Masaryk University
Faculty of Informatics

Automatic Refactoring of Large Codebases

Master's Thesis

Bc. Matúš Pietrzyk

Brno, Spring 2016

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Matúš Pietrzyk

Advisor: Bruno Rossi, PhD

Acknowledgement

I would like to thank my supervisor Bruno Rossi, PhD for his continuous feedback and support during the writing of this thesis. I would also like to thank Viktor Jablonský from FNZ for his support during practical part of the thesis.

Abstract

The aim of this thesis is to investigate different techniques for code refactoring using semi-automatic and automatic refactoring tools. The practical part focuses on providing automatic refactoring support for legacy source code using Roslyn compiler.

Keywords

Refactoring, Roslyn, Compiler, Code Smells, Legacy Code, SOLID, Design Smells, Large Codebase

Contents

1 Introduction

Technical debt that is present in many legacy codebases is a relevant problem for software companies. Often, in the beginning, the design of the system and actual code is unpremeditated which is then causing lot of problems in terms of code-readability and ability to extend the code when the new requirements are presented. This usually leads to poor design choices made by developers which clutters the code and performance of the system even further. Many companies try to solve this problem by complete project rewrites which takes huge amount of resources and are not always possible.

To mitigate these issues, various refactoring tools were introduced to the market. However, their usage is not always without the problems, especially when the code base is huge and new functionality is added on constant basis.

This thesis focuses on investigating different techniques for refactoring of such large codebases using refactoring tools and providing support for automatic refactoring in existing company.

1.1 Thesis Structure

Thesis is divided into five parts.

First part provides background of FNZ company and describes conditions of the codebase and reasons why it was necessary to refactor in the first place.

Second part introduces refactoring, describes its advantages and different refactoring strategies. Furthermore, it provides overview of SOLID principles, design and code smells and specific refactorings.

Third part provides description of three selected refactoring tools which are available on the market and their comparison and evaluation.

Fourth part describes .NET Compiler Platform "Roslyn" from Microsoft and its possible usage as a refactoring tool.

Fifth part provides description of a created software tool.

2 Issue Description

2.1 About FNZ

FNZ[1] is a market leading provider of investment and wrap platform technologies to major financial institutions in the wealth management and financial services sectors like life insurance companies, banks, asset managers and discretionary wealth managers. FNZ provides end-to-end technology, including investment front office, tax wrappers and investment back office, as cloud-based set of services which can be fully customized and branded according to customers' needs. Furthermore, it could be integrated into a customer's existing systems or created as a brand new, bespoke platform.

FNZ's existing customers include AXA Wealth, HSBC, Friends Life, Standard Life, Zurich, Barclays, AVIVA and many others, mainly investments companies. By second quarter of 2015 FNZ had over £58 billion of assets under management. FNZ is still growing rapidly and currently have over 900 people in offices in London, Edinburgh, Bristol, Brno, Wellington, Sydney, Honk King and Singapore.

2.2 Current State of the Codebase

FNZ started in Wellington in 2002 with the primary goal to deliver working platform in relatively short time. Because of this, design of the system and of the actual code was evolving rapidly, leading to large amount of technical debt. Large parts of the original code are still being used, which makes current development efforts more expensive than they would have to be.

For two main reasons, any attempt to address the problem is expected to be a large effort. Firstly, the legacy code has very low unit test coverage which makes any refactoring very risky and prone to errors. Secondly, multiple platforms have their own code base even though the legacy code structure originates from the same ancestor, and there is a degree of code duplication, too. As a result, many refac-

1. `http://www.fnz.com/`

toring changes done in one code base need to be replicated in the other code streams as well.

In the first projects, user interface was done in ASP.NET Web-Forms development model which was directly calling legacy code solution from its code-behind. Later, the service layer API was introduced which enabled connecting FNZ's legacy core solution with customers' own user interfaces and also enabled transition to MVC application model which is used for new customers. The controller part of MVC model is now calling services which execute legacy core code which is primarily written in VB.NET. In last few years, large part of the logically related functionality was grouped together and now form separate products. But in the deepest level of all these new products is still the legacy core code and new functionality must be constantly added into it. Often, developers want to avoid to add new code in VB.NET language so they create new and new projects. Currently, the typical legacy code base consists of more than 250 projects, which is causing the enormous build time. Using MSBuild diagnostic tools, it was shown that resolving dependencies between those more than 250 projects takes enormous time.

3 Refactoring

Martin Fowler [5] defined refactoring as "Process of changing a software system in such a way that is does not alter the external behavior of the code yet improves its internal structure."

Formally, refactoring can be defined as an ordered triple R = (**pre**, **T**, **P**) where **pre** is an assertion that must be true on a program for **R** to be legal, **T** is the program transformation, and **P** is a function from assertions to assertions that transforms legal assertions whenever **T** transforms programs. [4]

Refactoring can be categorized based on level of automation into three categories:

- **Fully manual refactoring** — done manually by programmers.
- **Semi-automatic refactoring** — using various refactoring tools available on the market such as ReSharper, CodeRush and JustCode which are described in chapter 4.
- **Automatic refactoring** -– refactorings which are prepared beforehand and applied to the whole solution/code/codebase by running script.

Good starting point on how to refactor is so called refactoring cycle which consists of three distinct steps as shown in figure 3.1. [8]:

1. **Adding a test (red phase)** — In this step, the failing unit test is added for a functionality that has not been built yet.
2. **Making it work (green phase)** -– make the failing test pass (go green) by implementing the necessary functionality simply but crudely.
3. **Making it clean (blue phase)** -– Use refactoring to ensure the overall code base is as clean and well-designed as possible for currently-implemented functionality.

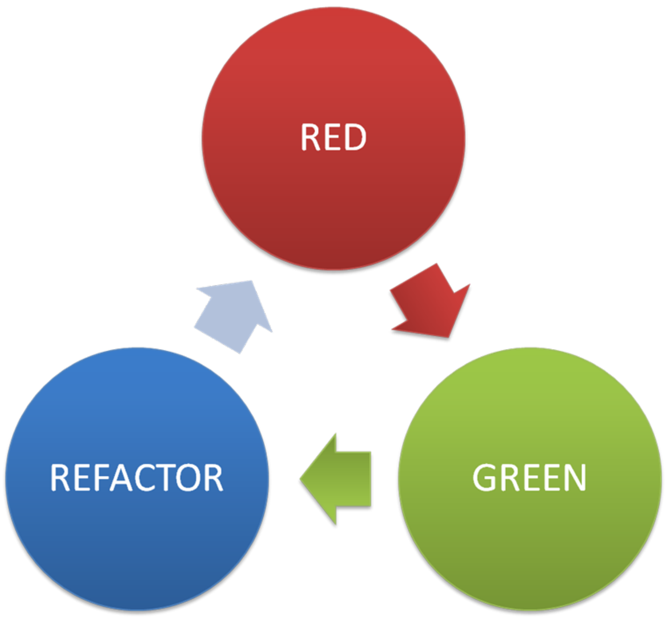

Figure 3.1: Refactoring Cycle. Image source: [8]

3.1 Key Advantages of Refactoring

According to Fowler [5], refactoring provides four key advantages:

- **Refactoring improves the design of software** – Without refactoring, the design of the program decays over time. When new things are added into the code, developers often do not have full knowledge of the original intended design which often leads to duplicate code.

- **Refactoring makes software easier to understand** – When developer wants to refactor certain part of code he firstly needs to have knowledge about what the code is doing and then he can change it which even further raises his level of understanding the code.

- **Refactoring helps you find defects** – When developer refactors certain part of the codebase, he is gaining deep knowledge about it which in return enables him to see possible defects.

- **Refactoring helps you program faster** – Poor design of the software often slows developers down because they spent lots of time trying to understand code itself and fixing defects. The benefits of refactoring is that it helps developers create software more rapidly, because it stops the design of the system form decaying.

3.2 Refactoring Strategies

Refactoring should be continuous process, not something that is done in specific chosen times. However, there are some guidance rules that states when to refactor [5]:

- **The Rule of Three** – Guideline by Don Roberts [5] that states: "The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor."
- **When a function is being added** – There are two possible reasons why the code should be refactored in this case. First is to better understand the code that needs to be changed when the new feature is added. Second reason is when the current code design makes adding the feature much more complicated than it could be with better design.
- **When a bug needs to be fixed** – This comes from a reason that when the bug is found, it is a sign that the code was not clear enough for developer to see the bug in the first place.
- **When a code review is being done** – Useful in small review groups consisting of author of the code and the reviewer who suggests changes and together they decide if these changes should be applied into the code or not.
- **When a design or code smell is identified** – Other time, when the refactoring should also be done whenever a design or code smell is found in the code.

- **Target error-prone modules** – one of the effective strategies of refactoring. These are modules that everybody is afraid of which is a good sign that they are more error-prone than others.

- **Target high-complexity modules** – These are modules that have the highest complexity ratings. Study done by Henry and Kafura showed that quality of the code improved dramatically when the focus was on these kind of modules.

- **Define an interface between clean code and ugly code, and then move code across the interface** – This is very effective strategy for old legacy systems. Every time the developer makes changes to a legacy system, some part of that code should be moved into the clean part of the code. If every developer in the team do this, the quality of the code will be significantly improved.

3.3 Design Smells

Software becomes rotted when it exhibits some of these traits [10]:

- rigidity,
- fragility,
- immobility,
- viscosity,
- needless complexity,
- needless repetition,
- opacity.

These smells are often caused by violation one or more SOLID design principles mentioned below.

3.3.1 Rigidity

Software is **rigid** when it is difficult to change, even in simple ways. "A design is rigid if a single change causes a cascade of subsequent changes in dependent modules. The more modules must be changed, the more rigid the design."[10]

3.3.2 Fragility

Fragility is the tendency of a program to break in many places even when a single change is made. The problems that arise in such situation are often in areas that have no logic relation with area that was changed. When developers try to fix those problems, the new problems occur and the modules are getting worse and worse the more they are being fixed. When the fragility reaches maximum, the probability that a change will introduce unexpected problems approaches certainty.

3.3.3 Immobility

Software is **immobile** when none of its parts can be used in other systems because the risk and effort involved with separating those parts from the original system are too great.

3.3.4 Viscosity

Viscosity can have two forms:

- **Software viscosity** – occurs when changes that preserve software design are more difficult to use than various hacks. In other words, it is easy to do the wrong thing but difficult to do the right thing.
- **Environment viscosity** – occurs when development environment is inefficient and slow. One example are long compilation times which causes that developers would rather make changes that do not require large recompilation even though those changes do not preserve software design. Other examples might be long times that are needed to check the code into

source control system. In this case developers would rather make changes that requires as few check-ins as possible.

3.3.5 Needless Complexity

Codebase is **needlessly complex** when it contains code that is currently not useful. This usually happens when developers try to anticipate future requirements and changes and put into the code parts that do not have to be there at the moment. This may seem as a good idea at the beginning but often the added parts are never used and they just make the code clumsier. One example is over-conformance to the SOLID principles mentioned below.

3.3.6 Needless Repetition

Needless repetition occurs when the same code appears in different locations, every time with slight change. This makes system hard to understand and maintain and it is sign that developers are missing abstraction. Another problem with such code is that when the change in the software is needed and there are bugs found in those repetitive units they have to be fixed in every repetition. But because of the fact that this repetition is always slightly different, the fix is not always the same.

3.3.7 Opacity

Opacity is a tendency of software code to be difficult to understand. Codebase that evolves over time tends to become more and more opaque with age. The constant refactoring is needed in this case to prevent this from happening.

3.4 SOLID Principles

Solid principles help to eliminate bad design smells mentioned above. It is necessary to add that they should not be used all over system because this leads to the design smell of needless complexity. Robert C. Martin defined them as follows [10]:

3.4.1 The Single-Responsibility Principle (SRP)

Single-Responsibility Principle tells that a class should have only one responsibility and not multiple. The responsibility in this context is defined as a reason to change.

The idea behind this is that "when the requirements change, that change will be manifest through a change in responsibility among the classes. If a class assumes more than one responsibility, that class will have more than one reason to change."

"If a class has more than one responsibility, the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class's ability to meet the others. This leads to design that is fragile and can break in unexpected way when some change is introduced."

Common violation of SRP principle is when the business rules and persistence control is mixed together in one class. Business rules and persistence control are changed for different reasons. To separate these responsibilities, design should be refactored using FACADE, DAO (Data Access Object) or PROXY patterns.

3.4.2 The Open/Closed Principle (OCP)

The Open/Closed Principle states that software entities such as modules, classes, functions and others should be open for extension but closed for modification. There are two basic attributes:

1. **Modules are open for extension** – behavior of the module can be extended when the requirement change.
2. **Modules are closed for modification** – "extending the behavior of a module does not result in changes to the source, or binary, code of the module."

To confirm to the OCP principle, the abstractions can be used, specifically STRATEGY or TEMPLATE METHOD design patterns. But confirming to OCP is expensive — it takes development time and effort to create the appropriate abstractions –- and those abstractions also increase the complexity of the software design.

In the past, the general opinion of software professionals was that the hooks should be put in the code that will make software more flexible. Unfortunately, this leaded many times needless complexity, because these hooks were never used.

Best approach is to initially write the code without expectations that it will change. Then, after the change occurs, the abstractions should be implemented which will protect code from future changes of that kind.

3.4.3 The Liskov Substitution Principle (LSP)

The Liskov Substitution principle tells that subtypes must be substitutable for their base types.

Violation of this principle can be easily recognized in the code by numerous type checking using if statement which is used to determine the type of an object so that the behavior appropriate to that type can be selected.

3.4.4 The Dependency-Inversion Principle (DIP)

Dependency-Inversion Principle definition consists of two parts [10]:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

High-level modules usually contain business models of an application and policy decisions. But when these modules depend on the lower-level modules, changes to the lower-level modules can have direct effects on the higher-level modules and can force them to change in turn. Thus, DIP principle states the opposite – high-level should not depend on low-level modules in any way.

3.4.5 The Interface Segregation Principle (ISP)

The Interface Segregation Principle states that clients should not be forced to depend on methods they do not use.

When a client depends on a class that contains methods that the client does not use but that other client use, that client will be affected by the changes that those other clients force on the class. ISP is about avoiding such couplings through separation of interfaces.

There are two ways to achieve ISP:

- Separation through delegation,
- Separation through multiple inheritance.

3.5 Code Smells

Code smells are similar in nature to design smells but at much lower level. The 10 most common code smells according to Fowler and Beck [5] are:

- **Duplicated code** – the most occurring code smell of all. There are various variants of this problem:
  - **Same expression in two methods of the same class** – solution is to use ***Extract Method*** refactoring and call the new code from both places.
  - **Same expression in two sibling subclasses** – can be resolved first by using ***Extract Method*** in both subclasses and then ***Pull Up Method*** refactoring.
  - **Duplicated code in two unrelated classes** – the easiest solution is to use ***Extract Class*** in one class and then use the newly created component in the other.
- **Long method** – Martin in his book Clean Code [9] stated that "the first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that. Functions should hardly ever be 20 lines long." Problem with long methods is that they are very hard to understand and change. There are couple of solutions to this problem based on complexity of the method. The easiest one is to use ***Extract Method***. This usually works fine when the method is simple and does not have lots of parameters or temporary variables.

Using Extract Method on this kind of method would introduce long parameter list which would make reading and understanding code even more complicated. In this case, it is best to use ***Replace Temp with Query*** in case of temporary variables and ***Introduce Parameter Object*** and ***Preserve Whole Object*** in case of long lists of parameters.

- **Large class** – Large classes are easily spotted when there are too many instance variables which is then often causing duplicated code. Solution is to ***Extract Class*** or sometimes ***Extract Subclass*** or ***Extract Interface*** refactorings.

- **Long parameter list** – In object-oriented programming parameter list should be as small as possible. According to Martin [9] "the ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification – and then shouldn't be used anyway." Problem is that when the new features are added, developers often see adding more (sometimes even optional) parameters as ideal solution. This makes code hard to read and understand. It also makes testing very difficult due to possible large combination of parameters. Solution here is to use ***Replace Parameter with Method, Preserve Whole Object*** or ***Introduce Parameter Object*** refactorings.

- **Divergent change** – occurs when a class is changed for different reasons. Goal is to have a design where each object is changed as a result of only one kind of change. Best approach is to identify all things that belong together (e.g. everything that changes for particular reason) and then use ***Extract Class*** refactoring.

- **Shotgun surgery** – opposite of divergent change which occurs when every desired change in functionality requires many changes to a many different classes. Problem is that changes like this are scattered in many places so it is easy to miss one. Solution here is to put all the changes into a new class using

Move Method and ***Move Field*** refactorings. Another possibility is to use ***Inline Class***.

- **Feature envy** – smell which can be easily spotted when one method calls methods of other classes more often than methods of the class that it is in. This usually happens with getting methods on another object to calculate some value. Solution is straightforward using ***Move Method*** or ***Extract Method*** refactorings.
- **Data clumps** – Some data items often appear together in lots of different places. Refactoring in this case consists of two steps. First is to use ***Extract Class*** refactoring to turn the data items into object. Second step is to slim down the method signatures with ***Introduce Parameter Object*** or ***Preserve Whole Object*** refactorings.
- **Temporary field** – This usually occurs when some algorithm in the class needs various variables to alter its behavior. Developers often solve this issue by introducing one or more temporary fields. Problem is that these variables are valid and only used in that specific algorithm, so their presence in other contexts is confusing and makes code difficult to understand. Ideal refactoring here is to put the specific variables together with all their code into a new class using ***Extract Class***.
- **Message chains** – can be spotted by series of message chains "when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another another object, and so on." [5]. This smell is often called "train wrecks" [9]. The solution here is to use ***Hide Delegate*** refactoring which can be used at various stages of the calling chain. It is possible to apply this to every object in the chain, but this introduces another code smell called middle man. Other alternative is to use ***Extract Method*** and then ***Move Method*** refactorings.

Other code smells which Fowler identified are:

- Primitive obsession,

- Switch statements,
- Parallel inheritance hierarchies,
- Lazy class
- Speculative generality,
- Middle man,
- Inappropriate intimacy,
- Alternative classes with different interfaces,
- Incomplete library class,
- Data class,
- Refused bequest,
- Comments.

3.6 Specific Refactorings

- **Extract method** – Turn the code fragment into a method whose name explains the purpose of the method.
- **Extract class** – Create a new class and move the relevant fields and methods from the old class into the new class.
- **Extract subclass** – Create a subclass for a specific subset of features.
- **Move method** – Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.
- **Move field** – Create a new field in the target class, and change all its users.
- **Introduce parameter object** – Replace group of parameters that naturally go together with an object.

- **Preserve whole object** – Instead of getting several values from an object and passing these values as parameters in a method call, the whole object should be send.
- **Hide delegate** – If a client is calling a delegate class of an object, methods on the server should be created instead to hide the delegate.

3.7 Big Refactorings

In contrast with little refactorings explained in part 3.6, big refactorings focus on the whole design of the system rather than on small individual cases.

Big refactorings can take months to finish so it is very important to achieve consensus between developers as well as between developers and managers.

3.7.1 Tease Apart Inheritance

Tease Apart Inheritance deals with case where there is an instance hierarchy which is doing two jobs at once. Solution is to create two hierarchies, and use delegation to invoke one from the other.

Recommended approach consists of six steps as follows [5]:

1. First step is to identify all the different jobs that are done by the hierarchy. A two-dimensional grid could be used with axes labeled with different jobs.
2. Next step is to decide which job is more important and should remain in the current hierarchy and which should be moved to the other hierarchy.
3. Using ***Extract Class*** refactoring in the common superclass, a new object should be created for the subsidiary job together with adding an instance variable which will hold this object.
4. A new subclass for the extracted object from the step three should be created for each of the subclasses in the original hierarchy. Then the instance variable also created in the step three should be initialized.

5. Using the ***Move Method*** refactoring, in each of the subclasses the behavior from the subclass should be moved to the extracted object.

6. Repeat steps above until all subclasses are eliminated.

3.7.2 Convert Procedural Design to Objects

This refactoring deals with converting code written in procedural style into the object-oriented one.

Recommended approach consists of four steps [5]:

1. First step is to turn all record types into a dumb data objects with accessors. In case of a relational database as a base for model, each table should be turn into a dumb data object.

2. Second step is to turn/take all the old procedural code and put it into a single class which can be either singleton or normal class with static methods.

3. Third step is to take the procedural code from second step and use ***Extract Method*** and ***Move Method*** refactorings to decompose it into dumb data classes created in first step.

4. Repeat steps above until all the behavior is removed from the original class.

3.7.3 Separate Domain from Presentation

In this case, there are GUI[1] classes that contain domain logic. Solution here is to separate domain logic into separate domain classes. Basically, it is transition from two-tier design where the data usually sits in the database and the logic are programmed in the presentation classes to the MVC pattern where there is clear separation between the user interface code (view part) and domain code (model part).

Recommended approach consists of four steps [5]:

1. Domain class should be created for each window.

1. GUI – Graphical User Interface

2. If there is a grid present, the class should be created which will represent rows on the grid. The row domain objects will be then stored in the collection on the domain class for the window.

3. Data should be treated based on the place where it is used as follows:

   - If it is used by the user interface, it should remain on the window.
   - If the data is used in the domain logic but not displayed on the window, the ***Move Method*** refactoring should be used to move it to the domain object.
   - When the data is used by the UI as well as the domain logic, the ***Duplicate Observed Data*** refactoring should be used. This will ensure that the data is properly sync between those two places.

4. Using the ***Extract Method*** refactoring, the domain logic should be separated from presentation logic in the presentation class. Once the domain logic is isolated, the ***Move Method*** refactoring should be used to move it to the domain object.

3.7.4 Extract Hierarchy

Extract Hierarchy deals with situation when there is one big class that is doing too much work through many conditional statements. This usually happens with evolutionary design when the new functionality is continuously added to the existing class and no refactoring is performed. Solution here is to create hierarchy of classes where each class will represent one special case. This particular refactoring can take months to finish.

Recommended approach consists of five steps [5]:

1. First step is to identify the specific variation which can be separated into the new subclass.

2. For the case identified in step one, a new subclass should be created and then ***Replace Constructor with Factory Method*** refactoring should be applied on the original class.

3. Methods that contain conditional logic should be copied (one at the time) to the new subclasses with usage of ***Extract Method*** refactoring.

4. Above steps should be repeated until all the special cases are moved to their new subclasses.

5. Final step is to delete bodies of all methods in the superclass which are overridden in the new subclasses and make whole superclass abstract.

4 Refactoring Tools

Since manual refactoring is enormously time consuming and costly activity, which is prone to many errors, various refactoring tools have been introduced to mitigate those issues. Refactoring tools attempt to resolve this issues by providing support for whole refactoring cycle -– identification of code smell; proposing a specific refactoring; and applying this refactoring.

These tools differ between each other by price, quality of provided documentation, subset of refactorings that can be applied and their complexity.

Refactoring tools can be divided into two categories:

- **Fully-automatic tools** – provide multi-stage refactoring without user's interactions. This is positive on one side, but on the other it introduces several problems like lack of customizability or negative impact on a user's current understanding of a system.
- **Semi-automatic tools** – unlike refactoring tools that are fully-automated, semi-automatic tools retain the user input to be able to customize some steps of the refactoring process but still automate those error-prone sub-tasks.

In this chapter, the three most common used refactoring tools for .NET and Visual Studio are presented and evaluated.

4.1 ReSharper

ReSharper is a refactoring and code analysis tool developed by company JetBrains[1] (formerly named IntelliJ) founded in Prague, Czech Republic in 2000. Current latest stable version -– 10.0.2 was released in December 2015.

ReSharper is available as a individual product or it can be purchased as a part **ReSharper Ultimate** package which contains following tools [14]:

1. `https://www.jetbrains.com/`

- **ReSharper**,
- **ReSharper C++** – version of the ReSharper for C and C++ developers.
- **dotTrace** – performance profiler used to detect performance bottlenecks in applications by analyzing application threads, call stacks, memory allocations, garbage collection, and I/O operations. When the bottleneck is found, dotTrace will find location of the piece of code that is causing it. dotTrace provides support for variety of .NET applications including ASP.NET, WCF, Windows services and others. Profiling of the unit testes and SQL queries is also supported.
- **dotMemory** – memory profiler used to analyze memory usage and memory leaks in desktop applications, Windows services, ASP.NET web applications, IIS, IIS Express and more.
- **dotCover** – unit test runner and code coverage tool that supports .NET framework from version 2.0 higher, as well as Silverlight 4 and 5. dotCover includes continuous testing which is a workflow where dotCover decides on-the-fly which unit tests are affected by the recent code changes, and then automatically re-runs those unit tests without developer's input. It can be also integrated with continuous integration servers, for example JetBrains's TeamCity. dotCover can then obtain coverage data directly from a TeamCity server without need to run analysis on the local machine.
- **dotPeek** – decompiler and assembly browser which can decompile any .NET assembly into equivalent C# or IL code. dotPeek includes support for multiple formats including libraries (*.dll*), executables (*.exe*), archives (*.zip*, *.vsix*, Windows metadata files (*.winmd*) and *.nupkg* formats) and folders. dotPeek can also identify local source code based on PDB files, or download source code from source servers such as Microsoft Reference Source Center so it can perform as a symbol server when debugging assembly code.

ReSharper also provides support for building **plugins** through usage of its own ReSharper API. This enables developers to create new code inspections and quick-fixes and many other things [12]. Once the plugin is created, it can be uploaded to **ReSharper Gallery**[2] to make it available to all users. Installation of the plugin is then done through Extension Manager.

Through **ReSharper Online Help**[3], JetBrains provides extensive documentation of ReSharper's services, description of language support, features and their usage. There are variety of tutorials available from JetBrains YouTube channel[4] as well as from **ReSharper Cookbook**[5].

Since convincing management about buying tools like ReSharper can be quite complicated, JetBrains provide **ReSharper Benefits**[6] document that describes ReSharper functionality from a ROI[7] perspective and provides an estimation framework about benefits from using ReSharper.

ReSharper offers quite complex model of pricing with different prices for individual customer and for business and organizations. Moreover, prices differ based on frequency of the payment (yearly or monthly). In addition to the basic price list shown in figure 4.1, JetBrains provides various discounts shown in figure 4.2.

4.2 CodeRush

CodeRush is a refactoring and productivity plugin by DevExpress that extends native functionality of Microsoft Visual Studio. Last version was released in July 2015. In addition to refactoring capabilities, CodeRush offers code visualizer, debugger, decompiler, automatic code generator and more.

2. `https://resharper-plugins.jetbrains.com/`
3. `https://www.jetbrains.com/resharper/help/`
4. `https://www.youtube.com/playlist?list=PLQ176FUIyIUa7pt4QLvXsspND1q-S_Zrx`
5. `https://confluence.jetbrains.com/display/NETCOM/ReSharper+Cookbook`
6. `https://www.jetbrains.com/resharper/docs/resharper_benefits.pdf`
7. ROI – Return On Investment

Version	ReSharper	ReSharper Ultimate
Price (pay yearly)	$299.00/1st year $239.00/2nd year $179.00/3rd year onwards	$399.00/1st year $319.00/2nd year $239.00 /3rd year onwards
Price (pay monthly)	$29.90/month	$39.90/month
ReSharper	✓	✓
ReSharper C++	-	✓
dotTrace	-	✓
dotMemory	-	✓
dotCover	-	✓

Table 4.1: Overview of ReSharper prices for individual customers. Source of data: [13].

Discount Type	Discount Amount
For startups	50% off
For students and teachers	free
For Open Source projects	free
For education and training	free

Table 4.2: Overview of ReSharper's discounts. Source of data: [11].

In June 2015, preview version of **CodeRush for Roslyn** was released [1] that sits on top of Microsoft's Roslyn engine described in detail in chapter 5. In that same release, CodeRush was renamed to **CodeRush Classic** and it was announced that there will not be any further releases apart from traditional security fixes for that product. In the future, once the CodeRush's core engine is replaced with Roslyn's engine and all the features are ported over, DevExpress plans to discontinue CodeRush Classic.

CodeRush for Roslyn is still in preview version, but it includes full support for C# 6 and VB 14 language features. If developer is working with Visual Studio 2015 and needs new language features in C# 6 or VB 14, DevExpress recommends installing CodeRush for Roslyn. If the features of CodeRush Classic that have not been ported yet are required, then CodeRush Classic should be installed. It is also possible to have them installed and use both.

CodeRush for Roslyn has extremely efficient use of memory and faster performance in comparison to CodeRush Classic.

Since every CodeRush feature is implemented as a plug-in using Visual Studio's designer and property browser, it is easy for developers to write their own plugins which can be commercial, free or open source. Once the plugin is ready, developer must contact DevExpress to make it available for other users. These plugins are then available from Google Code page or from company's own website.

In addition to basic documentation[8], DevExpress provides several free webinars[9] for developers. Topics range from basics on how to get started to more advanced subjects.

CodeRush is licensed on a subscription basis with the basic of $249.99 for 12 months per 1 developer for first year. On-time renewal rates for following years are substantially lower – $99.99. DevExpress also provides multi-user discounts shown in figure 4.3.

CodeRush is available as 30-days trial version which include full 30 days of free tech support. 60-day unconditional money-back guarantee is also offered.

8. `https://documentation.devexpress.com/#CodeRush/CustomDocument3519`
9. `https://www.devexpress.com/Support/Webinars/completed.xml`

Number of Licenses	Discount Amount
2–5	10% off
6–10	10% off
More than 10	Upon contacting DevExpress

Table 4.3: Overview of CodeRush's discounts. Source of data: [2].

4.3 JustCode

JustCode is a Visual Studio extension developed by Bulgarian company named Telerik which was acquired by Progress Software in December 2014. It provides several functions, such as refactoring, code generation, code analysis, error-checking and many others. [6]

JustCode supports creating plugins through its own JustCode API Visual Studio extensions. It is possible to download them directly from GitHub[10] or install directly from *.vsix* files. Then they can be customized through Extension Manager of Visual Studio.

To get started, Telerik provides basic introduction video and documentation[11].

JustCode can be purchased as a stand-alone product, or as a part of DevCraft which is .NET and HTML5 toolbox. The pricing itself is very complex with various discounts and different prices for renewals and upgrades from older versions. The license includes one year of updates and support. To continue receive updates, performance enhancements and support, this license must be renewed each year.

4.4 Comparison and Evaluation

Over last years, refactoring tools have evolved to the point when they are very similar to each other in terms of **reliability**. It is something that users expect automatically from them and thus it is no longer the main choosing criteria.

Major distinction between them are **number of supported refactorings** where ReSharper covers almost all defined by Fowler [5].

10. `https://github.com/telerik/justcode-extensions`
11. `http://www.telerik.com/help/justcode/introduction.html`

Version	JustCode	DevCraft Complete	DevCraft Ultimate
Price per developer	$249	$1,499	$1,999
Just Code	✓	✓	✓
Other Tools	-	13 other products	16 other products
Phone Assistance	-	-	✓
Remote Web Assistance	-	-	✓
Issue Escalation	-	-	✓
Ticket pre-screening	-	-	✓

Table 4.4: Overview of JustCode's prices. Source of data [7].

Number of Licenses	Discount Amount
2–5	10% off
6–10	10% off
11+	Upon contacting Enterprise Sales

Table 4.5: Overview of JustCode's discounts. Source of data [7].

Prices are also very similar and all companies offers their products on subscription basis. All refactoring tools can be purchased as a stand-alone product or as a part of bigger package with lots of **additional features** like different UI controls and various code analysis, debuggers and profilers.

In terms of **documentation and tutorials**, ReSharper offers the most thorough guides with its own YouTube channel and various start-up guides.

With the introduction of new .NET Compiler Platform "Roslyn", interesting distinction arose between tools that plan to support it in the future and those which do not. All refactoring tools need to understand the code and thus parse the solution source which requires huge amounts of **memory**. In tools like ReSharper which have decided not to support Roslyn, the required memory is doubled, since Visual Studio is storing similar results. On the other hand, CodeRush for Roslyn use the Roslyn data from Visual Studio so this doubling of memory is not happening. This has effect not only on memory usage but also **time for response**.

5 .NET Compiler Platform "Roslyn"

5.1 Overview

Traditional compilers can be seen as black boxes – they take source code as an input, process it and produce object files or assemblies. While the compiler is processing the source code it is gaining knowledge about the code itself. However, this knowledge is not available to developer. "Roslyn" exposes this knowledge through set of APIs, which enables programmers to use it in their code.

.NET Compiler Platform "Roslyn" is a complete rewrite of the C# and VB.NET compilers (which were written in C++) in those languages themselves. "Roslyn" is provided as an open source and can be found on the GitHub repository[1]. This allows third parties to build their own extension and tools to Visual Studio without having to implement all the language services by themselves.

Traditional compiler pipeline for .NET managed languages consists of four compilation parts as shown in figure 5.1 [3].

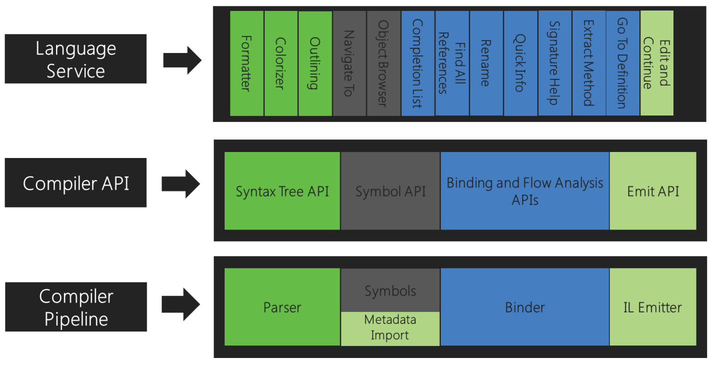

Figure 5.1: Compiler Pipeline. Image Source: [15]

1. **Parser** – takes in the source code and turns it into sequence of tokens which are then join together to form a syntax tree.

2. **Symbols and metadata import** – "source declarations and imported metadata are analyzed to form named symbols".

1. `https://github.com/dotnet/roslyn`

3. **Binder** – matches identifier to the right symbols.

4. **IL emitter** – takes all the information built up by the compiler and produces sequence of IL instructions which can be executed by the CLR[2].

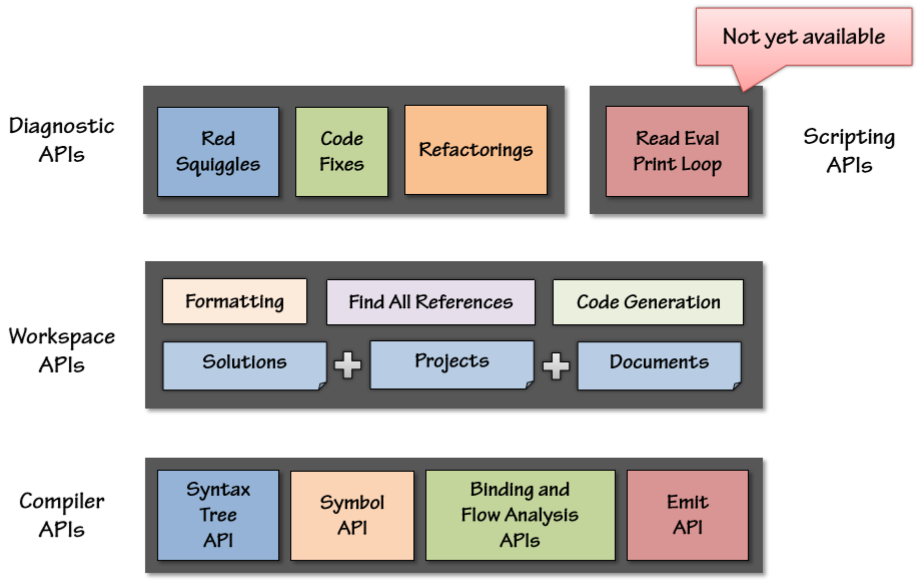

Figure 5.2: Roslyn APIs. Image Source: [3]

"Roslyn" provides mapping of all compilation stages as shown in figure 5.1 onto different APIs together called Compiler APIs. Overall, .NET Compiler Platform "Roslyn" consists of four main layers of APIs (Figure 5.2) [3].

- **Compiler APIs** – set of Syntax Tree API, Symbol API, Binding and Flow Analysis APIs and Emit API. These APIs are then mapped onto various language services (Figure X) which can be found in Visual Studio. All these services were also found in previous versions of Visual Studio but from version 2015 are built using Roslyn APIs.

2. CLR (Common Language Runtime) – the virtual machine component of .NET framework which manages the execution of .NET programs.

- **Workspace APIs** – represent sum of Solutions, Projects and Documents.
- **Diagnostic APIs** – represent diagnostic errors and warnings together with additional information, like location. In Roslyn, Diagnostic APIs are extensible so it is possible to build new “refactorings” and “code fixes”.
- **Scripting APIs** – allow to build “Read Eval Print Loops” which are basically much richer version of “Immediate windows” in Visual Studio. These APIs have been temporarily withdrawn from “Roslyn” in previous versions but are now available in pre-release package on NuGet server.

5.2 Syntax Analysis

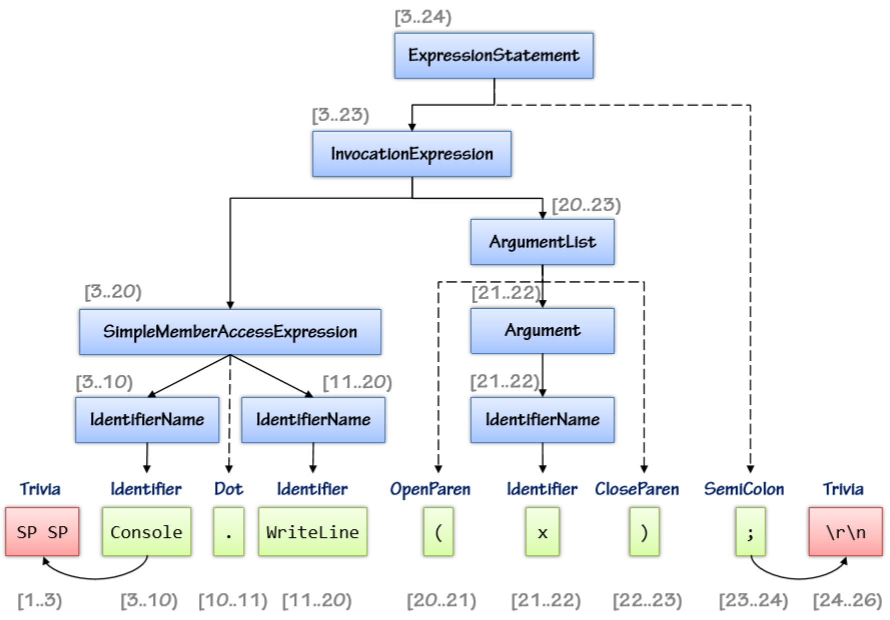

Figure 5.3: Syntax Tree. Image Source: [3]

The most fundamental structure exposed by the Compiler APIs is a **syntax tree** which represents the lexical and syntactic structure of the source code.

Syntax trees can be visualized in Syntax Visualizer tool[3] available in Visual Studio 2015 after installing .NET Compiler Platform SDK. This tool allows to visualize syntax trees in two different ways – as a textual representation or as a syntax graph.

In the code, syntax trees are represented by abstract *SyntaxTree* class which has two language specific derivatives — *CSharpSyntaxTree* for C# code and *VisualBasicSyntaxTree* for VB.NET code.

Syntax trees are made up of three basic components as shown in figure 5.3 [15].

- **Syntax Nodes** – represent various syntax constructs that exist in the language specification such as declarations, statements, clauses and expressions. Each type of syntax node is represented by its own class derived from *SyntaxNode*. Syntax nodes always have other nodes and tokens as children; they are never leaves of the syntax tree.

- **Syntax Tokens** – represent terminals of the language grammar. For example, identifiers (*Console*, *WriteLine*, etc.), operators (+, %, etc.), keywords (*class*, *struct*, *private*, etc.) and punctuation (*, . ;* etc.). They are never parents of other nodes in the tree. Syntax Tokens are represented by *SyntaxToken* type which is a CLR value type. This means that there is one structure for all different kinds of tokens which has various properties with different meaning based on the type of token that is being represented.

- **Syntax Trivia** – syntactically insignificant parts of the source code such as comments and whitespaces. Similar to syntax tokens, syntax trivia are CLR value types represented by *SyntaxTrivia* type. They are never children of other nodes or tokens in the syntax tree. Instead, they can be accessed by inspecting two properties of Syntax Tokens, named *LeadingTrivia* and *TrailingTrivia* which are collections of *SyntaxTrivia* types. Syntax trivia move with the associated token during syntax transformation.

3. `https://github.com/dotnet/roslyn/wiki/Syntax%20Visualizer`

In addition to three basic components shown above, syntax tree also contains exact locations of all nodes, tokens and trivia. These locations can be used for error reporting and they are represented as **Spans** which are half-open intervals. This information is available through two properties:

- ***Span*** – text span from the start of the first token in the node's sub-tree to the end of the last token. This span does not include any leading or trailing trivia.
- ***FullSpan*** – "text span that includes the node's normal span, plus the span of any leading or trailing trivia." [cit.]

Syntax Trees have two main properties:

- **Immutability** – once the syntax tree is created it cannot be changed. This has enormous advantage for concurrent access. Because of this, no locks are required and it allows for asynchronous access. To create a new tree or edit existing one, factory methods can be used. This process reuses existing subtrees in time and memory efficient way.
- **Full fidelity** – syntax trees represent all thee required aspects of the source code such as language constructs, trivia nodes and spans for error reporting. This means that it is possible to get back to the original source text from the syntax tree.

The initial point of creating a syntax tree is *CSharpSyntaxTree* or *VisualBasicSyntaxTree*. These classes expose several static factory methods which enable two ways of creating a syntax tree:

- **Creating syntax tree from existing source text** – using *ParseText* method which takes source text as a string parameter of itself.
- **Creating syntax tree programmatically** – using SyntaxFactory class which expose factory methods for all language constructs such as ClassDeclaration, MethodDeclaration and others. It is also possible to get all the required syntax factory calls to create

piece of code from the code itself using tools like Roslyn Code Quoter [4].

Traversing of the existing syntax trees can be done in one of the following methods [3]:

- **Using object model** – manual traversal which uses the properties of the *SyntaxNode* derived classes.
- **Using query methods** – defined on *SyntaxNode*. These are various methods such as *ChildNodes*, *Ancestors*, *AncestorsAndSelf*, *DescendantNodes* and others which all returns *IEnumerable<T>*, where *T* is *SyntaxNode*. There are also variants for tokens and trivia. Querying the results can be done by using LINQ to Objects.
- **Visitors** – standard pattern to deal with trees. It is useful when programmer wants to find all nodes of a specific type. Visitors use so-called double-dispatch approach:
    1. *SyntaxNode* class has an accept method that takes derivative of *SyntaxVisitor* type (language-specific) as a parameter.
    2. Visitors then can visit syntax nodes using *Visit* method of *SyntaxVisitor* class.

5.3 Semantic Analysis

While using syntax analysis it is possible to look at the structure of the program, this is often not enough. The deeper knowledge of the code is needed -– information about semantics or meaning of a program. This can be accessed using semantic analysis. "For example, many types, fields, methods, and local variables with the same name may be spread throughout the source. Although each of these is uniquely different, determining which one an identifier actually refers to often requires a deep understanding of the language rules." [15].

4. https://github.com/KirillOsenkov/RoslynQuoter

Compilation is analogous to a single project as seen by the compiler and represents everything that is needed to compile a C# or VisualBasic program such as assembly references, compiler options, and set of source files to be compiled. Compilation is an abstract class with two language-specific derivatives – *CSharpCompilation* or *VisualBasicCompilation*. Compilation is also immutable – once is created it cannot be changed. The new compilation must be created with desired changes. Compilation allows to obtain semantic model by calling *GetSemanticModel* method.

Symbols are entities, such as variables, members, namespaces, and types which names and other expressions refer to. The process of associating names and expressions with Symbols is called **binding**. Different kinds of symbols are represented by different interfaces, all derived from *ISymbol*. Each interface has its own set of methods and properties.

Symbols are similar to CLR type system represented by Reflection API. Difference is that Symbol API is executed at compile time of the code, not at runtime using CLR type system as reflection. This allows Symbol API to model not only language constructs like types, members, parameters that are found also in reflection, but also namespace, labels and others.

Analyzing of symbols can be done in three different ways [3]:

- Using interface type checks,
- Using *SymbolKind* enumeration,
- Using *SymbolVisitor* or *SymbolVisitor<TResult>*.

5.3.1 Data Flow Analysis

Data Flow Analysis [3] is an API that answers questions about data flow from and out of a specific region of code. The main entry point of data flow analysis process is *AnalyzeDataFlow* extension method on *SemanticModel* which has various different overloads. All these methods return *DataFlowAnalysis* type which can identify various data, such as [16]:

- Variables that are written inside the region of code (*WrittenInside* property).

- Variables that are written outside the region of code (*WrittenOutside* property).
- Variables that are declared inside the region of code (*VariablesDeclared* property).
- Pieces of data that flow into and out of the region of code.
- Variables that get captured from outer scope (for example in case of lambda expressions or anonymous methods).

5.3.2 Control Flow Analysis

Control Flow Analysis [3] provides information about transferring control in and out of a specific code region. The process is very similar to the data flow analysis. The starting point is *AnalyzeControlFlow* extension method on *SemanticModel* and its overloads which all return instance of *ControlFlowAnalysis* type. This instance then exposes the following properties [3]:

- Indication of reachability of the starting and ending point of the particular region of code (*StartPointIsReachable* and *EndPointIsReachable* properties).
- Identification of all entry and exit points of the analyzed region of code (*EntryPoints* and *ExitPoints* properties).
- List of all return statements that occur inside the analyzed region of the code (*ReturnStatements* property).

5.4 Workspaces

The Workspaces layer provide access to the Workspace API which assists programmers in organizing information about documents in projects and projects in solution into single object model. This model offers direct access to source text, syntax trees, semantic models and compilations without the need to parse the files themselves one by one. Workspace then provides enough content and information to perform operations like code analysis and code refactoring over entire solutions. For example, when renaming a symbol, it is necessary to know all the places across the solution where the symbol is used. This layer also provides set of APIs commonly used by Visual Studio IDE for implementing code analysis and refactoring tools, such as Find All References, Formatting and Code generation APIs.

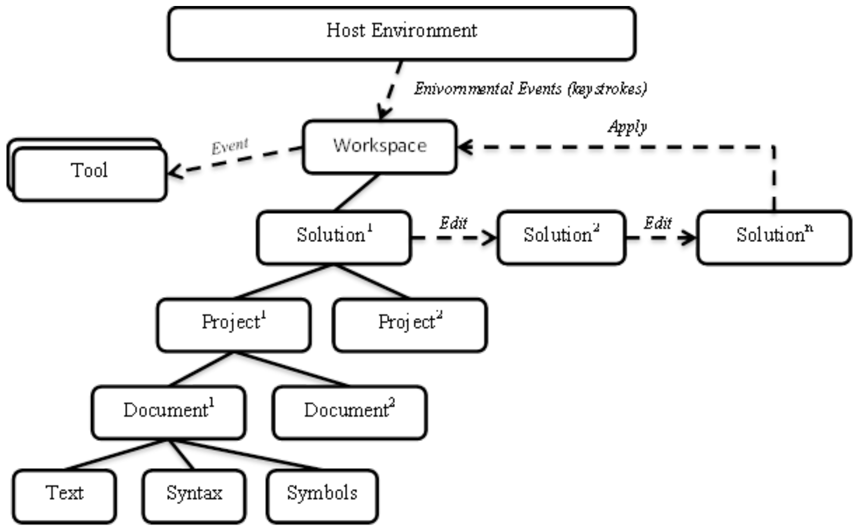

Figure 5.4: Roslyn Workspace Representation. Image Source: [15]

Workspace layer consists of four main components as shown in figure 5.4.

- **Workspace** – active representation of a solution as a collections of projects, each with a collection of documents together with inter-dependencies between projects. Workspace is typically tied to a host environment, but it is also possible to create stand-alone workspaces. Currently, there are four types of workspaces []:
  - **Workspace** – abstract base class for all other workspaces. It cannot be instantiated.
  - **MSBuildWorkspace** – workspace that can open and handle solution (*.sln*) and project (*.csproj*, *.vbproj*) files. However, current implementation of MSBuildWorkspace does not allow creation or editing solution, due to fact that it does not contain any solution file (*.sln*) writer.

  - **AdhocWorkspace** – formerly named CustomWorkspace. This type allows to add artefacts like solution and project files manually.
  - **VisualStudioWorkspace** – active workspace consumed within Visual Studio packages.

- **Solution** – immutable model of projects and documents.
- **Project** – represents all the source code files, both project-to-project and assembly references, and parse and compilation options. Substantial advantage of project is that from it, it is possible to access related compilation without need to parse any source files or resolve project dependencies.
- **Document** – represents a single source file from which the text of the file, the syntax tree, and the semantic model can be obtained directly.

Each one of these components is immutable data structure.

Workspace always maintains up-to-date model of the entire solution. Every time this model is changed, a *WorkspaceChanged* event is fired to signal this and *CurrentSolution* property is updated. This event also contains information about which document was modified. To get access to the current model of a solution there are two options. First, is to get the most recent solution from the event itself and the second, is directly from the workspace's *CurrentSolution* property. To save the changes, it is necessary to explicitly apply the changed solution back to the workspace.

It is possible to return to the previous versions of projects and documents. This can be achieved by the fact that each project or document have a unique ID. When the change occurs, new project (or document) is created with different ID that the previous one. Then it is possible to get to the desired version by calling *GetProject(ProjectId)* or *GetDocument(DocumentId)* methods.

Workspace APIs provide following services [3]:

- **Classification service** – supports the concept of rendering documents in a code editor which means classifying text spans into different categories such as identifiers, literals, comments, etc. This will allow things like syntax colorization in IDE. Classifying text span is done by calling *GetClassifiedSpans* method in static *Classifier* class.

- **Formatting service** – automatically formats code according to defined rules such as placement of whitespaces, curly braces and so on. There are several usage patterns of this service. One of them is directly calling *Format* or *FormatAsync* method in static *Formatter* class.

- **Symbol finding** – goal is to find occurrences of symbols which requires context across the whole solution. This allows to implement things like "find all references" (using *FindReferencesAsync* method in *SymbolFinder* class), "go to definition" (*FindDeclarationsAsync* method) or call hierarchy (*FindCallersAsync* method).

- **Recommendation service** – used for suggesting symbols at a cursor position. For example, IntelliSense functionality in Visual Studio. Recommendations service can be used by calling *GetRecommendedSymbolsAtPosition* method in static class *Recommender*.

- **Renaming service** – allows applying name changes across the whole solution. Renaming is done by calling *RenameSymbolAsync* method in static *Renamer* class. This method returns a new edited copy of the solution.

- **Simplifier service** – allows to simplify a piece of syntax in a semantically sound manner. This can be done by calling one of the methods in the *ReduceAsync* family of methods in static *Simplifier* class.

5.5 Analyzers and Code Fixes

Roslyn can also be used to extend the compiler pipeline as well as language services with customer analyzers and code fix providers. Basic concept that analyzers and code fixes use is called diagnostics.

5.5.1 Diagnostics

Diagnostics can be seen as an abstraction over potential outcomes of the analysis performed by the compiler. They are containers with various metadata such as warnings, errors, warning levels and others. Programmatically, diagnostic is an abstract base class defined in *Microsoft.CodeAnalysis* namespace with variety of properties.

Diagnostics can be retrieved in two ways [3]:

1. Calling *Emit* method on a compilation which returns *EmitResult* object. This object contains *Diagnostics* property which is basically a collection of diagnostics.
2. Using *AnalyzerDriver* class which has two methods named *GetDiagnostics* and *GetEffectiveDiagnostics*.

5.5.2 Analyzers

Core purpose of analyzers is to return set of diagnostics for a given piece of code. In practice, analyzers provide two primary usages:

- **Enforce certain desired coding styles and practices** – as a replacement for similar to tools like StyleCop[5] and FxCop.
- **Provide guidance to developers consuming a library** – as insurance that the consumers of the library will not do certain things.

Analyzers are types that implement the *IDiagnosticAnalyzer* interface which has many derivatives — *ICompilationAnalyzer*, *ISyntaxTreeAnalyzer*, *ISymbolAnalyzer*, *ICodeBlockAnalyzer*, *ISemanticModelAnalyzer* and others. Each one provides different type of analyses and they all return diagnostic objects.

Steps to build a custom analyzer are dependent upon the kind of analyzer that is being developed. For example, for a symbol analyzer, these steps are [cit.-plural]:

1. **Declaring the diagnostic descriptors** – by implementing *SupportedDiagnostics* property defined on *ISymbolAnalyzer*.

5. `https://stylecop.codeplex.com/`

2. **Declaring the symbol kinds to analyze** – by implementing *SymbolKindsOfInterest* property defined on *ISymbolAnalyzer*.

3. **Performing the desired analysis** – after checking for specific conditions of the analyzed symbol, the call to the *addDiagnostic* delegate should be made to report diagnostic instance to the environment.

4. **Exporting the analyzer** – by putting custom attributes in type definition of newly created analyzer.

5.5.3 Code Fix Providers

While analyzers and diagnostics provide way to complain about certain conditions in user's code, the code fix providers can suggest real fixes to those issues.

Providing code fixes is a two-step process consisting of:

1. **Perform analysis which returns set of diagnostics**.

2. **Apply code fixed to mitigate issues** – light bulb experience in Visual Studio.

Code fix providers work by implementing *ICodeFixProvider* interface defined in *Microsoft.CodeAnalysis.Workspaces* assembly. This interface has a single method called *GetFixesAsync* which returns set of *CodeActions* objects that can carry out edits to documents or whole solutions and they are also used for various services in Visual Studio.

Besides implementing *ICodeFixProvider* it is also necessary to annotate the code fix type using *ExportCodeFixProvider* attribute (similar to analyzers).

6 Refactoring of FNZ Codebase Using Roslyn

Practical part of this thesis is dealing with creation of automatic refactoring support for large FNZ legacy code base.

As was stated in part 2.2, the main reason behind creating this tool is that building whole code stream takes enormous amount of time, mainly due to large number of projects (more than 250) that are present there. Since the legacy code is replicated in various code streams the manual refactoring and projects transformation would be impossible. Various approaches to tackle this issue had been considered. One was to use manual XML transformation of project files and tackle the special refactorings problem with automatic text editing. This approach led to various problems, so it was decided to examine possibilities of Roslyn compiler described in chapter 5. From the description and official documentation, it would seem that the Roslyn compiler provides all the tools needed to solve all the different problems which were discovered during test implementation using XML transformation. Unfortunately, during development of the application, several problems occurred that made some of the things impossible to achieve. Initial thought was to do everything through Roslyn compiler, however it turned out, that many things Roslyn does not support yet. Furthermore, there are currently more than 1800 issues reported on GitHub[1] alone.

6.1 Tool Implementation

Tool is implemented as a script command line tool that takes path to the solution as a command line argument. All options are described in part 6.2.

Whole refactoring process consists of four steps which are divided into their own classes:

1. SolutionLoader Class
2. SolutionProcessor Class

1. https://github.com/dotnet/roslyn/issues

3. ReferenceProcessor Class

4. NamespaceProcessor Class

6.1.1 SolutionLoader Class

The main task of the ***SolutionLoader*** class is to open solution file and load all the referenced projects and files from the disk into the memory. The main method that used is Roslyn's *OpenSolutionAsync* method on workspace instance which takes one parameter – path to the solution. It is asynchronous method which uses task-based asynchronous pattern (TAP) which was introduced in the .NET Framework 4.0. It returns Task<Solution> as a result.

SolutionLoader class also contains time logging feature to provide information about time needed to open the solution.

The serious problem that was stumbled upon during development is enormous time it takes Roslyn to completely load a solution from a disk into a memory. As was stated in part 5.4, Roslyn provides several types of workspaces from which the developer can choose from. This tool is using MsBuildWorkspace, since it can open and handle MsBuild (*.sln*) and project (*.csproj, .vbproj*) files automatically. However, in some cases this type of workspace takes enormous time to open the solution. This is due to the fact that it recursively loads all the references of every project in the solution, even those that have been already loaded. The severity of this problem depends on number of references between the projects themselves. In solutions where the interconnection between projects is low, the time it takes to completely load the solution is between 10 and 30 minutes. However, when the interconnection is high, the total time can rise up to tens of hours – in one solution, the total time was over 53 hours.

There is also another type of workspace provided which enables adding project files manually – AdHocWorkspace. Unfortunately, the code behind this workspace is in the same as in MSBuildWorkspace and to resolve ambiguous references described in part 6.1.4, all the projects need to be included in the memory.

6.1.2 SolutionProcessor Class

SolutionProcessor class takes all files and based on type move them to new specific projects. It has one public method called *Run* which runs whole processing. First idea was to use Roslyn to get all these things done, but during development various problems were discovered. First problem is that Roslyn does not support creation of new project files due to fact that it does not contain any solution file writer. The solution was to create two final projects manually before the script can be applied:

- Legacy.CSharp,
- Legacy.VisualBasic.

Another problem is that Roslyn cannot move files on its own which results in using common .NET features to do this and then MsBuild API to edit the project files. More specifically, there are two relevant items in the project files in this step:

- **Compile** — represents the source files for the compiler.
- **Content** — represents resources that muse be copied unaltered.

6.1.3 ReferenceProcessor Class

ReferenceProcessor class copies all the required references from the old projects into the new ones. In the Microsoft project files there are two kinds of project references:

- **Reference** – represents an assembly (managed) reference in the project. The relevant items which are all optional are:
  - **HintPath** – relative or absolute path of the assembly.
  - **Name** – represents display name of the assembly.
  - **Private** – specifies whether the reference should be copied to the output folder.
  - **SpecificVersion** – specifies whether only the version in the fusion name should be referenced.

- **ProjectReference** – represents a reference to another project. The most relevant items (are all optional) are:
  - **Name** – display name of the reference.
  - **Project** – a GUID for the reference, in the format {12345678-1234-1234-1234-1234567891234}.
  - **Package** – represents the path of the project file that is being referenced.

Similar to the SolutionProcessor class, the initial idea was to use methods provided by Roslyn to copy these references. However, during development several bugs were found and raised. The most important one is that the full path to the assembly is added into the project while adding metadata references via Roslyn. This was again resolved by using MsBuild API.

6.1.4 NamespaceProcessor Class

The main purpose of the ***NamespaceProcessor*** class is to deal with ambiguous references which can arise during merging projects together as shown in previous steps. This situation can happen when there are classes or types with the same name in different projects. After merging those projects together, the ambiguous references can occur. Solution here is to replace them to also include the full namespace.

This step is currently being resolved using semi-automatic refactoring tool ReSharper presented in part 4.1 due to the various bugs found in Roslyn compiler during development.

6.2 Tool Usage

There are two prerequisites in order to use the tool:

1. VisualStudio 2015 must be installed,
2. Roslyn must be installed from the Nuget Package Manager.

Tool incorporates command line argument parser, which based on provided arguments enables or disables different options of the tool. Parameters use the following notation: **-paramName paramValue** and they can be entered in any order. Complete list of all parameters is shown in figure 6.1.

Since the real code that this tool is design to process cannot be provided due to the fact that it is property of FNZ, this thesis has also the sample solution included, which is able to demonstrate all the aspects of the code processing itself. This code is located in *Sample.zip* archive.

Parameter	Value	Description
ConsoleLoggingOn		Turns on logging to console (default is off)
FileLoggingOn		Turns on logging to file (default is off)
SolutionPath	<path> as string	Path to the solution. Mandatory parameter.
ProjectsListFilePath	<path> as string	Path to the file with project files that need to be processed. If this parameter is not present, the tool automatically processes every project in the solution.
LogFileDirectory	<path> as string	Path to the directory where the log files will be stored. If this parameter is not present, the tool automatically creates a directory "C: \LogRoslyn \".

Table 6.1: Command line arguments of the tool.

6.3 FNZ Code Solution Usage

After completely processing FNZ code solution, the time to run whole build was decreased in some cases by 30%. The future decrease should be possible with enhancements described in part 6.5.

6.4 Comparison with Semi-automatic Refactoring Tools

Semi-automatic refactoring tools described in chapter 4 provide several advantages over the compilers like Roslyn. The most important ones are **stability** and **functionality** since most of these tools have been on the market for quite some time. They also have very **large user base** so in case of any problem, the solution can be easily found on various developer's forums. In contrast with Roslyn, they have all the different **refactorings provided**, often with GUI. They can be described as **out of the box solutions** which are suitable for most of the situations.

However, if the **full customization** is the desired feature, then the Roslyn provides better solution. With all the available options, like code fixes and analyzers, it is possible to build refactoring tool from the scratch, which can be fully suited to the particular needs of the developer. This can be useful for special cases like the one described here – when the whole refactoring process must be fully automatized. Unfortunately, as it turned out, Roslyn has many bugs which is making the whole development process difficult and some things are even impossible to achieve. The approach that was presented here is combination of both, semi-automatic refactoring tool – ReSharper and Roslyn compiler.

6.5 Future Work

New versions of Roslyn compiler are being constantly released to the Nuget server with many bug fixes. This will allow to mitigate different issues described in previous parts of the chapter with attention to the most serious ones – long time to open the solution and bugs in namespaces processing.

Another proposed enhancement is to use Roslyn compiler to divide implementation from the API by further dividing the projects into four categories:

- Legacy.CSharp,
- Legacy.CSharp.Api,
- Legacy.VisualBasic,
- Legacy.VisualBasic.Api.

7 Conclusion

The focus of this thesis was to provide overview of different refactoring techniques that can be applied to large codebases. Several semi-automatic refactoring tools have been assessed and compare to each other. The specific problem of the FNZ code solutions however does not allow to use them, so the abilities of the Roslyn compiler have been presented and it was decided that it can be used to automate refactoring process in large company. The final approach that was used is the combination of ReSharper and the Roslyn compiler. The main advantage of the Roslyn compiler with comparison to the semi-automatic refactoring tools – the full customization was diminished by several issues with different severities which were found during development. This made some of the things impossible to achieve. However, as the new versions of the Roslyn compiler are being constantly released, these bugs will be eventually resolved and the whole tool will be updated in the future versions.

Bibliography

[1] *CodeRush for Roslyn*. 2015. URL: https://community.devexpress.com/blogs/markmiller/archive/2015/06/09/coderush-for-roslyn.aspx.

[2] *CodeRush Pricing*. 2015. URL: https://www.devexpress.com/products/coderush/pricing.xml.

[3] Bart De Smet. *Pluralsight: Introduction to the .NET Compiler Platform*. 2014. URL: https://www.pluralsight.com/courses/dotnet-compiler-platform-introduction.

[4] Donald Bradley Roberts, Ph.D. "Practical Analysis for Refactoring". PhD thesis. Urbana, Illinois: University of Illinois at Urbana-Champaign, 1999.

[5] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, Inc., 1999. ISBN: 0-201-48567-2. (Visited on 11/25/2015).

[6] *JustCode Overview*. 2015. URL: http://www.telerik.com/products/justcode.aspx.

[7] *JustCode Pricing*. 2015. URL: http://www.telerik.com/purchase/individual-justcode.aspx.

[8] Fowler Martin. *Workflows of Refactoring*. 2014. URL: http://martinfowler.com/articles/workflowsOfRefactoring/.

[9] Robert C. Martin. *Clean Code*. Pearson Education, Inc., 2011. ISBN: 0-13-235088-2. (Visited on 11/18/2015).

[10] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C#*. Pearson Education, Inc., 2007. ISBN: 0-13-185725-8. (Visited on 11/20/2015).

[11] *ReSharper Discounts*. 2015. URL: https://www.jetbrains.com/resharper/buy/#section=discounts.

[12] *ReSharper Plugins*. 2015. URL: https://www.jetbrains.com/resharper/plugins/.

[13] *ReSharper Subscription*. 2015. URL: https://www.jetbrains.com/resharper/buy/.

[14] *ReSharper Ultimate*. 2015. URL: https://www.jetbrains.com/dotnet/.

[15] *Roslyn Overview*. 2015. URL: https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview (visited on 09/30/2015).

[16] Josh Varty. *Learn Roslyn Now: Part 10 Introduction to Analyzers*. Apr. 2015. URL: `https://joshvarty.wordpress.com/2015/04/30/learn-roslyn-now-part-10-introduction-to-analyzers/`.

A Attachments

- **CodeRefactorerTool.zip** – code of the refactoring tool.
- **Sample.zip** – sample solution to demonstrate all the aspects of the code processing.