FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Martin Hubata

Conditional Branching Assistant for C#

Department of Software Engineering

Supervisor of the bachelor thesis:	Mgr. Ing. Robert Husák
Study programme:	Computer Science
Study branch:	Programming and Software Development

Prague 2024

First, I want to thank my supervisor for his patience and advice. I also want to thank my friends and parents for their support. Lastly, I want to thank my sister for proofreading.

Title: Conditional Branching Assistant for C#

Author: Martin Hubata

Department: Department of Software Engineering

Supervisor: Mgr. Ing. Robert Husák, Department of Software Engineering

Abstract: Convoluted `if` statements are often the reason why code written by beginners is hard to understand, and beginners might not even realize there are other options for writing functionally identical code. Existing code manipulation tools for C# do not focus on beginners. Their main focus is on saving time one would spend manually modifying the code. We developed an extension for Visual Studio and C# that presents the user with concrete small-sized refactoring options and recommends which ones to apply. The extension also helps the user understand the refactors through code highlighting in both the old and the previewed refactored code. The extension allows the user to weigh the changes themselves and decide if they are worth it. If they follow our recommendations, this results in more easily understandable and more maintainable code.

Keywords: Branching, Static analysis, Refactoring, C#, Microsoft Visual Studio 2022, Beginner programmers

Contents

Introduction

All beginners make mistakes, and beginner programmers are no different. There are two kinds of mistakes in code: functional and stylistic. Functional mistakes can be very difficult to fix, as they have a wide range, as wide as what you can do with programming. Getting rid of them constitutes a lot of what a programmer actually does. Stylistic mistakes are not isolated from them. If there are no functional mistakes but a lot of stylistic ones, the program will do what it is meant to, but the code itself will be hard to understand. Moreover, stylistic mistakes can make it unreasonably difficult or even impossible to implement any modifications.

Stylistic mistakes, such as bad naming conventions or overly long source code files, can be simple and easy to fix. However, some stylistic mistakes can also be very complex without a single objectively correct answer. One such complex and subjective area, and the subject of this thesis, are `if` statements and their conditions.

`if` statements are, in one way or another, necessary in every programming language. The question is: What can we improve about beginner-written `if` statements? The answer, of course, depends on a lot of context.

```
void m(int pipeCapacity, bool inUse,
        string maintenanceInstructions)
{
    if (pipeCapacity == 0)
    {
    /*replace it, it is too broken*/
    }
    else
    {
        if (inUse && maintenanceInstructions != "decide")
        {
        /*turn off water and do what instructed to*/
        }
        else if (inUse)
        {
            if (maintenanceInstructions == "decide"
                || pipeCapacity == 0)
            {
            /*turn off water, replace it just to be sure,
            or because nothing flows through*/
            }
            else if (pipeCapacity == 42)
            {
            /*investigate the magical pipe*/
            }
        }
    }
}
```

Listing 0.1 Introductory example.

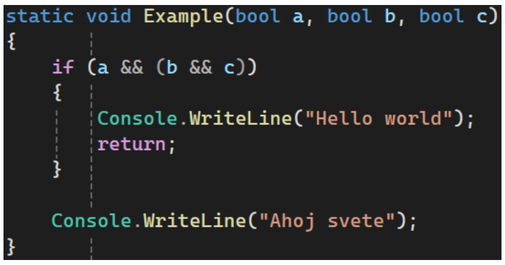
Listing 0.1 seems to be very questionably composed. There are purely stylistic choices, such as `if (pipeCapacity == 0)` lexically contains less code than its `else`, hence maybe they should be swapped and the condition negated. However, a more serious potentially functional mistake is that `else if (pipeCapacity == 42)` is actually unreachable. It also logically seems a little out of place, maybe because the beginner author actually wanted to place it a nesting level higher.

```
void m(int pipeCapacity, bool inUse,
        string maintenanceInstructions)
{
    if (pipeCapacity == 0)
    {
    /*replace it, it's too broken*/
    }
    else if (inUse && maintenanceInstructions != "decide")
    {
    /*turn off water and do what instructed to*/
    }
    else if (inUse)
    {
    /*turn off water, replace it just to be sure,
    or because nothing flows through*/
    }
}
```

Listing 0.2 Listing 0.1 refactored.

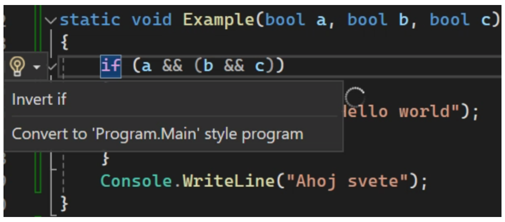
Listing 0.2 shows functionally equivalent code. Notice that `else if (pipeCapacity == 42)` completely disappeared. The issue with it was not actually a problem with the `pipeCapacity == 42` condition, but with the previous condition `maintenanceInstructions == "decide" || pipeCapacity == 0`, which always ends up being true in the place where it was in the code. This results in the code that was under `if (maintenanceInstructions == "decide" || pipeCapacity == 0)` now only being under the condition of `inUse`. We also flattened the whole structure so there are no nested `if` statements.

Beginners can currently make use of the many resources on the Internet to improve their code. They can also learn in courses or generally get help from an experienced programmer, but these options might be less accessible. However, if experienced programmers can identify and recommend solutions to these mistakes without much trouble, these mistakes could possibly be identified and fixed programmatically, at least to an extent.



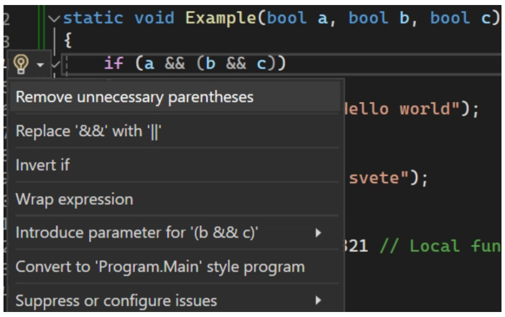
```
static void Example(bool a, bool b, bool c)
{
    if (a && (b && c))
    {
        Console.WriteLine("Hello world");
        return;
    }

    Console.WriteLine("Ahoj svete");
}
```

Figure 1 A top-level function.



```
static void Example(bool a, bool b, bool c)
{
    if (a && (b && c))
                              ello world");
    }
    Console.WriteLine("Ahoj svete");
}
```

(a) on `if` keyword



```
static void Example(bool a, bool b, bool c)
{
    if (a && (b && c))
                              ello world");
                              svete");
                              21 // Local fun
```

Remove unnecessary parentheses
Replace '&&' with '||'
Invert if
Wrap expression
Introduce parameter for '(b && c)'
Convert to 'Program.Main' style program
Suppress or configure issues

(b) on unnecessary opening parentheses

Figure 2 Examples of Quick Actions on Figure 1.

Visual Studio already offers a range of possible Quick Actions that enable the user to restructure their code stylistically. Figure 1 shows a short function in a program that utilizes top-level statements.

Figure 2a shows Quick Actions we get when opening the menu from the `if` keyword. We get the option to invert the `if`, which is a very local change. But we also get an option to convert the whole program to a `Program.Main` style, which does not directly concern the `if`, and is there due to the utilization of top-level statements.

Figure 2b shows Quick Actions available when accessed from the unnecessary opening parentheses in the `if` statement condition. We get many options with very different ideas and consequences. Wrapping expression only changes the whitespaces, removing unnecessary parentheses changes the condition a little, and introducing a parameter for part of the condition would change the code a lot. We also have the options that were available for the `if` keyword as well.

All of these options together can be overwhelming for the beginner user. Some options are only available very locally, while others are available in the whole code. Furthermore, they are just options. They are offered when they are possible but with no further commentary on whether it is a good idea to utilize them.

The goal of this thesis is: To help beginner programmers improve the `if` statements in their existing code and teach them how to implement these improvements themselves in the future. We create a tool that allows the users to explore the `if` statements and offers functionally equivalent refactors. We recommend these refactors based on whether they are generally better or worse than the unmodified code. We explain why the refactors are functionally the same by highlighting structures that make them possible in the user code. This can challenge the user to think about whether the current code does what they want if a refactor we offer does not seem to do so. The tool can also be used by experienced programmers who often come into contact with beginner code to help them refactor it.

We will now describe how this thesis is structured:

- **Background** describes existing adjacent works, both theoretical and practical. Theoretical works are primarily relevant for analysis, but even the practical works shaped some of our early discussions. The practical works descriptions also include the existing software we utilize in our implementation.

- **Analysis** discusses constraints we have to set on our analyses for them to be reasonably complex. We then discuss different ways we could go about our goal, both on the raw refactors side and also the user interaction side, while advocating the approach we think is the best.

- **Solution** introduces the general framework of our solution to the conclusions of **Analysis**. We also concretize specific code mistakes and describe them using the mentioned framework.

- **Implementation** describes specific implementation solutions that are a consequence of the framework introduced in **Solution**. We also give an overview of the whole implementation and its integration into Visual Studio.

- **Conclusion** summarizes what we managed to achieve, and gives ideas on how to extend our work further.

- **Appendix A** describes structure of the attachment.

- **Appendix B** describes how to go about installing and using the software we developed to achieve the goal of this thesis.

Chapter 1

Background

We will start by introducing existing works relevant to the thesis.

1.1 C# If Statement Syntax

Our primary focus in the thesis is `if` statements. Thus, we will describe some C# specific quirks of `if` statements, but also related code structures, such as boolean expressions that constitute conditions. A C# `if` statement has the basic anatomy[1] of:

```
if(conditon) ifEmbeddedStatement
```

With the optional `else`:

```
if(conditon) ifEmbeddedStatement else elseEmbeddedStatement
```

1.1.1 Condition

`conditon` is always a boolean expression; unlike C/C++, it cannot be, for example, an integer that gets interpreted as a boolean automatically.

Before it delves into non-boolean subexpressions, `conditon` is most often made out of lazily evaluated operators `&&` and `||`. Lazily evaluated means they act as their non-lazy variants `&` and `|`, but their second operand is only evaluated if it can change the result with respect to the result of the first operand. Importantly, `&&` has precedence over `||`.

Most `conditons` also delve into operating on non-boolean operands at some point. This is usually done by the operators `==`, `!=`, `<`, `<=`, `>`, and `>=`. These can be overloaded. It must be implemented in pairs of supposedly inverse operators, like `==` and `!=`. But there is no active enforcement of the implementations being the inverse of each other. One could, for example, make both always be true.

A final note on overloading: `&&` and `||` cannot ever be overloaded due to language rules. Moreover, they cannot be overloaded on the base `bool` type, as one simply cannot change the existing behavior of base types in C#. This means we can rely on the language-defined behavior of `&&` and `||`.

It is possible to declare a variable in the argument list of a method call using the syntax `out int variableName` (or any other specific type instead of `int`, or `var`). Due to methods being callable in expressions, this means that one can declare variables in `conditon` of an `if` statement.

```
void m(string unparsedNumber) {
    if (int.TryParse(unparsedNumber, out int number))
    {
        Console.WriteLine(number);
    }
}
```

Listing 1.1 Declaration of a variable in a condition.

```
void m(string unparsedNumber) {
    int number;
    if (int.TryParse(unparsedNumber, out number))
    {
        Console.WriteLine(number);
    }
}
```

Listing 1.2 Listing 1.1 reformulated.

Listing 1.1 is a common use case. It behaves exactly the same as Listing 1.2, meaning that the variable `number` exists after the `if` even in the first example.

1.1.2 If Embedded Statement

`ifEmbeddedStatement` does not have to be a block statement; it can directly contain a single non-block statement. That statement could even be another `if` statement.

Block statements are often used as the `ifEmbeddedStatement` to effectively allow multiple statements in a place where syntax rules allow only one. They are often used even when they contain only one statement to keep the code consistent. They also limit the scope of variables, which is sometimes necessary to make the code unambiguous. The problem with ambiguous code is that it simply cannot be run.

```
void m(bool greetEnglish, bool greetCzech) {
    if (greetEnglish) {
        string x = "Hello world";
        Console.WriteLine(x);
    }

    if (greetCzech) {
        string x = "Ahoj svete";
        Console.WriteLine(x);
    }
}
```

Listing 1.3 Example where identically named variable scopes do not overlap.

Listing 1.3 is an example where the blocks limit the scope of both variables `x`, which makes the scopes not overlap, and every usage of `x` clearly refers to one or the other.

```
void m(bool greetEnglish) {
    if (greetEnglish) {
        string x = "Hello world";
        Console.WriteLine(x);
    }

    string x = "Ahoj svete";
    Console.WriteLine(x);
}
```

Listing 1.4 Example where identically named variable scopes do overlap.

Listing 1.4 is an example where that is not the case. The scope of the variable `x` declared in `string x = "Ahoj svete"` is the whole method. This overlaps with the variable declaration in `string x = "Hello world"`, meaning the second `Console.WriteLine(x)` is ambiguous.

1.1.3 Optional Else

Everything we said about `ifEmbeddedStatement` also applies to `elseEmbeddedStatement`. Once again, `elseEmbeddedStatement` can be an `if` statement. An `else if` statement is just an `else` statement with `elseEmbeddedStatement` being an `if` statement; it has no special dedicated construct in the language.

The dangling `else` problem is solved in the usual way [2], with the `else` being attributed to the testually closest preceding `if` without an `else` yet.

1.2 Code Complexity Metrics

There are many established code complexity metrics, for example:

1. McCabe's cyclomatic complexity [3], which measures the number of linearly independent paths through the source code. It is important to note that complex conditions do not influence it; either the condition ends up being true or false. It only analyzes static code and not the actual running behavior.

2. The multiple vocabulary complexity metrics proposed by Halstead [4] which work with the total and distinct numbers of operators and operands.

3. The length of the code itself is also relevant.

The impact of these metrics was, among others, measured even using medical devices [5]. And while they certainly have their flaws, considering how we impact them is still valuable.

1.3 Roslyn

.NET Compiler Platform [6], codenamed Roslyn, is a C# and Visual Basic compiler that also exposes the tools and data structures used in the compilation. This is for the purpose of code analysis and refactoring, which is exactly what we want to do. There are two distinct types of analysis: syntactical and semantical. We primarily focus on the simpler syntax analysis. It is sufficient for analyzing the structure of the code. Still, we must delve into semantics at least a little.

1.3.1 Syntax Data Structures

Let's first describe the syntax data [7] structures Roslyn and we use. The syntax analysis in Roslyn parses the code into the following structures: `SyntaxTree`, `SyntaxNode`, `SyntaxTokens` and `SyntaxTrivia`.

Every statement, expression, declaration, and other complicated syntax language constructs get parsed into a descendant of `SyntaxNode`. Importantly, every `SyntaxNode` has a reference to its parent and also the whole tree it is in, which means even with a specific `SyntaxNode` in hand, we can analyze code around it. Every `SyntaxNode` also has a list of its children `SyntaxNodes`. This can mean different things for different constructs and the corresponding Roslyn types. For `BlockSyntax`, which represents a block, it includes the statements directly in it. But for an `IfStatementSyntax` it includes both the condition, and the embedded statement of the `if`, which might seem unexpected to a new user.

For many cases like `IfStatementSyntax` Roslyn exposes properties that make working with the child nodes easier than with the general `SyntaxNode`. `IfStatementSyntax`, as an example, has `IfStatementSyntax.Condition` and `IfStatementSyntax.Statement`. These are already very specifically typed, as, for example, `IfStatementSyntax.Condition` is `ExpressionSyntax`, which represents expressions.

If we want to further inquire if the condition is comprised of `&&`, we have to use the function `condition.IsKind(SyntaxKind.LogicalAndExpression)`, and then potentially cast the `condition` to `BinaryExpressionSyntax`. Notice that the `SyntaxKind.LogicalAndExpression` is more specific than the expression type we cast to, which is a design decision made by Roslyn to have less `SyntaxNode` descendants, but have them be a bit more parametrizable.

This parametrization is, among other things, done by `SyntaxTokens`. These represent small fragments of the code like keywords, identifiers, operators, and more. For example, `BinaryExpressionSyntax binExpr` is `binExpr.IsKind(SyntaxKind.LogicalAndExpression)` if `binExpr.OperatorToken.IsKind(SyntaxKind.AmpersandAmpersandToken)`.

Finally, there is trivia, which represents parts of the source code that do not change its functionality, examples being whitespaces and comments. Note that, for example, the whitespace in `return true;` is necessary in the source code. However, once Roslyn has parsed it into nodes and tokens, the whitespace is kept as trivia, but it is no longer necessary.

Roslyn also keeps the information about the textual span of the structures it parses the code to, which is completely unnecessary for code analysis. However, it is very useful when one also wants to work with the source code as a text, which we do during highlighting.

1.3.2 Creating and Changing Syntax Data Structures

So far, we have described the different Roslyn syntax structures and shown ways how to access information through them. However, the point of the thesis requires us to be able to change the code. When building nodes from scratch, Roslyn uses the class `SyntaxFactory`, which has methods that serve for the creation of new `SyntaxNode` or `SyntaxToken`. For example, `SyntaxFactory.BinaryExpression(SyntaxKind kind, ExpressionSyntax left, ExpressionSyntax right)`.

One could rebuild any existing code using just these methods, but a much more usual use case is that we use these when necessarily creating new nodes, but then we reuse existing structures instead of recreating them. It is important to note that the `SyntaxNode` and its descendants are strictly immutable. Any changes done to them produce a new `SyntaxNode`, while leaving the old one unchanged.

These changes are locally done by functions like `IfStatementSyntax`

`.WithCondition`, which lets us replace a certain part of a `SyntaxNode` while keeping the rest. In `IfStatementSyntax .WithCondition`, we replace the condition of an `IfStatementSyntax`, but the embedded statement stays the same. Note that these operations produce a whole new `SyntaxTree` where the node we changed is its root.

On a larger scale, they are done by the methods `SyntaxNode.ReplaceNode` and `SyntaxNode.RemoveNodes`. These once again make the copy of `SyntaxNode` we call them on the root of the newly created `SyntaxTree`. but if we call these on a root of a `SyntaxTree`, the result will be that root with the replaced or removed nodes.

1.3.3 Semantic Errors

Semantic analysis is necessary for detailed type analysis, variable tracking, and other more complicated code analyses outside of the scope of our thesis. However, semantic analysis is also necessary to determine if the code contains any semantic errors or specific ones. We have already shown an example of a semantic error that includes an `if` statement in Listing 1.4. We can access semantic errors of the analyzed code by querying `SemanticModel`, which represents the semantic analysis, for its diagnostics. These include errors but also warnings and informative messages. We can then filter those by severity or specific issues they diagnosed.

1.3.4 Tracking and Annotations

The immutability of `SyntaxNode` necessitates a way to track a `SyntaxNodes` across multiple `SyntaxTrees`, where they were duplicated simply because of the immutability. This is necessary when we want to make changes that cannot be done using only one Roslyn API call.

One way is to ask Roslyn to track a `SyntaxNode` using `SyntaxNode.TrackNodes`. We can get the current version of a `SyntaxNode` by passing the old version of the `SyntaxNode` to `rootOfNewTree.GetCurrentNode`. We utilize this approach sometimes.

Another way to “track” `SyntaxNode` in a sense is to use `SyntaxAnnotation`. Using `SyntaxNode.WithAdditionalAnnotations`, we can annotate a `SyntaxNode` with a specific `SyntaxAnnotation`, possibly multiple `SyntaxNodes` with the same `SyntaxAnnotation`. We can then get all of the annotated nodes using `SyntaxNode.GetAnnotatedNodes`.

1.3.5 Existing Tools

Roslyn already includes a way to provide refactors to users. This is done using Analyzers and CodeFixes [8]. The former detects when a rule in the source code is broken, while the latter provides a fix that refactors the code to adhere to that rule.

Our goal fits the goals of these APIs in some respects but not in others. The raw changes we propose could be considered CodeFixes, but the supporting structures around them that serve as guidance to beginners are not the primary focus of Analyzers and CodeFixes. Moreover, the implementations are not supposed to be IDE-specific.

1.4 Visual Studio

Visual Studio [9] is an integrated development environment. When used to develop C#, Visual Studio already runs a Roslyn instance under the hood for the purposes of syntax highlighting and such. We can borrow that instance of both syntax and semantic analysis without running our own, saving resources.

Visual Studio exposes several APIs that let us manipulate the usual windows that serve to display source code, including a way that lets us do our own highlighting.

We can integrate our own dedicated user interface into Visual Studio using their extensibility system [10], which allows us to even have our own window using WPF.

1.4.1 Visual Studio Community Toolkit

Visual Studio Community toolkit [11] is a package that aims to simplify writing Visual Studio extensions. The problem with many of the raw APIs for extensions is that they are cumbersome to access, often requiring a type parameter, a GUID, going through a web of properties, and/or numerous casting to different types. Visual Studio Community toolkit simplifies both access and manipulation of documents, which we use extensively. It also allows easier access to Visual Studio services, including the running Roslyn instance. For a more UI benefit, it allows the automatic theming of tool windows to align them with the rest of Visual Studio. Finally, it provides a simple way to log exceptions.

Chapter 2

Analysis

Our thesis aims to refactor `if` statements in beginner code without changing their behavior, help beginners understand why the behavior remains the same, and consequently help beginners improve any code they write in the future. There are many ways one could achieve that goal. First, we had to choose a language. We chose C# as the language to analyze because it is a common beginner language, but one could certainly do what we do in the thesis in another language. However, after choosing a language (where there are many similarly good options), there are many more design decisions to be made and limitations to be set. In this chapter, we describe those decisions and limitations, as well as our thought processes behind them.

2.1 Constraints

The refactoring part of our goal necessitates that we define a number of limitations as complicated approaches like symbolic execution [12], which analyzes what inputs cause what parts of the code to execute, fall outside of the scope of a bachelor thesis.

2.1.1 Semantic Correctness Assumption

Helping beginners refactor semantically incorrect code is a commendable endeavor in some cases, but it goes against our focus of not changing its behavior, as such code has no behavior at all. So, in the whole thesis, we assume the user code is semantically correct, and we want our refactors to also only produce semantically correct code.

2.1.2 Conditions

`if` statement conditions are, by their nature, complicated. They can contain many language constructs, forcing us to interact with them. Thus, we must impose some constraints on how we approach certain constructs directly in the conditions, and we often care about the constructs surrounding the `if` statements the condition is in.

No In-Between Run Code

```
void m(bool greetInEnglish, Brewery brewery)
{
    if (greetInEnglish) {
        Console.WriteLine("Hello World");
        return;
    }
    brewery.OrderBeer();
    if (!greetInEnglish) {
        Console.WriteLine("Ahoj svete");
    }
}
```

Listing 2.1 Two similar conditions, but with code between their evalutaions.

We only ever draw conclusions about one condition from another if there is no non-condition code run between them. Let us look at Listing 2.1, where there is such code present. In this specific case, it might be reasonable to assume that the second condition `!greetInEnglish` will always be evaluated as true if we get to it. That is because if `greetInEnglish` was true at the start of the function, the call ends with the `return`, and if it were false, `!greetInEnglish` is always true. However, we are assuming that `brewery.OrderBeer()` cannot change `greetInEnglish`. If we exclude some very unorthodox code tricks, it might be true in the case of Listing 2.1. But what if we passed `greetInEnglish` by reference to `brewery.OrderBeer()`? Then, we cannot assume anything about evaluating the second condition, as `greetInEnglish` might or might not be changed.

Attempts to check the in-between code for, in this case, occurrences of `greetInEnglish` will not solve this problem. For example, if `greetInEnglish` were a class property, and we passed `this` to `brewery.OrderBeer()`, it could still be modified between the conditions.

There are many other partial solutions to this problem, but they are bound to fall short in specific cases. Thus, the approach chosen in the thesis is not to assume anything from one condition about another if any non-condition code is run between them.

Side Effects

We discussed the problem of deducing information about conditions from one another if there is any in-between code between them. The problem stemmed from the in-between code possibly changing values used in the conditions. Unfortunately, this is a problem even conditions themselves can cause through side effects without any in-between code. For example, the condition `myClass.GetAndSetToTrueProp || myClass.GetAndSetToTrueProp` will always be true if the property does as advertised by its name. While from a logic point of view, we would want it to be simplifiable to `myClass.GetAndSetToTrueProp`, we cannot do that since the condition would not be a tautology anymore. Side effects are generally considered undesirable in getters. But we cannot just go hands off of with this problem, as, for example, debugging outputs are a very common side effect, and many existing standard libraries have side effects, especially ones that cost a lot of real resources.

Let us start with the debugging outputs. Let us assume an `if(myClass.GetAndSayHelloProp && myClass.GetAndSayHelloProp)`, where `myClass GetAndSayHelloProp` prints out something every time it is accessed. If we simplify it to just `if(myClass.GetAndSayHelloProp)`, we inadvertently reduced the number of hellos the code prints. However, we argue that these side-effect changes would not be a problem because the reason for them can be inferred even by a beginner programmer. We reduced the number of times we need to call the function, thus we reduced the number of printed outputs.

```
bool m(bool commsInEnglish, MyVeryCostlyAPI api)
{
    if (api.Connect()) {
        if (commsInEnglish) {
            Console.WriteLine("Hello world " + api.Username);
        }
        else {
            Console.WriteLine("Ahoj svete " + api.Username);
        }
    }
}
```

Listing 2.2 Costly function call in a condition.

```
bool m(bool commsInEnglish, MyVeryCostlyAPI api)
{
    if (api.Connect() && commsInEnglish) {
        Console.WriteLine("Hello world " + api.Username);
    }
    else if (api.Connect()) {
        Console.WriteLine("Ahoj svete " + api.Username);
    }
}
```

Listing 2.3 Listing 2.2 refactored at the cost of duplicating the costly function call, which might even break the code.

The general side effect problem is much more difficult to choose an approach for. Consider Listing 2.2 and Listing 2.3. If, for example, `MyVeryCostlyAPI.Connect()` does not expect to be called if already connected, Listing 2.3 is not a correct reformulation of Listing 2.2. Even if it can handle being called if already connected, it might still be a costly operation, which we do not want to duplicate unnecessarily.

Code with side effects is frequent enough that we must consider it in our analysis. We describe specific solutions to specific problems stemming from side effects later in this thesis. But generally, we only consider function calls and assignments side effect prone, and not, for example, properties. Whenever we risk duplicating or removing expressions with function calls, we are cautious.

Exceptions

Exceptions behave similarly to side effects in our problematics, hence they cause similar problems. We generally assume exceptions are not part of the usual code execution, and thus, we do not specifically consider them anywhere in our analysis outside of the already mentioned side effect considerations. One can certainly write code where exceptions are often thrown in conditions, but it is not a good practice. While we focus on beginners, who certainly do not always follow best practices (as is the point of our thesis), we argue that our proposed refactors are still worthwhile even with these limitations.

Operator Overloading

There is no way in C# to overload `bool` operators. However, the binary operators `==`, `!=`, `<`, `<=`, `>`, and `>=` operators can be overloaded by users on their classes as they please. They could, for example, break the implication that `a == b` implies `!(a != b)`. We argue that breaking that implication is generally difficult to justify. Furthermore, beginners are unlikely to do any operator overloading at all unless explicitly instructed to. Once they do, they are bound to pay attention to what happens to these operators. The reason we are mentioning operator overloading is that simplifying `!(a != b)` to `a == b` is much easier to digest, and we will not give it up just because one can technically make the operators asymmetrical.

2.2 Specific Code Complexity Metrics

We aim to work with the user and introduce changes to their code that follow best practices and make the code "better". Since most established metrics (described in Section 1.2) are very general, our refactors are bound to influence their outputs. However, with our main focus being `if` statements, there are two main smaller scope metrics that are almost always influenced when refactoring `if` statements: Usage of nesting and condition complexity.

Nesting is a necessary tool in programming. However, overusing it can lead to poorly understandable code. It increases both the length of the code in lines and the length of lines due to indenting. Moreover, it also increases the number of levels on which the programmer has to think about a problem; if there is a variable declared two nestings in, but three more nestings follow, it might get difficult to mentally keep track of where that variable does what. There are ways to reduce nesting:

- It is always possible to extract some of the nested code into a separate function. This reduces the nesting complexity, but it can result in functions with many parameters. It also puts related code further apart, which might cause trouble, especially for beginner programmers.

- Another possibility is to somehow reduce nesting by conjoining multiple constructs that cause it together. For example, two `for` statements whose purpose is to iterate over a matrix might be flattened. The cost here is that the code is less straightforward, as most programmers would choose the two `for` statements solution. In our case, where we are focusing on `if` statements, there might be cases where we are able to transform nested `if` statements into one or multiple on the same nesting level.

- And finally, we might be able to refactor the code to include less nesting due to the specifics of the problem. We think of a "better" solution. It can also help with the application of previously mentioned ways of reducing nesting.

All of these options influence the length of the code, which itself is important. They might also influence the other mentioned metrics; for example, extracting the body of an `if` statement influences the Halstead metrics because we have to introduce a new function, which has to have a name. In Listing 0.1, we found out that an `if` statement is unreachable, which influences the cyclomatic complexity. Even though it is unreachable while the program is running, in the static code, which McCabe focuses on, the `if` looks just as good as any other. We influence many other already used or imaginable metrics, but we argue that just the code length is reason enough to try to reduce nesting, as it plays a large part in comprehending code [5].

The other metric we focus on, mostly unique to `if` statements, is the complexity of conditions. Complicated boolean calculations can appear elsewhere but often naturally occur in conditions. Let us look at some constructs or their combinations that complicate conditions:

- Parentheses can both make a condition easier or harder to understand. A pair of parentheses that encloses another pair of parentheses only increases the complexity. Not even a beginner would directly write a condition like that but might end up with one after a few refactors. On the other hand, the pair of parentheses in `(x == 1 && y == 2) || z == 3` might be clearer to both a beginner and an experienced programmer, even though it is technically unnecessary.

- Similarly, negating a long part of a condition might be useful for understanding it in some cases. In others, we might want to only negate the smallest parts of the condition possible.

- Combining many nested `&&` and `||`, possibly with many duplicate subexpressions, is confusing even for experienced programmers, especially with the added complexity that the conditions behave a bit differently to logic operators due to their lazy nature.

- Textual length can also make conditions harder to understand. This can often stem from long member accesses but is also partly influenced by the previously mentioned points: an unnecessary pair of parentheses also makes the condition longer.

There are many other imaginable metrics relevant to `if` statements, but we will focus on the two mentioned: nesting and condition complexity. We might sometimes be able to improve one at no cost, but often, improving one comes at the cost of the other.

2.3 Helping Beginners Improve Their Code

We have outlined some constraints and reasoned why convoluted `if` statements might make code hard to understand. Now, we will discuss our approach to unraveling these convoluted `if` statements.

2.3.1 Example Use Cases

Let us start by presenting a few code examples we aim for our tool to be applicable to and show how. First we will improve upon one example sequentially from different angles. Then, we will show a number of examples that can be improved upon from the same point of view.

Intro Example Detailed Explanation

```
void m(int pipeCapacity, bool inUse,
    string maintenanceInstructions)
{
    if (pipeCapacity == 0)
    {
    /*replace it, it is too broken*/
    }
    else
    {
        if (inUse && maintenanceInstructions != "decide")
        {
        /*turn off water and do what instructed to*/
        }
        else if (inUse)
        {
            if (maintenanceInstructions == "decide"
                || pipeCapacity == 0)
            {
            /*turn off water, replace it just to be sure,
            or because nothing flows through*/
            }
            else if (pipeCapacity == 42)
            {
            /*investigate the magical pipe*/
            }
        }
    }
}
```

Listing 2.4 Introductory example (Listing 0.1 duplicated).

We will start with a larger example, where we improve the code for multiple reasons. That example is the one from the introduction. Let us introduce the story a bit. Imagine we are a learning repairman with questionable judgment who is called to repair a leaking pipe. We know how much it lets through and whether it is currently being used, and we got instructions from our boss on how to fix the pipe. But if the pipe is letting nothing through, it is too broken to be fixed, so we must replace it. If it lets at least something through, is it being used? If not, then we do not need to do anything. After all, no leaks are visible, so nothing is wrong! If it is in use and the boss did not tell us to decide how to fix it ourselves, we have to turn the water off and do what they told us. If they let us decide, we might as well turn off the water and replace the pipe, as that is easier than thinking about how to fix it. Wait, what if the pipe is letting nothing through? In that case, we also want to turn off the water and replace the pipe, so let us combine those thought processes. And finally, what if the pipe has a capacity of 42? We need to investigate it then.

Let us return to reality. The previous paragraph has several questionable lines of thought. Still, we argue that they are plausible, and Listing 0.1 is a piece of code representing those thoughts, not too distant from what a beginner might actually write. There are multiple mistakes to fix in the code and a number of alternative ways to express the same thing. The first is how the variable `inUse` is checked. We could only check it once, lift that check into the encompassing `else`, making it `else if(inUse)`, as shown in Listing 2.5.

```
void m(int pipeCapacity, bool inUse,
       string maintenanceInstructions) {
    if (pipeCapacity == 0) {
    /*replace it, it is too broken*/
    }
    else if(inUse) {
        if (maintenanceInstructions != "decide") {
        /*turn off water and do what instructed to*/
        }
        else if (maintenanceInstructions == "decide"
                 || pipeCapacity == 0) {
        /*turn off water, replace it just to be sure,
        or because nothing flows through*/
        }
        else if (pipeCapacity == 42) {
        /*investigate the magical pipe*/
        }
    }
}
```

Listing 2.5 Introductory example refactor part 1.

Now let us discuss the condition `maintenanceInstructions == "decide" || pipeCapacity == 0` in Listing 2.5. The right operand `pipeCapacity == 0` might seem reasonable. If the pipe has no capacity, we want to swap it out. However, exactly this is already handled by `if (pipeCapacity == 0)`, meaning it cannot contribute to fulfilling the `||`. Perhaps the beginner added this case here "just to be sure," or we arrived at this through multiple rounds of debugging. But no matter how this code came to be, we can remove the `|| pipeCapacity == 0` part, but the remaining left operand could still fulfill the condition.

```
void m(int pipeCapacity, bool inUse,
       string maintenanceInstructions) {
    if (pipeCapacity == 0) {
    /*replace it, it is too broken*/
    }
    else if(inUse) {
        if (maintenanceInstructions != "decide") {
        /*turn off water and do what instructed to*/
        }
        else if (maintenanceInstructions == "decide") {
        /*turn off water, replace it just to be sure,
        or because nothing flows through*/
        }
        else if (pipeCapacity == 42) {
        /*investigate the magical pipe*/
        }
    }
}
```

Listing 2.6 Introductory example refactor part 2.

But looking at Listing 2.6, where we removed the unnecessary part of the condition, this idea obviously does not hold. First, we inquire `maintenanceInstructions != "decide"`, then `maintenanceInstructions == "decide"`. This is overly complicated. If the first condition is true, the second one never gets evaluated. If the first one is false, the second one is true. This means that we always branch into `else if (maintenanceInstructions == "decide")` if we get there. It can thus be turned into a simple `else` with no condition. But what about `else if (pipeCapacity == 42)`? We cannot just keep it there. It cannot follow after `else`. This might be exposing a problem in the initial code composition, maybe this `else if` should have been a nesting higher, or in a completely different place. But if the code really was structured as intended, we can delete `else if (pipeCapacity == 42)`.

```
void m(int pipeCapacity, bool inUse,
       string maintenanceInstructions) {
    if (pipeCapacity == 0) {
    /*replace it, it is too broken*/
    }
    else if(inUse) {
        if (maintenanceInstructions != "decide") {
        /*turn off water and do what instructed to*/
        }
        else {
        /*turn off water, replace it just to be sure,
        or because nothing flows through*/
        }
    }
}
```

Listing 2.7 Introductory example refactor part 3.

The result is shown in Listing 2.7. It is important to note that the simplification of `else if (maintenanceInstructions == "decide" || pipeCapacity == 0)` could also start with `maintenanceInstructions == "decide"`. We have shown it this way because then the simplification in Listing 2.6 seemed very easy. An experienced programmer could argue it is almost as easy to simplify `maintenanceInstructions == "decide"` in Listing 2.5, but we argue that it might not be for many beginners. Combining these two mistakes makes getting rid of them much harder.

The final result with less nesting shown in Listing 0.2 can be achieved by flattening the `else if(inUse)` with its embedded `if` statements, at the cost of duplicating the usage of `inUse`.

```
void m(int pipeCapacity, bool inUse,
        string maintenanceInstructions)
{
    if (pipeCapacity == 0)
    {
    /*replace it, it's too broken*/
    }
    else if (inUse && maintenanceInstructions != "decide")
    {
    /*turn off water and do what instructed to*/
    }
    else if (inUse)
    {
    /*turn off water, replace it just to be sure,
    or because nothing flows through*/
    }
}
```

Listing 2.8 Listing 2.4 refactored (Listing 0.2 duplicated).

Flattening Nested If Statements

Let us now look at multiple examples of a similar problem in different codes. Our motivation is that we want to flatten nested `if` statements.

```
void m(bool x, bool y) {
    if (x) {
        if (y) {
        /*Body1*/
        }
    }
}
```

Listing 2.9 Flattening `if` statements basic example.

```
void m(bool x, bool y) {
    if (x && y) {
    /*Body1*/
    }
}
```

Listing 2.10 Listing 2.9 refactored.

Listing 2.9 shows code most programmers would agree is much better if written as shown in Listing 2.10. `if` statement that only contains another `if` statement can always be simplified by merging them into one and combining their conditions.

```
void m(bool x, bool y) {
    if (x) {
        if (y) {
        /*Body1*/
        }
        else {
        /*Body2*/
        }
    }
}
```

Listing 2.11 Flattening `if` statements example with nested else.

```
void m(bool x, bool y) {
    if (x && y) {
    /*Body1*/
    }
    else if (x) {
    /*Body2*/
    }
}
```

Listing 2.12 Listing 2.11 refactored.

Listing 2.11 differs from Listing 2.9 by the inner `if` having an `else`. It is still possible to apply our idea as we did in Listing 2.10, but the cost is that the outer condition now has to be included with the `else` as well, turning it into `else if (x)`, shown in Listing 2.12.

```
void m(bool x, bool y, bool z)
   {
    if (x) {
        if (y) {
        /*Body1*/
        }
        else if (z) {
        /*Body2*/
        }
    }
}
```

Listing 2.13 Flattening `if` statements example with two nested `if` statements.

```
void m(bool x, bool y, bool z)
   {
    if (x && y) {
    /*Body1*/
    }
    else if (x && z) {
    /*Body2*/
    }
}
```

Listing 2.14 Listing 2.13 refactored.

Listing 2.13 is similar to Listing 2.11, but with the `else` already being an `else if`. In this case, refactoring them as shown in Listing 2.14 is correct, but what if the conditions were not as simple?

```
void m(MyClass x, MyClass y,
    MyClass z) {
     if (x.m()) {
          if (y.m()) {
          /*Body1*/
          }
          else if (z.m()) {
          /*Body2*/
          }
     }
}
```

Listing 2.15 Flattening `if` statements example with potential side effects.

```
void m(MyClass x, MyClass y,
    MyClass z) {
     if (x.m() && y.m()) {
     /*Body1*/
     }
     else if (x.m() && z.m()) {
     /*Body2*/
     }
}
```

Listing 2.16 Listing 2.15 sometimes correctly refactored, depending on side effects.

Listing 2.15 differs from Listing 2.13 by having classes instead of booleans as parameters and the conditions accessing a function instead of directly taking the boolean value. The refactoring shown in Listing 2.16 might be correct in some cases, but not always. As discussed in Section 2.1.2, this refactor is functionally equivalent only if `x.m()` has no side effects. Note the same problem would appear if we swapped out booleans for classes in Listing 2.11, but the basic idea of flattening `if` statements shown in Listing 2.9 is correct even with side effects, due to us not having to duplicate the potential side effect inducing condition.

So far, we have shown examples of refactoring that is always correct in Listing 2.10, two that can be correct in Listing 2.12 and Listing 2.14, but we must watch out for side effects as shown in Listing 2.16. Let us now look at a negative example where "flattening `if` statements" cannot be reasonably done. Our first trivial negative example could be one where no nested `if` is present. We certainly cannot "flattening `if` statements" in that case, as there is nothing to flatten.

```
void m(bool x, bool y, bool z)
    {
    if (x) {
        /*Some code*/
        if (y) {
        /*Body1*/
        }
        else if (z) {
        /*Body2*/
        }
    }
}
```

Listing 2.17 Flattening `if` statements example with two nested `if` statements and additional code.

```
void m(bool x, bool y, bool z)
    {
    if (x) {
    /*Some code*/
    }
    if (x && y) {
    /*Body1*/
    }
    else if (x && z) {
    /*Body2*/
    }
}
```

Listing 2.18 Listing 2.17 almost always incorrectly refactored.

Now for a more interesting negative example. Listing 2.17 differs from Listing 2.13 by having some code inside of the outer `if`. We discussed in-between code being a problem in Section 2.1.2 in relation to making a conclusion about one condition from another. However, while the problem here stems from in-between code between conditions, the reason is different. The problem here is where to put the in-between code if we want to do some `if` flattening. Listing 2.18 shows one option, but it is almost certainly incorrect to refactor the code like that. That is because if `/*Some code*/` changes the values of the boolean, the code will behave differently than in Listing 2.17. Finding out if such changes will occur would require above the scope of the thesis.

```
void m(bool x, bool y) {
    if (x) {
        if (y) {
        /*Body1*/
        }
    }
    else {
    /*Body2*/
    }
}
```

Listing 2.19 Flattening `if` statements example with outer else.

```
void m(bool x, bool y) {
    if (x && y) {
    /*Body1*/
    }
    else {
    /*Body2*/
    }
}
```

Listing 2.20 Listing 2.19 almost always incorrectly refactored.

Listing 2.19 differs from Listing 2.9 by the outer `if` having an `else`. We cannot do what we did in Listing 2.10 and keep the `else` as is because in the case that `x` is true, but `y` is false, no code outside of the condition evaluation is supposed to run, whereas if we used the proposed refactor in Listing 2.20 the else body would be run.

2.3.2 Primary User Interaction

As discussed in Section 2.2, we mainly focus on nesting and condition complexity. We could provide the user with a button to "reduce nesting" that uses many approaches in our arsenal, including what was described in Section 2.3.1, to reduce the nesting across the whole solution, file, or function. There are two main problems with this approach.

Imagine an example like Listing 2.13, but with ten `else if` statements instead of one. Flattening the `if` statements there would be very unwise. Thus, we would need some setting that relativizes the metrics of nesting and condition complexity to each other. However, such a setting would be too abstract and subjective to both the author and the user.

The second problem is that such a button would teach very little. The user could go through the code before and after using the button, but the code might become almost unrecognizable to a beginner programmer. Encouraging users to write good code by making them understand our proposed refactors is an important part of the thesis.

Thus, the thesis does not introduce any metrics to the user at all. Instead, we, at one point in time, focus on one concrete code refactor with a singular idea behind it. That refactor could be done manually in a few minutes or even seconds of work by an experienced programmer. Still, a beginner programmer might not even think of it. These refactors might change both nesting and condition complexity. It is then up to the user to consider if these refactors are worth it in their specific case.

2.3.3 Parts of a Specific Case Focused Refactor

Let us investigate the approach of focusing on specific small-sized refactors we arrived at in Section 2.3.2 using a concrete example. We will use the problem of flattening `if` statements described in Section 2.3.1. But we quickly arrive at a problem: What to do when presented with code such as in Listing 2.21 below?

Size of Refactors

```
void m(bool x, bool y, bool z) {
    if (x) {
        if (y) {
            if (z) {
            /*Body1*/
            }
        }
    }
}
```

Listing 2.21 Flattening example with double nesting.

We have two possible approaches here. We could propose immediately flattening the three `if` statements into a single `if` statement with the condition `x && y && z`. We will call this approach recursive.

Or we could give the choice of flattening the `if (x)` and `if (y)`, or `if (y)` and `if (z)`, and then, if the user decides, they could further flatten the remaining two `if` statements to get the same result as the recursive approach, that is an `if` with the condition `x && y && z`. We will call this approach singular.

We argue that the singular approach is better. The recursive approach is straightforward in the case of Listing 2.21, but what if `if (y)` had an `else` like shown in Listing 2.22 below? `if (x)` and `if (y)` can be flattened, as shown in Listing 2.23, but further flattening them with `if (z)` is impossible for the same reason it was not possible in Listing 2.19.

```
void m(bool x, bool y, bool z)
   {
    if (x) {
        if (y) {
            if (z) {
            /*Body1*/
            }
        }
        else {
        /*Body2*/
        }
    }
}
```

Listing 2.22 Flattening example with double nesting and middle else.

```
void m(bool x, bool y, bool z)
   {
    if (x && y) {
        if (z) {
        /*Body1*/
        }
    }
    else if (x) {
    /*Body2*/
    }
}
```

Listing 2.23 Listing 2.22 flattened as much as possible.

The recursive approach might be confusing to a beginner programmer by sometimes recursing deep or sometimes stopping much earlier. We could, of course, explain this case to the user just like we do in this thesis. However, we argue that the bite-sized refactors and their explanations proposed in the singular approach are friendlier to beginners. It is an extension of the argument and conclusion in Section 2.3.2. Even the small singularly minded refactors should be split into the smallest parts possible.

Attributing the Refactor

The possibility of general refactors is an attribute of the code as a whole. However, our refactors are small in size. And if we were able to pin any refactors we make onto a specific place in the code, it would be very useful for UI design. Making the refactors accessible from that place in the code would make user interactions very natural. And it turns out we can achieve exactly that. We attribute all refactors to the textually earliest changed language construct, usually an `if`.

Combined with the singular approach from Section 2.3.3, we have two possible refactors to offer in our running example of flattening `if` statements in Listing 2.21: flattening `if (x)` and `if (y)`, which is attributed to `if (x)`, and flattening `if (y)` and `if (z)` which is attributed to `if (y)`. The singular approach makes it so that `if (x)` only flattens with `if (y)`, and not also with `if (z)`.

Can We Do It

We now have a concrete refactor idea we want to apply to a concrete thing; in our running example, we want to flatten an `if` with its nested `if`, and we want to apply that to a concrete `if`. But first, we have to decide whether that refactor even makes sense on that concrete thing. For example, we trivially cannot flatten `if (z)` with its nested `if` in Listing 2.21 because it has no nested `if`. There are, of course, more possibilities that make flattening impossible, some described in Section 2.3.1, but it is important to note that even these trivial ones exist and are the ones that disqualify refactors most often.

Should We Do It

If a refactor is impossible, then we obviously cannot carry it out. But if it is possible, that still does not mean it is a good idea to do so. As previously discussed in Section 2.3.2, we want the users to gauge the refactor themselves and decide if it makes sense in their particular case. But we still also want to offer guidance on whether it is generally a good idea. For example, flattening `if` statements in Listing 2.9 is almost certainly a good idea. The only instance where it could not be is if the user plans to add code into the outer `if` but outside the inner `if`. But we have no way of knowing that, so we assume the user is not planning on changing the code's functionality, and in that case, we want to recommend flattening them. Flattening the `if` statements in Listing 2.13 is a mixed case. It sometimes makes sense, sometimes not. It also depends on the future extension of the program, which we cannot predict. So it is best left up to the user to decide whether it is worth it; we are indifferent. Finally, if the nested `if` in Listing 2.13 did not have one `else if`, but let us say ten, it would almost certainly be a bad idea to flatten the `if` statements there. However, it would still be possible to use the tools we must already have in place for the other cases, so we might as well offer that refactor to the user but heavily discourage it.

Explaining Refactors to Beginners

Explaining refactors is a complex issue. All of the refactors we offer should function the same as the original code. This is a stable fact the user can rely on while judging our refactors. Still, providing further hints on why the functionality did not change is necessary if we want to guide the user in their learning.

```
void m(bool x, bool y) {
    if (x) {
        if (y) {
        /*Body1*/
        }
    }

    if (x) {
        /*Body2*/
    }
}
```

Listing 2.24 Pairing negative example.

A useful tool we can utilize to provide these hints is somehow pairing related constructs. We can do that pairing either on just the original code or on the original code and the refactored code at the same time. In Listing 2.11, the variable `x` appears once in the code before refactoring and twice after it in Listing 2.12. Since these appearances are directly related, it makes sense to emphasize that fact by somehow pairing them together, making that fact apparent to the user. On the other hand, in Listing 2.24, if we are analyzing flattening `if (x)` and `if (y)`, there is no reason to pair the first `if (x)` with the second `if (x)`. It has no bearing on the refactor.

There might also be cases where parts of the refactored code are completely new instead of being pairable with other code. Inversely, as a part of our refactoring, we might find out that some code can be simply deleted. We ought to somehow point these additions or deletions to the user, as not doing so would be confusing.

Combining Certain Code Changes

Sometimes, it might make sense to combine certain refactors together. Let us extend our running example of `if` flattening for the following case shown in Listing 2.25 and Listing 2.26.

```
void m(bool x, bool y, bool z)
    {
     if (x) {
          if (y || z) {
          /*Body1*/
          }
     }
}
```

Listing 2.25 Flattening with an `||` example.

```
void m(bool x, bool y, bool z)
    {
     if (x && (y || z)) {
     /*Body1*/
     }
}
```

Listing 2.26 Listing 2.25 refactored.

The parentheses in Listing 2.26 are necessary because of operator precedence. But if we swapped the `||` in `y || z` for an `&&`, the parentheses would not be necessary. This is something we obviously have to get right in our refactors. We do not want to produce code that is functionally different. But it is also something we can teach the user about.

At first, we should assume the user is a complete beginner, and we should be as consistent as possible. In that case, we will always put the parentheses on both of the operands, making the actual `if` in the refactored code `if ((x) && (y || z))`. The parentheses in `(x)` are obviously unnecessary, but, again, if the expression in parentheses was more complicated, they might be necessary.

Once the user has understood this, we might offer them an option to only include the parentheses when necessary. This breaks the principle of small-sized refactors, but we will only apply this to relatively simple expression simplification, like with parentheses.

2.4 UI

In Section 2.3.3, we discussed how to make the educational part of this thesis accessible to the users through pairing. Still, there are a number of decisions to be made on how to make other functionalities, both primary and supporting, accessible to the user.

2.4.1 Location of the UI Best for Beginners

The first question is if we even require a dedicated user interface. Experienced programmers might do well with a command line tool, but our primary target audience is beginners. A well-designed user interface will be much more approachable to them.

The next question is where we should put the user interface. An external tool that serves only to recommend, explain, and perform our recommended refactors but does not let the user edit the code themselves seems like an unnatural solution. We are changing the code. Why should the user not be able to? If the user should also be able to change the code close to our user interface, the most natural solution is to put it somewhere in the existing development environment. This carries the benefit of it being a mostly familiar environment, except for our own refactors.

Still, there are many ways to offer our refactors to the user. We could follow the existing CodeFixes described in Section 1.3.5, and just extend those. This has many disadvantages from the point of view of this thesis:

- The user might mix up our custom CodeFixes with the existing CodeFixes, which is a problem. Mixing up our and other options might lead to not meeting user expectations. A big part of our work is explaining it to the user, but not all CodeFixes are like that. Moreover, our CodeFixes might behave differently from the existing ones due to differences in the assumptions we make. Ours are mostly justified by focusing on beginners, but not all CodeFixes can do that.

- The user also has to actively look for CodeFixes. They are only available in the immediate textual vicinity they actually affect. If we want the user to consider a specific refactor, we must hope they discover it on their own.

- Continuing with the previous point, if we expanded the availability of our CodeFixes, too many might be available. If we offered parentheses simplification of the condition of an `if` statement in all of the nested statements of the condition, we could easily end up with dozens or hundreds of options. And most of our changes aim to be very local.

- There are also technical limitations to this approach, as our CodeFixes have to align with the already existing CodeFixes implementation. For example, they are not supposed to be IDE-specific.

The other solution is to have a fully-fledged user interface that is tightly interconnected with the code that the user currently has open. This allows us more freedom to express recommendations to the user. For example, if we see a glaringly obvious and easy change we could implement, we can make it very apparent to the user, and the only thing they need to do is to look at our user interface. We also have more freedom in general and can explore the code with the user in different ways. For example, we can extract just the parts about which we can improve something from the existing code.

2.4.2 Main Part of UI

Now, the question is what we should actually show in the user interface. The possibility to attribute all our changes to the textually earliest changed language construct that is discussed in Section 2.3.3 is important here. If all of our changes can be attributed to a small subset of language constructs, `if` statements and `else` statements, then we might use them in the UI as a kind of summary of the code. We can, for example, omit a variable declaration from the UI, as we cannot do anything about it specifically while taking it into consideration in our analysis under the hood.

Moreover, many of our proposed changes interact with nesting, which is a natural outcome as nesting is one of the two main complexities we focus on, as discussed in Section 2.2. Thus, it might make sense to use that nesting in the user interface as well.

Together, this results in a user interface structured like the actual code, with nested `if` statements nested under their parents but only with the parts of the code that are important to us. This approach both serves as a concise summary of the code constructs we focus on and exposes our tools to the user.

Chapter 3

Solution

In this chapter, we will build upon the conclusions we arrived at in Chapter 2. We will present the overarching solutions to most of our goals in the form of a *refactorer*. Then, we will present concrete refactorers, which constitute the main "content" the users can utilize while interacting with our tool.

3.1 General Limitations and Focus

First, we must set a few general limitations to our analysis that shape the API and functionality of all refactorers.

3.1.1 When Semantic Analysis Is Necessary

We can solve plenty of problems while focusing on just the syntactical analysis. However, we will only begin our analysis if the code is compilable, which requires both syntactical and semantical correctness. For example, even if our analysis stays on the syntactical level, it is still useful to assume that if the code contains `if (x)`, then `x` is a boolean. That is, however, something that can only be uncovered by semantic analysis. A much simpler reason is that we want to offer functionally equivalent versions of the code, and that makes little sense when the code is not in a compilable state, as the functionality is not there.

Another problem for which we have to delve into semantics is the problem of variable declarations in a block. Consider the example in Listing 3.1 below. The condition of the inner `if` will always be true due to the immediately previously evaluated condition of the outer `if`. This means we can transform the code as shown in Listing 3.2. But we must keep the block around `int x = 0`, as removing it would make the scopes of the two variables `x` overlap. However, if we replaced one of the declarations with something else, even just renamed the variable, the block around `int x = 0;` or its replacement would be unnecessary.

```
void m(bool a, bool b) {
    if (a) {
        if (a) {
            int x = 0;
        }
        if (b)
        {
            string x = "Hello
    world";
        }
    }
}
```

Listing 3.1 Example where a block is for variable scope limitation.

```
void m(bool a) {
    if (a) {
        {
            int x = 0;
        }
        if (b)
        {
            string x = "Hello
    world";
        }
    }
}
```

Listing 3.2 Listing 3.1 refactored, but the block must stay.

We do not want to always defensively keep the sometimes unnecessary block because unembedded blocks are an unusual sight in programming. Blocks are most often used inside statements that allow only one embedded statement, for example, the `if` statements we focus on in this thesis. But we would not want to produce uncompilable code by always removing the blocks. The chosen solution in this thesis is to do a bit of semantical analysis, but just over the file where the block/variable scope problem arose. First, we try to remove the block and check if any semantical errors arising from overlapping variable scopes are present. There could not have been any before, as we only begin our analysis once the code is without semantical errors. If there are any such errors, we caused them by our proposed change. Thus, we need to keep the block, as it is important to limit the variable scope.

3.1.2 Intraprocedural Analysis

In this thesis, we focus on valuable but small changes to code concerning mainly `if` statements. Small in the sense that the changes could be somewhat quickly done by hand. All of our proposed refactors are limited to a single function. Some refactors could be made quickly, even if they manipulate multiple functions or create and delete them, but those fall outside the scope of this thesis. It could be argued that, for example, extracting the body of an `if` statement into a separate function is a problem that we should solve, as it crops up quite often when writing long `if` statements. Such extracting also intends to reduce the function length and nesting, which are metrics we focus on. However, that would require extensive semantical analysis and interaction with many more language constructs, which falls outside of the scope of this thesis.

3.1.3 Boolean Binary Operators

Lazily evaluated `&&` and `||` constitute most conditions by both experienced and beginner programmers. The not lazily evaluated boolean binary operators `&`, `|`, and `^` are rarely seen in conditions. Thus, we focus our analysis on `&&` and `||`. This does not mean we do no analysis when the non-lazy operators are present, but we do not recurse under them.

The benefit of focusing on `&&` and `||` is that we can rely on the set order of evaluation due to their laziness. None of our refactors should reorder operands of `&&` and `||`, as without complicated analysis, we must assume the ordering is important. Looking at the first item of a list and then checking if the list contains any items is unwise.

3.2 Refactorers

In Section 2.3.2, we arrived at the conclusion that we want to focus on refactoring instances of specific mistakes in specific places in the code. The specific mistake is directly represented in the implementation by refactorers. But how is the specific place in the code represented? We utilize the existing syntax nodes from Roslyn (see Section 1.3), usually one representing an `if` statement. Note that even if we only pass a concrete syntax node around, we still have access to the whole syntax tree, as discussed in Section 1.3. This means we have the context of the surrounding code.

For a specific syntax node, each refactorer can decide if the refactoring it represents is possible there. It can also decide if we should recommend it. The refactorer can also apply the refactoring. Finally, it can also facilitate explaining the refactoring through highlights on either just the pre-refactored code or both the pre-refactored and post-refactored code.

3.2.1 Applicability

As discussed in Section 2.3.3, the first thing a refactorer has to be able to answer is if the refactoring is even possible. If it is impossible, then there is nothing more to be done using this refactorer. It could be that the refactoring is impossible because it is unnecessary, as the code might already be “perfect” from the point of view of this refactorer. Or it could be due to some implementation or constraint limitations. Note that we do not attempt to show how the user should change the code to make some refactorer applicable. If the changes required did not change the functionality, we could simply extend the implementation for that case. If the changes required did change the functionality, that is something we want to completely avoid.

After we find out the refactoring is possible, we immediately want to determine to what extent the refactoring is a good idea. This is because guiding the user plays an important part in this thesis. And there is no instance where considering a recommendation without first finding out the possibility of a refactor makes sense. Thus, our solution is to merge the possibility of applying a refactor with the recommendation. Combining these two questions, we get four possible answers:

1. The refactoring is impossible, either by the nature of the existing code or due to some limitations of the refactorer.
2. We discourage applying the refactoring because it produces generally worse code, but it is possible.
3. The resulting code and the original code are about the same complexity. Thus, it is mainly left up to the user to decide.
4. We encourage applying the refactoring because it produces generally better code, but we do not force the user to apply it. It might not make sense in their case.

Inverse Refactors

We outlined that we sometimes have a refactoring available that does not necessarily improve the code, it just makes the code different. It is especially important for refactorers that allow such cases to also have an inverse refactorer that enables the user to switch between the two similarly complex code variants. We would not want to lock the user into one of the options after presenting them as similarly good.

3.2.2 Refactor

If the user decides to do so, we have to be able to carry out the refactoring. But what does that entail? Replacing the actual code the user sees in their IDE can be done in multiple ways. The cleaner approach is not to immediately use the code in the refactorers but to return some intermediate data structure, which we use to replace code when and where needed. The obvious choice for that structure is to use the Roslyn syntax nodes again, which we use as the input for the refactorer and in the actual analysis. It might also allow direct layering of analysis if we use the Roslyn syntax nodes everywhere. But which syntax node or nodes do we return?

```
void m(bool x, bool y) {
    if (x) {
        if (y) {
        /*Body1*/
        }
    }
}
```

Listing 3.3 Flattening basic example.

```
void m(bool x, bool y) {
    if (x && y) {
    /*Body1*/
    }
}
```

Listing 3.4 Listing 3.3 refactored.

Let us look at a basic example of combining `if` statements in Listing 3.3 and Listing 3.4. We pass the node representing `if (x)` to the refactorer. It might seem sufficient that the refactorer returns the new merged `if (x && y)` statement with which we should replace the old one. In this case, this approach is good enough, but elsewhere it will fail.

```
void m(bool x) {
    if (x) {
    /*Lots of code*/
    }
}
```

Listing 3.5 Invert basic example.

```
void m(bool x) {
    if (!x) {
        return;
    }
    /*Same Lots of code*/
}
```

Listing 3.6 Listing 3.5 refactored.

`if (x)` in Listing 3.5 can be "inverted", which results in Listing 3.6. In this case, there is no single node we want to replace the old `if (x)` with. We do want to replace the old `if (x)` with the inverted `if` statement that just has `return` inside. But we also need to put the code that used to be inside `if (x)` after the new inverted `if (!x)`. So, what do we need to return from refactorers refactor to solve this? For the mentioned case of inverting `if` statements, a list of nodes to replace it would be sufficient. But there are cases where even that would be insufficient.

After the discussion in Section 3.1.2, returning a replacement for the function that the inputted syntax node was in might seem like a good idea. But this is an unnecessarily middling solution with no actual benefits. A general solution when we are sure that we are only changing one document is to return the syntactical root of the whole document. This approach would be insufficient only if we ever decided to do refactors across multiple documents, but that falls outside of the scope of this thesis and would nontrivially complicate the code everywhere.

3.2.3 Highlighting

A big part of this thesis is not only carrying out a refactoring but also explaining it. As described in Section 2.3.3, pairing related pieces of code in both just the old or the old and the new code is a good way to do so. Pairing can be shown in many ways, for example, by pointing arrows between the paired pieces of code. But said arrows might be too intrusive and take away a lot of attention. The already used way of drawing attention to specific parts of code in programming is somehow styling the text. We follow that notion and highlight parts of the code by changing the background color of those parts.

So, how does a refactorer provide this highlighting? A solution akin to the ones we arrived at earlier is to return a list of pairs consisting of a Roslyn syntax node and some type of a highlight. This, however, ends up having too little granularity. Looking at Listing 3.5, the Roslyn syntax node representing `if (x)` contains not only the `if` keyword and the `(x)` condition but also the body as well. If we want to only highlight the expression `x`, we could do so, as that is its own syntax node. But what if we wanted to highlight the whole line `if (x)`, but not the body? That is impossible if we limit ourselves to syntax nodes as the main result of refactorer highlighting. Even if we tried to piece it together from multiple nodes, that would not work, as, for example, just `if` in `if (x)` is not a syntax node but a syntax token. It might seem that the solution is to return a list of syntax nodes or tokens and their highlight types, but even then, the granularity would be an issue. Furthermore, we also need to consider trivia. The solution that evades all of the mentioned problems, and the one chosen in this thesis, is to return a list of pairs of textual spans and the accompanying highlight types. This is the most general solution, where we can basically do what we want. These spans might often directly come from syntax nodes, but do not need to.

3.3 Code Reusability Through Annotations

The applicability, refactoring, and highlighting analysis might share a lot of logic and, thus, code. For example, many of our refactorers use recursion but differ in actions that should be taken at different points in the recursion. As code duplicity is undesirable, we want to avoid it as much as we can. This is usually done by using type parameters, callbacks, or other generalizations of code. We use them to a degree, but also use a different way of generalizing, specific to the problem we solve. We use Roslyn syntax annotations.

Let us use parentheses simplification as an example. When highlighting, we want to find all pairs of parentheses that can be simplified and highlight them. When actually refactoring the code, we want to take these pairs and remove them.

This could, of course, be reasonably solved through the normal ways of code generalization. We could generalize the code using a callback, to which we would pass any unnecessary parentheses we find during recursing.

In highlighting this callback would “log” any of the pairs given to it, and then we would highlight the “logged” pairs. Refactoring would be a bit more difficult. One solution would be to utilize the callback in the middle of the recursing to remove the parentheses as we go. This, however, would require the callbacks to be able to change the code we are refactoring while recursing. The highlight callback could just “log” the unnecessary parentheses and not make any changes. This is not a clean solution from a design point of view. Another solution is to have two callbacks, one that only “logs” and one that can do changes while we are recursing. This is, in our opinion, too overly complicated for a relatively simple goal.

Another option is that we could solve the refactoring by “logging” too. First, we would “log” the unnecessary parentheses and then remove them after the “logging” is done. This removal would require more recursion due to the nature of expressions but separates the problem nicely. One part of the code finds the interesting parts, unnecessary parentheses in this case, and others do something with the list of the interesting parts, highlight or remove them in this case.

The approach from the previous paragraph is basically what we do with annotations. Instead of a callback that fills a list, we, in most cases, have one piece of code that annotates all the simplifiable pairs and returns the syntax tree with the annotations. This is the equivalent of the “logging”. Then in highlighting we go through these annotated pairs and highlight them, when refactoring we remove them, same as we described we could do after “logging”.

The advantage of annotations over the “logging” approach is that we do not need a “logging” callback that would always do almost the same thing. We simply use annotations, as they are intended for, among other things, this exact problem. The drawback is that the flow of information in the code is not as clear as a more direct solution. Any syntax tree or node can include any number of annotations. However, the code still has obvious purposes: some parts annotate, some go through the annotations, and some do something concrete with them.

3.4 Walk Over a Syntax Tree

`if` statement syntax often includes recursive structures. Even a simple expression like `A && B && C && ...`, which could be a condition of an `if` statement, turns out to be recursive, even though programmers often think about it as a list of conditions `A`, `B`, `C`... Provided we have to work with these recursive structures, there are multiple ways to approach them. Recursive functions, or their unraveling into loops, are one option. Another one, specific to our problem, is Roslyn syntax walkers and rewriters.

The main difference between recursion and the Roslyn options is who manages the recursing. If we do it ourselves, we must explicitly state all language constructs we recurse through. If we do not explicitly include something, we will not recurse into it. But managing the recursion ourselves is more verbose and duplicate. If we use the Roslyn walker or rewriter, the recursion management is done over the whole syntax tree for us, but it only stops if we explicitly say so. This fact dictates the difference between their usage. The Roslyn way is useful when we want to do something in many similar nodes, and we do not really care what is between these nodes or how the syntax tree generally looks. In all other cases, we manage the recursion ourselves, so we have complete control and only recurs when we actually want to.

3.5 Concrete Refactorers

3.5.1 Extract Start

Introductory Example

```
void m(bool a, bool b, bool c,
    bool d) {
     if (a && b) {
     /*body1*/
     }
     else if (a && c) {
     /*body2*/
     }
     else if (a && d) {
     /*body3*/
     }
}
```

Listing 3.7 Extract start basic example.

```
void m(bool a, bool b, bool c,
    bool d) {
     if (a) {
          if (b) {
          /*body1*/
          }
          else if (c) {
          /*body2*/
          }
          else if (d) {
          /*body3*/
          }
     }
}
```

Listing 3.8 Listing 3.7 refactored.

Listing 3.7 and Listing 3.8 show the basic idea behind "Extracting start" of `if` statements. This is the ideal case where we slightly increase the nesting but reduce the multiple instances of an identical expression at the start of the conditions.

Possibility

```
void m(bool a, bool b, bool c) {
    if (a && c) {
    /*body1*/
    }
    else if (b && c) {
    /*body2*/
    }
}
```

Listing 3.9 Extract start example with different starts.

The conditions of the `if` and the following `else if` statements have to share the same start in some sense to use the Extract start refactorer. We cannot use this refactorer on `if (a && c)` in Listing 3.9, because it does not share the start with `else if (b && c)`. Sharing the same start means that if the expressions in the conditions consist of sequences of `&&` operators, the leftmost operand of the leftmost `&&` in such sequence must be equivalent for all of the conditions. If the condition of an `else if` does not consist of `&&`s at the top level, it might still be possible to extract the whole condition out, and turn said `else if` into just an `else`, as shown in Listing 3.10 and Listing 3.11 below.

```
void m(bool a, bool b) {
    if (a && b) {
    /*body1*/
    }
    else if (a) {
    /*body2*/
    }
}
```

Listing 3.10 Extract start example with `else if`.

```
void m(bool a, bool b) {
    if (a) {
        if (b) {
        /*body1*/
        }
        else {
        /*body2*/
        }
    }
}
```

Listing 3.11 Listing 3.10 refactored.

But what if there was another `else if` after `else if (a)` as shown in Listing 3.12 below? The refactored code would result in Listing 3.13, which is just like Listing 3.11, except the function parameter `bool c` is still there, as potentially omitting it falls outside of the scope of this refactorer, and generally the whole thesis. `else if (a && c)` can be deleted because if the condition `a` is not met, then the condition `a && c` also cannot be met. Even a simple refactorer like this can uncover an unreachable part of the code in this way.

```
void m(bool a, bool b, bool c)
    {
     if (a && b) {
     /*body1*/
     }
     else if (a) {
     /*body2*/
     }
     else if (a && c) {
     /*body3*/
     }
}
```

Listing 3.12 Extract start example with unreachable `else if`.

```
void m(bool a, bool b, bool c)
    {
     if (a) {
          if (b) {
          /*body1*/
          }
          else {
          /*body2*/
          }
     }
}
```

Listing 3.13 Listing 3.12 refactored.

So far, all of the conditions of `if` and `else if` statements shared the common beginning `a`. But what if the condition of the last `else if` statement had nothing in common with the others, as shown in Listing 3.14 below? The proposed refactor in Listing 3.15 is incorrect, because when `a` and `d` are true, but `b` and `c` are false, the original code would have run `/*body3*/`, but the proposed refactor would result in no body being run. When a `else if` condition does not share the start with the preceding `else if` conditions, refactoring the preceding `if` statement or `else if` statements are impossible.

```
void m(bool a, bool b, bool c,
    bool d) {
     if (a && b) {
     /*body1*/
     }
     else if (a && c) {
     /*body2*/
     }
     else if (d) {
     /*body3*/
     }
}
```

Listing 3.14 Extract start example with unextractable from `else if`.

```
void m(bool a, bool b, bool c,
    bool d) {
     if (a) {
          if (b) {
          /*body1*/
          }
          else if (c) {
          /*body2*/
          }
     }
     else if (d) {
     /*body3*/
     }
}
```

Listing 3.15 Listing 3.14 incorrectly refactored.

However, it does not prevent using the refactoring on the following `else if` statements. So far, we have always considered and shown applying the refactoring on the first `if` statement. But, for example, in the first shown example in Listing 3.7, it could be applied to both `else if` statements, resulting in Listing 3.16 and Listing 3.17.

```
void m(bool a, bool b, bool c, bool d) {
    if (a && b) {
    /*body1*/
    }
    else if (a) {
        if (c) {
        /*body2*/
        }
        else if (d) {
        /*body3*/
        }
    }
}
```

Listing 3.16 Refactored Listing 3.7 on first `else if`.

```
void m(bool a, bool b, bool c, bool d) {
    if (a && b) {
    /*body1*/
    }
    else if (a && c) {
    /*body2*/
    }
    if (a) {
        else if (d) {
        /*body3*/
        }
    }
}
```

Listing 3.17 Refactored Listing 3.7 on second `else if`.

Criterion

```
void m(bool a, bool b) {
    if (a && b) {
    /*body1*/
    }
}
```

Listing 3.18 Extract start discouradged example.

```
void m(bool a, bool b) {
    if (a) {
        if (b) {
        /*body1*/
        }
    }
}
```

Listing 3.19 Listing 3.18 refactored.

Listing 3.18 and Listing 3.19 show an example usage of the refactorer that we should discourage but falls under this refactorer and thus should be possible through it. The difference between this example and the ideal example in Listing 3.7 and Listing 3.8 is the number of conditions we are extracting the start from. And exactly that constitutes our criterion. If it is one, we discourage the refactoring, for two we are indifferent, and for three or above we encourage it.

Limitations

If the start of conditions we extracted had side effects, we would affect the behavior of the code if we extracted it from more than one condition. Thus, if we are extracting from more than one condition, and the start is side effect prone (see Section 2.1.2), we consider the refactoring impossible.

3.5.2 Flatten If

Introductory Example

```
void m(bool x, bool y) {
    if (x) {
        if (y) {
        /*Body1*/
        }
    }
}
```

Listing 3.20 Listing 2.9 duplicated.

```
void m(bool x, bool y) {
    if ((x) && (y)) {
    /*Body1*/
    }
}
```

Listing 3.21 Listing 3.20 refactored but with parentheses.

We already described flattening in Section 2.3.1 before we knew we would implement it as a refactorer. The main difference between the analysis refactor idea in Listing 2.10 and the Flatten `if` refactorer refactoring in Listing 3.21 is that we always keep the parentheses around the former conditions that constitute the new condition. We discussed the reason for this in Section 2.3.3.

Possibility

The possibility was discussed in detail in Section 2.3.1.

Criterion

The criterion is the inverse of the criterion of the Extract start refactorer. If the inner construct is only an `if` statement, we encourage the refactoring; if the inner `if` statement has one `else if`, we are indifferent; and if it has two or more, we discourage the refactoring.

Limitations

Again, similarly to the Extract start refactorer, if the outer condition is side effect prone, and the inner `if` statement has at least one `else if`, the refactoring is impossible because we risk duplicating a side effect expression.

3.5.3 Invert If

Introductory Example

```
void m(bool a) {
    if (a) {
    /*Lots of code*/
    }
}
```

Listing 3.22 Invert `if` basic example.

```
void m(bool a) {
    if (!(a)) {
        return ;
    }
    /*Lots of code*/
}
```

Listing 3.23 Listing 3.22 refactored.

Listing 3.22 and Listing 3.23 sum up the idea behind the Invert `if` refactorer. This refactorer allows us to, in some cases, massively reduce the nesting of the code at the cost of possibly adding return statements. Note that the return type is `void`. The parentheses in `!(a)` are there because they would be necessary for any binary expression.

Possibility - If With a Lot of Code Inside

Adding code before the `if` statement we aim to invert does not impact the refactoring in the case of Listing 3.24 and Listing 3.25 below. We keep the code where it was and do what we did in the previous example.

```
void m(bool a) {
    Console.WriteLine(a);
    if (a) {
    /*Lots of code*/
    }
}
```

Listing 3.24 Invert `if` example with preceding code.

```
void m(bool a) {
    Console.WriteLine(a);
    if (!(a)) {
        return ;
    }
    /*Lots of code*/
}
```

Listing 3.25 Listing 3.24 refactored.

However, consider Listing 3.26 below. The two inner blocks in the example contain a declaration of `x`. This means keeping the block after the refactoring is necessary, as shown in Listing 3.27. We discussed this problem and its solution in Section 3.1.1. To sum it up, we try semantic analysis on the refactored code without the block and check for errors that signify overlapping variable scopes. If there are none, we do not need the block; if there are any, we need it.

```
void m(bool a) {
    {
        int x = 0;
    }
    if (a) {
        /*Lots of code 1*/
        string x = "hello world
    ";
        /*Lots of code 2*/
    }
}
```

Listing 3.26 Invert `if` example with a block/scope issue.

```
void m(bool a) {
    {
        int x = 0;
    }
    if (!(a)) {
        return ;
    }
    {
        /*Lots of code 1*/
         string x = "hello
    world";
        /*Lots of code 2*/
    }
}
```

Listing 3.27 Listing 3.26 refactored.

Finally, note that even if the condition of the inverted `if` statement contains a function with an `out` parameter, our refactoring is sound (semantics discussed in Section 1.1.1). An example is shown in Listing 3.28 and Listing 3.29 below.

```
void x() {
    if (int.TryParse("1", out
    var result)){
        Console.WriteLine(
    result);
    }
}
```

Listing 3.28 Invert `if` example with out parameter.

```
void x() {
    if (!(int.TryParse("1", out
     var result))) {
        return;
    }
    Console.WriteLine(result);
}
```

Listing 3.29 Listing 3.28 refactored.

Possibility - If With Only a Return Inside

As discussed in Section 3.2.1, having inverse variants for our proposed refactors is good. Let us look at the first example discussed under this refactorer Listing 3.22 and Listing 3.23, but in reverse order. If we started with Listing 3.23, transforming it into Listing 3.22 seems like a refactor that could reasonably be described as "inverting an `if`." It might not be advisable if `/*Lots of code*/` is indeed lots of code, but what if it was just one line? It might then be the cleaner version of the code. And because even the inverse refactoring is in the spirit of "inverting an `if`", the refactorer itself can play the part of its own inverse refactorer.

Possibility - Generalization

We can generalize the idea of this refactorer to encompass all of the previously described cases. Consider the example shown in Listing 3.30 below. If `a` is true `/*Lots of inner code*/` is run, if it is false `/*Lots of following code*/` is run. We can transform to Listing 3.31 while keeping the functionality.

```
void m(bool a) {
    if (a) {
        /*Lots of inner code*/
        return ;
    }
    /*Lots of following code*/
}
```

Listing 3.30 Invert `if` example with following code.

```
void m(bool a) {
    if (!(a)) {
        /*Lots of following
    code*/
        return ;
    }
    /*Lots of inner code*/
}
```

Listing 3.31 Listing 3.30 refactored.

Once again, any preceding code before `if (a)` in Listing 3.30 would only affect the refactoring about the potential block removal. Other than that, it would not change how the refactoring works; we always keep the preceding code, and it sometimes makes us unable to remove the block around `/*Lots of inner code*/`.

If `/*Lots of following code*/` is empty, we do not actually require `return` immediately after the `/*Lots of inner code*/` in Listing 3.30. This special case is how the generalized approach handles some previously shown examples, which contain no following code after the inverted `if`.

A final improvement is to remove unnecessary `return` statements after the refactor. There is no need to have a `return` statement at the absolute end of a function, and there is no need to have it at the end of the `if` we just refactored if no code follows after the `if`.

Possibility - Prohibitive Summary

We discussed the many applications of this refactorer, but what prohibits it from being used? Firstly the `return ;` at end of `if` in Listing 3.30 is important if `/*Lots of following code*/` is not empty. If it was not there both `/*Lots of inner code*/` and `/*Lots of following code*/` would be run if `a` was true, and we cannot reasonably refactor the code from the point of view of "inverting an `if`." Furthermore, the `if` needs to be directly in a function's body. It being inside any other block prohibits the use of this refactorer.

Criterion

We show the criterion for recommendation on Listing 3.30. The criterion is simply the difference between the number of lines of `/*Lots of inner code*/` and `/*Lots of following code*/`. If the difference is a large positive number, swapping them means reducing the nesting of a lot of lines, which we recommend. Contrary to that, if the difference is a large negative number, swapping them would greatly increase the amount of nested code, so we discourage it. Finally, there is a middle ground, where the size of the `/*Lots of inner code*/` and `/*Lots of following code*/` is about the same, where we are indifferent.

Complex Example

Listing 3.32, Listing 3.33 and Listing 3.34 below show how two usages of this refactorer can unravel the code to reduce the nesting of all of the code in the function to its absolute minimum.

```
void ConditionalHello(bool polite, bool multilingual) {
    if (polite) {
        Console.WriteLine("Hello world");
        if (multilingual) {
            Console.WriteLine("Dobry den");
        }
    }
}
```

Listing 3.32 Complex invert `if` example.

```
void ConditionalHello(bool polite, bool multilingual) {
    if (!(polite)) {
        return ;
    }
    Console.WriteLine("Hello world");
    if (multilingual) {
        Console.WriteLine("Dobry den");
    }
}
```

Listing 3.33 Listing 3.32 refactored.

```
void ConditionalHello(bool polite, bool multilingual) {
    if (!(polite)) {
        return ;
    }
    Console.WriteLine("Hello world");
    if (!(multilingual)) {
        return ;
    }
    Console.WriteLine("Dobry den");
}
```

Listing 3.34 Listing 3.33 refactored.

Limitations

The refactorer is limited to just `void` functions but could potentially be extended to others in the cases where the necessary `return` statements were already present.

3.5.4 Simplify Condition

Introductory Example

Listing 3.35 below shows one use case of the Simplify condition refactorer. The problem is that once we get to the evaluation of `!a && b`, `a` must have been false, meaning that the second condition can be simplified to just `b`, as shown in Listing 3.36. No seasoned programmer would write this code. However, a beginner might, especially if we imagine more `else if` statements between the two `else if` statements in the example.

```
void m(bool a, bool b) {
    if (a) {
    /*Code1*/
    }
    else if (!a && b) {
    /*Code2*/
    }
}
```

Listing 3.35 Simplify condition basic example.

```
void m(bool a, bool b) {
    if (a) {
    /*Code1*/
    }
    else if (b) {
    /*Code2*/
    }
}
```

Listing 3.36 Listing 3.35 refactored.

Atoms

Atom is what we call a simple expression that is the endpoint of our analysis. They are binary expressions with operators ==, !=, <, <=, > or >=. Identifier names, like variable or property names, are also atoms. If our analysis ever reaches an identifier name, it necessarily means the underlying type of them is `bool` because we do not recurse under any operators that compare non `bool` types. Finally, simple member accesses, which include property and field accesses, are also atoms. However, invocations of class methods are not. This is due to our assumptions about side effects made in Section 2.1.2.

Possibility

The refactorer approaches this problem from the point of view of the `if` whose condition can be simplified, called in this chapter *main if*. In the case of Listing 3.35, the main if is `if (!a && b)`. Next, we need a list of conditions with a clear outcome at the point we are evaluating the condition of the main if. We will call this *simplifying list*. In the case of Listing 3.35, the list has just one condition `a`. We will use these for the simplification. Note that these do not need to just come from the preceding `if` and `else if` conditions in the same `if` and `else if` chain, but can also be a part of an enclosing `if`, shown in Listing 3.37 and Listing 3.38.

```
void m(bool a, bool b) {
    if (a) {
        if (a && b) {
        /*Code1*/
        }
    }
}
```

Listing 3.37 Simplify condition example with enclosing `if`.

```
void m(bool a, bool b) {
    if (a) {
        if (b) {
        /*Code1*/
        }
    }
}
```

Listing 3.38 Listing 3.37 refactored.

A limit to our solution is that we will only put atoms or their negations into the simplifying list. A general solution could use a logical solver to do more, but that is outside of the scope of this thesis.

So far, the examples only had one condition in the simplifying list, but consider Listing 3.39 below. Here, we can simplify `!a && b && c` based on `a` being false and `b` being true, and we get Listing 3.40.

```
void m(bool a, bool b, bool c)
    {
    if (a) {
    /*Code1*/
    }
    else if (b) {
        if (!a && b && c) {
        /*Code2*/
        }
    }
}
```

Listing 3.39 Simplify condition example with two entries in simplifying list.

```
void m(bool a, bool b, bool c)
    {
    if (a) {
    /*Code1*/
    }
    else if (b) {
        if (c) {
        /*Code2*/
        }
    }
}
```

Listing 3.40 Listing 3.39 refactored.

Note that if there was any code between `else if (b)` and `if (!a && b && c)`, the values of `a` and `b` might have been changed by that code, so we no longer put them in the simplifying list. This refactorer would not be able to simplify that code in any way.

The main if so far always consisted of a sequence of `&&`, but we allow `||` as well, an example shown in Listing 3.41 and Listing 3.42.

```
void m(bool a, bool b) {
    if (a) {
    /*Code1*/
    }
    else if (a || b) {
    /*Code2*/
    }
}
```

Listing 3.41 Simplify condition basic example with `||`.

```
void m(bool a, bool b) {
    if (a) {
    /*Code1*/
    }
    else if (b) {
    /*Code2*/
    }
}
```

Listing 3.42 Listing 3.41 refactored.

Consider Listing 3.35 and Listing 3.41. We know that `a` is always false when applying the refactorer on the `else if` statements, which neither helps nor hinders fulfilling the conditions `!a && b` and `a || b`, so we can just delete the respective part about `a`. So far, all the examples were like that. Now consider Listing 3.43. It is like Listing 3.35, but without the negation in `a && b`. This makes the `/*Code2*/` unreachable, and we can delete it as shown in Listing 3.44.

```
void m(bool a, bool b) {
    if (a) {
    /*Code1*/
    }
    else if (a && b) {
    /*Code2*/
    }
}
```

Listing 3.43 Simplify condition example with always false condition.

```
void m(bool a, bool b) {
    if (a) {
    /*Code1*/
    }
}
```

Listing 3.44 Listing 3.43 refactored.

Listing 3.45 below is an example just like Listing 3.35 but with an `||` instead of `&&`. The change means that `/*Code2*/` is always executed if we reach the `else if (!a || b)`. Thus, we can transform the `else if` into just an `else`, shown in Listing 3.46.

```
void m(bool a, bool b) {
    if (a) {
    /*Code1*/
    }
    else if (!a || b) {
    /*Code2*/
    }
}
```

Listing 3.45 Simplify condition example with always true condition.

```
void m(bool a, bool b) {
    if (a) {
    /*Code1*/
    }
    else {
    /*Code2*/
    }
}
```

Listing 3.46 Listing 3.45 refactored.

The last two examples are special cases that stem from simplifying the main if condition. Simplifying `if(a && b)` in Listing 3.43 is internally rewritten as `if(false)`. Conversely `if(!a || b)` in Listing 3.45 is internally rewritten as `if(true)`. The structural changes in Listing 3.44 and Listing 3.46 are just a more natural and overall clean version of the `if` with a boolean constant as a condition.

Note that if Listing 3.43 had more `else if` statements after `a && b`, we would leave those in place, as their statements might still be reachable. If Listing 3.45 had more `else if` statements after `!a || b`, the following `else if` statements were actually unreachable, so there is no reason to include them.

Finally, note that if the `if` we are refactoring is not a part of an `else if` (we have already shown an example like that in Listing 3.37), the behavior in the special cases where the condition is always true or always false is a bit different. If we find it is always true, we replace the whole `if` statement with its embedded statement. We also try the block simplification as described in Section 3.1.1. If it is always false, we replace it with the `if` statement in its `else if`.

To summarize, when is the refactoring of main if possible? It is possible when we are able to pair at least one member of the simplifying list to an atom or the negation of an atom in the condition of the main if.

Criterion

If it is possible, we recommend it. No inverse refactorer is present, as it does not apply to this refactorer.

Limitations

```
void m(bool a, MyClass x) {
    if (a) {
    /*Code1*/
    }
    else if (x.m() && a) {
    /*Code2*/
    }
}
```

Listing 3.47 Simplify condition example with potential side effect.

```
void m(bool a, bool b) {
    if (a) {
    /*Code1*/
    }
}
```

Listing 3.48 Listing 3.47 refactored.

Let us look at Listing 3.47. When refactoring `if(x.m() && a)`, we know `a` is false. We could evaluate that condition as always false and then get rid of it. This, however, would be incorrect if `x.m()` has any side effect. It could, in a similar example, even change the value of `a`.

We already talked about how we only consider expressions that are not side-effect-prone as atoms. This means that the simplifying list only includes what we consider side-effect-free expressions, as we only put atoms there. But to solve the problem in Listing 3.47, we must also make sure the main if condition is not side-effect-prone. Finally, when recursing into preceding conditions to find out the simplifying list, whenever we encounter a side-effect-prone condition, we must stop. Otherwise, we risk that the side effect could interfere with our presumed function of the code.

The main focus of this refactorer is on simple cases like Listing 3.35 and Listing 3.37, which are occasionally found in beginner code. The described functionality is a natural extension of these cases in a few directions but could be improved much further using semantic analysis, advanced code analysis, and mathematical logic techniques.

3.5.5 Return Condition

Introductory Example

```
bool m(bool a)
{
    if (a)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Listing 3.49 Return condition basic example.

```
bool m(bool a)
{
    return a;
}
```

Listing 3.50 Listing 3.49 refactored.

Listing 3.49 and Listing 3.50 show the basic application of the Return condition refactorer, which simplifies the `if` to just a `return`.

Possibility

The only requirement for the usage of this refactorer is that the bodies of the `if` and `else` contain only `return true;` and `return false;`. The boolean constants may be swapped. In that case, we negate the condition that is now in the `return`, shown in Listing 3.51 and Listing 3.52 below. The parentheses in `!(a)` are unnecessary but would be necessary if `a` was a binary expression, so we leave them there, as discussed in Section 2.3.3.

```
bool m(bool a)
{
    if (a)
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

Listing 3.51 Return the condition example with swapped boolean values.

```
bool m(bool a)
{
    return !(a);
}
```

Listing 3.52 Listing 3.51 refactored.

Criterion

If the refactoring is possible, we always recommend it. The only reason not to refactor is if we predict we will need to add some code that should only be run if the condition is either true or false. However, we cannot predict that, so we always recommend the refactoring.

Limitations

An inverse refactorer is not completely necessary here, as, without seeing into the future, applying this refactorer is always a good idea. But if the user later realizes they want to, for example, log something when the condition is true, it would be nice to have the option to go back to the `if` statement form, instead of having to do it manually. This would, however, require including `return` statements in the UI to have a place for the inverse refactorer. We chose not to include `return` statements in the UI to not overly complicate it for one small use case. But if we wanted to do so in the future, the refactorer interface and the surrounding structures are easily extensible to allow for `return` statement refactorers.

3.5.6 Simplify Parentheses

Introductory Example

```
void m(bool a, bool b, bool c)
{
     if (a && (b && c))
          Console.WriteLine("
    Hello World");
}
```

Listing 3.53 Simplify parentheses basic example.

```
void m(bool a, bool b, bool c)
{
     if (a && b && c)
          Console.WriteLine("
    Hello World");
}
```

Listing 3.54 Listing 3.53 refactored.

The idea behind this refactorer is summed up by its name Simplify parentheses: sometimes parentheses get overly complicated, and it is handy to have a tool that simplifies them. A basic example is shown in Listing 3.53 and Listing 3.54.

Possibility

The refactoring is possible if we find any pair of parentheses that can be simplified. We obviously need to respect operator precedence. For example, parentheses in the expression `a && (b || c)` cannot be omitted without changing the meaning.

Criterion

If it is possible, we always recommend it.

Limitations

During analysis, we focus primarily on operators like `&&`, `||`, negations, and parentheses. If we encounter anything uncommon, like `^` (XOR), we might keep parentheses even though they are unnecessary. We also do not recurse under these uncommon constructs.

3.5.7 Propagate Negations

Introductory Example

```
void m(bool a, bool b)
{
    if (!(a && b)) {
        Console.WriteLine("
    Hello World");
    }
}
```

Listing 3.55 Propagate negations basic example.

```
void m(bool a, bool b)
{
    if (!a || !b) {
        Console.WriteLine("
    Hello World");
    }
}
```

Listing 3.56 Listing 3.55 refactored.

Once again, this refactorer is summed up by its name Propagate negations. An example is shown in Listing 3.55 and Listing 3.56.

Possibility

The refactoring is possible whenever there is at least one negation that can be propagated into its operand.

Criterion

If it is possible, we always recommend it.

Limitations

Once again, as with the Simplify parentheses refactorer, we only focus on a select subset of expressions that constitute most conditions. When we encounter an uncommon operator like `^` (XOR) during the propagation, we simply wrap it in parentheses and a negation.

3.5.8 Combine Else and Embedded If

Introductory Example

```
void m(bool a, bool b)
{
    if (a) {
        Console.WriteLine("
    Hello World");
    }
    else {
        if (b) {
            Console.WriteLine("
    Ahoj svete");
        }
    }
}
```

Listing 3.57 Combine else and embedded if basic example.

```
void m(bool a, bool b)
{
    if (a) {
        Console.WriteLine(
        "Hello World");
    }
    else if (b) {
        Console.WriteLine(
        "Ahoj svete");
    }
}
```

Listing 3.58 Listing 3.57 refactored.

Listing 3.57 and Listing 3.58 show the basic application of this refactorer. It removes the unnecessary block between the `else` and `if` statement and thus reduces the nesting. Note we attribute this refactoring to the `else`.

Possibility

The refactor is possible only if the embedded block of the `else` contains only an `if` statement. If there is any other code before or after the `if` statement, this refactoring is impossible. If the `if` statement has an `else` or `else if`, we simply move those with the `if` statement during the refactoring, as shown in Listing 3.59 and Listing 3.60 below.

```
void m(bool a, bool b)
{
    if (a) {
        Console.WriteLine(
        "Hello World");
    }
    else {
        if (b) {
            Console.WriteLine(
            "Ahoj svete");
        }
        else {
            Console.WriteLine(
            "Hallo Welt");
        }
    }
}
```

Listing 3.59 Combine `else` and embedded `if` basic example.

```
void m(bool a, bool b)
{
    if (a) {
        Console.WriteLine(
        "Hello World");
    }
    else if (b) {
        Console.WriteLine(
        "Ahoj svete");
    }
    else {
        Console.WriteLine(
        "Hallo Welt");
    }
}
```

Listing 3.60 Listing 3.59 refactored.

Criterion

If it is possible, we always recommend it.

Limitations

Due to the simplicity of this refactorer, no limitations really apply.

Chapter 4

Implementation

In this chapter, we describe the implemented refactorer API and some interesting consequences and solutions stemming from that API. We then describe the core parts of Visual Studio integration.

4.1 Refactorers Implementation API

We will start by describing the refactorer API because it has consequences for many of our interesting implementation solutions.

4.1.1 API

What we have so far called a refactorer in this thesis is a class that implements the interface `IRefactorer<T>`. The interface has four methods, which all share the parameter `T node`, for which the given operation should be carried out:

- `GetRecommendation(T node)` which returns the recommendation for the refactoring on `node`. That recommendation is the `enum ERefactoringRecommendation`, which has four values: `Impossible`, `Recommended`, `Sidegrade` (an alternative that is neither better nor worse), and `Discouraged`. This directly implements the conclusion we arrived at in Section 3.2.1.

- `GetRefactoredRoot(T node)` which carries out the refactoring on `node` and returns the root of the syntax tree where the changes have been carried out. The implementations also annotate select nodes solely for paired highlighting without any effect on the source code.

- `GetHighlights(T node)` which returns a list of pairs, where each pair represents where and what type of highlight should be applied in the code pre-refactor. The highlight type is represented by the `enum EHighlight`, which has values that signify newly added code, deleted code, and then several values that are used for pairing similar constructs in code when appropriate. Pairing similar constructs is done by putting multiple pairs into the returning list, with different locations but the same `EHighlight`.

- `GetPairedHighlights(T node, SyntaxNode annotatedRefactoredRoot)` is similar to `GetHighlights` but is expected to be used when a preview of the refactored code is also available. It is the only method in the interface that has another parameter. The parameter `annotatedRefactoredRoot` contains the root of the tree where the refactoring has already been carried out, and it (usually) contains annotations solely present to facilitate paired highlights, which were put there during `GetRefactoredRoot`.

4.1.2 Type Parameter

The type parameter in `IRefactorer<T>` is `IfStatementSyntax` for most implemented refactorers. The one usage in this thesis where it is something else is in the Combine else with embedded if refactorer described in Section 3.5.8, where it is `ElseClauseSyntax`. `IRefactorer<T>` type parametrization is helpful because, in combination with other type parametrization, we can use the same code in the extensions view model transformation for both `if` and `else` statements. It would also be useful in any future attempts to generalize the tool. If there were any other constructs we wanted to have refactorers for, for example, variable declarations, we would require very few changes to the existing code.

4.1.3 Non-Parametrization of Refactorers

The one disadvantage of the `IRefactorer<T>` interface is that the refactors cannot be parameterized. For example, imagine we wanted to extend the Extract start refactorer described in Section 3.5.1 to be able to transform Listing 4.1 to Listing 4.2 below in one go. We would somehow need to pass the information that we want to extract a start of "length 2" to the refactorer. A possible workaround that uses the existing API would be to pass that additional information as an annotation. That is, however, only a workaround solution. A cleaner approach would be a further generalization of `IRefactorer<T>` interface that included such additional information in the method signatures, which is not implemented.

```
void m(bool a, bool b, bool c)
{
    if (a && b && c) {
        Console.WriteLine("
    Hello World");
    }
}
```

Listing 4.1 Extract start example with two && in the condition

```
void m(bool a, bool b, bool c)
{
    if (a && b) {
        if (c) {
            Console.WriteLine("
    Hello World");
        }
    }
}
```

Listing 4.2 Listing 4.1 possible refactor through our tools, but no in one step.

4.2 Concrete Solutions

4.2.1 Auto-Simplification

While using the Flattening refactorer described in Section 3.5.2, we might produce a condition like `(a || b) && (c || d)`. The parentheses here are necessary. But even when they are not necessary, such as if the resulting condition was just `(e) && (f)`, it is sometimes a desirable result that we want to show to the user. That is because it keeps the refactoring consistent. The refactorer always keeps the parentheses there. However, once the user understands this, we might want to offer an option to remove these unnecessary parentheses. In one way, we do that by having a dedicated refactorer that simplifies parentheses. But it would be inconvenient for the user to have to use it all the time. So we want an option to use it automatically. This presents several technical problems.

Why

We already mentioned unnecessary parentheses as one problem we want to auto-simplify. Another one is negations and their propagation, for similar reasons. Both of these are worthy of their own refactorer, which we described in Section 3.5.6 and Section 3.5.7. However, we should also be able to do them automatically, and that automation presents problems. Finally, there is the option of including clarity parentheses. Parentheses in `a || (b && c)` are technically unnecessary, but they lessen the mental load for even experienced programmers. We separate the problem of clarity parentheses from the parentheses simplification because then the responsibilities are clear. One auto-simplification option only removes parentheses, and the other only adds parentheses.

Limiting What

The first problem is which conditions to auto-simplify. The first thing to note is that it might not always be just one condition. This fact influenced the decision to return the root of the refactored tree from `IRefactorer<T>.GetRefactoredRoot`, as discussed in Section 3.2.2. Even when we only modify one condition and thus want to auto-simplify one condition, we still lose the information on which one it was due to returning the root.

We certainly do not want to auto-simplify conditions the refactorer did not work with at all, as that would be confusing for the user. In fact, we only want to automatically simplify the conditions that were directly important to the refactor. Some conditions might be inside blocks of code we are moving during the refactoring, but if they were not directly involved in the possibility and carrying out of the refactoring, we should not auto-simplify them. This means that we cannot, for example, check the syntax tree pre-refactor and post-refactor and auto-simplify based on their differences because we cannot easily differentiate the conditions important to the refactoring from the ones that just happened to be there.

The chosen solution is to use a concrete annotation across all refactorers in every `IRefactorer<T>.GetRefactoredRoot`, with which we tag all of the conditions that the refactorer has modified. These conditions might or might not be automatically simplified based on user preferences. This is a sort of hidden contract of `IRefactorer<T>`, but if it is not fulfilled, the only thing that happens is that we will be unable to auto-simplify the conditions where the annotation is missing.

Wrappers

Refactorers now annotate the conditions we want to auto-simplify. A further decision that needs to be made is how to act on these annotations. A way to seamlessly integrate the auto-simplifications among refactorers is to make them refactorers. We call these wrapper refactorers, but they still fulfill the same `IRefactorer<T>` interface. We call them “wrapper” because, contrary to so far mentioned refactorers, these need an inner refactorer they wrap around. When we ask a wrapper refactorer for `GetRecommendation`, it simply asks the inner refactorer for its `GetRecommendation` on the node it was given. The more interesting case is what the wrapper refactorer does for `GetRefactoredRoot`. First, we ask the inner refactorer for its `GetRefactoredRoot`. Then, we go through all of the conditions that were annotated as candidates for auto-simplification, and we do just that. Then, we return the result. Note that this approach allows for multiple wrapper refactorers.

Highlighting

While the wrapper refactorers could theoretically add additional highlights in `GetHighlights`, there is not a good use case for that. Hence, the reality is that just like `GetRecommendation`, they just return what the inner refactorer returns.

The paired highlights are much more interesting. We mentioned that, unlike other `IRefactorer<T>` methods, `GetPairedHighlights` has an additional parameter `annotatedRefactoredRoot`. The original reason for including it was that every `GetPairedHighlights` implementation started with calling the `GetRefactoredRoot(node)`. But paired highlights only make sense if we already have the refactored code somewhere, so we might as well pass it to `GetPairedHighlights` and save some resources.

This is where wrapper refactorers come into play and exploit this design choice. The normal refactorers implement the post-refactor highlights by annotating nodes during `GetRefactoredRoot`, and then translating the annotations to highlights in `GetPairedHighlights`.

If we calculated the post-refactor highlights based on the code before the auto-simplification, it would not visually (or logically) line up with the code we show to the user with the auto-simplification, so that is not an option. This only leaves us with the option to somehow calculate the post-refactor highlights based on the auto-simplified code.

However, there is a problem. Consider that the parentheses simplification wrapper might remove a pair of parentheses that were annotated for highlighting. The solution is to do auto-simplification carefully regarding annotations. When we, for example, remove a pair of parentheses in auto-simplification, we copy the annotation of the removed parentheses to its child. Note that both parentheses and negations are unary operators, so whenever we omit them, it is straightforward who should receive their annotations.

Usage Summary

So what do we do if we want to follow up a Flattening refactorer described in Section 3.5.2 with all three auto simplification options? That is, we want to simplify parentheses, but include clarity parentheses, and propagate negations. We create the Flattening refactorer, then wrap that refactorer by the wrapper one that takes care of the negations, then one for the parentheses simplification, and then the one that adds the clarity parentheses back. The ordering is important. The negation one might introduce many unnecessary parentheses, which means the parentheses simplification has to be after it. And there would be no point in adding clarity parentheses before omitting all of the unnecessary ones.

This approach allows us to use this triple-wrapped refactorer as if it were just the inner refactorer without any wrapping. Only the part of the code that translates user preferences regarding auto-simplification knows about the wrappers.

Simplify Parentheses and Clarity Parentheses Quirk

When we wrap the Simplify parentheses refactorer in the clarity parentheses wrapper refactorer, we get a package that, when refactoring, removes unnecessary parentheses and then adds the clarity ones. This, however, means that the package will be applicable to whatever it itself produces because it will contain unnecessary clarity parentheses. This is a quirk of our decision to separate the clarity parentheses from the parentheses simplification instead of making a combined "parentheses adjuster" or such. We could easily have special behavior for this special case. But we chose to keep this behavior, as it keeps the functioning of auto-simplification options consistent. The best solution would be to notify the user somehow why the refactor seemingly does not do anything, but we have no such notification system in place and chose not to implement it just for this special case.

4.2.2 Trivia

Most of our refactorers move code around. As discussed in Section 1.3, trivia, which includes comments and white spaces, falls under nearby syntax nodes. Note that the nodes or tokens Roslyn assigns trivia to might not be the actual reason it was put there, especially with comments.

Comments

We try to keep comments intact using methods like `IfStatementSyntax.WithCondition(ExpressionSyntax)`. These only replace the given part, in this case, the condition, while keeping everything else the same, including comments. However, consider the Flattening refactorer described in Section 3.5.2. As a part of our refactor, we must eliminate one of the `if` statements. Generally, whenever we delete or combine nodes as part of a refactor, we risk losing comments if they are attached to the nodes we are manipulating. We could tag the deleted comments onto another node, but it adds another layer of complexity, as they have no set structure, unlike code. Any implementation would require user adjustments to the comments in most cases, anyway. Thus, the implementation only keeps trivia to the extent that the family of methods like `IfStatementSyntax.WithCondition(ExpressionSyntax)` allows, but we never intentionally move trivia around to keep it.

This presents a problem if we take a different route during refactoring the intro example, which we analyzed in Section 2.3.1. By extracting `inUse` first, we can quickly arrive at the code in Listing 4.3.

```
void m(int pipeCapacity, bool inUse, string
    maintenanceInstructions) {
     if (pipeCapacity == 0)
     {
     /*replace it, it's too broken*/
     }
     else
     {
          if (inUse)
          {
               if (maintenanceInstructions != "decide")
               {
               /*turn off water and do what instructed to*/
               }
               else
               {
                    if (maintenanceInstructions == "decide"
                         || pipeCapacity == 0)
                    {
                    /*turn off water, replace it just to be sure,
                         or because nothing flows through*/
                    }
                    else if (pipeCapacity == 42)
                    {
                    /*investigate the magical pipe*/
                    }
               }
          }
     }
}
```

Listing 4.3 Introductory example alternative path.

The comments in the example are meant to stand for some functional code. But if we take them literally, and apply the Simplify condition refactorer on `if (maintenanceInstructions == "decide" || pipeCapacity == 0)`, we get Listing 4.4 below. This is because block removal, discussed in Section 3.1.1, comes into play. If no statements are inside a block, any comments inside the block must be attributed to the block. When we remove that block, we pay no special attention to the comments, and we remove them as well.

```
void m(int pipeCapacity, bool inUse, string
    maintenanceInstructions) {
     if (pipeCapacity == 0)
     {
     /*replace it, it's too broken*/
     }
     else
     {
          if (inUse)
          {
               if (maintenanceInstructions != "decide")
               {
               /*turn off water and do what instructed to*/
               }
               else
               {
               }
          }
     }
}
```

Listing 4.4 Listing 4.3 refactored using Simplify condition refactorer, with comments taken literally.

Whitespaces

In the case of whitespaces, we cannot just try to keep them as they are. We often increase or lessen the nesting of code, which we need to signify by increasing or decreasing the indentation of the code, which means changing whitespaces. Another problem is that `SyntaxFactory` methods, by default, do not include any trivia. For example, `SyntaxFactory.ReturnStatement(SyntaxFactory.LiteralExpression(SyntaxKind.TrueLiteralExpression))` results in `returntrue;`, without the space that is necessary when we turn the syntax tree back to code.

We utilize the function `SyntaxNode.NormalizeWhitespace()`, which does exactly as its name implies. It is, however, very aggressive in that normalization and has very few settings.

A first idea to limit the impact of `NormalizeWhitespace()` would be to only apply it as locally as possible. This does not work because the function does not know at what indentation we are, as it assumes the node we call it on has none. We could, of course, manually increase or decrease that indentation, but the more non-conforming code is compared to the whitespaces `NormalizeWhitespace()` prescribes, the more difficult these manual changes will be.

It is also unnecessary, as we are not the first to have a problem with too heavy-handed whitespace normalization. The usual approach is to utilize the `Formatter`

`.Format()` function from `Microsoft.CodeAnalysis.Formatting`. This function requires a workspace from which it takes formatting settings. It can either format a syntax node we directly pass to it or format all nodes with a specific annotation. The important difference is that when formatting a node that is part of a larger tree, it takes its context into account, mainly the indentation. Thus, it can have the limited impact `NormalizeWhitespace()` cannot easily have.

We, however, currently do not utilize `Formatter.Format()`. Currently, all implemented refactorers directly use `NormalizeWhitespace()` in their implementation. A cleaner way to do whitespace normalization would be to create a whitespace normalizing wrapper refactorer. We could reuse the existing auto-simplification annotations or create new ones just for the whitespace normalizing. This wrapper could then use `Formatter.Format()` on either a user workspace or one with our custom format.

Implementing the solution described in the previous paragraph would not be that difficult by itself. However, it would require changing most of the refactor tests we use. Early on, we decided to make the refactor tests fit the output exactly, including whitespaces. This, however, means that changing the way we normalize those whitespaces would necessitate adjusting the tests.

We could also implement a wrapper refactorer that uses `Formatter.Format()`, and another one which would use `NormalizeWhitespace()` to keep the tests relevant. However, the best way, but one that would take a lot of effort, would be to remake the tests and test what we will actually present to the user.

4.2.3 VS Integration

We will now describe the integration into Visual Studio. Figure 4.1 shows a simplified UML diagram that demonstrates the relationships between classes. All relationships without explicit cardinality are one-to-one. Some classes are omitted, so the diagram is not overly complicated, but we describe them in the text.

WPF and View Models

The front end of the extension uses WPF, as is usual for VS extension. We utilize binding to keep our UI up to date with our analysis. The source for that binding is `MainViewModel`, which has bindable properties for settings, property for the text we display, and, most importantly, it contains a bindable list of `BranchingStatementViewModel` (`BSVM`) representing `if`, `else if` and `else` statements directly in the body of a function we are analyzing. Every `BSVM` has its own list of nested `BSVM` that represent the nested code constructs.

Every `BSVM` has a list of `RefactorerViewModel`, which represent a potential application of a concrete refactorer on the concrete `if`, `else if` or `else` the given

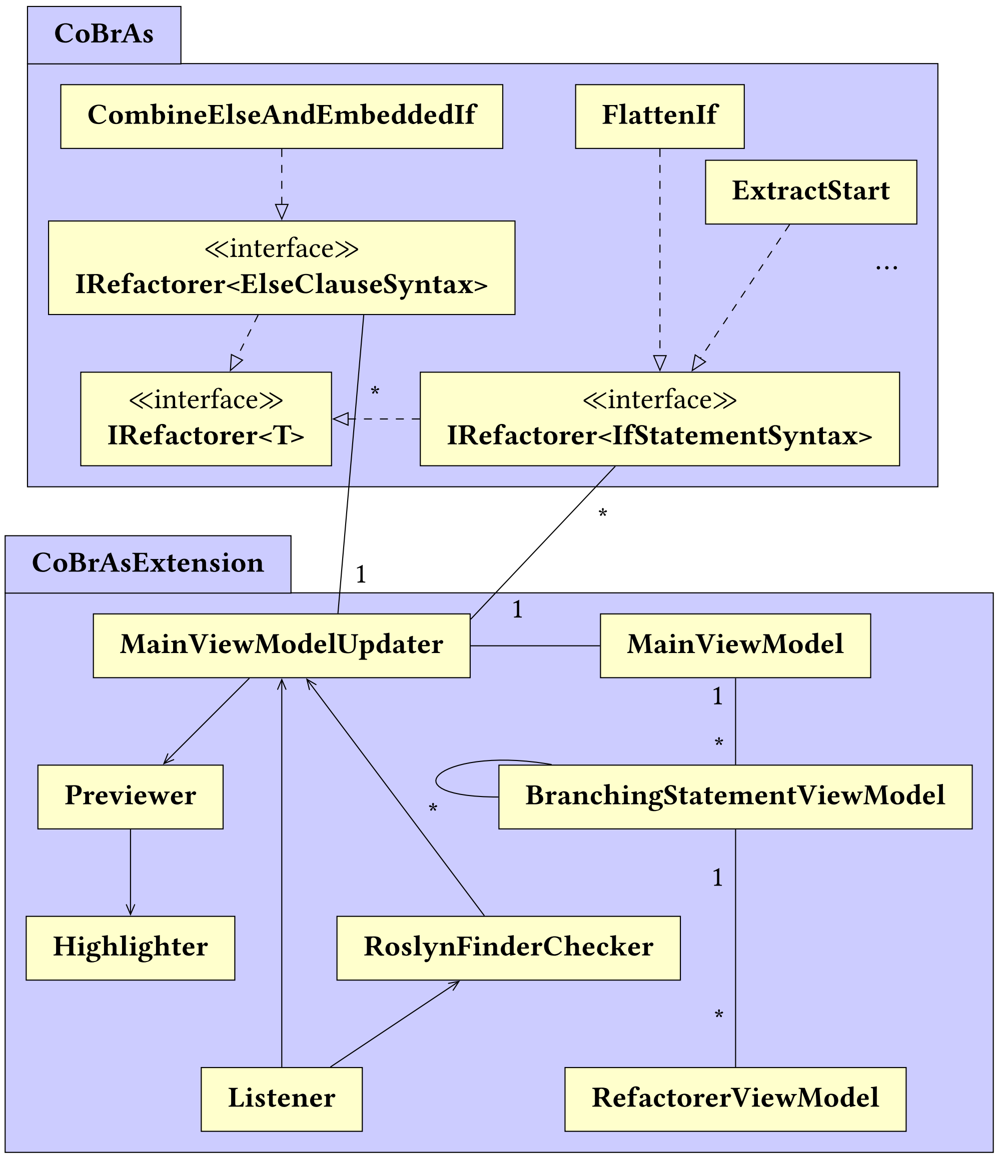


Figure 4.1 Visual Studio integration simplified UML diagram.

`BSVM` represents. We bind buttons to these. The refactor being impossible means the button is invisible, and if it is possible, recommendations result in different colors. We also use the binding to bind to `ICommand` that carries out the highlighting and refactoring.

`MainViewModel` is only ever manipulated by `MainViewModelUpdater`. We separate the data from the manipulation because `MainViewModel` is already busy with being WPF binding compliant, and `MainViewModelUpdater` contains non-trivial logic that transforms our analysis into the bindable structures of `MainViewModel`, `BSVM`, and `RefactorerViewModel`.

Previewing and Highlighting

`MainViewModelUpdater` utilizes `MainViewModel` to express ourselves in our own tool window. However, highlighting in the old code and previewing and highlighting in the refactored code requires us to express ourselves outside of our tool window. `MainViewModelUpdater` utilizes `Previwer` for that.

`Previwer` utilizes `Highlighter` for highlighting on `DocumentView` (which represents an open text document) with user code, with our preview also being `DocumentView`, meaning we can use the same API to highlight on both the user code and our preview. `Previwer` is also responsible for creating that preview.

`Highlighter` encapsulates the raw VS highlighting (further just called RAW). Such an encapsulation is required because we are using it somewhat unnaturally. RAW expects the analysis required to be done inside of its implementation. Whereas, in our case, we want to highlight something occasionally, but a good encapsulation and splitting of responsibilities necessitates the code that does analysis cannot be inside of our RAW implementation.

In a more low-level description, RAW provider is automatically instantiated for an open document. But we need to access that RAW provider from the outside to put our highlights in. However, due to the automatic instantiation, we found it difficult to gain that access. A solution we found is that `Microsoft.VisualStudio.Text.ITextBuffer`, which represents a sequence of characters and is utilized by `DocumentView`, has the so-called properties (not to be confused with the code construct). Any type can be put into those properties, and it can also be retrieved using the type.

We can put the RAW provider into those properties during instantiation as `ITextBuffer` is easily accessible there. It is also easily accessible from `Highlighter` using `DocumentView.TextBuffer`, whose properties we then query for the RAW provider type.

Servicing Updates

We have yet to talk about what starts or refreshes our analysis. That is the responsibility of `Listener`. On startup, we use it to start listening to workspace events. These include both big events, such as loading a different solution, and small ones, such as just writing a single character. We also periodically use `Listener` to start listening for the currently open documents moving of the caret. Note that the movement as a result of the usual text edits is not considered moving the caret, meaning our two listening points do not overlap. We let the workspace changes settle for a short period of time before we start our analysis simply because normal programming often results in non-compilable code before one finishes. To most caret changes, we react immediately, but not when the user is moving the caret as a part of a selection. In that case, we do not react at all.

`Listener` only has access to `MainViewModelUpdater` for the purposes of reporting reasons we could not analyze, for example, that we are waiting for the workspace changes to settle. If `Listener` finally decides it is time to analyze, it does not call `MainViewModelUpdater` but calls `RoslynFinderChecker`.

The responsibility of `RoslynFinderChecker` is to primarily retrieve the Roslyn instance already running in VS and check it for both syntax and semantic errors. It also retrieves other internal VS information required for the actual file changes, previewing, and such. If it does not find any errors, it takes the retrieved information and finally passes it to `MainViewModelUpdater`.

Transforming Refactorers to View Models

`MainViewModelUpdater` has a hardcoded list of implemented `IRefactorer<IfStatementSyntax>` and `IRefactorer<ElseClauseSyntax>` it uses when creating the view models. There is a number of `IRefactorer<IfStatementSyntax>` implemented, but only one `IRefactorer<ElseClauseSyntax>`, but there would be no problem in integrating more.

We will now describe the transformation of a Roslyn syntax tree into the view models. First, we summarize `if`, `else if`, and `else` statements into descendants of a class `BranchingStatementSummary<T>` (not displayed in the diagram for the sake of brevity). These summaries are IDE-independent and simply serve as a summary of conditional branching statements without any of the code between them. Contrary to the view models, they follow the recursiveness of code constructs. Importantly, they include the `SyntaxNode` the summary represents.

`MainViewModelUpdater` then goes through these summaries. It has to transform them from the language rule recursiveness to the one we need in the tree view and thus view models. We want to display an `else if` statement on the same level as its ancestor `if` statement, even though in the language rules, their relationship is descendant-ancestor.

When we are view modeling a summary of `if` or `else if` statements, the hardcoded list of `IRefactorer<IfStatementSyntax>` applies. For `else` statements, the hardcoded list of `IRefactorer<ElseClauseSyntax>` applies.

For a given summary, we first create its `BranchingStatementViewModel`. Then we go through all of the refactorers that apply to it. For each refactorer, we first get its `GetRecommendation` on the node the summary represents. If it is not impossible, that refactorer is not applicable to the current node. If it is not impossible, we also speculatively try to calculate the refactored code and highlights and declare it impossible if any exceptions appear. This is to increase the stability of our implementation.

If the refactor is possible, we transform it into a `RefactorerViewModel` under the current `BranchingStatementViewModel`. The recommendation we calculated translates into the color of the button we display. The actions on hovering and clicking on the button are (under some wrappers necessary for WPF) facilitated through C# `Actions`.

As a part of these `Actions`, we wrap the base refactorer with wrapper refactorers. The wrapper refactorers are not shown in the diagram for the sake of brevity, and we will now describe the reality of their instantiation. The `MainViewModelUpdater` simply has the wrapper refactorers hardcoded in a local function, which conditionally does the wrapping based on the settings. We utilize variable capturing to keep them up to date with the current settings. This means we do not need to recalculate the view models when the settings change.

Mock View Model and Theming

We utilize a design time mock view model that runs our analysis on a set piece of code. It is not presented in the diagram to avoid overcomplicating it. It inherits from `MainViewModel` and fills out its properties in the constructor.

The usage of a design time mock view model influenced the design a bit due to the technical constraints of the mock binding. It, for example, required `BranchingStatementViewModel` and `RefactorerViewModel` to not be type parametrized. However, the mock view model allows easier redesigning of the tree view part of the UI.

We use the Visual Studio Community toolkit to align the theming of our tool window with the rest of the Visual Studio. This means the design time mock viewed in the designer will not be how the tool window looks exactly because the automatic theming needs a VS instance to function. But it will still look very similar.

Exceptions

We utilize the Visual Studio Community toolkit to log any exceptions we encounter. The log is in the Output window of Visual Studio, in a selection named Extensions.

Conclusion

In our analysis, we identified that the main problem with existing ways of providing refactors, such as CodeFixes, is that they do not focus on beginners. Instead, they focus on improving the efficiency of programmers who already know what they want to do. We also identified a problem with the availability and approachability of CodeFixes for beginners.

As a solution, we implemented a Visual Studio extension that lets the user explore the already present `if` statements in their C# code. Whenever the extension has a refactor to offer, it is clearly presented to the user in the dedicated part of the UI of the extension, which summarizes the `if` statements from the user code. This is in contrast to CodeFixes, which are only locally available and hidden behind a few menus.

The extension not only offers the refactor but also recommends them in a beginner-friendly simple color coding. Moreover, the extension explains the reasoning why the refactors are possible using highlighting in the user code. Whenever the user is considering a refactor, they can utilize this highlighting on either the old code or on both the old code and a preview of the refactored code to better understand the refactor. The comparable feature of CodeFixes is a simple diff, which might be hard to understand for new users, and simply describes the textual changes instead of the ideas behind them.

Overall, the goal of this thesis was met. We put emphasis on our users being beginner programmers and designed the extension to cater to them. The extension is easy to install and use, which is important for beginners.

Future Work

The most valuable addition would be a gentler handling of user white spaces, for which we already outlined a solution in Section 4.2.2.

The existing refactor analyses could be deepened to cover more cases. Some of them could also utilize much more advanced code analysis techniques. Test coverage could also always be improved.

The implementation is easily extensible with refactors of new mistakes for `if`, `else if`, and `else` statements, albeit with the caveat of refactor non-parametrization described in Section 4.1.3. Our infrastructure could also reasonably be extended to accommodate refactors on, for example, `return` statements.

Textual explanations alongside refactors could help beginners better understand them. However, if done in the detail necessary to be helpful, they would immensely increase the complexity of a simple refactor analysis. Similarly, explaining why a refactor idea does not apply in a specific case could be helpful but also very complex.

Bibliography

[1] Microsoft. *If statement language specification.* URL: `https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/statements#1382-the-if-statement` (visited on 07/12/2024).

[2] Paul W Abrahams. "A Final Solution to the Dangling else of ALGOL 60 and Related Languages". In: *Communications of the ACM* 9.9 (1966), pp. 679–682.

[3] T.J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: `10.1109/TSE.1976.233837`.

[4] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc., 1977.

[5] Norman Peitek et al. "Program Comprehension and Code Complexity Metrics: An fMRI Study". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* 2021, pp. 524–536. DOI: `10.1109/ICSE43902.2021.00056`.

[6] Microsoft. *Roslyn GitHub repository.* URL: `https://github.com/dotnet/roslyn` (visited on 06/27/2024).

[7] Microsoft. *Roslyn syntax introduction.* URL: `https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/work-with-syntax` (visited on 07/12/2024).

[8] Microsoft. *Analyzer and CodeFix overview.* URL: `https://learn.microsoft com/en-us/visualstudio/code-quality/roslyn-analyzers-overview?view=vs-2022` (visited on 07/12/2024).

[9] Microsoft. *Visual Studio 2022 homepage.* URL: `https://visualstudio.microsoft.com/cs/vs/` (visited on 07/12/2024).

[10] Microsoft. *Visual Studio 2022 extensibility resources.* URL: `https://learn.microsoft.com/en-us/visualstudio/extensibility/?view=vs-2022` (visited on 07/12/2024).

[11] Mads Kristensen and community. *Visual Studio Community toolkit*. URL: `https://github.com/VsixCommunity/Community.VisualStudio.Toolkit` (visited on 07/08/2024).

[12] James C. King. “Symbolic execution and program testing”. In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: `10.1145/360248.360252`. URL: `https://doi.org/10.1145/360248.360252`.

Appendix A

Attachment structure

`CoBrAsExtension.vsix` is the file using which you can install our extension.

`repository` contains a copy of the repository that was used during the development. It includes all of the git files, including the history of the repository.

Source code, solution and project files, and other necessary files needed to produce and test our extension are present in folders `repository\CoBrAs`, `repository\CoBrAsExtension`, and `repository\CoBrAsTests`. The solution file specifically is present in `repository\CoBrAs`.

An example solution on which we recommend trying out our extension is in `repository\Examples`.

Finally, `repository\Analysis` contains a few proof concept solutions and informal notes about use cases. In `repository\SourceOfMistakes` is a solution written by a beginner programmer, which served as a source for most of the use cases we focus on in the thesis.

Appendix B

Using CoBrAs (User Documentation)

Conditional Branching Assistant is a Visual Studio 2022 extension that aims to help beginners refactor the `if` statements in their C# code. It not only provides the refactors but also helps the user understand them through highlighting.

B.1 Installing VS

To find out the requirements and install Visual Studio 2022, follow the steps on their installation guide[1]. Version 17.10.4 or higher is required. As our extension only focuses on C#, you must install the appropriate workload (as of writing, named ".NET desktop development") in Visual Studio Installer to utilize our extension. If the workload is not installed, you will see an error when trying to open the extension as described in Appendix B.4.1.

B.2 Installing Our Extension

Our extension is attached as a .vsix file, see Appendix A for more details. Once you have VS2022 installed, it should set itself as a default application to manipulate .vsix files. You can simply double-click it to start the installation and follow the instructions from there[2].

[1]https://learn.microsoft.com/en-us/visualstudio/install/install-visual-studio?view=vs-2022

[2]https://learn.microsoft.com/en-us/visualstudio/ide/finding-and-using-visual-studio-extensions?view=vs-2022#install-without-using-the-manage-extensions-dialog-box

Building Our Extension

To produce the .vsix file, you need to build the source code, which is available as an attachment; see Appendix A for more details. Extension development requires an additional workload "Visual Studio extension development". The solution contains three projects: one with the analysis that is not IDE specific (also happens to be a console project for debugging), one with the extension, and one with tests. We utilize a number of packages through NuGet, but those should not require any explicit actions to work.

B.3 Example Solution

We prepared an example solution/project/file, available as an attachment. See Appendix A for details. We encourage opening that example while following the rest of this documentation.

The only .cs file in the solution contains a number of functions. Each function contains code tailored for one of the use cases we focus on. Some of the use cases are covered by multiple functions. Even though the examples are tailored to a specific use case, others may be available.

B.4 Using the Extension

B.4.1 Opening the Tool Window

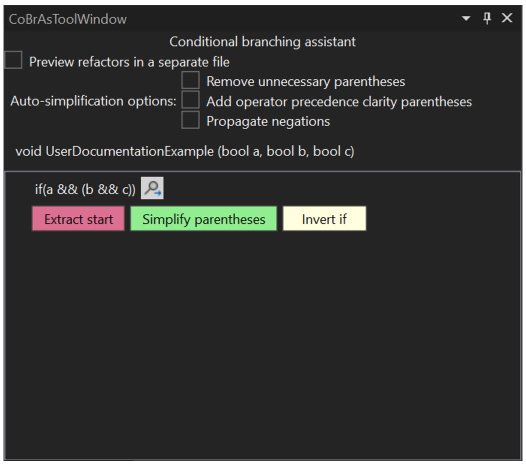
The tool window can be opened by clicking **View** > **Other Windows** > **CoBrAs**. After clicking into a function, the tool window should look similar to Figure B.1.

B.4.2 Settings

The top part of our tool window contains settings in the form of four checkboxes. The first one labeled "Preview refactors in a separate file" and the latter three have different functions.

Preview

Clicking the "Preview refactors in a separate file" checkbox will immediately create a new document, similar to one with user code. We recommend moving it side by side with your code, so you can compare the differences between our preview and your code.


CoBrAsToolWindow

Conditional branching assistant

☐ Preview refactors in a separate file

Auto-simplification options: ☐ Remove unnecessary parentheses
☐ Add operator precedence clarity parentheses
☐ Propagate negations

void UserDocumentationExample (bool a, bool b, bool c)

if(a && (b && c))

Figure B.1 CoBrAs tool window

If you accidentally close the preview window, we will automatically open it again, provided the checkbox is still checked, once we have something to show in it. We automatically close the preview window if you uncheck the checkbox.

Auto-Simplification

The rest of the settings concern themselves with the refactor we propose. Some of the refactors we produce may require adding parentheses or a negation to conditions. These parentheses might sometimes be unnecessary, and the operand of the negation might be a long expression, while the negation could be propagated into the long expression. The auto-simplification options "Remove unnecessary parentheses" and "Propagate negations" are for that exact purpose. The remaining auto-simplification option we have yet to talk about is "Add operator precedence clarity parentheses". This option results in adding parentheses to conditions like `a || b && c`, in this case resulting in `a || (b && c)`. The parentheses are unnecessary due to operator precedence but might sometimes be helpful so the reader does not have to think about operator precedence.

If checked, whenever we refactor, all of the conditions that were directly part of the refactor get the unnecessary parentheses removed, clarity parentheses added, or negations propagated, depending on the settings. Note that these options might make some of the refactors harder to understand.

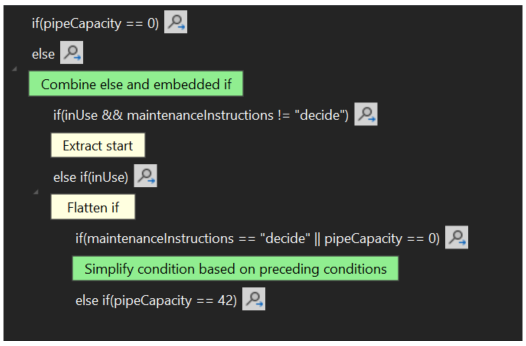
B.4.3 Main Part of the UI

The non-setting part of the UI always relates to the part of user code where the caret is.

Function Signature

Directly under the settings, we display either a message about why our analysis was not completed or (in the usual case) the signature of the function inside of which the caret is and that our analysis applies to.

Tree View

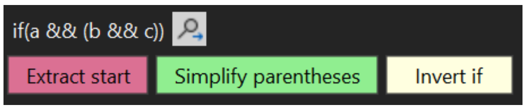


```
if(pipeCapacity == 0)
else
Combine else and embedded if
  if(inUse && maintenanceInstructions != "decide")
  Extract start
  else if(inUse)
  Flatten if
    if(maintenanceInstructions == "decide" || pipeCapacity == 0)
    Simplify condition based on preceding conditions
    else if(pipeCapacity == 42)
```

Figure B.2 Tree view for Listing 0.1

A tree view with custom items follows, which only shows `if`, `else if` and `else` statements from the function we are analyzing. The nesting in the tree view follows the nesting in the code, as shown in Figure B.2 for Listing 0.1.

Tree View Item

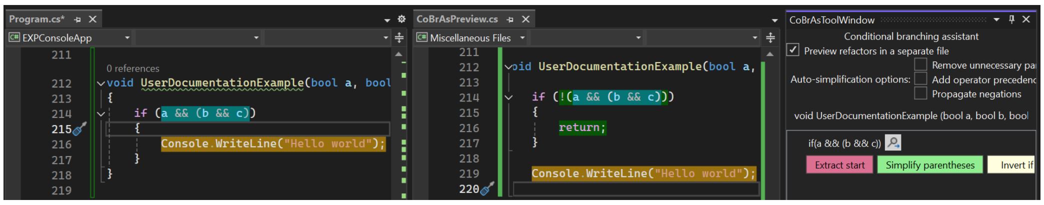


```
if(a && (b && c))
Extract start   Simplify parentheses   Invert if
```

Figure B.3 Tree view item

Every tree view item, for example, in Figure B.3, contains the text of the conditional branching statement it represents. Next to it is a button with an icon of a magnifying glass with an arrow, which, on clicking, jumps to the conditional branching statement the tree view item represents.

After that, every tree view item contains a variable number of buttons, where each of the buttons represents a possible refactor. The buttons are color-coded based on how much we recommend the refactor. Red means we discourage it, yellow means we neither discourage nor recommend it, and green means we recommend it. Whenever you hover one of these buttons, we utilize highlight in both your code and the preview to explain what is happening in the refactor, as shown in Figure B.4 below, where we hovered over the "Invert if" button. We automatically scroll to the highlights, but you can also manually scroll around.



```
void UserDocumentationExample(bool a, bool
{
    if (a && (b && c))
    {
        Console.WriteLine("Hello world");
    }
}
```

```
oid UserDocumentationExample(bool a,

    if (!(a && (b && c)))
    {
        return;
    }

    Console.WriteLine("Hello world");
```

[✓] Preview refactors in a separate file

Auto-simplification options: [ ] Remove unnecessary pa [ ] Add operator precedenc [ ] Propagate negations

void UserDocumentationExample (bool a, bool b, bool

if(a && (b && c))

Figure B.4 Full example of highlighting and preview

B.5 Disabling or Uninstalling Our Extension

To disable (which allows later reactivation) or uninstall our extension, you have to use the **Manage Extensions** dialog, accessible in VS2022 from **Extensions** > **Manage Extensions**. Once there, you need to find our extensions named CoBrAsExtension and disable or uninstall it. We utilize no persistent settings or files you could potentially want to keep, so the uninstallation process is straightforward.