`devenv.exe` is the main Visual Studio 2015 executable):

   devenv.exe /rootsuffix Roslyn

   After the IDE starts, close it again and proceed to the next step.

2. Find the `Installation.zip` attachment and navigate to the extensions folder for the Roslyn root suffix instance. This folder is usually located on the following path:

   ```
   C:\Users\[user]\AppData\Local\Microsoft\VisualStudio
   \14.0Roslyn\Extensions\
   ```

3. Extract the `Installation.zip` into this folder. It should create a `Marek Linka` folder containing the actual distribution files.

4. To make Visual Studio "notice" the installed packages, navigate to the following path:

   ```
   [VS2015 installation path]\Common7\IDE\Extensions
   ```

Open the `extensions.configurationchanged` file located in this folder in any text editor and save it – it is only necessary to change the file's edit timestamp. This will force Visual Studio to refresh its extension cache upon next start.

Performing these steps should ensure that the toolset is properly installed and registered in the Roslyn-enabled instance of Visual Studio. To check this, simply run **devenv.exe /rootsuffix Roslyn** and bring up the Extensions window. Figure A.1 shows the Extensions window with the installed components.

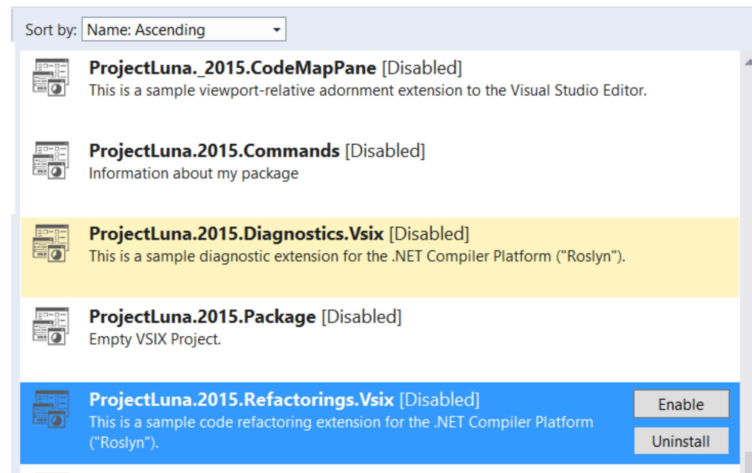


Figure A.1. Post-installation Extension manager

The installation process might leave the toolset's components installed but disabled, in which case simply manually enable them and restart the IDE (don't forget the “Roslyn” root suffix).

The installation process is rather complicated due to the fact that VS 2015 is currently still a preview and that Roslyn is only enabled by a special root suffix. Once the IDE is released in final form, Roslyn will be enabled by default and the installation will only rely on the VSIX package, no manual configuration will be necessary.

Uninstallation of the toolset is possible using the Extensions window in a standard way.

A.2. Usage

Performing code transformations

Once the installation process is complete, the implemented diagnostics and facilities should be available in any C# code. Code diagnostics are executed as the user types, highlighting any discovered issues immediately using “squiggly lines”, illustrated in Figure A.2.

```
0 references
public int test { get; set; }
```

Figure A.2. Code breaking a stylistic rule

Placing the caret into the highlighted text brings up a context-sensitive icon at the left side of the code editor. Clicking this icon brings up additional information about the diagnosed issue, including the ability to perform a fix. Figure A.3 shows an example of the user interface rendered for this task.

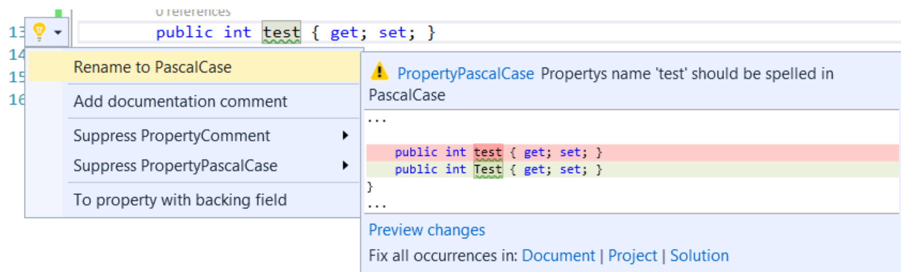


Figure A.3. Application of a code fix (including a preview of the proposed change)

Applying a fix subsequently transforms the code. It is possible to revert the transformation by the *Undo* command (bound to **Ctrl**+**Z** by default).

Code refactorings are not evaluated automatically by the IDE, it is necessary to trigger them manually. This action is available from the code editor's context menu (shown in Figure A.4) and is also bound to a keyboard shortcut (**Ctrl**+**.** by default):

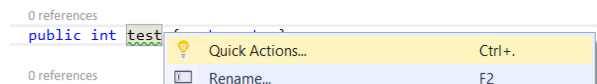


Figure A.4. Quick actions context menu item

Opening the quick fix menu will bring up the context sensitive menu at the left side of the code editor. This menu will be populated with the currently available code actions, including refactorings. The UI for applying a code refactoring is shown in Figure A.5:

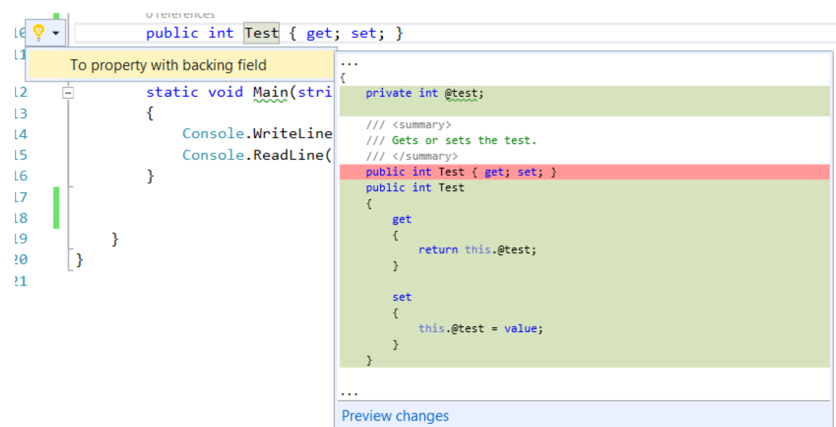


Figure A.5. Application of a refactoring fix (including a preview of the proposed change)

Clicking the refactoring will perform the transformation (as demonstrated in the preview pane). This action is revertable by using the *Undo* command.

Code map

The code map is automatically displayed at the right side of the code editor window. Figure A.6 shows the code map's UI.

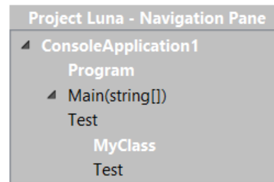


Figure A.6. The code map

The code map reflects declared members of the code file, including namespaces, classes, properties, constructors, and methods (including their signatures). These declarations are displayed hierarchivaly to reflect their nesting and parents can be collapsed to save space. The header of the code map is clickable and doing so hides the entire code map from view, as illustrated by Figure A.7.

Figure A.7. Code map in collapsed state

Clicking the header again will expand the code map back into full view.

Managing settings

The toolset's settings are accessed using the *Project Luna Settings* menu item located in the *Tools* menu, as shown in Figure A.8.

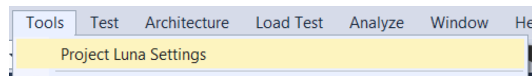


Figure A.8. Settings menu item

Clicking this menu item brings up the settings manager window, shown in Figure A.9.

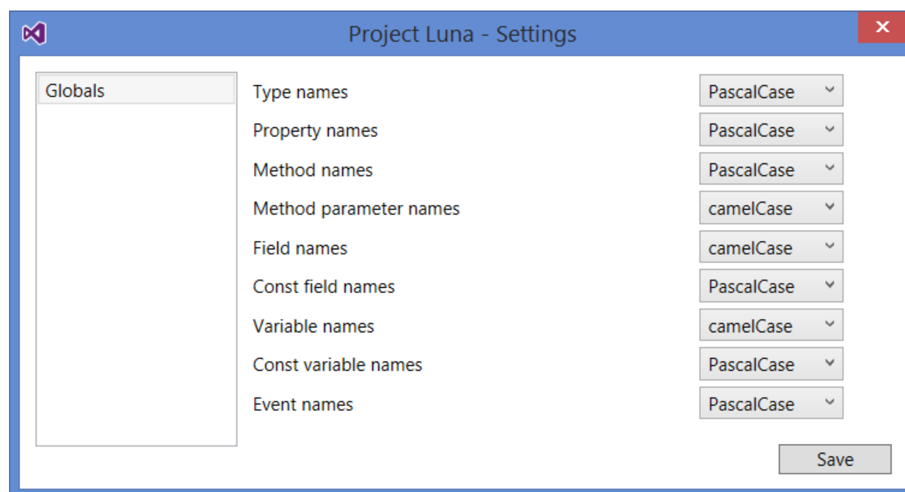


Figure A.9. The settings window

The left side of the window displays all the available setting *sections*. Sections are used to organize settings into smaller groups by their focus to simplify orientation. Plugins might also declare their own sections, in which case these sections would also be displayed in the left pane of the window.

The main part of the window displays the settings available in the selected section. Clicking the *Save* button saves the current setting values. Closing the window without clicking the *Save* button discards all changes.

A.3. Implemented code transformations

A.3.1. Diagnostics

There are currently 24 code style diagnostic rules whose validation is supported by the toolset. These are divided into four categories: *naming*, *comments*, *layout*, and *miscellaneous*.

Naming

Diagnostic rules in the *naming* category analyze that code elements' names follow the casing specified in the settings (camelCase, PascalCase). The following code elements' names are checked:

- Constant fields (default camelCase)
- Constant local variables (default PascalCase)
- Events (default PascalCase)
- Non-constant fields (default PascalCase)
- Methods (default PascalCase)
- Method parameters (default camelCase)
- Properties (default PascalCase)
- Types (default PascalCase)
- Non-constatnt local variables (default camelCase)

Layout

The diagnostics in the *layout* category monitor block language constructs for missing braces and display a warning when braces are missing. Example:

```
// the following
if (true)
  return;

// should look like this
if (true)
{
  return;
}
```

The following language constructs are validated:

- `if` statements
- `using` statements
- `for` statements
- `foreach` statements

Comments

Diagnostics in the *comments* category monitor the presence of documentation comments on code member declarations and the layout used for single-line comments. Example:

```
// missing documentation comment
private void Example()
{
}

// invalid comment layout (missing space after //)
//this is an invalid comment layout
```

The following code constructs are checked for documentation comments:

- Enum members
- Events
- Exceptions explicitly thrown by methods
- Fields
- Methods
- Properties
- Types

Additionally, single-line comments are checked against the following rules:

- Comment must be separated from the leading `//` by white space
- Comment must be preceded by an empty line

- Comment must not be followed by an empty line

Miscellaneous

The *miscellaneous* category contains two validations:

- Source file must have a file header
- Fields must be declared as `private`

A.3.2. Refactorings

The toolset implements 16 refactorings:

- Convert auto-property to a property with backing field
- Convert a property with backing field to auto-property
- Convert a class into a data contract, including property annotation with `[DataMember]` attributes
- Convert a class to exception, including necessary constructor declarations
- Implement `INotifyPropertyChanged` interface for a class
- Change access modifier (e.g. `private` to `internal` etc.)
- Check method argument for `null`
- Check variable for `null`
- Move class to a separate source file
- Organize `using` declarations
- Propend `this` to local member access calls
- Add change notification to a property
- Rename source file to match class name
- Split declaration and assignment
- Convert string concatenation to a call to `string.Format`
- Explicit to implicit type declaration (e.g. `int i;` to `var i;`)

Appendix B. Plugin Development Guide

This appendix summarizes the process of creating a new plugin that integrates with this thesis and supplies an enirely new code style diagnostic.

B.1. Prerequisites

Before starting the actual plugin development, it is first necessary to install and configure the necessary tools. Plugin development relies on the following:

- Visual Studio 2015 Preview or newer

  Microsoft Roslyn is only available in VS 2015, therefore plugin development must be done within this pre-release version of the IDE.

- Corresponding Visual Studio 2015 SDK

  Development for Roslyn requires the VS SDK to be present on the development machine. The SDK version should correspond to the installed version of Visual Studio 2015.

- Roslyn SDK Project Templates

  The Roslyn project templates simplify the development of Roslyn-enabled VSIX packages. Installing these templates is strongly recommended to make development easier.

Additional information about the steps necessary to install and configure the aforementioned tools, together with download links, can be found at Roslyn's GitHub page [10].

B.2. Developing a plugin

The diagnostic

The first step in creating a Roslyn code diagnostic is to create a new solution and add a new project based on the “Diagnostic with Code Fix” project template. This template is installed as part of the Roslyn SDK project template and resides in the “Extensibility” category of the *New Project* window, shown in Figure B.1.

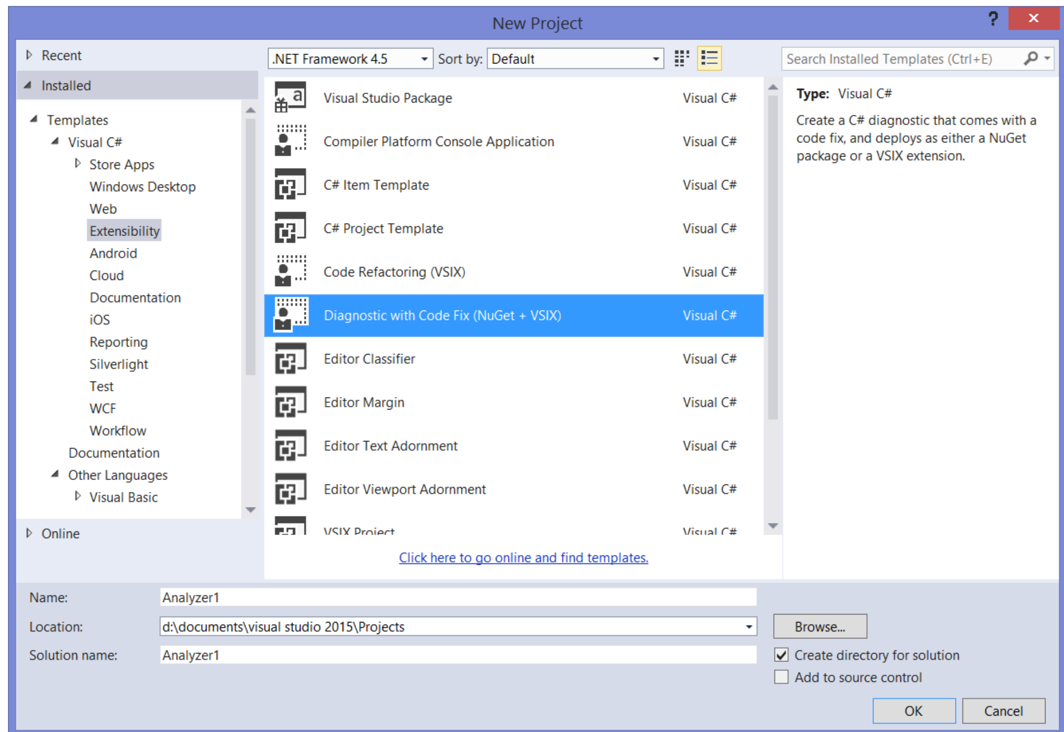


Figure B.1. New Diagnostic with Code Fix project

Creating a new project from this template will automatically create a solution containing several projects: the diagnostic implementation, unit test project for the implementation, and a VSIX package project. The unit test project contains skeleton code for creating tests for the implemented code diagnostic and the VSIX project is simply used to create an installable package from the diagnostic.

The core implementation of a plugin lies in the main diagnostics project (its name depends on the name specified when creating the project). This project also contains skeleton code with a simple code diagnostic (`DiagnosticAnalyzer.cs`). This example code scans types declared in a code file and reports a warning whenever a type name is not all uppercase. This is, of course, a rather primitive and useless diagnostic, but it serves as a very simple example of how to use the Roslyn APIs.

This guide will reimplement the skeleton code to scan declared classes for properties and report a warning when the number of properties in a class is odd. As a fix, the diagnostic will declare a new property in the affected class, making the number even. Such a diagnostic is utterly useless in a production environment, but it will serve as a complex example for using the Roslyn API in combination with facilities provided by the toolset implemented in this thesis. The plugin will also feature a setting to turn the diagnostic on and off.

The provided skeleton code diagnostic inherits from the `DiagnosticAnalyzer` class. The base class declares two abstract members, as shown in Figure B.2.

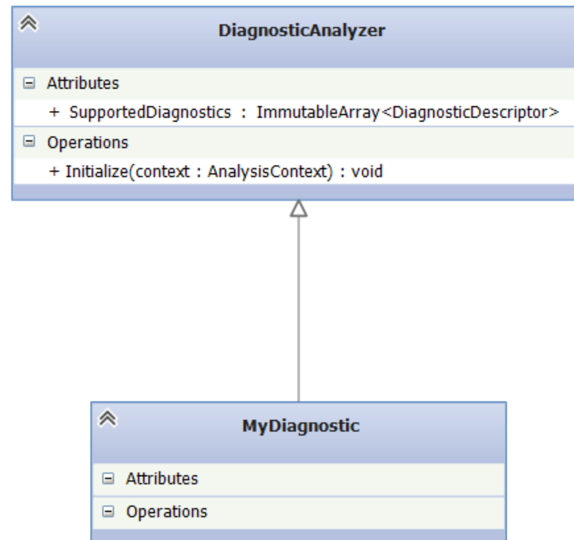


Figure B.2. Diagnostic inheritance

Both the abstract members are already overriden. The `SupportedDiagnostics` returns a descriptor for the diagnostic that is consumed by the Roslyn runtime when executing the diagnostic. The `Initialize` method serves to tell the runtime what kind of analysis the diagnostic performs.

There are also several fields declared in the class, containing information about the diagnostic used by the `SupportedDiagnostics` property. Let's change these fields to reflect accurate information about the new diagnostic:

```
string DiagnosticId = "EvenOddAnalyzer";
string Title =
  "Type must have an even number of properties";
string MessageFormat =
  "Type name '{0}' contains odd number of properties.";
string Category = "Demo";
```

The `DiagnosticId` field contains the diagnostic's ID. This value is used to uniquely identify the diagnostic at runtime and to find a code fix that corresponds to the diagnostic. It should remain public so it can be accessed from other assemblies, if necessary. The `Title` field stores the title displayed in the UI when presenting the diagnostic's result. The `MessageFormat` field contains the formatting string used to create the main message describing the diagnosed issue. The `Category` simply serves to group related diagnostics in the UI for better orientation.

After setting the diagnostic's information, the next step is to implement the `Initialize` method. This method receives a parameter of type `AnalysisContext` that can be used to register analysis actions that the runtime should perform. The demo diagnostic will scan types for properties, therefore it's necessary to tell the Roslyn runtime to run analysis on types:

```
context.RegisterSymbolAction(
  Analyze,
  SymbolKind.NamedType);
```

The `RegisterSymbolAction` method tells the runtime that the specified method (`Analyze`) should be called on any symbol that represents a named type (classes, structs). There are other `Register` methods with similar signatures that can be used to register analysis on different entities – syntax nodes, trivia (comments, whitespace etc.), semantic model, and more.

The final step to implementing a diagnostic is to implement the `Analyze` method. This method receives an argument of type `SymbolAnalysisContext` (named `context`) that contains all the information about the encountered symbol (in this case, a named type symbol). The diagnostic must take this symbol and find the properties declared within:

```
var t = (INamedTypeSymbol)context.Symbol;

var properties =
    t.DeclaringSyntaxReferences[0]
      .GetSyntax()
      .ChildNodes()
      .OfType<PropertyDeclarationSyntax>()
      .Count();

if (properties % 2 != 0)
{
  var diagnostic =
      Diagnostic.Create(Rule, t.Locations[0], t.Name);
  context.ReportDiagnostic(diagnostic);
}
```

This code first converts the generic `Symbol` instance provided by the `context` argument to the `INamedTypeSymbol` interface. The second line of code counts the number of properties declared in the type. It does this by first finding the syntax node declaring the type and then finding its `PropertyDeclarationSyntax` child nodes.

Finally, if the number of declared properties is odd, the code reports a warning using the `ReportDiagnostic` method. When this diagnositc is executed, the runtime will pass any encountered named type symbols to the `Analyze` method and if this method reports a problem, display a “squiggly line” to indicate a problem, illustrated in Figure B.3.

```
0 references
public int test { get; set; }
```

Figure B.3. A diagnosed code style issue

It is worth noting that the code diagnostic does not use any facilities provided by the implemented toolset, it only uses the standard Roslyn APIs. This is caused by the fact that Roslyn APIs for reading code and traversing syntax trees are rather well designed and simple to use. Therefore the core toolset provides no custom API for this task, as it was deemed unnecessary.

The code fix

Code diagnostics may have fixes associated with the problems they diagnose. In context of the implemented example diagnostic, the code should take a class with odd number of properties and declare a new one in it, thus satisfying the diagnostic rule. The fix will make use of the code synthesis API implemented as part of this thesis' toolset.

The project template also created a skeleton code for a code fix. This code is located in the `CodeFixProvider.cs` file. The original implementation takes a problem reported by the original diagnostic ("type name is not all uppercase") and changes the type name to uppercase. This walkthrough will reimplement this code fix to create a new property instead.

Every code fix must inherit from the `CodeFixProvider` class. This abstract class declares three members, as shown in Figure B.4.

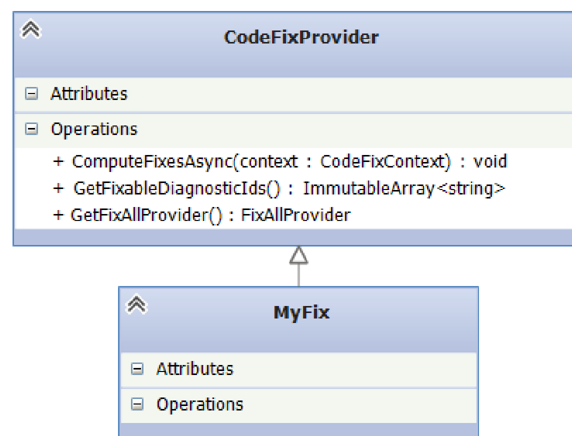


Figure B.4. Code fix inheritance

The `GetFixableDiagnosticIds` method tells the runtime what diagnostics this fix applies to. A single fix might apply to several different diagnostics, therefore this method returns an array of IDs. The original implementation already specifies that the code fix applies to the desired diagnostic so it is possible to simply leave the code as is.

The `GetFixAllProvider` method is used to get a batch fixer for the diagnostic. This method is used when the user decides to fix all the diagnosed issues across the whole code file, project, or solution. The Roslyn libraries provide a default implementation of a batch code fixer and it is usually safe to leave the original code as is:

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

However, if a special behavior is required to perform a "fix all" operation of a code fix, it is possible to implement a custom `FixAllProvider` and return it from this method.

The `ComputeFixesAsync` method is the most important of the three. This method is called by the Roslyn runtime to get the information about fixes available for a previously diagnosed issue. This method receives an instance of the `CodeFixContext` class (named *context*) as a parameter. The parameter contains all the information about the diagnosed issue and also serves to report the available fixes. This is the method that has to be rewritten to create a new property in the diagnosed class.

The first few lines of code in this method analyze the diagnostic and find the class declaration syntax node:

```
var root = await
  context.Document
    .GetSyntaxRootAsync(context.CancellationToken)
    .ConfigureAwait(false);

var diagnostic = context.Diagnostics.First();
var span = diagnostic.Location.SourceSpan;

var declaration = root
  .FindToken(span.Start)
  .Parent
  .AncestorsAndSelf()
  .OfType<TypeDeclarationSyntax>()
  .First();
```

The code first retrieves the analyzed document's syntax root, then finds the span (area of text) where the diagnosed problem occurs. The last line then finds the `TypeDeclarationSyntax` node residing at the span. This is the syntax node of the class with odd number of properties.

Registering a fix with the API is achieved using the provided *context* parameter:

```
context.RegisterFix(
  CodeAction.Create(
    "Add property",
    c => AddProperty(context.Document, declaration, c)),
  diagnostic);
```

This code informs the Roslyn runtime that there is a code fix available, the fix is called “Add property”, is performed by invoking the `AddProperty` method, and relates to the diagnostic stored in the *diagnostic* variable.

When the runtime executes the `ComputeFixesAsync` method and receives a fix, the user interface then allows the user to apply it. This is done by invoking the `AddProperty` method.

In case of the example diagnostic, the `AddProperty` needs to create a new property declaration in the affected class. This can be done by only using the Roslyn APIs, but the process can be greatly simplified by using the code emission facilities implemented in this thesis' core toolset. These facilities are implemented in the `ProjectLuna.2015.PlatformServices.dll` library. To make use of them, the project needs to reference this library.

After adding the necessary reference, the code emission API is accessible through the `ProjectLuna.PlatformServices.Synthesis.Luna` class. Creating a new property declaration using the `Luna` class looks like this:

```
var property =
  Luna.Property("Demo")
    .Type("string")
    .Public()
    .ToNode();
```

The created property is called "Demo", is of type `string`, has a `public` access modifier, and has both its accessors (`get`, `set`) are empty. The final `ToNode` call creates a syntax node that can be further used in calls to the Roslyn APIs.

The final step in fixing the odd number of properties in a class is to actually insert the newly created property into the class. This is achieved by using the following code:

```
var newType = typeDecl.InsertDeclaration(property);
var newRoot = root.ReplaceNode(typeDecl, newType);
var newDoc = document.WithSyntaxRoot(newRoot);

return newDoc.Project.Solution;
```

The `InsertDeclaration` method is part of the platform services library from this thesis. It takes a type and inserts a new member into it. The next calls then take this modified class and create a modified document (`WithSyntaxRoot`). The final line of code returns the modified code base back to the Roslyn runtime.

And that is everything that needs to be done to create a simple code fix. For detailed information about the APIs used in the examples, see Roslyn's documentation and the programmer's documentation of this thesis.

Using settings

The toolset implemented in this thesis also provides a unified way for plugins to store and use settings. The final part of this walkthrough will describe the process of integrating a plugin with this settings manager.

The first step necessary for a plugin to make use of the settings manager is to reference the `ProjectLuna.Settings.Composition.dll` library. This library implements all the necessary classes and interfaces to access the settings manager.

The settings manager uses a concept of "sections" to store settings. A section is basically a subset of the settings set that groups relates settings. For example, a plugin must declare its own section to store its settings. These sections are identified by GUIDs, so the plugin first needs to generate a valid GUID and store it for later access.

Settings can be accessed from anywhere within the plugin's code base using the `ServiceContainer` static class declared in the referenced `Settings.Composition` library. An example:

```
var settings = ServiceContainer
  .TryResolve<ISettingsProvider>();

var isDisabled = settings
  .GetSetting<bool>(
    Helpers.SettingsSectionGuid,
    "IsDisabled");
```

The code first retrieves the `ISettingsProvider` instance used to access the settings, then uses this instance to ask for a setting “IsDisabled” located in the section identified by the provided GUID. The setting will be read as a boolean value. This way of accessing settings works universally and does not require any configuration on the plugin's part.

The toolset implemented in this thesis also provides a unified user interface for viewing and managing settings using a single window. To allow a plugin to integrate with this settings window, the `ISettingsEditor` interface must be implemented in the plugin. The settings manager uses MEF composition to discover the instances of this interface, therefore the implementing class must also be decorated with the `[Export(typeof(ISettingsEditor))]` attribute.

The interface declares two properties – called `SectionName`, `Control` – and a method called `GetSettings`. The `SectionName` property is used to identify the setting section to the user and it should be kept short and clear. The `Control` property returns a WPF `UserControl` class that is rendered in the settings window and displays the UI elements used to edit the settings in the corresponding section.

The `GetSettings` method is called when the user decides to save the current settings. It must construct an instance of the `Section` class and fill it with current setting values. This section is subsequently stored in the settings file by the settings manager.

Unfortunatly, the `ISettingsEditor` is not portable, therefore it cannot be implemented in the same assembly as the code diagnostic. Implementing this interface thus requires a separate, non-portable project to be added to the solution. A simple Class Library project is sufficient. This project needs to reference the `ProjectLuna.Settings.dll` library, which contains the aforementioned interface. This project also needs to be included in the VSIX package as a MEF component to allow its participation in MEF composition.

Implementing the necessary interface should subsequently be rather straightforward. Please refer to the programmer's documentation for details about the required APIs.

Putting it all together

With both a diagnostic and a code fix implemented and integrating with the settings manager, it is time to test the implementation by running it inside Visual Studio. Simply run the solution with the VSIX project as the startup project. Doing so will start a new instance of Visual Studio, including the Roslyn runtime, and load the content of the VSIX package. It's improtant to note that the toolset implemented in this thesis must also be installed for all the plugin's features to work properly.

The Visual Studio's Extension manager should properly display the plugin, similarly to Figure B.5.

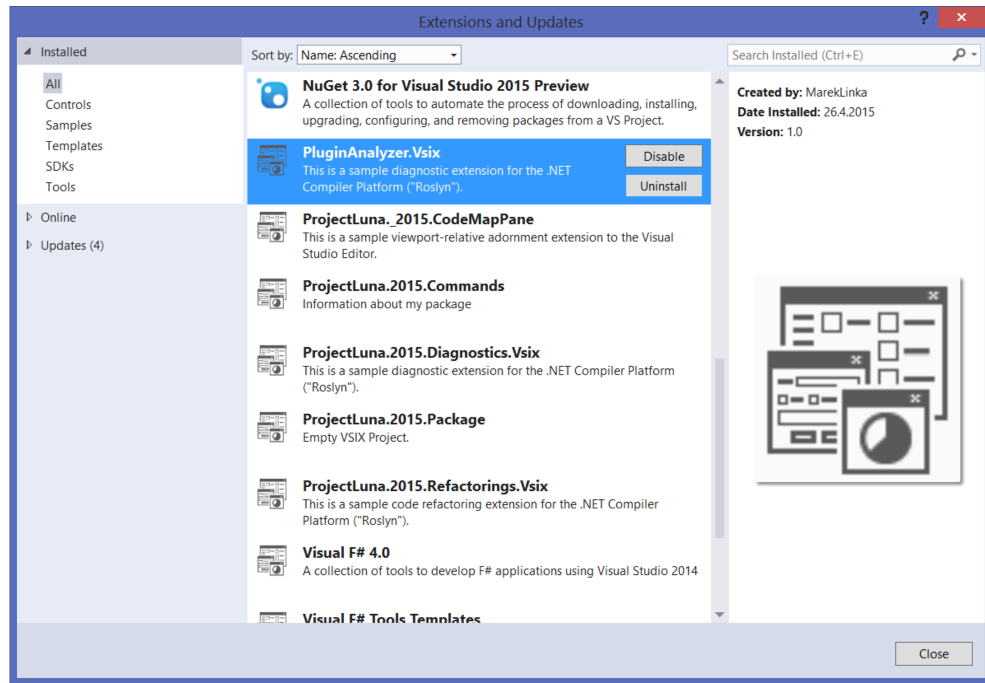


Figure B.5. Plugin installed in Extensions window

The settings manager (accessed using the *Project Luna Settings* menu item in the *Tools*) menu should also display the new settings section, as shown in Figure B.6:

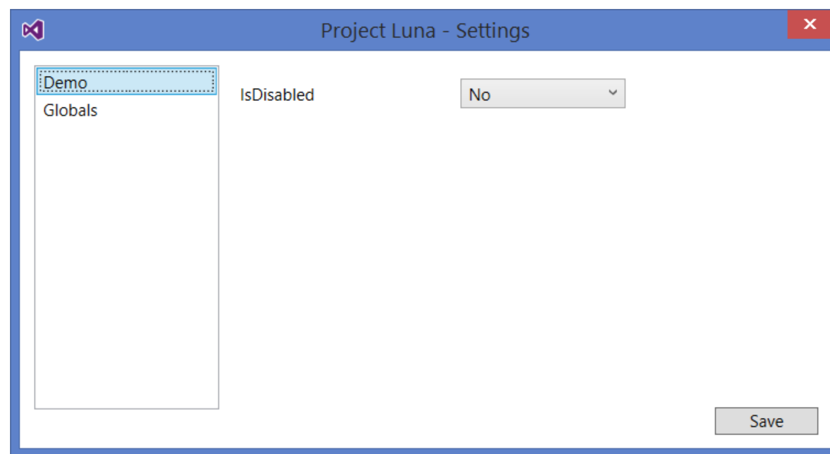


Figure B.6. Plugin in Settings window

If both of these windows display the plugin properly, the whole project is configured properly and should work well with the core toolset. There are more advanced features implemented in the toolset that can be used in plugin development but these are outside the scope of this guide. Please refer to the programmer's documentation for information about the usage of these advanced APIs.

Appendix C. References

Table C.1. References

[1]	C# Coding Conventions at MSDN	https://msdn.microsoft.com/en-us/library/ff926074.aspx
[2]	Homepage for the StyleCop VS extension	https://stylecop.codeplex.com/
[3]	JetBrains ReSharper homepage	https://www.jetbrains.com/resharper/
[4]	DevExpress CodeRush homepage	https://www.devexpress.com/products/coderush/
[5]	SharpDevelop IDE homepage	http://www.icsharpcode.net/OpenSource/SD/
[6]	C# Parser and CodeDOM homepage	http://www.inevitablesoftware.com/Products.aspx
[7]	NRefactory C# parser homepage	https://github.com/icsharpcode/NRefactory
[8]	Microsoft Roslyn project homepage	https://github.com/dotnet/roslyn
[9]	CodePlex discussion about diagnostics' MEF composition	http://roslyn.codeplex.com/workitem/467
[10]	Roslyn development introduction and tools	https://github.com/dotnet/roslyn#build-tools-that-understand-c-and-visual-basic
[11]	Microsoft Public License text	http://www.microsoft.com/en-us/openness/licenses.aspx

Appendix D. Content of the enclosed CD

The enclosed CD contains the following:

- Electronic version of the master thesis (this document) in the form of a PDF document
- The distribution package - a ZIP archive containing the compiled solution output
- The source code package - a ZIP archive containing the complete source code
- Programmer's documentation - two CHM files (compiled HTML help file) containing the programmer's documentation for the portable and non-portable code