Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Engineering (FEM)

Master's Thesis

Developing a refactoring extension for Rider IDE

Mykhailo Kryhin

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

DIPLOMA THESIS ASSIGNMENT

Mykhailo Kryhin

Informatics

Thesis title

Developing a refactoring extension for Rider IDE

Objectives of thesis

The main objective of the thesis is to develop a refactoring extension for Rider IDE. The extension will find, classify, and suggest fixes for the most widespread code organization problems known as 'code smells'. The secondary objective is to categorize mentioned code issues from different sources into one.

Methodology

The thesis will be based on the 'Code smells' materials – small signs of a bad code that can be improved. We will model the extension by using BPMN methodology and UML tools. That will give us the ability to achieve a clear and easily understandable process of code fixes. As a developing language, the C# will be used. To create an IDE extension we also will use custom open-source SDKs (software development kits) of the selected IDE.

The proposed extent of the thesis

60-80 pages

Keywords

Code smells, IDE extension, Refactoring, Code improvement, Software quality tooll

Recommended information sources

Design patterns and refactoring. SourceMaking [online]. [Accessed 20 May 2022]. Available from:
https://sourcemaking.com/refactoring

Fowler, M., 2019. Refactoring. Boston: Addison-Wesley.

KOCH, Matthias. Writing plugins for Resharper and rider: The .net tools blog. The JetBrains Blog [online].
14 February 2019. [Accessed 20 May 2022]. Available from:
https://blog.jetbrains.com/dotnet/2019/02/14/writing-plugins-resharper-rider/

MARTIN, Robert Cecil. Clean code. Milano : Apogeo, 2018.

MCCONNELL, Steve. Code complete. Unterschleißheim : Microsoft Press, 2012.

Suryanarayana, G., Samarthyam, G. and Sharma, T., 2015. Refactoring for software design smells.
Amsterdam: Elsevier/Morgan Kaufmann,

Expected date of thesis defence

2022/23 SS – FEM

The Diploma Thesis Supervisor

Ing. Marek Pícka, Ph.D.

Supervising department

Department of Information Engineering

Electronic approval: 4. 11. 2022

Ing. Martin Pelikán, Ph.D.

Head of department

Electronic approval: 28. 11. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Dean

Prague on 28. 03. 2023

Declaration

I declare that I have worked on my master's thesis titled "Developing a refactoring extension for Rider IDE" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the master's thesis, I declare that the thesis does not break any copyrights.

In Prague on 31.03.2023 _____

Acknowledgement

I would like to extend my heartfelt appreciation to my supervisor, Ing. Marek Pícka, Ph.D., for giving me the opportunity to undertake this work under his supervision and for providing invaluable guidance and support throughout the process. His advice and encouragement have been instrumental in the successful completion of this work.

Developing a refactoring extension for Rider IDE

Abstract

The Rider Integrated Development Environment (IDE) is a popular tool for developing .NET applications. However, it does not currently have an extension for quickly refactoring code. In this project, we propose to develop such an extension for Rider. Our extension will allow users to perform common refactoring operations directly from within the IDE, such as renaming variables and extracting methods.

It will also provide automatic suggestions for refactoring opportunities and allow users to preview the changes before applying them. To ensure the extension is easy to use and reliable, we will follow established best practices for developing Rider extensions and thoroughly test the extension on various codebases. Overall, this project aims to create a valuable tool for Rider users that makes it easier to improve the quality of their code through refactoring.

Keywords: Code smells, IDE extension, Refactoring, Code improvement, Software quality tool

Czech version of abstract

Abstrakt

Integrované vývojové prostředí (IDE) Rider je oblíbeným nástrojem pro vývoj aplikací .NET. V současné době však nemá rozšíření pro rychlou refaktorizaci kódu. V tomto projektu navrhujeme vyvinout takové rozšíření pro Rider. Naše rozšíření umožní uživatelům provádět běžné refaktorizační operace přímo z prostředí IDE, například přejmenovávat proměnné a extrahovat metody.

Bude také poskytovat automatické návrhy možností refaktorizace a umožní uživatelům zobrazit náhled změn před jejich použitím. Abychom zajistili snadné používání a spolehlivost rozšíření, budeme se řídit zavedenými osvědčenými postupy pro vývoj rozšíření Rider a rozšíření důkladně otestujeme na různých kódových bázích. Celkově je cílem tohoto projektu vytvořit pro uživatele aplikace Rider cenný nástroj, který jim usnadní zlepšování kvality jejich kódu pomocí refaktorizace.

Klíčová slova: Pachy kódu, Rozšíření IDE, Refaktoring, Zlepšování kódu, Nástroj pro kvalitu softwaru

Table of content

10

1 Introduction

Code refactoring is a vital practice in software development that helps improve code design and structure without changing its functionality. It can make code more readable, easier to understand, and easier to maintain, as well as more efficient and scalable. However, performing refactorings manually can be time-consuming and error-prone, especially in large codebases. This is where tools like the Rider Integrated Development Environment (IDE) come in.

Rider is a popular and powerful IDE for developing .NET applications, but it currently lacks an extension for quickly refactoring code. In this diploma paper, we propose to fill this gap by creating a refactoring extension for Rider. Our extension will allow developers to perform common refactoring operations directly from within the IDE, such as renaming variables and extracting methods. It will also provide automatic suggestions for refactoring opportunities and allow users to preview the changes before applying them.

The development of this extension presents several challenges. First, we must ensure that the extension is easy to use and reliable, so that developers can trust the refactorings it suggests and applies. Second, we must ensure that the extension integrates seamlessly with Rider and does not negatively impact its performance. Finally, we must ensure that the extension is able to handle a wide variety of codebases and refactoring scenarios. To address these challenges, we will follow established best practices for developing Rider extensions and thoroughly test the extension on a variety of codebases. Our goal is to create a valuable tool for Rider users that helps them to improve the quality of their code through refactoring. In the following sections, we will provide a background on refactoring and its importance in software development, describe the features and functionality of the extension, and present the results of our testing and evaluation.

2 Objectives and Methodologies

2.1 Objectives

The main objective of the thesis is to develop a refactoring extension for Rider IDE. The extension will find, classify, and suggest fixes for the most widespread code organization problems known as 'code smells'. The secondary objective is to categorize mentioned code issues from different sources into one.

1. To develop a refactoring extension for the Rider IDE that makes it easier for developers to perform joint refactoring operations and to receive automatic suggestions for refactoring opportunities.

2. To ensure that the extension is easy to use, reliable, and seamlessly integrated with Rider.

3. To evaluate the extension's effectiveness and reliability through testing on various codebases.

2.2 Methodology

The thesis will be based on the 'Code smells' materials - small signs of a bad code that can be improved. We will model the extension by using activity diagram UML tools. That will give us the ability to achieve a clear and easily understandable process of code fixes. As a developing language, the C# will be used. To create an IDE extension, we also will use custom open-source SDKs (software development kits) of the selected IDE.

1. Research: To identify common refactoring operations known as 'code smells' and best practices for developing Rider extensions.

2. Design: To design the features and functionality of the extension, including integration with Rider.

3. Implementation: To implement the extension using the Rider Extension SDK.

4. Testing: To test the extension on various codebases and refactoring scenarios using both manual and automated testing.

5. Evaluation: To evaluate the testing results and gather feedback from users of the extension.

3 Literature overview

3.1 Refactoring Signs

Refactoring is the process of changing the structure of code without changing its behavior. This typically involves making the code easier to read, easier to understand, and easier to maintain. It can also involve making the code more efficient or more scalable. Refactoring is an essential practice in software development because it helps improve the codebase's quality over time. It is usually done on an ongoing basis as part of the normal development process and can be performed by both individuals and teams [1 p. 9].

There are many different techniques for refactoring code, such as extracting methods, renaming variables, and reorganizing code into smaller, more cohesive units. Refactoring can be done manually, or it can be automated using tools like refactoring browsers or integrated development environments (IDEs) [1 p. 328]. It is typically done in small increments, with the goal of making incremental improvements to the codebase over time.

Refactoring is important for a number of reasons [1]:

- It helps improve the design of the code, making it more maintainable and easier to understand. This is especially important in large codebases where multiple people may be working on the same code.

- It can help fix bugs and improve the performance of the code. By restructuring the code and eliminating unnecessary or inefficient parts, you can often resolve issues causing problems.

- It can make it easier to add new features to the code. By having a well-organized and maintainable codebase, it's much easier to make changes and add new functionality without introducing new bugs or breaking existing features.

Overall, refactoring helps ensure that the code is of high quality and can evolve over time to meet the project's changing needs [1 p. 51].

Code smells are characteristics of code that may indicate a deeper issue with the design or structure of the code. There are many different types of code smells, and different developers may have their own definitions and examples [1 p. 11].

Here are a few common types of code smells [1]:

1. **Long Method**: A method that is too long and complex, making it difficult to understand and maintain.

2. **Duplicate Code**: Duplicate similar blocks of code that could be refactored into a single, reusable method.

3. **Large Class**: A class with too many responsibilities or too much code makes it difficult to understand and maintain.

4. **Inappropriate Intimacy**: Tight coupling between classes that should be more independent.

5. **Switch Statements**: Complex switch statements that could be refactored using polymorphism.

6. **Lazy Class**: A class that does very little and could potentially be eliminated or combined with another class.

7. **Speculative Generality**: Code that is implemented to handle potential future requirements, but which is not currently being used.

These are just a few examples of code smells. There are many others, and different developers may have different opinions on which code smells are the most important to address.

Code smells can be categorized in various ways, depending on the perspective of the person doing the categorization. Here are a few examples of ways that code smells could be categorized:

By type of code element: Code smells can be categorized based on the type of code element that they affect, such as classes, methods, variables, or statements. For example, "Long

Method" is a code smell that affects methods, while "Large Class" is a code smell that affects classes.

By type of issue: Code smells can be categorized based on the type of issue they indicate, such as design issues, readability issues, maintainability issues, or performance issues. For example, "Duplicate Code" is a code smell that indicates a maintainability issue, while "Speculative Generality" is a code smell that indicates a design issue.

By severity: Code smells can be categorized based on their severity, with some being more serious than others. This can be subjective and may depend on the code's context and the development team's goals.

By level of abstraction: Code smells can be categorized based on the level of abstraction at which they occur. For example, some code smells may occur at the class level, while others may occur at the method level or the statement level.

There is no "correct" way to categorize code smells, and different developers may have different opinions on how to categorize them. The important thing is to identify and address code smells in a way that helps improve the codebase's quality and maintainability. But we need somehow to navigate throw code smells. For that we will use primary classification suggested by Martin Fowler [1 p. 85].

3.1.1 Bloaters

Bloaters are a type of code smell that occur when code elements become too large or complex. Bloaters can make code difficult to understand and maintain, and can be a sign of deeper issues with the design or structure of the code [8].

Here are a few examples of code smells that are classified as bloaters [1]:

Long Method: A method that is too long and complex, making it difficult to understand and maintain. Programmers should break down methods into smaller ones based on the intention of the code, even if it results in longer method calls, as long as the method name clearly explains the purpose of the code.

Large Class: A class with too many responsibilities or too much code makes it difficult to understand and maintain.

Long Parameter List: A method with too many parameters, which can make it difficult to understand and use.

Primitive Obsession: The overuse of primitive data types, which can make code less expressive and more error-prone.

Data Clumps: Groups of data that are always used together, which could be refactored into a single class.

Bloaters can be particularly difficult to deal with because they often require significant refactoring to fix. However, addressing bloaters can significantly improve the quality and maintainability of the codebase [17].

3.1.2 Object-oriented abusers

Object-Orientation Abusers is a term that is sometimes used to describe code smells that indicate a misuse or abuse of object-oriented programming (OOP) concepts. OOP is a programming paradigm that is based on the concept of "objects," which are data structures that contain both data and behavior. OOP aims to provide a clear and modular structure for code, making it easier to understand, maintain, and reuse. However, it can be misused or abused in ways that lead to code smells [9].

Some examples of "Object-Orientation Abusers" code smells include [1]:

Refused Bequest: A subclass that does not use or override methods inherited from its superclass. This can indicate a lack of modularity and can make the subclass unnecessarily dependent on the superclass.

Alternative Classes with Different Interfaces: Multiple classes that could be combined into a single class with multiple interfaces. This can indicate a lack of code reuse and can make the codebase more complex.

Data Clumps: Groups of data that are always used together, which could potentially be refactored into a single class. This can make the code more difficult to read and maintain.

Incomplete Library Class: A class that is missing functionality that is expected in a library class. This can lead to confusion and may make the class challenging to use.

Data Class: A class that contains mostly data and very little behavior. This can make the class difficult to reuse and may indicate a lack of modularity.

"Object-Orientation Abusers" code smells can be difficult to fix because they often require significant codebase refactoring. However, addressing these code smells can significantly improve the quality and maintainability of the code [10].

3.1.3 Change Preventers

Change Preventers is a term that is used to describe code smells that make it difficult to change or modify the codebase [16].

Code that is difficult to change is generally less maintainable and more prone to errors, as it requires more effort to update or fix. Change Preventers code smells can occur for a variety of reasons, such as a lack of modularity, a lack of encapsulation, or a lack of abstraction [10].

Here are examples of code smells that are classified as Change Preventers:

- **Divergent Change**: Code that is changed in multiple, unrelated ways at the same time. This can make it difficult to understand and maintain the code [20].

- **Shotgun Surgery**: Code that requires many small changes in multiple, unrelated places in order to make a single, larger change. This can make the codebase more difficult to modify and more error-prone [1].

- **Parallel Inheritance Hierarchies**: Multiple inheritance hierarchies that are related, but not identical. This can make the codebase more difficult to understand and maintain [1].

3.1.4 Dispensables

Dispensables are a type of code smell that occur when code elements are unnecessary or redundant. Dispensables can make the codebase more difficult to understand and maintain, and can be a sign of deeper issues with the design or structure of the code [14].

Here are all the code smells that are classified as Dispensables:

18

- **Dead Code**: Code that is no longer used or needed and can be safely removed [14].

- **Commented Out Code**: Code that has been commented out and is no longer in use, but has not been fully removed [1].

- **Lazy class**: A class that was designed with a purpose, but now his some of his functions are not used [18].

- **Data class**: A class that contains only fields and no functions working on the this them [1].

- **Speculative Generality**: A functionality (group of classes) is created for future [1].

3.1.5 Couplers

Couplers are a type of code smell that occurs when code elements are too closely coupled or dependent on one another. High coupling can make the codebase more difficult to understand, maintain, and modify, as changes in one part of the code may have unintended consequences in other parts of the code [14].

Here are all the code smells that are classified as Couplers [1]:

Feature Envy: A method that accesses the data of another class more frequently than its own data. This can indicate that the method would be better placed in the other class.

Inappropriate Intimacy: Tight coupling between classes that should be more independent. This can make the code more difficult to understand and maintain.

Message Chain: A long chain of method calls, with each method calling the next. This can make the code more difficult to understand and maintain.

Middle Man: A class that serves as a middleman between two other classes, but adds no additional functionality. This can make the code more difficult to understand and maintain.

Incomplete Library Class: A class that is missing functionality that is expected in a library class. This can lead to confusion and may make the class difficult to use.

19

3.2 Refactoring techniques

Refactoring techniques are specific methods or approaches that can be used to improve code structure without changing its behavior. Many different refactoring techniques can be used, depending on the code's specific needs and the development team's goals [1 p. 85].

According to Fowler, we will classify techniques in the next order [1]:

Composing Methods: Techniques for improving the structure and organization of methods (9 techniques).

Moving Features Between Objects: Techniques for improving the organization of code by moving features between objects (8 techniques).

Organizing Data: Techniques for improving the organization and structure of data within a program (15 techniques).

Simplifying Conditional Expressions: Techniques for improving the readability and maintainability of conditional logic (8 techniques).

Simplifying Method Calls: Techniques for improving the readability and maintainability of method calls (14 techniques).

Dealing with Generalization: Techniques associated with moving functionality in the hierarchies of objects (12 techniques).

Due to the vast number of techniques available and the limited scope of this discussion, it is impossible to provide an in-depth description of each one, and thus, some of the techniques will be left out. However, the most important techniques will be thoroughly explained, ensuring that the reader gains a comprehensive understanding of the topic.

3.2.1 Composing Methods

Composing Methods is a refactoring technique that involves breaking down a large method into smaller, more focused methods. This technique aims to make the code more readable and maintainable by reducing the complexity of the method and making it easier to understand what the method is doing and how it works [15].

The key benefit of composing methods is that they help to make code more modular and less monolithic, which makes it easier to change, test, and reuse. When a method is broken down into smaller methods, it becomes more focused, which makes it easier to understand and reason about. Additionally, composing methods help to improve the cohesion of the code, which means that the code is more tightly focused on a specific task or responsibility [1 p. 89].

When composing methods, it is important to ensure that the new methods are clearly named and have a single, well-defined responsibility. This makes it clear what the method does, and makes it easy to understand how it fits into the larger codebase. Additionally, when composing methods, it is important to ensure that the new methods are properly encapsulated and have the appropriate visibility (e.g. public, protected, private) [1 p. 89].

Extract method

There are different techniques for composing methods, like extracting methods, inlining methods, or moving methods. Each one serves a different purpose and need a different approach. For example, extracting a method is to break a block of code into a new method with a name that reflects what the code is doing, while inlining a method is combining the code in the method being called with the code that calls it. Moving method is used to move a method to a class that is more semantically related to it [1 p. 90].

It's important to note that code-level refactoring techniques like composing methods should be done in addition of larger-scale techniques like code organization and design patterns, in order to have a clean, maintainable and easy to read codebase [1 p. 90].

The "Extract Method" technique is a refactoring technique that involves breaking down a large method into smaller, more focused methods by extracting a section of code from the larger method and placing it in a new, separate method. The goal of this technique is to make the code more readable and maintainable by reducing the complexity of the larger method and making it easier to understand what the code is doing and how it works [1 p. 90].

The process of extract method can be summarized in these steps [1 p. 90]:

- Identify a section of code that performs a single, well-defined task.

- Create a new method with a clear and descriptive name that reflects the task the extracted code performs.

- Move the selected code into the new method.

- Replace the original code with a call to the new method, passing any necessary variables as arguments.

- Make sure that the visibility of the extracted method is set according to the needs of the program and the extracted method only access to the variables it needs.

The key benefits of extract method technique include:

- Improving code readability by making it easier to understand the purpose of the code.

- Reducing code duplication by creating reusable methods.

- Improving code maintainability by making it easier to identify and fix bugs.

- Improving code flexibility by making it easier to add new features or make changes to the codebase.

Balancing between making the code readable and maintainable, and making it easy to navigate and understand is crucial while implementing the Extract Method technique, as it could create methods that are too small and trivial, commonly referred to as the "Method too small" anti-pattern. Such methods could lead to a proliferation of tiny, hard-to-understand methods, making the codebase more complex. Hence, developers should be mindful of the trade-offs involved while using this technique. [4, p. 176].

Let's consider an example:

Source code 1: Code before refactoring [1]:

```
void PrintOwing(double amount) {
    PrintBanner();

    //print details
    Console.WriteLine("name:" + _name);
    Console.WriteLine ("amount" + amount);
}
```

22

Source code 2: Code after refactoring [1]:

```csharp
void PrintOwing(double amount) {
    PrintBanner();
    PrintDetails(amount);
}

void PrintDetails (double amount) {
    Console.WriteLine ("name:" + _name);
    Console.WriteLine ("amount" + amount);
}
```

Other methods in the group:

- **Inline Method:** This technique involves replacing a method call with the method's body. This can be used to remove unnecessary method calls and simplify the code [1].

- **Extract Variable**: This technique involves taking an expression and assigning it to a new variable. The variable is then used in place of the original expression [14].

- **Inline Variable**: This technique involves replacing a variable with its value. This can be used to remove unnecessary variables and simplify the code [14].

- **Replace Temp with Query**: This technique involves replacing a temporary variable with a query method. This can be used to make the code more readable and understandable [1].

- **Replace Method with Method Object**: This technique involves replacing a method with an object that contains the method and its associated data. This can be used to make the code more readable and understandable [1].

- **Replace Conditional with Polymorphism**: This technique involves replacing a conditional statement with polymorphism. This can be used to make the code more readable and understandable [1].

- **Replace Assignments to Parameters**: This technique involves using a local variable instead of a parameter [14].

3.2.2 Moving Features between Objects

The Moving Features between Objects technique is an effective approach to refactoring code that can significantly enhance the structure and maintainability of software projects. By moving methods, properties, or fields from one class to another, developers can create a more organized and streamlined codebase that is easier to understand and modify. The technique aims to ensure that each class or object has a clear and distinct set of responsibilities, which can improve the overall functionality of the software and reduce the likelihood of errors or bugs. However, it is important to carefully consider the impact of moving features between objects, as it can also introduce new dependencies or increase the complexity of the codebase. As with all refactoring techniques, a thoughtful and systematic approach is required to achieve the best possible results. [1 p. 115].

Here are a few examples of techniques for moving features between objects [1]:

- **Move Method**: This technique involves moving a method from one class to another class where it better belongs, this is usually done when a method is more closely related to another class than the class it currently belongs to.

- **Move Field**: This technique involves moving a field from one class to another class where it better belongs, this is usually done when a field is more closely related to another class than the class it currently belongs to.

- **Extract Class**: This technique involves creating a new class and moving methods and fields from the current class to the new class. This can be used when a class has grown too large and has multiple responsibilities.

- **Inline Class**: This technique involves moving the fields and methods of a class to its parent class and then deleting the original class. This can be used when a class is no longer needed or is too small.

- **Hide Delegate**: The goal of the technique is to remove unnecessary dependencies between classes by hiding the delegate behind a simpler interface.

- **Remove middle man**. The "Remove Middle Man" aims to remove unnecessary intermediate classes or methods that add little or no value to the codebase and simply act as intermediaries between two other classes or methods. By removing these classes or methods, the codebase becomes simpler, more efficient, and easier to understand and maintain.

- **Introduce Foreign Method**. The "Introduce Foreign Method" is used when you need to add a new method to a class but cannot modify the class directly. Instead, you create a new method in a different class that takes an instance of the original class as a parameter and then calls the new method on that instance.

- **Introduce Local Extension** is a refactoring technique that allows you to add new functionality to a class without modifying it directly. It's similar to the "Introduce Foreign Method" technique, but instead of creating a new method in a different class, you create a new method in the same class as the original method, but in a local scope.

These techniques can help to improve the code design, organization, and readability. By moving functionality to the appropriate classes or namespaces, the code becomes more maintainable, and it's easier to understand how the code fits together. It should be emphasized that these techniques require careful consideration and analysis of each individual case to assess their impact on the overall codebase. [14].

3.2.3 Organizing Data

Organizing Data is a crucial refactoring technique that is used to improve the structure and organization of the data in a codebase. These techniques aim to make the data more understandable, manageable, and maintainable by improving the way it's represented, stored, and accessed [14].

The goal of these techniques is to simplify the codebase, reduce redundancy, and make the code more modular, which ultimately leads to better performance, maintainability, and extensibility. By applying these techniques, developers can also eliminate unnecessary duplication and reduce the likelihood of errors, which ultimately leads to better quality code [14].

Here are some examples of "Organizing Data" techniques [1]:

- **Replace Data Value with Object**: This technique involves replacing a simple data value, such as a string or integer, with a new object that encapsulates the value and its behavior. This can be used to add functionality to the data and make it more understandable.

- **Replace Array with Object**: This technique involves replacing an array with an object that can better represent the data. This can be used to add functionality to the data and make it more understandable.

- **Encapsulate Field**: This technique involves making a field private and adding getter and setter methods to access the field, thus encapsulating the data, this can be used to ensure that the data is accessed and modified in a controlled and predictable way.

- **Encapsulate Collection**: This technique involves encapsulating a collection in a new class, this can be used to add functionality to the data and make it more understandable.

- **Replace Type Code with State/Strategy**: This technique involves replacing a type code with a state or strategy object, this technique can be used to make the code more maintainable, flexible and to avoid using an enumeration.

- **Replace Subclass with Fields**: This technique involves replacing a subclass with fields, this can be used to simplify the codebase, improve performance and to make it more maintainable.

- **Replace Record with Data Class**: This technique involves replacing a record with a data class, this can be used to improve readability and maintainability, making the data more understandable and explicit.

- **Replace Magic Number with Symbolic Constant**: This technique involves replacing a magic number (a number with a special meaning) with a symbolic constant, this can be used to make the code more readable and maintainable by making the meaning of the number explicit.

26

These techniques can help to improve the code design, organization, and readability. By organizing the data in a better way, the code becomes more maintainable, and it's easier to understand how the data is represented and used in the application. It's important to note that these techniques should be used in combination with each other and tailored to each specific case, in order to achieve the desired results and improve the codebase [1 p. 138].

3.2.4 Simplifying Conditional Expressions

"Simplifying Conditional Expressions" is a fundamental category of refactoring techniques that software developers use to improve the readability and maintainability of conditional logic in the code. The main objective of this category of techniques is to simplify complex conditional statements that can be challenging to read and understand. These techniques enable developers to eliminate redundant code, reduce code duplication, and make the code more expressive and readable. [1 p. 192].

Here are a few examples of "Simplifying Conditional Expressions" techniques:

- **Consolidate Conditional Expression**: This technique combines several conditional expressions into a single, more readable expression [1].
- **Consolidate Duplicate Conditional Fragments**: This technique consolidates the duplicate parts of a conditional expression into a single, reusable fragment [19].
- **Replace Nested Conditional with Guard Clauses**: This technique involves replacing nested conditional statements with a series of guard clauses [19].
- **Decompose Conditional**: This technique involves breaking a complex conditional expression into several simpler expressions [1].
- **Replace Conditional with Polymorphism**: This technique involves replacing a conditional expression that depends on the type of an object with polymorphism.

It's important to keep in mind that these techniques should be applied when it makes sense and can help improve the code readability and maintainability, it's only sometimes the case that applying these techniques will lead to a better code [1 p. 192].

3.2.5 Simplifying Method Calls

"Simplifying Method Calls" is a category of refactoring techniques that are focused on improving the readability and maintainability of method calls in the code [5].

27

Simplifying method calls can also make the code more efficient by reducing the number of parameters passed to a method, and thus reducing the amount of memory and processing power required to execute the method. This can lead to better performance, faster execution times, and an overall better user experience for users of the software. [1 p. 220].

Here are a few examples of "Simplifying Method Calls" techniques [1]:

- **Replace Method with Method Object**: This technique involves replacing a method with a separate class that holds the method's local variables.
- **Introduce Parameter Object**: This technique replaces multiple parameters with a single object.
- **Preserve Whole Object**: This technique involves passing an entire object as a method parameter instead of just selected fields of the object.
- **Replace Parameter with Method**: This technique involves replacing a method parameter with a call to a method on the same object.
- **Replace Parameter with Explicit Methods**: This technique involves replacing a parameter with explicit methods on the object being passed.

It's important to keep in mind that these techniques should be applied when it makes sense and it can help to improve the code readability and maintainability, it's not always the case that applying these techniques will lead to a better code [1 p. 220].

3.2.6 Dealing with Generalization

"Dealing with Generalization" is a category of refactoring techniques that are focused on improving the design of the code by making it more generic, reusable, and extensible [1 p. 259].

Here are a few examples of "Dealing with Generalization" techniques [1]:

Extract Superclass: This technique involves extracting common functionality from two or more subclasses into a new superclass.

Pull Up Field: This technique involves moving a field from a subclass to a superclass.

Pull Up Method: This technique involves moving a method from a subclass to a superclass.

Push Down Method: This technique involves moving a method from a superclass to a subclass.

Extract Interface: This technique involves extracting an interface from a class to define a common set of methods that must be implemented by all classes that implement the interface.

Form Template Method: This technique involves pulling the skeleton of an algorithm into a superclass and let subclasses fill in the details.

It's important to keep in mind that these techniques should be applied when it makes sense and it can help to improve the code design and maintainability, it's not always the case that applying these techniques will lead to a better code [14].

3.3 Testing

The essential precondition for refactoring is having solid tests. You still need tests even if you are fortunate enough to have a tool that can perform automation refactorings. It will be a long time before all possible refactorings can be automated in a refactoring tool [1 p.73].

Many different types of software tests can be used to ensure the quality and correctness of a software system [1 p.73].

Here are a few examples of common types of software tests:

- **Unit tests**: Unit tests are automated tests that are written to test the individual units or components of a software system. They are typically focused on testing a small, specific piece of functionality [1].

- **Integration tests**: Integration tests are automated tests that are written to test the interaction between different units or components of a software system. They are typically focused on testing how the different parts of the system work together [11].

- **Functional tests**: Functional tests are automated tests written to test a software system's overall functionality. They are typically focused on testing the system as a whole rather than individual units or components [1].

29

- **Performance tests**: Performance tests are automated tests written to test a software system's performance. They are typically focused on measuring the system's performance under different conditions, such as high load or low memory [11].

- **User acceptance tests:** User acceptance tests (UATs) are tests that end users or stakeholders conduct to ensure that the software meets their needs and expectations [11].

Many other types of software tests can be used, depending on the software system's specific needs and the development team's goals. We will focus on the unit tests.

Unit tests are a fundamental component of software testing and are designed to ensure that each individual unit or component of a software system is functioning as expected. They are automated tests that are typically written by developers and executed during the development process, before the software is released to users [15].

Unit tests are critical for maintaining software quality and are an essential part of a comprehensive testing strategy. By isolating and testing individual units of code, developers can catch and fix errors early in the development process, which can save significant time and resources later on. Unit tests also provide a safety net for developers, allowing them to make changes to the codebase with confidence, knowing that any changes will be quickly detected if they introduce bugs or regressions [15].

In addition, unit tests can help to improve code design and maintainability, as they encourage developers to write modular, loosely-coupled code that is easier to test and maintain [15].

There are several benefits to writing unit tests [12]:

- **Improved code quality**: Unit tests can help to ensure that the code is correct and meets the requirements. This can help prevent bugs and defects and improve the codebase's overall quality.

- **Improved maintainability**: Unit tests can help to ensure that code changes do not break existing functionality. This can make the codebase more maintainable over time, as it is easier to make changes without introducing new bugs.

- **Improved confidence**: By having a suite of unit tests, developers can have more confidence in the codebase and can be more productive.

- **Improved documentation**: Unit tests can serve as documentation for the code, as they provide examples of how it is intended to be used.

Writing unit tests requires some additional effort upfront, but it can pay off in the long run by improving the quality and maintainability of the code [1 p. 74].

Martin Fowler provides practical advice for writing tests to ensure that code is correct and of high quality. Here are a few key points about writing tests according to Fowler [1]:

- **Tests should be automated**: Automated tests are easier to run and maintain than manual tests, and they can be run more frequently to ensure that the code is still correct.

- **Tests should be written before the code**: Test-driven development (TDD) involves writing tests before writing the code itself. This helps ensure that the code meets the requirements and is correct.

- **Tests should be simple and concise**: Tests should be easy to understand and should only test a single aspect of the code.

- **Tests should be repeatable**: Tests should always produce the same result, regardless of when or where they are run.

- **Tests should be independent**: Tests should not depend on the system's state or the results of other tests.

Fowler's book covers many other topics in addition to these and is a valuable resource for anyone looking to improve their skills in writing tests.

3.4 Code inspections

A code inspection is a specific kind of review that is highly effective in detecting defects and is relatively economical compared to testing. [2 p. 485].

Static analysis involves examining the source code of a program to identify potential defects, security vulnerabilities, and other issues that could impact the program's performance

or reliability. This process is often used by developers and integrated development environments (IDEs) to improve code quality, readability, and maintainability. One effective technique for identifying defects during static analysis is through code inspections. By analyzing code early in the development process, code inspections can help identify problems that might otherwise be missed by other testing methods, such as unit tests or integration tests. This can ultimately save time and resources by catching defects before they have a chance to propagate through the system and become more difficult to fix. Furthermore, code inspections can also help ensure that the code meets the requirements, is consistent with best practices, and is easy to understand and maintain over time [13].

A team can conduct code inspections. This helps to ensure that multiple people with different perspectives and expertise review the code. Code inspections should also follow a specific process. This includes preparing for the inspection, conducting the inspection, and following up after the inspection. Code inspections should focus on the code, not the person who wrote it. They also should be objective and focus on the code's quality rather than the person who wrote it [2]

3.5 Rider

Rider is a software development tool that runs the Rider IDE process as a thin layer on top of ReSharper. Rider IDE provides the user interface, including an editor and text caret in source files, while some languages like JavaScript and TypeScript have a full implementation in Rider. For other languages like C#, VB.NET, and F#, the front-end relies on the ReSharper back-end process for language information. The front-end and back-end communicate with each other, exchanging information for features like code completion and code inspections. The Rider IDE front-end provides editing and other infrastructure for some languages, while relying on the ReSharper back-end for information when needed [7].

In the developing we will mostly focus on back-end process.

3.5.1 Rider IDE

JetBrains Rider IDE is a cross-platform integrated development environment (IDE) for .NET and web development. It is built on the IntelliJ Platform and uses the same codebase as

IntelliJ IDEA, JetBrains' flagship Java IDE. However, Rider is optimized for C#, VB.NET, and F# development and provides a feature set tailored to the needs of .NET developers [6].

Rider supports the development of various types of applications, including Windows desktop applications, ASP.NET and ASP.NET Core web applications, and Unity game development. It provides features such as advanced code editing and navigation, debugging, profiling, and unit testing. It also offers built-in support for popular frameworks and technologies, such as .NET Framework, .NET Core, and Mono, as well as web technologies like JavaScript, TypeScript, HTML, and CSS. Additionally, Rider offers tight integration with version control systems, issue trackers, and other tools and services that are commonly used in .NET development [6].

3.5.2 Rider SDK

The Rider SDK (Software Development Kit) is a set of tools and libraries that allows developers to create custom integrations and plugins for JetBrains Rider, a cross-platform integrated development environment (IDE) for .NET and web development. The SDK provides access to the internal structure and functionality of the IDE, such as its code editor, language services, debugging tools, and project management features. Developers can use the SDK to create custom code completion, refactoring, and navigation tools, as well as integrate with other tools and services, such as version control systems, issue trackers, and continuous integration systems [3].

In the context of the ReSharper Platform SDK, refactoring refers to the process of improving the structure and organization of existing code without changing its external behavior. ReSharper SDK provides a set of APIs and tools for performing various refactoring tasks on code written in different programming languages [3].

Some of the refactoring features provided by the ReSharper SDK include:

- Renaming variables, methods, and classes

- Extracting methods, classes, and interfaces

- Inlining variables and methods

- Moving types and members between namespaces

- Changing method signatures

- Introducing and inlining local variables

- Converting loops to LINQ expressions

ReSharper SDK also provides a wide range of code generation features that help developers to quickly create new code [3].

These features include:

- Generating constructors, properties, and methods

- Implementing interfaces and abstract classes

- Generating equality members and overrides

- Generating event handlers

- Generating comparison methods

- Generating hash codes

- Generating string representations

ReSharper SDK also provides a powerful navigation and search feature that enables developers to quickly find and navigate to specific code elements, such as classes, methods, and properties [3].

3.5.3 Rider Refactorings

Refactorings are undoubtedly one of the most powerful and essential features of ReSharper, but they are also the most complex. Triggered by a variety of events, from a simple edit or deliberate invocation, refactorings can involve all sorts of operations. Some are relatively straightforward, involving only local code changes, while others may involve the entire project, such as renaming a file, or changing multiple files in many projects [3].

At the simplest level, a refactoring consists of three main components: the workflow, the workflow provider, and the refactoring itself. The workflow is a series of steps or actions that transform the code in a specific way, such as extracting a method, renaming a variable, or changing the signature of a method. The workflow provider is responsible for creating and managing the workflow, while the refactoring is the high-level operation that combines both the workflow and the workflow provider [3].

To execute a refactoring, ReSharper typically follows a standard process. First, it analyzes the code to determine the context in which the refactoring should be applied. This analysis can involve examining the syntax tree of the code, performing semantic analysis, and even analyzing the version control system to ensure that all changes can be properly tracked. Once the context has been determined, ReSharper creates an instance of the workflow provider and passes it the necessary information. Finally, the workflow provider executes the workflow, which makes the desired changes to the codebase [3].

The workflow is a class that manages the steps involved in performing the refactoring. Most refactorings are "driven," meaning that the user has to navigate through menus that present conflicts or other considerations related to the refactoring [3].

Typically, a workflow is implemented as a class that inherits from DrivenRefactoringWorkflow. This class has various members that perform specific functions related to the refactoring. The workflow is executed in three stages: PreExecute, Execute, and PostExecute. Each of these methods, as well as others such as IsAvailable, returns a boolean value indicating whether the current step is successful. If any of these methods return false, the entire transaction is rolled back, and no refactoring takes place [3].

The workflow provider is a critical component in the implementation of refactorings in ReSharper. It is responsible for providing workflows that ReSharper can consume and use during the refactoring process. The workflow provider is implemented as a class that satisfies two requirements: it must implement the IRefactoringWorkflowProvider interface and be decorated with the [RefactoringWorkflowProvider] attribute [3].

The IRefactoringWorkflowProvider interface defines a single method that the workflow provider class must implement: CreateWorkflow(). This method returns an IEnumerable of one

or more workflows that the provider wishes to offer to ReSharper. This method is called by ReSharper at various times, such as when a user initiates a refactoring or when the provider needs to refresh its list of workflows [3].

The workflow provider is a crucial component in the implementation of refactorings in ReSharper, as it provides the workflows that ReSharper uses to perform the refactoring. By implementing the IRefactoringWorkflowProvider interface and creating a class that derives from DrivenRefactoringWorkflow, developers can create new refactorings that are fully integrated with ReSharper [3].

Overall, refactorings are a fundamental tool for improving code quality, maintainability, and readability. They provide a powerful way to make complex changes to the codebase quickly and efficiently, while also reducing the risk of introducing errors or breaking existing functionality [3].

3.5.4 Rider PSI

Another important part of the Rider SDK is PSI model. Rider PSI, or the Platform Specific Implementation, is a feature of the JetBrains Rider development environment. PSI is the underlying API that JetBrains Rider uses to provide support for different programming languages and technologies, such as C#, Java, and TypeScript, among others. The PSI provides a tree structure of the code, where elements such as classes, methods, and properties, are represented as nodes in the tree. Developers can use the PSI to navigate the codebase, find specific code elements, and make changes to the code. The PSI also provides a rich set of functionality for working with the code, such as code completion, navigation, refactoring, and code generation. Developers can use the PSI to perform various code-related tasks, such as renaming variables or methods, extracting methods, and creating new classes or interfaces. In summary, the PSI is an essential part of the JetBrains Rider development environment, it serves as the backend of the JetBrains Rider providing functionality to the front-end, allowing developers to easily navigate, understand, and change the codebase, with a rich set of functionality to enhance the development experience when working with syntax tree [3].

The PSI consists from:

- **Abstract syntax trees (AST)** – detailed description can be found in the 3.5.4 section.

- **Declared elements** - serve as the gateway to the semantic model view of a codebase. They are essential components for any item that has a declaration, ensuring that there is a declared element to correspond with it. The IDeclaredElement interface defines a base type for language-agnostic declared elements and establishes a cross-language method of referring to declared elements. For the Common Language Runtime (CLR), the ITypeElement interface offers a comprehensive semantic overview of a type, encompassing members, base types, and links to the underlying AST declarations and constructs. However, declared elements are not solely limited to types. In fact, ReSharper has extensive support for a wide range of items, including CSS classes, HTML elements, colors, and even file system paths [3].

- **References** - provide a powerful linkage between an AST node and a declared element, allowing for all type usages to reference back to the declared element of the type. This mechanism is extensively used in ReSharper, with Ctrl+Click navigation simply following a reference, find usages performing a reverse lookup, and renaming being implemented by renaming both the target and any incoming references. The use of references ensures that changes made to a declared element are reflected throughout the entire codebase, allowing for easy and efficient code maintenance [3].

- **Type systems** - Language-specific declared element interfaces utilize type systems to properly model type usage beyond type declarations. While declared elements can provide a semantic view of a type, they cannot fully represent all type usages, such as arrays, pointers, or generic types. Typically, a type system is employed to capture this information, often based on a declared element and any necessary substitutions required to satisfy a generic type [3].

- **Caching** – It offers a framework for persistent caching on a per-file basis, enabling the caching of essential information about a file or serving as a key-value store to cache data that is not tied to a specific file, such as unit test results. Additionally, the PSI provides

37

support for in-memory, temporary caches that can be invalidated when the codebase undergoes changes [3].

ReSharper's ASTs are primarily based on the contents of a file, but they can also work with secondary and injected ASTs, known as secondary and injected PSIs, respectively. A secondary PSI is a PSI syntax tree constructed from a secondary file, which is an in-memory "code-behind" file generated by ReSharper from the primary code file [3].

The secondary PSI tree is mapped to the original file, allowing ReSharper to handle files such as .aspx and Razor files. In addition to maintaining a primary PSI tree for the main file, ReSharper generates an in-memory representation of the file that is produced when the .aspx or Razor template is compiled. This generated file is referred to as a code-behind file, but it is not an ASP.NET code-behind file [3].

A second PSI tree is created for the generated C# file, and the areas of C# in the .aspx or Razor file are mapped to the equivalent areas in the generated file. Since the generated file has full context of the class it belongs to, code completion and navigation are possible by mapping between the areas of the files. On the other hand, an injected PSI serves a different purpose. Instead of providing extra context for a whole file or for isolated content within a file, it creates a new AST by parsing the contents of a single node of a host AST [3].

For example, ReSharper 9.0 introduced support for regular expressions by creating an AST for the regular expression represented by a string literal in a C# file. Injected PSIs are intended to handle the "embedded DSLs" of languages written inside a string literal of another language, such as regular expressions, SQL, or even Angular JS expressions [3].

3.5.5 Rider Abstract syntax trees

An abstract syntax tree (AST) is a hierarchical data structure that represents the contents of a file. In this structure, each node denotes a particular construct in the source code, such as a method declaration, a comment, whitespace, or a string literal. For instance, a method declaration node may have child nodes representing accessibility, return type, method name, parameters, and the method body, which itself may have child nodes representing method instructions and nested code blocks [3].

To create an AST, the Program Structure Interface (PSI) defines common base interfaces that are shared across all languages supported by ReSharper. These interfaces provide a common base for sharing information across files and languages. ReSharper provides utility methods for traversing and manipulating the tree, such as adding or deleting nodes, and moving nodes around the tree. These methods help to analyze and modify the code structure efficiently [3].

An important feature of ReSharper is its use of abstract syntax trees (ASTs), which are data structures that represent the contents of a file as a hierarchical tree of nodes, where each node corresponds to a construct in the source code. The tree structure allows for efficient traversal and manipulation of code, making it a valuable tool for automated code analysis and refactoring [3].

Each node in the AST corresponds to an ITreeNode interface, while the root file node corresponds to the IFile interface. As a file is parsed, an AST is constructed and directly mapped to the underlying text buffer. The PSI is responsible for ensuring that both the tree and text buffer remain synchronized. This means that any modifications made to the tree are reflected in the text buffer, and any changes made to the text buffer are reflected in the tree[3].

Additionally, ReSharper is designed to be highly efficient, only parsing the code blocks that have changed rather than parsing the entire file. For example, when a C# method body is modified, only the method body is re-parsed, and not the entire containing file. This helps to ensure that ReSharper remains responsive and doesn't introduce unnecessary overhead when working with large codebases [3].

3.5.6 Rider actions

ReSharper offers several options for modifying or correcting your code, including a pop-up menu that appears on the left-hand side of the screen. This menu, which can be accessed with the Alt+Enter shortcut, contains a variety of menu items that are provided by plugin writers. There are two distinct ReSharper constructs that can supply items to appear in this menu [3]:

- Context actions are possible code modifications that may be applicable at a specific point in your code. They offer a way to improve your code without making drastic changes. Context action items are displayed with pencil icons [3].

- Quick-fixes, on the other hand, are modifications that appear in association with a particular highlighting, such as an error or warning. They provide a way to correct a specific problem in your code. Quick-fix items are displayed with red or yellow light bulb icons, with red bulbs fixing errors and yellow bulbs fixing warnings, suggestions, and hints [3].

In both cases, a context action or quick-fix can provide one or more menu entries for the user to act upon. Starting from ReSharper 7, these entries can be hierarchical, allowing for nested or submenu items in addition to top-level ones [3].

4 Practical part

4.1 Motivation

Our approach will involve presenting a problem, followed by an analysis of potential solutions. Specifically, we will focus on the code smell known as "Long Method," which was discussed in the section 3.1.1 It's important to note that there is no universally accepted standard for determining what constitutes a "long" method. Instead, this can vary depending on the specific needs and preferences of different teams, as well as the particular context in which the code is being developed. Nonetheless, for the purposes of our discussion, we will assume that any method containing more than 20 instructions will be considered as "long," in order to provide a clear and concrete threshold for analysis.

Consider the next code fragment:

Source code 3: Long method before refactoring.

```csharp
public static void GetSquareRoot() {
// 6 statements
string ans = "";
double a = Convert.ToDouble(Console.ReadLine());
double b = Convert.ToDouble(Console.ReadLine());
double c = Convert.ToDouble(Console.ReadLine());
double identifier = b * b - (4 * a * c);
double root1, root2;

// 6 statements
if (identifier > 0)
{
    root1 = (-b+(Math.Sqrt(identifier)/(2*a)));
    root2 = (-b - (Math.Sqrt(identifier) / (2 * a)));
    string r1 = Convert.ToString(root1);
    string r2 = Convert.ToString(root2);
    ans = "Root1 =" + r1 + "Root2 = " + r2;
}

// 6 statements
if (identifier < 0)
{
    double Real = (-b / (2 * a));
    double Complex = ((Math.Sqrt((identifier*(-1.00))) / (2 * a)));
    string SReal = Convert.ToString(Real);
```

41

```
    string SComplex = Convert.ToString(Complex);
    ans = "Roots = " + SReal + "+/-" + SComplex + "i";
}

// 4 statements
if (identifier == 0)
{
    root1 = (-b / (2 * a));
    string Root = Convert.ToString(root1);
    ans = "Repeated roots : " + Root;
}
// 1 statement
Console.WriteLine(ans);
}
```

The following source code demonstrates a double square equation solving algorithm. It has 23 statements, that exceeds our border value. Let's apply "Extract method" to see how it will improve the readability of the code. Good candidates for the technique are constructions like "if", "for", "switch" or any other nesting constructions. We will use "Extract method" for all three "if" statements by clicking appropriate button that appears:

Figure 1: Refactoring context menu
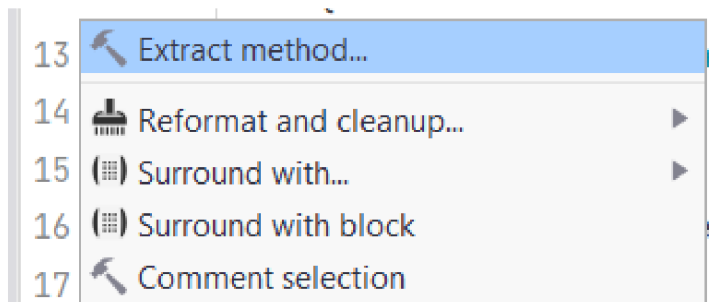
After that the Extract method dialog will be opened that allows us to perform additional manipulations.

We need to specify new method name, optionally set up parameters for our function and finally press 'Next' button to finish the refactoring. The result for all three extractions will be the next:

Source code 4: Long method after refactoring

```csharp
public static void GetSquareRoot()
{
    string ans = "";
    double a = Convert.ToDouble(Console.ReadLine());
    double b = Convert.ToDouble(Console.ReadLine());
    double c = Convert.ToDouble(Console.ReadLine());
    double identifier = b * b - (4 * a * c);

    if (identifier > 0)
        ans = GetRegularAnswer(b, identifier, a);
    if (identifier < 0)
        ans = GetComplexAnswer(b, a, identifier);
    if (identifier == 0)
        ans = GetZeroCaseAnswer(b, a);

    Console.WriteLine(ans);
}

private static string GetZeroCaseAnswer(double b, double a)
{
    double root1;
    string ans;
    root1 = (-b / (2 * a));
    string Root = Convert.ToString(root1);
    ans = "Repeated roots : " + Root;
    return ans;
}

private static string GetComplexAnswer(double b, double a, double
identifier)
{
    string ans;
    double Real = (-b / (2 * a));
    double Complex = ((Math.Sqrt((identifier * (-1.00))) / (2 * a)));
    string SReal = Convert.ToString(Real);
    string SComplex = Convert.ToString(Complex);
    ans = "Roots = " + SReal + "+/-" + SComplex + "i";
    return ans;
```

```
}
private static string GetRegularAnswer(double b, double identifier,
double a)
{
    double root1;
    double root2;
    string ans;
    root1 = (-b + (Math.Sqrt(identifier) / (2 * a)));
    root2 = (-b - (Math.Sqrt(identifier) / (2 * a)));
    string r1 = Convert.ToString(root1);
    string r2 = Convert.ToString(root2);
    return "Root1 =" + r1 + "Root2 = " + r2;
}
```

Now we can see that our method contains only 12 statements instead of 23. We can move even further and apply some techniques, but let stop on this. So, our goal is to develop an extension for Rider that will suggest such fixes and apply them automatically with ability to configure additional parameters like threshold value for statements count.

4.2 Analysis

4.2.1 Goals

The topic of this diploma project is to investigate the potential benefits of adding a code smells detection extension to the Rider Integrated Development Environment (IDE). The aim is to enhance Rider's existing refactoring capabilities by identifying problematic code patterns, commonly referred to as code smells, and suggesting refactoring options to address them.

The work will start with a comprehensive review of existing code smells detection tools and the various code smells that they identify. Based on this review, the project will then focus on developing a code smells detection extension for Rider IDE. The extension will be designed to identify common code smells and provide suggestions for refactoring.

The extension will be evaluated through a series of experiments, where it will be used to analyze software parts. The results of these experiments will be compared with other existing code smells detection tools to determine the effectiveness of the extension in detecting code smells.

In addition to the technical aspect of developing the code smells detection extension, this project will also focus on the user experience of using such a tool within the Rider IDE. The goal is to make the code smells detection and refactoring process as seamless and intuitive as possible, ensuring that the extension is accessible and useful to a wide range of developers.

In conclusion, the aim of this diploma project is to contribute to the development of better code refactoring tools, by exploring the potential benefits of adding code smells detection to Rider IDE. The project will explore the technical aspects of developing such an extension, as well as its practical implications for developers. The results of this project will contribute to the ongoing efforts to improve software development practices, and help make software development more efficient and effective.

4.2.2 Requirements

For simplicity we will arrange requirements for successful implementation:

- **Integration with Rider IDE**: The extension should seamlessly integrate with the Rider IDE and be easily accessible to users through the user interface. The criterion will be displaying the name of the extension in the plugin list of Rider.

- **Code smells detection**: The extension should be able to detect at least two code smells in the source code. Can be tested on the long method and long class examples. Should show a warning if method contains more than 4 lines.

- **Refactoring Support**: The extension should provide support for commonly used refactoring techniques, such as extracting methods.

- **User-Friendly Interface**: The extension should have a user-friendly interface that allows users to easily use the code smells detection and refactoring features.

- **Compatibility**: The extension should be compatible with a variety of types of projects, such as console applications, web applications, and Windows Forms applications.

- **Documentation**: The extension should be well-documented and include clear and concise instructions for users on how to use the code smells detection and refactoring features.

4.3 Design

At present, the Rider documentation that is available is not particularly informative, as it is still undergoing development and subject to frequent changes. As a result, a significant amount of the information that is currently available has been obtained through a combination of reverse engineering techniques and consulting with the developers of Rider. While this process can be time-consuming and require a certain level of expertise, it is often necessary in order to gain a more comprehensive understanding of how the program works and how best to utilize its various features and capabilities.

Additionally, given the fluid nature of the development process, it is important to remain vigilant and stay up-to-date on any new changes or updates that may be released in order to ensure that you are making the most of the program and taking advantage of all of its available functionalities.

To build an extension, we will concentrate on the following most important entities:

- **Problem analyzers** - a group of classes/interfaces that were primarily designed to determine whether the code has problems or not.

- **Highlightings** - represents a group of classes and interfaces that our realization should implement to provide a proper way to highlight problem ranges in the source code. Problem analyzers create the highlightings.

- **QuickFixes** - represents a group of entities that aim to provide a sufficent algorithm to fix a problem that was determined by the problem analyzers.

- **Registrators** - connects QuickFixes to appropriate highlightings. Used only on the plugin startup.

Let's consider an activity diagram demonstrating the algorithm for problems detection.

Figure 2: Problem detection activity diagram
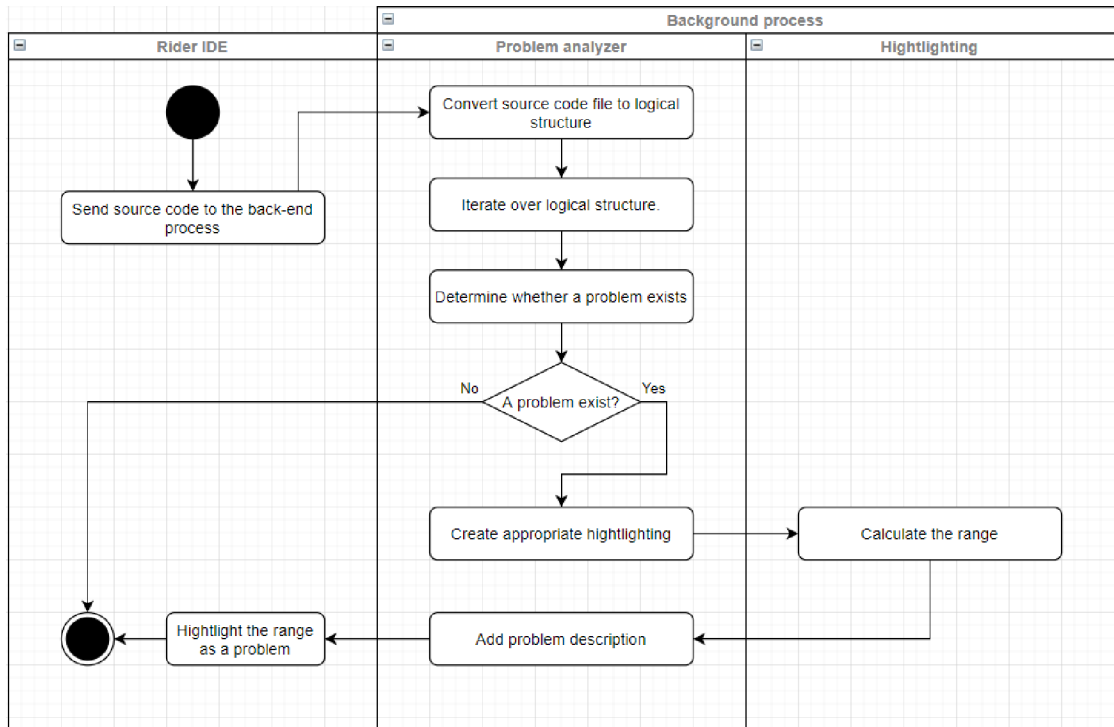
The problem detection process starts after the project is loaded or source code changes are made in the Rider IDE. It triggers the back-end rider process by calling it via custom protocol. The back-end process starts source code analysis by converting the source code to a logical structure and iteratively passing elements to existed problem analyzers. The purpose of which to detect a problem and add appropriate highlighting. After that a highlighting will calculate the range of problem. Problem analyzer can also add problem description and notify Rider IDE about problem detection complete. If the Rider IDE receives problems, they will be immediately displayed on the screen.

The process of applying a quick fix can be described on the following diagram

Figure 3: Quick fix applying activity diagram.
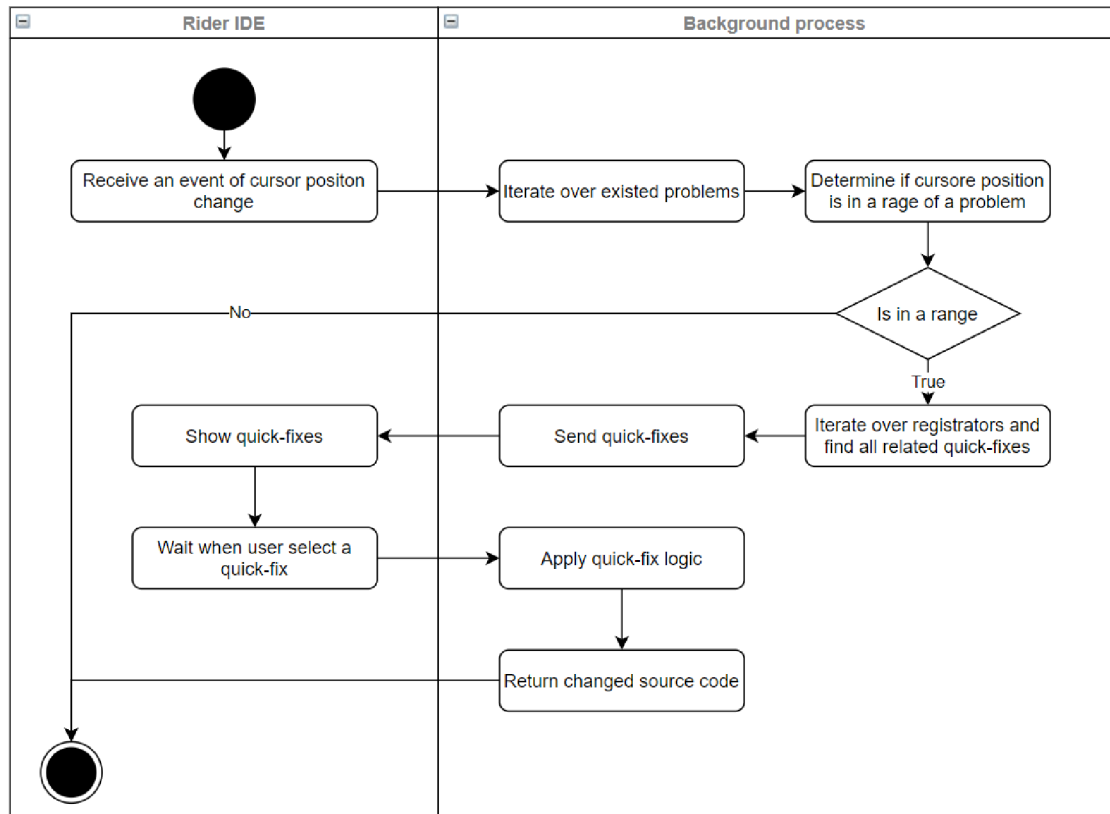
The process started by receiving a cursor position change. From the previous step, we still hold the problems in memory, which allows us to send them to find appropriate quick fixes in Quick fix Registrators. Then we can show all possible quick fixes for the selected problem and wait for the user to choose. From that point, the user can ignore and continue to edit the code. In such cases, the problem still will be displayed as not resolved. Otherwise, the user can select a quick fix and press the button to perform it. This action will trigger a refactoring logic.

4.3.1 Problem analyzers

Figure 4: ElementProblemAnalyzer<T> class diagram
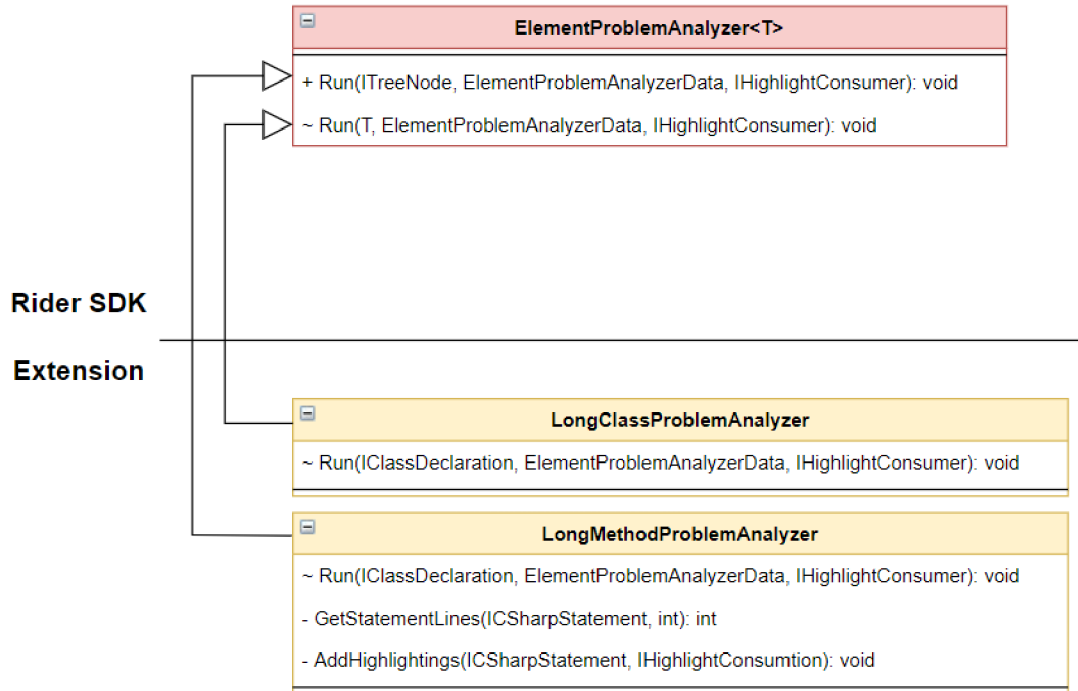
A problem analyzer is usually represented as a class that inherits ElementProblemAnalyzer<T>, where T is a type of a specific entity class such as IMethodDeclaration. The ElementProblemAnalyzer<T> is an abstract class with one single method to implement

Source code 5: ElementProblemAnalyzer<T>

```
public abstract class ElementProblemAnalyzer<T>
{
    protected abstract void Run(
    [NotNull] T element,
    [NotNull] ElementProblemAnalyzerData data,
    [NotNull] IHighlightingConsumer consumer);
}
```

The "Run" method is used when the problem analysis of an entity is required.

49

4.3.2 Highlightings

Highlightings are represented by IHightlighting interface that has the next form:

Figure 5: IHightlighting class diagram
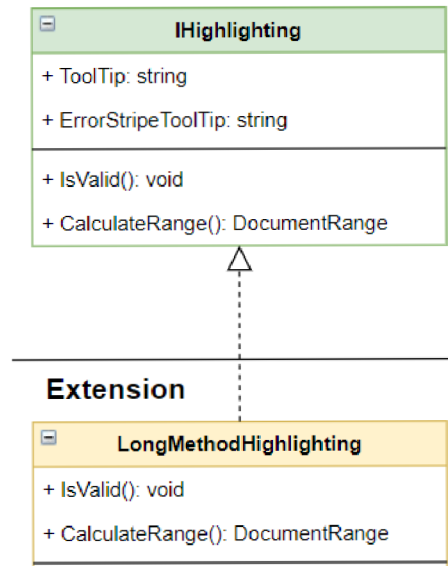
Any highlighting class should implement IHighlighting interface in order to be detected by Rider as a hightlighting logic for the particular set of problems.

Source code 6: IHightlighting interface

```csharp
public interface IHighlighting
{
  string ToolTip { get; }
  string ErrorStripeToolTip { get; }
  bool IsValid();
  DocumentRange CalculateRange();
}
```

The ToolTip and ErrorStripeToolTip are not relevant for us.

4.3.3 QuickFixes

Quick fixes can be described in the next class diagram:

Figure 6: Quick fixes class diagram.
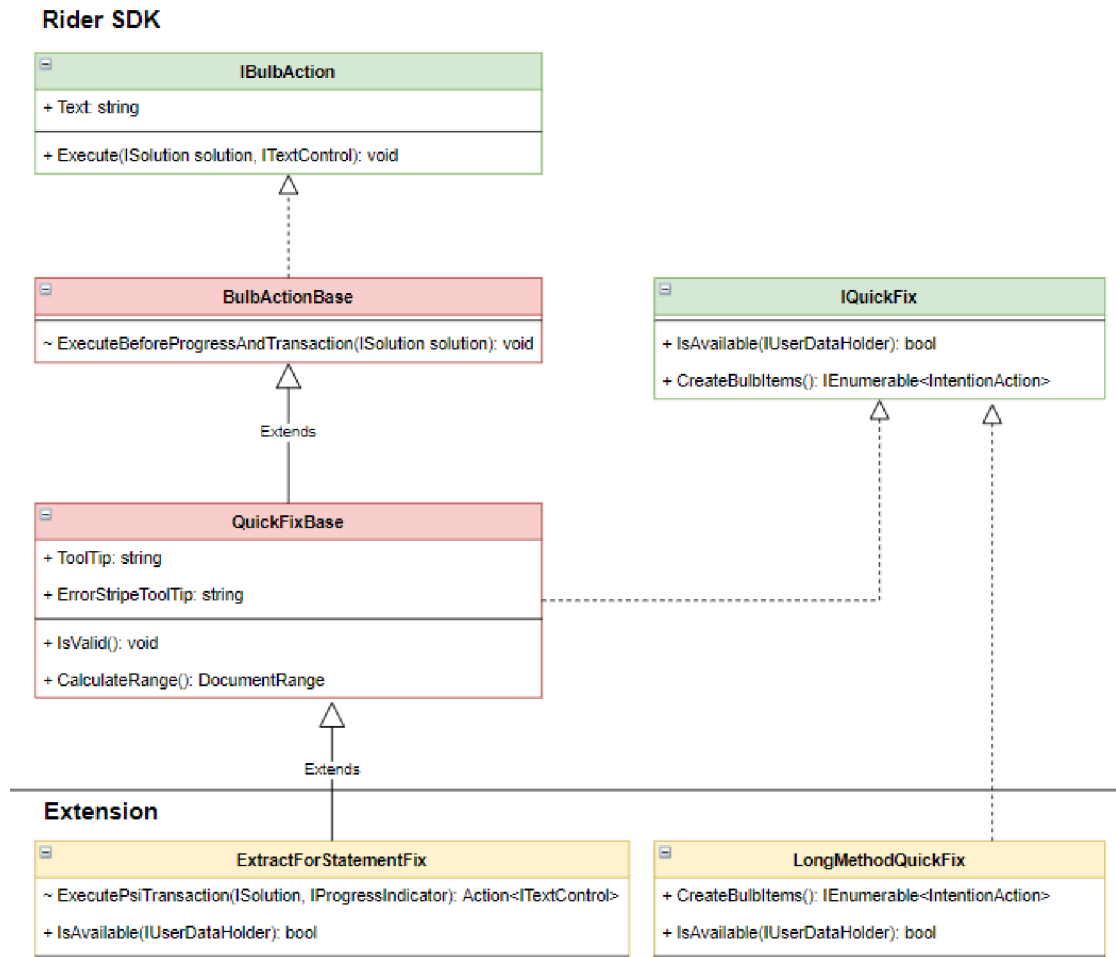
A quick fix in our term is a class that extends/implementsQF1 any of the classes above. Here we should distinguish different types of quick fixes:

- **IBulbAction** – is an interface that are used by RiderIDE to show it in the context menu. Usually can be seen as an icon and short text. Also specified a Execute method signature – action that should be executed on pressing the item in the context menu.

- **BulbActionBase** – an abstract class that implements IBulbAction and simplifies IBulbAction implementation because it contains a few already implemented helpful features. It is preferred to use this class instead of implementing IBulbAction.

- **IQuickFix** – an interface that provide a contract for simplest code fix.

- **QuickFixBase** – an abstract class that merges functionality of bulb actions and quick fixed into single entity. Contains most convenient way to manipulate the project and separate files.

We should mention that QuickFix depends on BulbAction and not reverse, because BulbAction can be used not only for quick fixes, but showing information or code navigation. To determine which class to use as basic we need to understand what we need to implement. In most scenarios we should prefer QuickFixBase as it leaves us more robust tools to create our own fixes, but it has lack of flexibility.

On the diagram above we showed how two quick fixes can be related to existed Rider SDK architecture.

Usually quick fixes are classes that inherit QuickFixBase abstract class.

Source code 7: QuickFixBase

```
public interface QuickFixBase
{
  IEnumerable<IntentionAction> CreateBulbItems();
  bool IsAvailable();
  void Execute(ISolution solution, ITextControl textControl);
}
```

The "CreateBulbItems" – creates context menu items with bulb icon and fix description that should be displayed in Rider IDE.

The "Execute" method contains an actual logic of changing something. It has two parameters:

- ISolution solution – project that is currently edited.

- ITextControl – object that contains information about edited document, cursor position and so on.

4.3.4 Regisrtators

A registrator is a class that implement IQuickFixeProvider. It contains single method Register(IQuickFixesRegistrar) that associates highlightings with quick fixes.

Figure 7: Registrators class diagram.
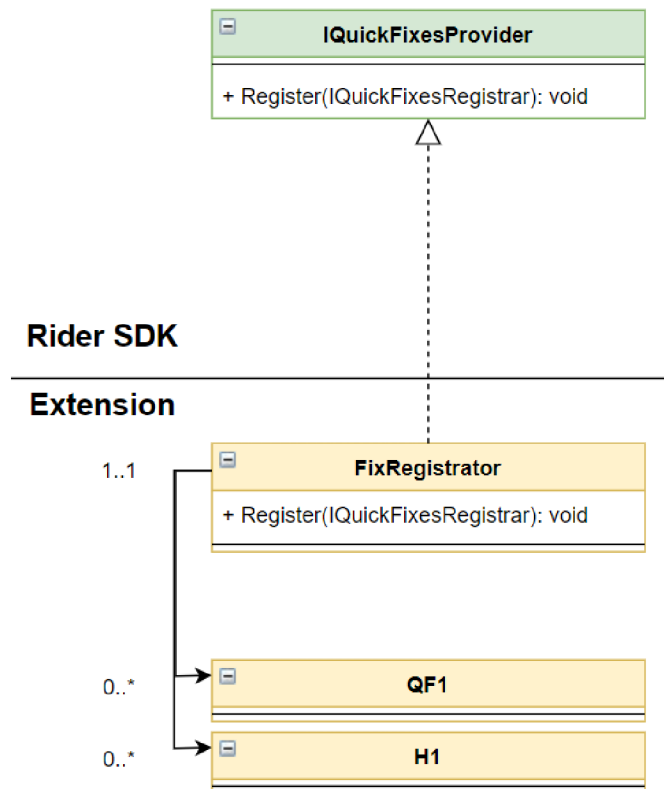
In the following diagram we can see how our FixRegistrator implements IQuickFixesProvider. It also important to see that it associated with a QF1 and H1 classes. We can have only one FixRegistrator but zero or many QF1 and H1 classes. It makes possible showing the QF1 in the context menu when the highlighting H1 is showed.

4.4 Implementation

When developing an extension, it is crucial to define the name and author before beginning the implementation process. In the case of a code smells refactoring extension, the author of the extension will also be the author of the work. Defining a clear and concise extension name is important not only for branding purposes, but also for easier identification and tracking during the evaluation process. By clearly defining the extension name and author, developers can establish a more organized and efficient workflow, while also ensuring proper attribution and recognition for their contributions.

4.4.1 Preparation

Setting up a project for developing an extension for the Rider IDE is an essential step that should be taken into consideration. There are various official instructions available on the https://github.com/JetBrains/resharper-rider-plugin page that developers can follow to set up their project.

It is highly recommended to use the latest version of the Rider IDE to avoid any compatibility issues. It is crucial to follow the instructions carefully to ensure that the project is set up correctly and ready to be developed further. Keeping the link to the official instructions as a reference is beneficial as the instructions may be updated frequently.

After the set up process we will be able to see next three folders:

Figure 8: Initial folder structure
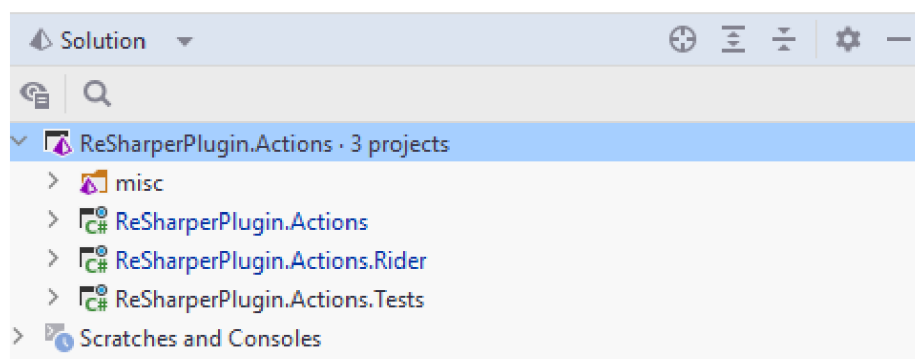
- **ReSharperPlugin.Actions** – the plugin version for Resharper

- **ReSharperPlugin.Actions.Rider** – the plugin version for Rider

- **ReSharperPlugin.Actions.Tests** – the project with unit tests

- **Misc** – the folder with configurations and infrastructure tools.

The ReSharperPlugin.Actions and ReSharperPlugin.Actions.Rider is depends on each other. If a developer does a change in the first project it will be automatically applied for the second one to ensure consistency. Now we are ready for the actual implementation.

4.4.2 Long method problem

The long method code smell was described in the section 3.1.1. For now, we will concentrate on fixing it by extract method technique, described in the section 3.2.1. We should begin with the highlightings.

As it was already mentioned, the highlighting class should implement IHighlighting interface.

The highlighting classes have been shown below:

Source code 8: ExtractForStatementHighlighting.cs

```csharp
public class ExtractForStatementHighlighting : IHighlighting
{
    private const string SeverityId =
"ExtractForStatementInspection";
    public const string Message = $"ReSharper SDK: Extract For
statement";
    public const string Description = "A for statement can be
extracted to a new method";

    public ExtractForStatementHighlighting(
        IForStatement forStatement)
    {
        ForStatement = forStatement;
    }
    public IForStatement ForStatement { get; }

    public bool IsValid()
    {
```

55

```csharp
            return true;
        }

        public DocumentRange CalculateRange()
        {
            return ForStatement.GetHighlightingRange();
        }

        public string ToolTip => Message;
        public string ErrorStripeToolTip => Message + " StripeToolTip";
}
```

Source code 9: LongMethodHighlighting.cs

```csharp
public class LongMethodHighlighting : IHighlighting
{
    public const string SeverityId =
"LongMethodDeclarationInspection";
    public const string Message = $"ReSharper SDK: Long method";
    public const string Description = "A method contains too many
lines of code. Generally, any method longer than ten lines should
make you start asking questions.";

    public LongMethodHighlighting(IMethodDeclaration declaration)
    {
        Declaration = declaration;
    }

    public IMethodDeclaration Declaration { get; }

    public bool IsValid()
    {
        return Declaration.IsValid();
    }

    public DocumentRange CalculateRange()
    {
        return
Declaration.NameIdentifier.NotNull().GetHighlightingRange();
    }

    public string ToolTip => Message;

    public string ErrorStripeToolTip => $"ReSharper SDK:
```

```
{nameof(LongMethodHighlighting)}.{nameof(ErrorStripeToolTip)}";
}
```

We passing an IMethodDeclaration object to the constructor for further calculation of DocumentRange object in CalculateRange method. It is used to determine the problem area and highlight only required parts of the code. In our case the method name will be highlighted.

Now we are ready to implement problem analyzer. To do so, we need to extend ElementProblemAnalyzer<IMethdoDeclaration> abstract class.

Source code 10: Long method problem analyzer

```
public class LongMethodProblemAnalyzer :
ElementProblemAnalyzer<IMethodDeclaration>
{
    protected override void Run(
      IMethodDeclaration element,
      ElementProblemAnalyzerData data,
      IHighlightingConsumer consumer)
    {
        var statements = 0;

        foreach (var statement in element.Body.Statements)
            statements = GetStatementsLines(statement, statements);

        if (statements >= 5) {
            consumer.AddHighlighting(new
LongMethodHighlighting(element));

            foreach (var statement in element.Body.Statements)
                AddHighlightings(statement, ref consumer);
        }
    }

    private int GetStatementsLines(ICSharpStatement statement, int
totalLines)
    {
        foreach (ICSharpStatement substatement in
statement.Substatements())
            totalLines = GetStatementsLines(substatement,
totalLines);

        if (!statement.Substatements().Any())
```

```csharp
            totalLines++;

        return totalLines + statement.Substatements().Count();
    }

    private void AddHighlightings(
      ICSharpStatement statement,
      ref IHighlightingConsumer consumer)
    {
        if (statement is IForStatement)
            consumer.AddHighlighting(
            new ExtractForStatementHighlighting(statement as
      IForStatement)
            );

        foreach (ICSharpStatement substatement in
statement.Substatements())
                AddHighlightings(substatement, ref consumer);
    }
}
```

The Run method has been implemented to incorporate the LongMethodHighlighting if the number of statements exceeds five. Additionally, an extra logic has been included to integrate the ExtractForStatementHighlighting. This feature is intended to highlight "for" statements, enabling the extract method technique to be applied at a later stage.

Now we are able to detect long method problems and we need to implement a quick fix logic to provide a quick button to fix the problem.

To do it, we created the ExtractForStatementFix class that extends QuickFixBase.

Source code 11: ExtractForStatementFix

```csharp
public class ExtractForStatementFix : QuickFixBase
{
    public readonly IForStatement ForStatement;

    public ExtractForStatementFix(IForStatement forStatement)
    {
        ForStatement = forStatement;
    }
```

```csharp
    protected override Action<ITextControl> ExecutePsiTransaction(
      ISolution solution,
      IProgressIndicator progress)
    {
        var workflow =
            (CSharpExtractMethodWorkflowBase)new
CSharpExtractMethodFromStatementsWorkflow(solution, (string)null);

        return (Action<ITextControl>)(textControl =>
        {
            IList<IDataRule> datarulesAdditional = DataRules
                .AddRule<ISolution>(
                "ManualExtractMethod",
                ProjectModelDataConstants.SOLUTION, solution)
                .AddRule<IDocument>(
                "ManualExtractMethod",
                DocumentModelDataConstants.DOCUMENT,
            textControl.Document)
                .AddRule<ITextControl>(
                "ManualExtractMethod",
                TextControlDataConstants.TEXT_CONTROL, textControl);

        using (LifetimeDefinition lifetimeDefinition =
        Lifetime.Define(Lifetime.Eternal))
            {
                var context =
JetBrains.ReSharper.Resources.Shell.Shell.Instance
                .GetComponent<IActionManager>()
                    .DataContexts
                    .CreateOnSelection(
                      lifetimeDefinition.Lifetime,
                      datarulesAdditional);

                textControl.Selection.SetRange(ForStatement.GetHighli
ghtingRange());

                RefactoringActionUtil.ExecuteRefactoring(context,
workflow);
            }
        });
    }

    // ... Other class parts
}
```

In order to facilitate additional usage, we include an IForStatement object as a parameter in the constructor. When the user clicks the quick button fix, the ExecutePsiTransaction method is triggered, which in turn creates a CSharpExtractMethodFromStatementWorkflow object that acts as the refactoring provider. Before executing the refactoring, a context object is created. Finally, the RefactoringActionUtil.ExecuteRefactoring method is called, passing in the context and the previously created workflow object as parameters. By doing so, the refactoring can be executed.

It is left to register the quick fix to the problem. To do so we need to implement IQuickFixesProvider in the next way:

Source code 12: FixRegistrator

```
public class FixRegistrator : IQuickFixesProvider
{
    public void Register(IQuickFixesRegistrar qf)
    {
        qf.RegisterQuickFix<LongMethodHighlighting>(Lifetime.Eternal,
h => new LongMethodQuickFix(), typeof(LongMethodQuickFix));

qf.RegisterQuickFix<ExtractForStatementHighlighting>(Lifetime.Eternal
, (h) => new ExtractForStatementFix(h.ForStatement),
typeof(ExtractForStatementFix));
    }

    ...// Other methods and properties.
}
```

It assigns the ExtractForStatementHighlighting to the ExtractForStatementFix and LongMethodHighlighting to LongMethodQuickFix. It will allow user to see the fix when the problem occurs.

The purpose of the LongMethodQuickFix is not to automatically fix the long method problem, but to provide the user with a detailed description of the issue and a step-by-step solution for resolving it. It should be noted that the "QuickFix" prefix is required by the Rider SDK naming convention for any class that implements the IQuickFix interface. This interface

is part of the Rider API. The LongMethodQuickFix is an example of such a quick fix, which provides a helpful tool for addressing the long method problem and improving code quality.

Source code 13: LongMethodQuickFix

```csharp
public class LongMethodQuickFix : IQuickFix
{
    public IEnumerable<IntentionAction> CreateBulbItems()
    {
        var b = new List<IBulbAction>()
        {
            new MessageBoxInfoAction("Extract Method info", "Info:\nA
method that is too long and complex, making it difficult to
understand and maintain \n\nTo resolve it you have two options:\n\n
1.Select a range inside your function.\n Use CTRL + ALT + M to call
'Extract Method' dialog \n\n 2. Put the cursor of existing for
statements and call context menu to perform 'Extract method'
refactoring")
        };
        return b.ToQuickFixIntentions();
    }

    public bool IsAvailable(IUserDataHolder cache)
    {
        return true;
    }
}
```

4.4.3 Long class problem

In order to tackle the problem of long classes, which was previously described in section 3.1.1, we have decided to mark a class as "long" if it contains more than 20 lines of code. To detect these long classes, we will implement the "LongClassProblemAnalyzer" logic. This analyzer will scan the codebase for classes that exceed the 20-line threshold and flag them as potential candidates for refactoring. By using this analyzer, we can easily identify and prioritize the classes that require attention, allowing us to focus our efforts on the most critical parts of the codebase.

Source code 14: LongClassProblemAnalyzer

```csharp
public class LongClassProblemAnalyzer :
ElementProblemAnalyzer<IClassDeclaration>
{
    protected override void Run(IClassDeclaration element,
ElementProblemAnalyzerData data, IHighlightingConsumer consumer)
    {
        if (element.ClassMemberDeclarations.IsEmpty)
            return;
        var linesSum = 0;
        foreach (var memberDeclaration in
element.ClassMemberDeclarations)
        {
            if (memberDeclaration is IFieldDeclaration)
            {
                linesSum++;
            } else if (memberDeclaration is
ICSharpFunctionDeclaration)
            {
                linesSum += (memberDeclaration as
ICSharpFunctionDeclaration).Body.Statements.Count;
            }
        }
        if (linesSum >= 20)
        {
            consumer.AddHighlighting(new
LongClassDeclarationHighlighting(element));
        }
    }
}
```

The code defines a class called "LongClassProblemAnalyzer" that extends the "ElementProblemAnalyzer" class and is used to analyze instances of "IClassDeclaration". The "Run" method of this class iterates over the members of a class (fields and methods) and calculates the number of lines in the method bodies. If the total number of lines is greater than or equal to 20, it adds a "LongClassDeclarationHighlighting" highlighting to the "consumer" parameter.

The highlighting can be represented in the next way:

Source code 15: LongClassDeclarationHighlighting.cs

```csharp
public class LongClassDeclarationHighlighting : IHighlighting
{
    private const string SeverityId = "SampleDeclarationInspection";
    public const string Message = $"ReSharper SDK: Long class";
    public const string Description = "A class contains many
fields/methods/lines of code.";

    public LongClassDeclarationHighlighting(IClassDeclaration
declaration)
    {
        _declaration = declaration;
    }
    private IClassDeclaration _declaration { get; }

    public bool IsValid()
    {
        return _declaration.IsValid();
    }

    public DocumentRange CalculateRange()
    {
        return
_declaration.NameIdentifier.NotNull().GetHighlightingRange();
    }

    public string ToolTip => Message;
    public string ErrorStripeToolTip => Message + " StripeToolTip";
}
```

4.4.4 Final plugin overview

Taking to consideration the implementation of every described entity as highlightings, problem analyzers, quick fixes and registrator, we should have the next folder structure:

Figure 9: Final plugin structure.
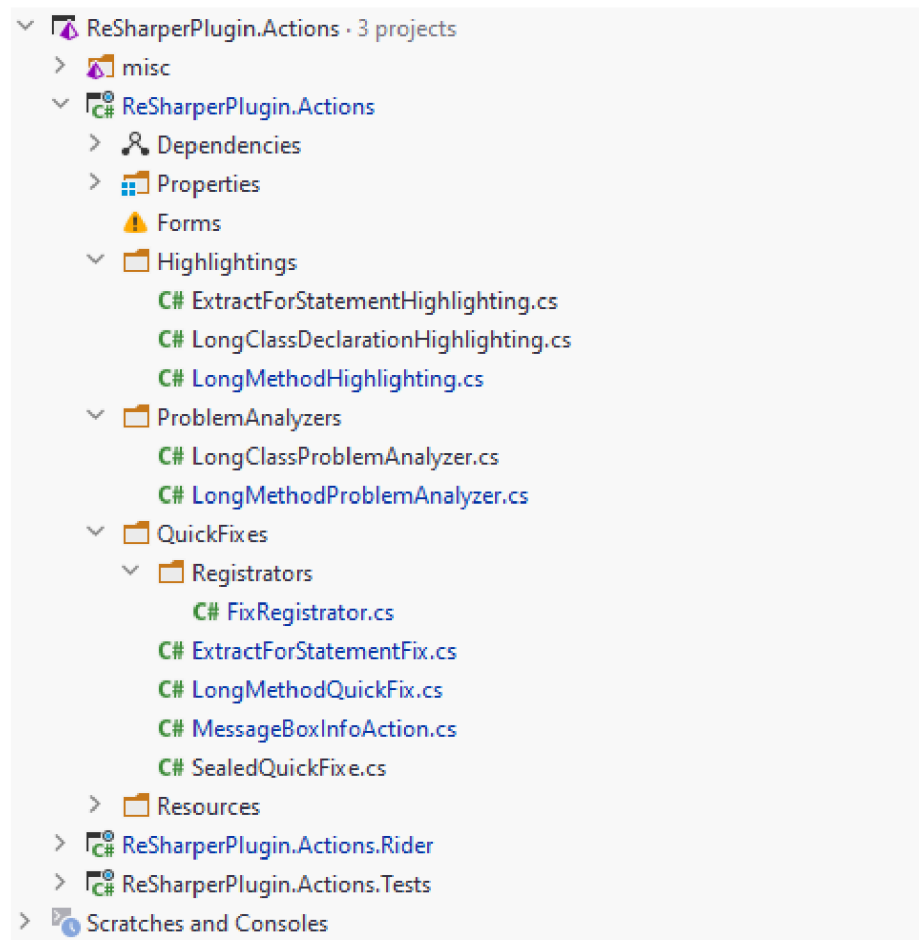
When developing a plugin that works both in Visual Studio with ReSharper and in Rider IDE, it's important to ensure that the codebase is structured in a way that is compatible with both platforms. This can be achieved by maintaining a consistent structure between the ReSharperPlugin.Actions and ReSharperPlugin.Actions.Rider namespaces, as this allows the plugin to be easily ported between the two environments. Additionally, developers should consider the specific features and limitations of each platform when designing the plugin, in order to ensure that it functions properly in both environments. By taking a thoughtful and strategic approach to plugin development, developers can create tools that are highly effective and widely accessible, regardless of the platform being used.

The remaining code, which was not covered in this work, can be found in the appendix section.

4.5 Evaluation

4.5.1 Integration with Rider IDE

The extension should seamlessly integrate with the Rider IDE and be easily accessible to users through the user interface. The criterion will be displaying the name of the extension in the plugin list of Rider. We can prove it by going to settings/plugins and observing the "Code smells refactoring extension" plugin among others.

Figure 10: Code smells refactoring extension in plugins section
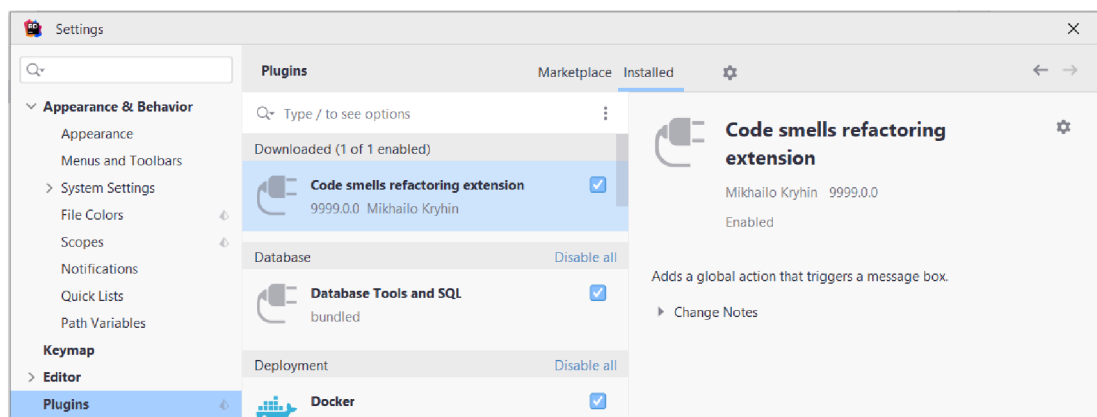
4.5.2 Code smells detection

The extension should be able to detect common code smells in the source code. Can be tested on the long method example. Should show a warning if method contains more than 4 lines. To evaluate it, we can create a simple c# console project with the class Program with the next code:

Source code 16: Example of long method code smell detection

```
private static void MethodExample()
{
    Console.WriteLine("Press any key to continue");
    Console.ReadKey();
    for (int i = 0; i < 5; i++) {
        Console.WriteLine("Count: " + i);
    }
}
```

The following code should not fire warning because we have no more than 4 lines.

Figure 11: No warnings example

```
5          internal class Program
6          {
7              public static void Main()
8              {
9                  Console.WriteLine("Press any key to continue");
10                 Console.ReadKey();
11                 for (int i = 0; i < 5; i++) {
12                     Console.WriteLine("Count: " + i);
13                 }
14             }
15         }
```

    C# ExtensionTestSolution   () ExtensionTestSolution

Problems: Current File 1   All Solution Files   Toolset, Environment 1

C# Program.cs C:\Users\Study\RiderProjects\ExtensionTestSolution\ExtensionTestSolution 1 problem
        ✔ Class 'Program' is never instantiated :5

According to the screenshot above, there are no such warnings. Let add one more line and see problems list.

Figure 12: Warnings example.

```
5          internal class Program
6          {
7              public static void Main()
8              {
9                  Console.WriteLine("Press any key to continue");
10                 Console.ReadKey();
11                 for (int i = 0; i < 5; i++) {
12                     Console.WriteLine("Count: " + i);
13                 }
14                 Console.ReadKey();
15         }
```

    C# ExtensionTestSolution   () ExtensionTestSolution

Problems: Current File 3   All Solution Files   Toolset, Environment 1

C# Program.cs C:\Users\Study\RiderProjects\ExtensionTestSolution\ExtensionTestSolution 3 problems
        ⚠ ReSharper SDK: Long method :7
        ⚠ ReSharper SDK: Extract For statement :11
        ✔ Class 'Program' is never instantiated :5

66

At present, it is evident that the method name "Main" has been highlighted, and a tooltip displaying the message "ReSharper SDK: Long method" is visible. Furthermore, the same warning can be observed in the problem list located at the bottom of the screen.
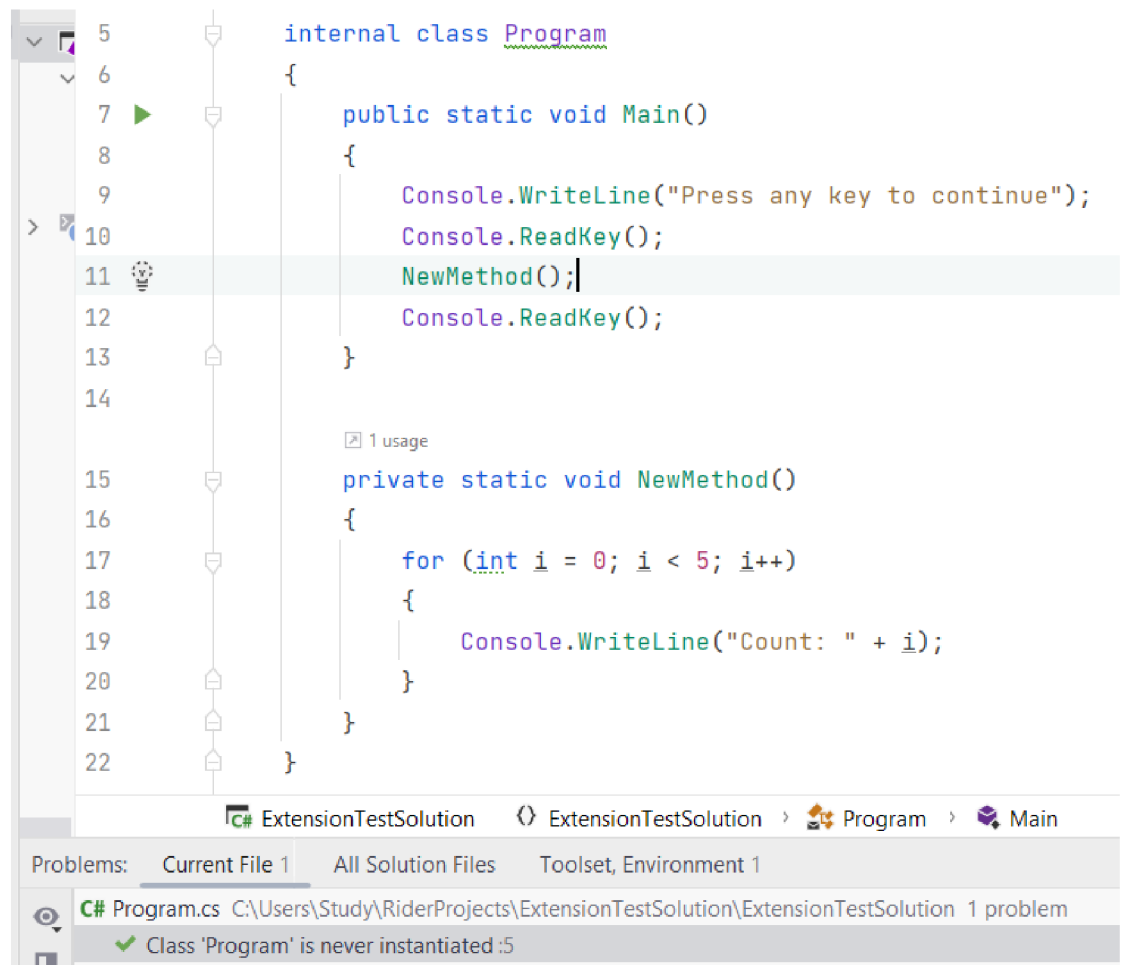
4.5.3 Refactoring support

In order to facilitate the process of refactoring code, it is important for the extension to have the capability to support commonly used techniques, such as method extraction. As evidenced by the previous example, there exists a noticeable yellow region that spans from line 11 to 14, which serves to highlight a particular issue. Additionally, there is a supplementary notification labeled "ReSharper SDK: Extract for statement," which can be seen alongside the highlighted section. By simply hovering the cursor over the yellow segment, a bulb action will appear, allowing the user to quickly and easily perform the desired refactoring operation.

Figure 13: Extract for statement context action.
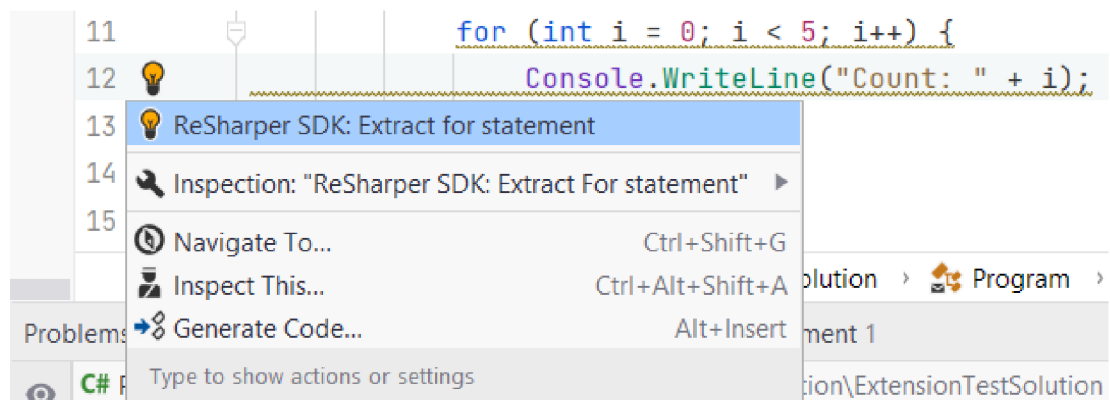
After clicking on it and process the refactoring dialog we will see the result of the refactoring.

67

Figure 14: Extract for statement refactoring result.

```
 5        internal class Program
 6        {
 7            public static void Main()
 8            {
 9                Console.WriteLine("Press any key to continue");
10                Console.ReadKey();
11                NewMethod();
12                Console.ReadKey();
13            }
14

     1 usage
15            private static void NewMethod()
16            {
17                for (int i = 0; i < 5; i++)
18                {
19                    Console.WriteLine("Count: " + i);
20                }
21            }
22        }
```

C# ExtensionTestSolution   () ExtensionTestSolution › Program › Main

Problems: Current File 1   All Solution Files   Toolset, Environment 1

C# Program.cs C:\Users\Study\RiderProjects\ExtensionTestSolution\ExtensionTestSolution 1 problem
        ✔ Class 'Program' is never instantiated :5

After moving the "for" statement to its own function, named "PrintCounts," no warnings are generated.

4.5.4 User friendly interface

Based on the information presented in Figure 11, it is apparent that the "Extract for statement" context action is displayed using Rider's native controls. Additionally, the Extract Method dialog employs Rider's native UI kit, which indicates that it meets the required acceptance criteria. It is worth noting that any changes made to the Rider theme by the user will also impact the appearance of the relevant dialogs and context actions.

Figure 15: Extract for statement context action (light theme).
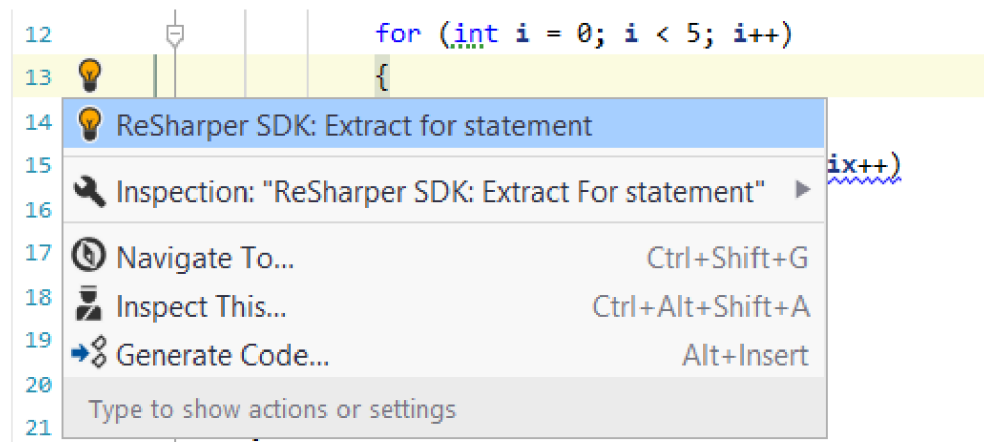

4.5.5 Compatibility

In section 4.6.2 we created a console type project and demonstrated the plugin usage in it. Also, it is possible to use the same plugin in any other c# project types like win forms or web app. This versatility of the plugin makes it useful for a wide range of development projects, allowing developers to identify and refactor code smells in various contexts. By applying the techniques and tools provided by the plugin to different types of projects, developers can maintain code quality and readability across their entire codebase.

4.5.6 Documentation

To demonstrate the documentation abilities, we can use the previous example and instead of performing actions, we can use "Extract method info" context action. It will lead us to an info box information.

Figure 16: Extract method info box.
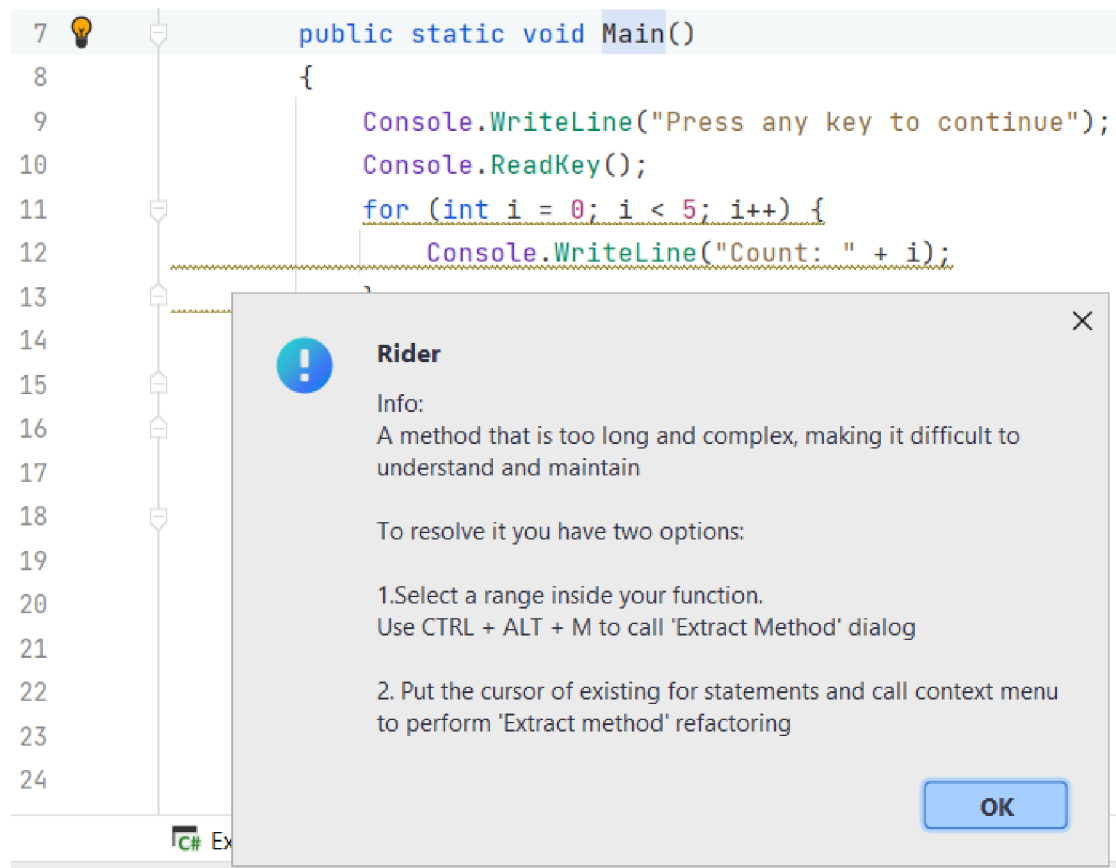
5 Conclusion

In conclusion, the development of a refactoring extension for Rider IDE based on code smells detection has been an interesting and challenging task. The proposed extension has been designed to help developers to identify and fix code smells that can lead to potential problems in their codebase. The extension uses algorithms to detect code smells and provide recommendations for possible refactorings. The evaluation of the extension has shown promising results in terms of its usefulness of its recommendations and native interface. The implementation of the extension has required a good understanding of the Rider IDE architecture, the principles of refactoring, and the techniques for code smell detection. Overall, the development of this refactoring extension has contributed to the improvement of the software development process by providing developers with an efficient and reliable tool for detecting and fixing code smells. The extension can be further improved by incorporating more sophisticated algorithms, expanding the set of supported code smells and provided more documentation. The result of the thesis can be used by any developers who are willing to use Rider IDE and simplify the process of code analysis.

6 References

1. FOWLER, Martin. *Refactoring: Improving the Design of Existing Code*. Boston, Addison-Wesley, 2019.

2. McConnell, Steve. *Code Complete: A Practical Handbook of Software Construction. 2nd ed.,* Microsoft Press, 2004.

3. JetBrains. *Resharper SDK Welcome* [online]. www.jetbrains.com/help/resharper/sdk/welcome.html. Accessed 01.11.2023.

4. Martin, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall Professional, 2008.

5. SourceMaking team. *Simplifying Method Calls* [online], *https://sourcemaking.com/refactoring/simplifying-method-calls.* Accessed 26.02.2023.

6. JetBrains. *Rider* [online]. https://www.jetbrains.com/rider/. Accessed 08.10.2022.

7. JetBrains. *Building a .NET IDE with JetBrains Rider* [online]. https://www.codemag.com/article/1811091/Building-a-.NET-IDE-with-JetBrains-Rider. Accessed 10.02.2023.

8. Santiago Mino. *How to Identify Code Smells* [online]. 29 Apr 2022 https://www.jobsity.com/blog/how-to-identify-code-smells. Accessed 11.02.2023.

9. Anu Viswan. *Code Smells: Object Oriented Abusers* [online]. 25 February 2018. https://bytelanguage.com/2018/02/25/code-smells-object-oriented-abusers. Accessed 15.10.2022.

10. Refactoring.Guru. *Change Preventers* [online]. https://refactoring.guru/refactoring/smells/change-preventers. Accessed 22.12.2022.

11. Sten Pittet. *The different types of software testing* [online]. https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing. Accessed 13.11.2022.

12. CSW Solutions Team. *5 Benefits of Unit Testing and Why You Should Care* [online].15 Dec 2022. https://cswsolutions.com/blog/posts/2022/december/5-benefits-of-unit-testing-and-why-you-should-care. Accessed 13.11.2022.

13. Richard Bellairs. *What is Static Code Analysis? Static Analysis Overview* [online]. *10 Feb 2020.* https://www.perforce.com/blog/sca/what-static-analysis. Accessed 13.11.2022.

14. JavierEscacena. *Refactoring - Moving features between objects* [online]. 2 Dec 2021. https://codersnack.com/js-clean-code-refactoring-catalog-move-features-between-objects/. Accessed 25.02.2023.

15. Scrutinizer. *Compose Method* [online]. https://scrutinizer-ci.com/docs/refactorings/compose-method#:~:text=A%20composing%20method%20is%20nothing,time%20naming%20the%20extracted%20methods. Accessed 14.01.23.

16. Manh Phan. *Dealing with change preventers* [online]. 15 Jan 2020. https://ducmanhphan.github.io/2020-01-15-Dealing-with-change-preventers/#:~:text=Change%20Preventers%20means%20when%20code,want%20that%20for%20obvious%20reasons. Accessed 16.12.22.

17. Pavan Kumar S. *Refactoring 101: Code Smells – Bloaters* [online]. 14 Dec 2021. https://medium.com/testvagrant/refactoring-101-code-smells-bloaters-f80984859340. Accessed 24.02.23.

18. Anu Viswan. *Code Smells: Dispensable* [online]. 19 May 2018. https://bytelanguage.com/2018/05/19/code-smells-dispensable. Accessed 26.02.23.

19. Yoan Thirion. *Refactoring journey* [online]. https://ythirion.github.io/refactoring-journey/journey/4-simplifying-conditional-expressions.html. Accessed 26.02.23.

20. Gavin Douch. *Code Smells* [online]. https://code-smells.com/change-preventers/divergent-change. Accessed 26.02.23.

7 List of pictures, tables, graphs and abbreviations

7.1 List of figures

7.2 List of source code