Source Code Quality Enhancer

Tristan Percy Read

001151378

Project Supervisors:

Cornelia Boldyreff

John Ewer

BEng (Hons) Software Engineering

University of Greenwich

7th May 2024

Abstract

This paper considers the current state of source code quality and the importance of maintaining a high quality codebase. It discusses how there is a lack of uniformity in coding styles and how this can lead to a decrease in efficiency within development teams and thereby cause increased costs and time to develop a project. The paper also shows an in-depth implementation of a program that can aid in increasing software quality and how it can be used to reformat source code to a user defined style with support for object oriented programming principles in combination with design patterns to create a well structured and higher quality codebase from a pre-existing codebase, or from scratch. Throughout this paper the use of the spiral software development methodology is used as it suits the project area well.

Preface

This project was undertaken as part of the requirements for the degree in BEng Hons, Software Engineering at the University of Greenwich. The motivation to researching this area comes from an interest in developing tools that aid programmers in creating software as individuals and in a team, so that uniformity can be maintained, and increasing the readability of a program that may one day be discontinued or passed onto another developer or set of developers.

Acknowledgements

Technical material used within the literature review and project planning was sourced from the course material provided and learnt in the BEng Hons, Software Engineering course at the University of Greenwich. This includes methodologies and techniques used during the research and development of this project. During the practical implementation of the project, the use of the internet was essential to the development of the project. This includes the use of StackOverflow, GitHub Gists, Microsoft documentation, and other sources of information that were used to help solve problems and provide information on how to use specific features of the C# language and Roslyn API. Debugging tools were also of a significant use in areas of development where online resources and documentation were either not available, not helpful or very sparse and limited in information with many of the discoveries made during development obtained through debugging and inspecting runtime objects. Peer reviews were used during the creation of this report to help improve the quality of the report and to ensure that the report was of a high standard.

Contents

1 Introduction **4**

1.1 Background . . . 4
1.2 Aims and objectives . . . 4
1.3 Project overview . . . 4
1.4 Justification . . . 4

2 Literature review **5**

2.1 Domain research . . . 5
2.1.1 Introduction . . . 5
2.1.2 Problem domain . . . 5
2.1.3 Review . . . 6
2.1.4 Conclusions . . . 6
2.2 Methodology . . . 6
2.2.1 Related work . . . 6
2.2.2 Defining software quality . . . 6
2.2.3 Considerations . . . 7
2.2.4 Requirements . . . 7
2.3 Design approach . . . 8
2.3.1 Overview . . . 8
2.3.2 Techinical details . . . 9
2.3.3 Prototypes . . . 9
2.3.4 Reflection . . . 10

3 Project development **11**

3.1 Evaluation . . . 11
3.2 Design philosophy . . . 12
3.2.1 Object oriented programming principles and design patterns . . . 12
3.2.2 Project management . . . 12
3.3 Integration . . . 13
3.3.1 User defined configuration . . . 14
3.3.2 Naming engine . . . 15
3.3.3 Document formatting . . . 17
3.3.4 Language features . . . 20
3.4 Testing . . . 21
3.4.1 Unit testing . . . 21
3.4.2 Integration testing . . . 23
3.4.3 Standalone testing . . . 24

3.5 Product evaluation 24

4 Epilogue **26**
4.1 Summary 26
4.2 Conclusion 26
4.3 References 26
4.4 Appendices 28
4.4.1 Custom unit test file parser 28
4.4.2 Manual testing 29
4.4.3 Debugging research process 29
4.4.4 Additional figures 30

List of Tables

1 Simplified Regular Expression Syntax 11
2 File Parser Syntax 28

List of Figures

1 Code TODO List 8
2 Token Breakdown 9
3 YAML Configuration 14
4 YAML Configuration for Punctuation 19
5 Unit Tests 23
5a Unit Test Results 23
5b Failing Unit Test 23
6 Integrated IDE Hints 24
7 Standalone Tool 24
8 Debugging research process 30
9 Project Timeline 31
9a Timeline table 31
9b Gantt chart 31
10 Using GitHub to track changes and issues 32
11 YAML Schema 32
12 Regex Token Recursive Parse 32
12a Group Construct Recursive Parse 32
12b Recursive Token Parser 32
13 Character Range Conform 32
14 Quantifier Conform Parameters 33

1 Introduction

1.1 Background

This project seeks to present a solution to the problem of software source code quality by creating a tool that can analyze source code and reformat it to a user defined style with support for object oriented programming principles in correlation with design patterns to create a well structured and higher quality codebase from a pre-existing codebase or to aid in the development of a new project with enforced coding standards.

1.2 Aims and objectives

The goals of this project are to create a tool that can: interpret source code; reformat source code to a user defined style; support object oriented programming principles; supply integration with existing popular IDEs; and provide a user interface for batch reformatting operations. By the end of this paper a series of existing solutions will have been evaluated, expressing their pros and cons, which will be used to formulate a set of requirements for a new tool that should help to address these issues.

1.3 Project overview

This paper will discuss the current state of source code quality and the importance of maintaining a high quality codebase. It demonstrates how there is a lack of uniformity in coding styles and how this can lead to a decrease in efficiency within development teams and thereby cause increased costs and time to develop a project. The paper also shows an in-depth implementation of a program that can aid in increasing software quality and how it can be used to reformat source code to a user defined style with support for object oriented programming principles in combination with design patterns to create a well structured and higher quality codebase from a pre-existing codebase, or from scratch. Throughout this paper the use of the spiral software development methodology is used as the development framework and the continuous integration pipeline is used during the practical implementation of the project.

1.4 Justification

The choice to work on the discussed domain was made due to the lack of uniformity in coding styles and how this can lead to a decrease in efficiency within development teams, thereby causing increased costs and time to develop a project. The use of the spiral model methodology is beneficial in this project as source code analysis tools have to be made up of various components that run independently from each other due to the complexity of programming languages and their respective syntaxes. Through the use of a spiral model methodology, the project can be broken down into smaller components that can be developed and tested independently by integrating a continuous integration (CI) model to the development pipeline.

2 Literature review

2.1 Domain research

2.1.1 Introduction

This project seeks to provide a solution to the problem of creating high quality code and structured code. Research was made into the area of software quality and the different approaches that have been proposed to measure this using papers and resources found primarily on on IEEE Xplore and ACM Digital Library. The research conducted on these sites have been specifically conducted on the area of software source code quality and the development processes around it. Additional areas of research that were relevant looked into the fields of code schemas, code analysis and developer preferences. There is a lack of standardization in the way software quality is measured while there exists a demand for high quality code. While there have been many proposed approaches to measuring software quality, not many of them have not been widely adopted and there is a lack of agreement on how to do this (Saraiva 2013).

2.1.2 Problem domain

The object oriented programming paradigm has become one of the most popular in the industry (Saraiva 2013). The wide adoptation of this paradigm has resulted in many large projects being written in an object oriented language. Unfortunately, IT programs often fail to deliver intended results due to poor program management and governance (Kummamuru & Hussaini 2015). It becomes increasingly important to maintain high quality source code as it becomes harder to find bugs in later stages of development (Ashfaq et al. 2019) as well as potentially becoming too expensive and increasingly difficult to maintain (Liu 2019, Singh et al. 2013, Pecorelli et al. 2020, Saraiva 2013, Kummamuru & Hussaini 2015), which then means programs can't be kept up-to-date which in turn could then introduce security vulnerabilities. Developing healthy software is always a challenge (Ashfaq et al. 2019) and research indicates that there is a lack of standardization in the way software quality is measured (Saraiva 2013, Ashfaq et al. 2019), making it difficult for developers to decide what methodologies to adopt (Saraiva 2013). These issues all exist while there remains a high demand for high quality code (Saraiva 2013).

Many of the papers in this area cover how important software quality is and related problems but only few provide approaches and solutions to solve the issues. For example, in the research conducted by Saraiva (2013) they investigated a large number of papers that looked into software quality. Amongst those papers only 14.4% of them went on to discuss the real world problems of software quality and of those papers only 2.7% proposed solutions to solve the problem Saraiva (2013).

Teachers are struggling to educate students how to write high quality code (Kirk et al. 2020), having a knock on effect as students will be entering the industry with a lack of understanding of how to write high quality code, with poorly written code becoming more common despite the amount of research that is being conducted in the area of software quality.

It is clear, then, that there is a problem with software quality and while many papers have been written

on the topic and the issues have been discussed, there is evidently a lack of real world solutions to the problem.

2.1.3 Review

This project will seek to present a solution to the problem of software quality by creating a tool that can analyze source code and reformat it such that it makes use of object oriented programming principles in combination with design patterns to create a well structured and higher quality codebase from a pre-existing codebase. However, such a tool cannot be created without researching further into the following topics: Existing approaches; Defining software quality; Object oriented programming principles and design patterns. Further research will be conducted into these topics in the methodology section of the literature review. Studies in this field are inconclusive on uniformity of coding styles and software quality measurement though to progress this project some of the alternatives available will need to be used.

2.1.4 Conclusions

The research, and that conducted by others, shows that there is no standardized way to measure software quality. Kirk et al. (2020), Stegeman et al. (2014) said that "current approaches do not take the developer's perception of design issues into account", therefore we cannot just create a tool that coheres to a standard. Instead the tool should be able to accept a schema that defines patterns and rules that the tool will attempt to adhere to. This is discussed more in section 2.2.3 where other research undertaken is referenced in this paper when applicable to the section being discussed.

2.2 Methodology

2.2.1 Related work

Ashfaq et al. (2019) covers how there are many tools that have been created to measure software quality. Some existing tools used to aid developers in maintaining particular coding styles include, but are not limited to:

- Lint4j - Checks the performance of code;
- Checkstyle & Codacy - Indicates errors and flags when language conventions are not followed;
- PMD - Inspects for code duplication;

These tools can be very helpful when used in conjunction with each other, however with the sheer number of tools available it can become difficult to decide which tools to use (Saraiva 2013).

2.2.2 Defining software quality

Saraiva (2013) said that "Software Engineering (SE) has very peculiar characteristics that strongly relate it to social sciences that encourage the implementation of empirical studies that are able to assess the effectiveness

of techniques, methodologies and processes proposed in the area". This can interpreted this as meaning that software quality is a broad topic and that the definition of what is considered high quality software can vary from person to person. In the paper by Kirk et al. (2020), it is inferred that teachers define high quality code by how readable the source code is to other developers. Kirk et al. (2020), Stegeman et al. (2014) defined code quality as what can be determined by "just looking at the source code, i.e. without checking against the specification".

2.2.3 Considerations

Before undertaking the development of this project, a number of considerations must made to ensure that the project would be successful. A first consideration is the choice of programming language. It has been decided that the project would be developed in and for the C# language as this is a widely used making this a valid candidate for this project. While it would be desirable to create a tool that supports multiple languages, this would have a significant impact on the complexity of the project and so it was decided that the project would be limited to C# due to the time restraints of the project in addition to my knowledge of other languages not being as strong as that of C#. As previously discussed, software quality does not have a set-in-stone definition, so in order to measure the quality of the software that is produced a baseline of things will be measured, such as consistency of the code, maintainability through the use of up-to-date language features, the use of object oriented programming principles and design patterns, and the ease of readability of the code. These are all important factors that can be used to measure the quality of the software that is produced in this project.

2.2.4 Requirements

For the development of this project, a multitude of base requirements should be laid out so that project has a structure to follow. These requirements will be used to guide the development of the project and ensure that the project is on track to meet the goals that have been set out. As the CI pipeline is to be used this will require continuous updating of the "todo" list as the list will change as new features get marked off when started, completed or dropped from the project pipeline as the project progresses (see figure 1). By having an adaptive todo list it allows for the project to be more flexible and allows for the project to be more easily adapted to changes in requirements, goals or new discoveries that may be made during the development of the project. The downside to such an approach is that it is less structured and so it can be harder to follow and find a set goal. However, it can still be suitable for this project as the goals and features are modular, as will be seen in section 3.2, how there is no set way to restructure code and so the tool will be updated with new features as they are discovered or needed. Another major requirement is the analysis of how to measure the current progress of the project through time management. While this is not as easy to do due to the nature of a CI pipeline, it is still important to make assumptions and estimates on how long each task will take to complete and how much of the alloted time should be spent on certain tasks. A project timeline will be used to aid in the time management of the project which will be updated as the project progresses (see figure 9a). It is important to follow this timeline, otherwise if the project time went unmanaged, it

Figure 1: Code TODO List

```
Priority: (4)
    ☐ Abstract generator @started(24-04-22 16:05)
    ☐ Auto-documenation generator (based on node tree)
    ☐ Generate example code to demonstrate with
    ☐ Fields and properties at the top of a type declaration

Code: (12)
    ☐ Brace line @started(24-03-20 11:56)
        ✓ New line after brace @done(24-04-14 02:05)
        ☐ Ignore single line blocks
        ☐ Last line brace after
        ☐ Move to source line @started(24-04-15 02:31)
    ✓ Indentation @done(24-04-16 08:13)
    ☐ Single statment block
    ☐ Nested node file splitter (split classes into individual files)
    ✓ Naming convention @done
    ✗ Large parameter multi-line (somewhat achieved through operator location) @cancelled(24-04-22 07:15)
    ☐ Comment style
        ✓ Space after declaration @done(24-04-17 04:37)
        ☐ Single to multi-line
        ✓ Full stop at end @done(24-04-17 11:11)
        ✓ Capital letter at start @done(24-04-19 14:33)
        ☐ Multi to single-line
    ✓ Operator location @done(24-04-22 07:14)
        ✓ Move && and || to start/end of line @done(24-04-22 07:14)
        ✓ Move ? and : to start/end of line @done(24-04-22 07:14)
        ✓ Add/remove spaces around operators @started(24-04-21 23:03) @done(24-04-22 04:38) @lasted(5h35m21s)
    ☐ Chain functions on new lines
    ✓ Explicit access modifier @done(24-04-22 04:38)
    ✗ Explicit type (already built into the IDE). @cancelled(24-04-22 04:38)
    ☐ Correct line spacing
    ✓ Batch fix diff UI @done(24-04-21 02:08)
    ☐ Abstract own code
    ✓ Implicit new keyword (C#8+) @done(24-04-23 23:26)
```

may result in incomplete or rushed features that are not up to the standard that is expected Additionally, too much time may be spent on tasks and potentially not leaving enough time for others. A Gantt chart can also be useful as it also shows interdependencies of activities and so delays in some areas may delay the overall project and this can be equally seen when the tasks are updated (see figure 9b). This can then help identify if other tasks can be completed sooner to avoid delays if changes are required or the overall delay is accepted.

2.3 Design approach

2.3.1 Overview

To create a tool that can analyze source code and reformat it such that it makes use of object oriented programming principles in combination with design patterns to create a well structured and higher quality codebase from a pre-existing codebase, C# language will be used. This is a suitable choice as this is object oriented and is popular in the industry, as well as having many features that can be used to create a well structured codebase.

A codebase will need to be provided as well as a set of rules to adhere to, however as noted, there is no standardized way to measure software quality. Ashfaq et al. (2019) proposed definition of software quality being "the degree of conformance to explicit or implicit requirements and expectations". By allowing a user

to define a schema that sets the requirements and expectations of the codebase, an individual or team can define their own standards and then use this tool to enforce those standards.

The output of this tool will attempt to provide a reformatted codebase that adheres to the set schema, from which a report can be produced. The report will contain information about the codebase such as UML diagrams as a visual overview of the codebase as well as documentation. This metric can allow developers to get an overview of the codebase so they can more easily figure out what a program does and how it works, thereby saving time and money when it comes to maintaining and migrating a codebase between developers and teams.

2.3.2 Techinical details

Static code analysis is the process of analyzing source code without executing it (Liu 2019). Dewhurst et al. (2021) discuss various methods for static code analysis, some of which include: Data Flow Analysis (Dewhurst et al. 2021) and Lexical Analysis (Dewhurst et al. 2021).

There are also various ways of representing the data that is collected from the static code analysis. One way is to use an Abstract Syntax Tree (Liu 2019) and another is to use a Node Graph.

Node graphs can be used to represent the links between classes and data within a program. Making it allows one to see how data is linked between classes and methods. This also gives on the ability to build a UML diagram to present to the user.

Abstract syntax trees (ASTs) can be used to represent the structure and flow of how a program works (Liu 2019). Abstract Syntax Trees could be used to break down the flow and instructions of a program. This can then be used to detect code smells such as duplicated code, long methods, poor flow control, cyclical dependencies, etc.

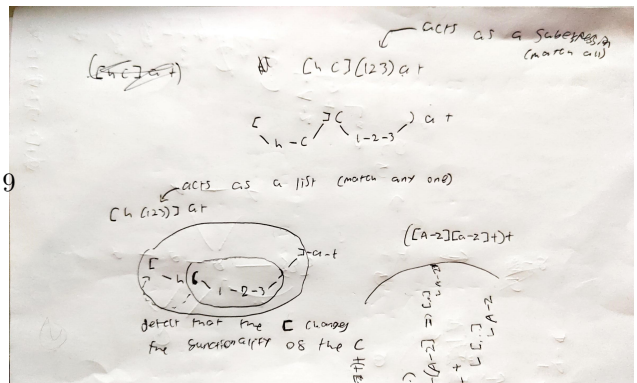
By combining the use of ASTs and node graphs an algorithm can be made to detect the relationship between classes and methods as well as the flow of the program. The algorithm can then decide how to modify and restructure the source code. It is important that both relation and flow are taken into consideration when refactoring the source code otherwise the algorithm may worsen the code by creating code smells such as god classes, or removing data that is required by other classes.

YAML and JSON are credible candidates for providing the schema, however as YAML is a more human friendly data format it will be used for this tool. YAML will allow the user to define their own schema that the tool will then attempt to adhere to. The schema will allow for the definition of things such as: Naming conventions; Use of Object Oriented (OOP) principles; Code deduplication; etc.

2.3.3 Prototypes

One of the major goals of this project was to create a module for the program that would correct the naming of objects, members and variables to match a user-defined style, the point of this is to ensure consistency of the schema throughout the pro-

Figure 2: Token Breakdown



gram and because of this importance it is once of th major goals of the project. Regular expressions (regex) is a very powerful algorithm that allows for checking if an input string matches a pattern using a specific syntax. While there are many regex tools available, either through third-party packages or built-in to the programming language, the majority of them are only capable of testing if a string matches a pattern and performing simple replacements on matches within a string based on group constructs in the provided regex pattern. However, converting an input to conform to an output style is a much more complex task as the algorithm needs to be run "in reverse", though it is not as simple as just reversing the regex pattern. With this in mind a new type of regex engine had to be created from scratch to suit the needs of this task. To start out, the existing syntax for regex was noted, specifically the syntax from the IEEE POSIX standard (Wikipedia contributors 2024) and was investigated. Theoretical prototypes were created on paper to show how the regex pattern could be deconstructed and reconstructed (see figure 2). From this syntax, a simplified set was created that would be used for the new regex engine (see table 1). This simplified expression set was picked to be simple to implement and easy to read. An advanced language set is not nessecaraly needed for matching single strings on object names, however there are cases where it could be useful.

2.3.4 Reflection

From what was discovered in the literature review, it is clear that there are many ways to improve the quality of a codebase. The use of object oriented programming principles and design patterns can be used to create a well structured codebase. It is important to note that these principles and patterns can be abused and lead to code smells which affect program comprehensibility, maintainability and testability, however the ultimate result of the code reproduced by this program will be down to the configuration that is provided by the end user. From the prototypes that were created, the goals of this project have been updated to facilitate the integration of a new, custom, regex based naming engine that can be used to convert an input string to a desired output string based on the regex pattern. Further research will have to be conducted into the technical details of how to achieve this. Good areas to look into include: Practical applications of static code analysis; Software engineering standards; Object oriented programming principles; Design patterns; Cyclic references and ideal flow control; etc.

Table 1: Simplified Regular Expression Syntax

Symbol	Description	Example
Quantifiers		
?	Zero or one	ab?c matches ac and abc
*	Zero or more	ab*c matches ac, abc, abbc, abbbc, etc.
+	One or more	ab+c matches abc, abbc, abbbc, etc.
{}	Exact number	ab{2}c matches abbc
{n,m}	Range	ab{2,4}c matches abbc, abbbc, abbbbc
Group Constructs		
()	Group	ab(cd) matches abcd
Meta sequence		
\s	Whitespace	
\S	Non-whitespace	
\d	Digit	Matches 0 through to 9
\D	Non-digit	
\w	Word	Matches letters A to Z case insensitive
\W	Non-word	
Character Range		
[]	Character range	[a-c] matches a, b, or c
^	Negated character range	[^ a-c] matches any character except a, b, or c
Atom		
.	Any character	A matches the capital letter A

3 Project development

3.1 Evaluation

In order to evaluate the developed tool the following will be assessed: function; comprehensibility; and performance.

The function of the reproduced code should maintain the result of the original code. This can be measured by running the original code and the reproduced code either through manual or automated testing and then comparing the results. Within these results we are looking for an output that matches a predetermined output defined by us which we can use to measure the accuracy and consistency of the reproduced code.

The comprehensibility of the reproduced code should be no harder to digest than the original code. This is a harder metric to measure as it is subjective, so in order to evaluate this peer review can be used as well as tools that measure cyclomatic complexity.

Finally, the performance of the reproduced code should be on-par or better than the original code. This can be measured by running the original code and the reproduced code and comparing the time taken for each. While this tool will not aim to optimize the performance of the code, it should not make it worse, i.e. the timetables should be the same or less.

In the paper by Ashfaq et al. (2019), it was said that "no tool succeeds in all respects". By this they

mean that no tool is perfect and that each tool has its own strengths and weaknesses. This statement is something that should be kept in mind with this project as it will not aim to create a tool that succeeds in all respects. Instead it will aim to create a tool that can combine the strengths of each tool to create a more complete tool.

All these aspects will be considered in the overall project evaluation.

3.2 Design philosophy

3.2.1 Object oriented programming principles and design patterns

There are many object oriented programming principles and design patterns that can be used to create a well structured codebase. In the paper Kirk et al. (2020) a few principles and patterns are discussed in relation to code quality. The principles and patterns discussed are as follows:

- Documentation - Comments;
- Presentation - Layout & formatting;
- Algorithmic - Flow, idiom & expressions; and
- Structure - Decomposition & modularization;

Ideally this project will make use of as many of these principles and patterns as possible in order to improve the quality of the code that can be produced. Abstraction and inherence can be very useful when it comes to the structure of a program and the deduplication of code as well as maintainability, which was also mentioned by Ashfaq et al. (2019) where it was stated "Modularization of code marks better reuse of the code and compilation time" meaning that the more modular the code is the easier it is to reuse and faster it is to compile thereby reducing the time it takes to develop the program.

However it is important that one doesn't abuse these abilities as they can lead to code smells which affect program comprehensibility (Ashfaq et al. 2019, Abbes et al. 2011), maintainability (Ashfaq et al. 2019, Khomh et al. 2012, Palomba et al. 2017) and testability (Ashfaq et al. 2019, Grano et al. 2019). As indicated in the paper by Neto et al. (2022) there are several types of code smells, such as: comments; duplicated code; feature envy; large class/god class; long method; lazy class; long parameter list; and shotgun surgery. Therefore, in considering a solution, care will be needed regarding the use of these principles and patterns such as abstraction and inheritance and automated comments as they can lead to these undesirable code smells.

3.2.2 Project management

When undertaking large projects, it is often recommended to use a project management tool to help keep track of the progress and changes made to the project. Throughout the development lifecycle of this project GitHub was used extensively to track changes and issues (see figure 10). GitHub is a web-based platform that uses the Git version control system to track changes made to a project. For this project in particular it

allowed for updating existing features with new experiential approaches without the risk of making permanent breaking changes to the existing code. This is because branches can be created via Git that are separate from the main tree and roll back changes if necessary. This is a useful feature as it allows for the project to be developed in a more agile way, where changes can be made quickly and easily without the risk of having to spend time reintegrating old code if the experimental branches do not work out.

3.3 Integration

The product that was developed was split into various modules. Each module was designed to be independent from one another. This has multiple advantages, such as making the code easier to maintain and allowing for easier debugging and modification. This was achieved through the use of various design patterns from the object oriented programming paradigm.

This section discusses the integration of the various modules that were developed, the ways that the user interacts with the system, and the ways that the system provides responses.

3.3.1 User defined configuration

As already noted, the lack of standardization in code formatting has lead to a key aim of this tool to be able to be configured by the user. It was also noted that this would be achieved through the use of a YAML configuration file. As configuration parsing isn't the goal of this work, the well-known, open source YAML parsing library YamlDotNet was used. This library is under the MIT license which grants full permission to use the library in this project.

YAML was chosen as the configuration language due to its human readability, as well as being very easy to implement and understand by the end user. A custom schema was created for the configuration to aid the user in creating the configuration file by providing IDE hints and validation on their configuration (see figure 11).

As with the program, the configuration is split up into multiple sections. Each section corresponds to a different module in the program with each module having a wide range of configuration options to allow for fine tuning of the program. In Figure 3 we can see an example of a configuration file, the visible snippet showing the configuration for the custom naming engine. Each module inherits from a base set of rules that allow the user to either enable or disable a module and set the severity of the issue that the module is checking for. By default, all modules are disabled and the severity is set to 'info'. Depending on how strict the work environment to be made, the severity can be set to cause an error which will result in not allowing the program to compile. This can be beneficial in a large team where you want to enforce a strict code style to improve readability and maintainability of the codebase.

Figure 3: YAML Configuration

```yaml
  namespace:
    severity: "error"
    pattern: "[A-Z][a-z]+(?:\\.?[A-Z][a-z]+)*$" #Pascal.Case
  generic_parameter:
    severity: "error"
    pattern: "T([A-Z][a-z]+)+" #TPascalCase
formatting:
  curly_braces:
    severity: "error"
    new_line: true
  indentation:
    severity: "error"
    size: 4
  comments:
    severity: "info"
    new_line: true
    leading_space: false
    trailing_full_stop: true
    capitalize_first_letter: true
    comment_detection_sensitivity: 0.5
  punctuation:
    space_around:
      - left: true
        right: true
        severity: "warning"
        tokens:
          - "+"
          - "-"
          - "="
    new_line:
      - left: true
        right: false
        severity: "info"
        tokens:
          - "&&"
          - "||"
  object_structure:
    properties_at_top: true
inferred:
  access_modifier:
    severity: "error"
    mode: "explicit"
  constructor:
    severity: "error"
    mode: "implicit"
```

The configuration file is loaded into the program at runtime. The way the configuration is loaded varies by the program implementing the tool due to the limitations of certain platforms, the configuration class has been created as a partial class that inherits from an interface where the configuration loader can be tweaked depending on the platform it is being compiled for. The configuration class is implemented as a singleton, as the code parser that is being used works in a static context, where configuration is loaded once and only once. This does have the downside of meaning any changes that the user wants to make to the configuration will require a restart of the program, which is a limitation of the Roslyn API and is not something that can easily be worked around.

Taking the configuration implementation from the VSIX project as an example, the configuration is loaded when the extension is loaded into the IDE. The configuration also has to get loaded into a separate process

that Visual Studio launches to run code analysis. This inherently caused a problem as it was discovered through debugging that the child-process does not have the context of the loaded workspace and so the user-defined configuration would not be found. To work around this issue, a shared memory space is setup between the two processes to allow for communication of the configuration to be shared (see figure **??**).

3.3.2 Naming engine

The naming engine turned out to be one of the most complex parts of the project. The naming engine is responsible for checking the names of various parts of the codebase, such as classes, methods, and variables and then converting them to match the user-defined configuration. The naming engine is split into multiple modules: the diagnostic module, code fix module; and the regex engine module. The diagnostic module is responsible for finding the issues in the codebase, the code fix module is responsible for fixing the issues, and the regex engine module is responsible for converting the names to match the user-defined configuration.

The diagnostic module uses the Roslyn API to parse the codebase and analyze objects, classes, methods, and variables. In order to reliably verify a name, the object name is check through the built-in C# regex engine. When the name does not match the user-defined configuration, a diagnostic is created and added to the list of diagnostics that are returned to the user. The diagnostic is then displayed in the IDE, and the user can then choose to fix the issue or not.

The code-fix module acts as a wrapper for the regex engine, receiving the diagnostic from the diagnostic module and returning the corrected name.

As discussed in the prototype, a custom regex engine was necessary to allow for the conversion of the names to match the user-defined configuration. The regex engine is an instantiable object that, when created, is passed a regex pattern that will be used to match an input to.

The regex engine has three main functions: parse; convert; and match. The parse function is responsible for converting the regex pattern into a list of tokens, the convert function is responsible for converting the tokens into a tree structure, and the match function is responsible for matching the input to the tree structure.

Pattern parsing During construction of the regex engine object, a regex pattern is to be provided which will be parsed into a token tree. The choice for representing the pattern data in a tree structure was made to allow for easy manipulation of the pattern. For example, if a quantifier token is found, it makes it significantly easier to determine what the quantifier token is linked to as one can either navigate up the tree or back one token to find the relevant token or set that needs to be quantified. Due to current limitations with the regex parser, the regex pattern gets wrapped in a group construct. The reason for this is that the regex parser uses group constructs as the base token for the tree structure. All recognized token classes within this regex engine inherit from the base `AToken` abstract class which contains common properties and interfaces that all tokens must implement. This base class contains properties for the local pattern that the token represents, its parent, children, previous and next tokens in addition to the following abstract methods: `CanParse`; `Parse`; `Test`; and `Conform`. And the protected helper methods for recursively parsing children and reading individual tokens from a string.

The parse method is responsible for converting a pattern into a tree structure of tokens. Every token class has a different implementation of this method, for example, the group construct class will first check the construct type which could be for example a positive lookbehind. After this the helper, recursive parse method is called to parse the children within the group construct. This recursive parse method takes three parameters: a list of valid child tokens, a consumable pattern to parse, and a closing token to look for. The method will then proceed to consume the pattern and parse tokens (see figure 12b). For any valid tokens found a new instance of that token class is constructed and the parse method on that child will be called. After the child token has been parsed, the child tokens parent gets updated to the calling contexts token. The previous token of the child is set to the last token in the parent's children list, and the child token is added to the parent's children list. The method will then continue to consume the pattern until the closing token is found. If the closing token is not found, the method will throw an exception due to the pattern being invalid. If the closing token is found, the method will return the parsed tokens. The parse method is called recursively on the root token to parse the entire pattern into a tree structure of tokens.

Input conversion The second part of the regex engine is the convert function. This function will attempt to convert an input string to conform to the pattern that the object was initialized with. Along with the input string, various options can be passed to the convert function to allow for fine-tuning of the conversion process. These options exist as there may be scenarios where certain parts of the pattern need to behave differently, for example, for literal tokens, the user may want to remove mismatched tokens or replace them with a valid token. For the scenario of renaming code objects, one will, more often than not, want to remove mismatched literals with a blank. To provide an example of this, if we want to conform the input `__my_variable` to the pattern `([A-Z][a-z]+)+`, it should produce a desired output of `MyVariable`. If mismatched tokens were not dropped the algorithm would fail to conform to the pattern as the pattern provides no way to match the literal characters `__` to the pattern. By dropping these mismatched characters, the algorithm effectively ignores them from the output and matches where it can on valid tokens, producing an output of `MyVariable`.

The conversion process is a recursive process that starts at the root token of the tree structure. Each implementation of the abstract conform method takes a single parameter of the `ConformParameters` object. This object contains information about the current state of the conversion process, the input string, the current index in the input string, the current state of the newly generated output string and other options specific to other tokens.

The most complex operation in this process will be shown using the code from the `Quantifier` class. The quantifier parse method starts out by getting any quantifier options for the current index in the parsing process. This method will search up the tree for a quantifier token and return the options for that quantifier, which will become apparent later on. Following this, the method will determine if the quantifier is greedy or not. A greedy quantifier is one that matches a pattern up to infinity times. Within the conform parameters a value can be set to configure a greedy quantifier to break under certain conditions. The method will then attempt to run the conform method on its children, if the conform method fails the quantifier will break and return the current index in the input string if the quantifier was configured in the zero-or-more mode. After successfully parsing the children at least once, checks are then run to see if the quantifier should break. If

it is greedy the quantifier will run until it can no longer match the pattern. If it is not greedy the quantifier will run either until the upper bound is reached or until the break condition is met. Currently, this break condition allows for breaking on: character case changes; alphanumeric changes; and delimiter changes. If the quantifier breaks the method will return the current index in the input string while if the quantifier does not break the method will return the index of the last successful match (see figure 14). The quantifier needs to initially search up the tree because it needs to know its current context depth, if it is a nested quantifier then it should only conform to its local context and not a parent one, as the context is shared within the conform parameters object. When this method returns, the parent context will continue to parse the input string from the index that the quantifier has set within the conform parameters object.

To give an example of how an implementation of the conform method generates a corrected string, we can look at the `CharacterRange` class. The character range class in regex is used to match a single character to a range of valid characters. In the instance of this classes conform method. Firstly, the character at the current index in the input string is checked to see if it is within the range of valid characters, which is achieved by comparing the characters unicode value to the unicode values of the start and end characters of the range. If the character is within the range, the character is added to the output string and the index is incremented. If the character is not within the range the method will attempt to see if the character can be converted to a valid character by changing the case of the character. For example, if the valid range of characters was `[a-c]` and the input character was `A`, the method would convert the character to its lowercase variant and see if that character is within the range. If the character is within the range after the case change, the character is added to the output string and the index is incremented. If the character is still not within the range after the case change, the method will either drop the character if configured to do so, or return false indicating that it was unable to conform the input string to the pattern (see figure 13).

Output validation The final part of the regex engine is the match function. This function will validate wether the result from the conform function matches the pattern that the object was initialized with. Usually this function does not need to be called as the conform method will either succeed of fail. The method exists as a validation step to ensure that the output is correct for the provided pattern using the custom regex engine algorithm as it may possibly differ from the standard C# regex engine.

3.3.3 Document formatting

The tool also provides various document formatting options to pick from where these options are designed to help the user maintain a consistent formatting style throughout their document. The tool provides the following formatting options: brace location; indentation; comment style; punctuation; and object structure. All of these options make use of the abstract syntax tree (AST) that is generated for each document. The AST contains all information about the document in a programmatic way as provided by the Roslyn application programming interface (API). Navigating and interpreting the AST as necessary for each of the formatting options is needed.

Comments Comments can be styled in various ways. The tool provides options to force comments to be on their own lines, or inline with code and additionally choose wether comments should have a leading space or a trailing full stop. Each item in the AST contains a relative location to the document. Using this location span one can determine what line the comment is on. If the user has configured comments to be on a new line then this can be navigated backwards through the AST to check if there are any non-whitespace tokens on the same line as the comment. If there are then a new line token can be inserted before the comment and add any necessary indentation. Minor performance improvements can be made by not checking tokens after the comment as single-line comments defined by using the double backslash trivia as this will always be the last non-whitespace token a line. Before styling comments, the tool will check to see if the comment token contains a mostly-text based comment or a commented out line of code. This is achieved by using a regex pattern on the token which captures words but not code. Following C#'s syntax, a code comment is considered to be any comment that contains more camel case words with trailing punctuation than it does regular words. The regular expression pattern used, `(?<![.])\b(?:[A-Z][a-z]*|[a-z]+|)\b(?![.(][A-z])`, can be broken down as follows:

- `(?<![.])` Don't capture when the text follows a full-stop;
- `\b` Limit the search to be between word boundaries (i.e. is a letter followed by a non-letter);
- `(?:` Capture everything inside as a single match;
  - `[A-Z][a-z]*` Capture a Capital and optinally unlimited lowercase letters, or...
  - `[a-z]+` Capture one or more lowercase letters, or...
  - Include spaces in the search captures.
- `(?![.(][A-z])` Don't capture when the sequence ends with either a . or (as these are common code tokens that often directly proceed text, so long as it is followed by another A-z character.

Within the user configuration file, an additional option can also be specified to set the sensitivity of the comment detection. By default this value is set to 50% meaning that if the number of characters matched in the pattern is greater than 50% of the total number of characters in the comment token string, it is to be considered a comment. If the result determines that the token is a code comment, the tool will not apply any formatting to the comment as it is assumed that the comment is a commented out line of code. Otherwise, if the comment token is a mostly-text based comment, the tool will proceed to run additional checks on the token. The benefit to using a detection algorithm like this is that it works across human languages without the need for a dictionary of common words to be implemented. The downside, however, is that it may not always work if the text does not match the pre-defined pattern rules, though through testing it more often than not works as expected not producing false positives.

Indentation To determine indentation levels for a given line the document is navigated line-by-line. A last-in-first-out (LIFO) stack is created to hold information about the current context level. A context level

is determined by the following tokens: `{ } ( ) [ ]`. When a token is found that increases the context level the current context level is pushed onto the stack, and when a token is found that decreases the context level the stack is popped. The indentation level is then determined by the number of items in the stack multiplied by the user-defined indentation size, allowing for the tool to determine the correct indentation level for each line in the document. A simple text search for the tokens cannot be relied upon as they may be part of a string or comment, therefore the AST has to be navigated to find the tokens. Additionally, the indentation level cannot properly be calculated if the stack contains an invalid sequence of tokens. For example, if the stack contains `{ [ }`, the indentation level cannot be calculated as the stack is in an invalid state. However, this should not occur as the custom analyzers are configured to only be run on syntactically valid C# code which is determined by the compiler.

Punctuation and brace location The punctuation sub-module allows the user to configure properties of language punctuation. In C#, punctuation is any token that is not a keyword or object variable, for example `;` `+` `-` etc. The punctuation sub-module allows the user to configure the spacing around punctuation, and wether punctuation this should be on the same line as the previous token or on a new line. Within the YAML configuration file the user has the ability to define as many punctuation rules as they desire for the following options: space around, and new line. The tool will then navigate the AST to find any matching punctuation and adhere it to the configuration. For each of these options within the YAML file, the user should provide an array of configurations. Each of these configurations are to contain options for the formatter, a severity and a list of tokens that the rule should be applied to (see figure 4).

Figure 4: YAML Configuration for Punctuation

```yaml
punctuation:
  space_around:
    - left: true
      right: true
      severity: "warning"
      tokens:
        - "+"
        - "-"
        - "="
  new_line:
    - left: true
      right: false
      severity: "info"
      tokens:
        - "&&"
        - "||"
object_structure:
  properties_at_top: true
```

The brace location works in a very similar way to the punctuation, however the brace location has to work slightly differently for the closing brace because the closing brace, if configured to be on a new line, must have one less indentation than what the tokens would otherwise have.

Object structure The object structure sub-module is rather minimal but can have a major impact on code readability. It provides the user with the option to force object properties and fields to be at the top of the object declaration with all methods at the bottom. This has a significant impact on readability and debugging as it contributes to reducing the amount of spaghetti code. Spaghetti code is code that is difficult to read and understand due to poor structure, which can be caused by having methods and properties mixed together in a class. By separating out the methods and properties it makes it far easier to understand what members a class has as there is no need to randomly dig through a class or struct to find potentially tucked away properties and fields between large method bodies. As before the sub-module will traverse the abstract syntax tree and, if any object declarations are encountered, a sub-loop will be run to check the order of the object members. If, for example, a method declaration is found and further down the object declaration a

property is found, the tool will create a diagnostic and return it to the user hinting that the object should be restructured. If the suggested change is accepted by the user the sub-module will move any methods to the bottom of the object declaration. During this operation various operations are made on the AST and with each subsequent modification a new AST is generated. This introduces a new problem of keeping track of other nodes that we may want to move. Fortunately, the Roslyn API provides a way to keep track of the original node that was moved and from this one can track nodes that need to be moved, deleted or replaced without having to re-traverse the AST for similar matching nodes to those produced by the diagnostic from before.

3.3.4 Language features

The language features module provides automation for updating code to support new language specific features. Currently, this module contains support for features such as inferred access modifiers and the implicit object initialization syntax that was introduced in C# 9.0.

Inferred access modifiers In C#, when a top level class or struct is declared, it is assigned a default access modifier of `internal`, while any nested declarations within the class or struct are assigned a default access modifier of `private`. For both of these instances an explicit access modifier can be provided to override the default. Using implicit access modifiers can reduce the amount of code written ever so slightly. To many it is likely not known what the default access modifiers are, so by providing the option to require explicit access modifiers, it can help to improve the readability of the codebase by making it clear what the access modifier is for a given declaration. The sub-module will be triggered for all object declaration syntax. For all declaration syntax that is passed to the sub-module, a check is run to determine what modifiers the node has. If the node is missing access modifiers a diagnostic is created and returned to the user with an automated code fix. The code fix will insert the missing access modifier into the list of modifiers for the node. The access modifier to be inserted is determined by the level of the node and if the node is a top level node the access modifier will be `internal`, while if the node is a nested node the access modifier will be `private`. A node is considered top-level if it is within a document that is not using top-level statements or if the parent of the node is a namespace declaration.

Implicit object initialization Introduced in C# 9.0, the implicit object initialization syntax allows for the omission of the object type when initializing an object whose underlying type is already known at compile-time. This can help to reduce the amount of code written and can help to improve readability by reducing the amount of boilerplate code that is written. For example, prior to C# 9.0, to initialize a new object one would have to write `var obj = new MyObject();`. With the new syntax one can write `MyObject obj = new();`. If enabled, the sub-module will be triggered for all object creation expressions. Unlike the previous modules in this tool, the AST cannot be worked on directly as context of the compilation is needed to determine the type of the object that is being created. Therefore, the compilation event provided is hooked by the Roslyn API. From this context we checks are made for any object creation expressions. On these expressions there are two ways that the underlying type can be determined: the first is by checking if

the object creation expression belongs to a variable declaration; and the second is by checking if the object creation expression is being assigned to a variable. For the former case, it is rather easy to determine if the object is being assigned to a new variable as the parent of the object creation expression can be checked to see if it is a variable declaration and if it is then it can be checked to see if the expression is being assigned to a `var` keyword or a type. If it is being assigned to a `var` keyword then the type of the object can be determined by checking the type of the variable declaration. For the latter case, the parent object is also assessed, though what the parent object is can vary significantly. Take the example of a method call: the object creation expression is a child of the method call. In this instance the definition of the method call with the relevant overload that the object creation expression is being assigned to needs to be found. Once the method call is found via the compilation context, is it necessary to acquire the index of the argument that the object creation expression is being assigned to, and from this index the type of the object that is being created can be determined. Once the type of the object is determined, the sub-modules code fix provider will either convert the object creation expression to or from the implicit object initialization syntax. When converting to the implicit syntax the type of assignment must be checked again and if the object creation expression is being assigned to a `var` keyword, the code fix will convert the object creation expression to the implicit syntax and replace the `var` keyword with the explicit type of the object. This has the added benefit of cleaning up the declarations in the program as while the `var` can reduce the amount of code written for variable declarations, it can often be misleading due to the variable type not being known to the programmer without searching through the code to see what is being assigned to it in order to determine the variable type. Likewise, if the module is configured to not allow implicit object initialization, the code fix will convert the implicit object initialization syntax to the explicit syntax and insert the type of the object.

During the development of this module, there was a significant lack of documentation and resources online to aid in this area, so a debugging research process conducted to help with this. This method of research and debugging was used a significant amount throughout the development of the program, the details of this process can be found in the appendix section 4.4.3.

3.4 Testing

The testing of the project was conducted in three main stages: unit testing; integration testing; and standalone testing. The unit testing was conducted using .NET Unit testing framework, integration testing was conducted using the Visual Studio Integrated Development Environment (IDE), and standalone testing was conducted by running the application in a standalone environment.

3.4.1 Unit testing

Unit tests were the primary method of testing the project. Unit tests are particularly useful for an algorithmic system as it allows for the testing of individual components in isolation. In addition to this regression testing, which is the process of testing a program to ensure that changes to one part of the program have not affected other parts of the program, has been used. This testing pattern is particularly useful for this project as it has been build in a modular fashion.

To ensure that the custom regex engine was robust, a comprehensive set of unit tests were written. These tests were designed to test the engine under a variety of conditions, including invalid inputs and a plethora of inputs to be converted to various output formats. The tests that were designed to validate the conversion process used a series of common naming patterns such as camel case, snake case, and pascal case. These tests were as simple as providing the custom engine with an input and comparing the output to an expected result to ensure that the engine was functioning correctly and as expected.

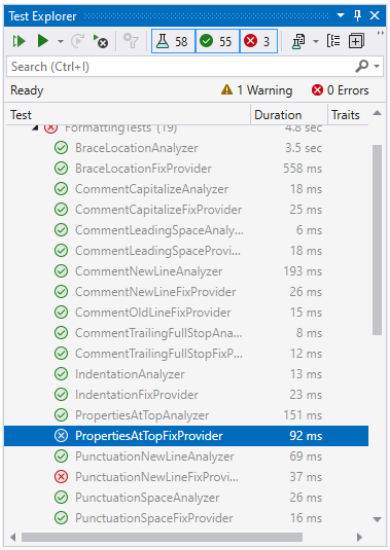
For testing the code analyzers and fix providers the MSBuild testing framework was used. This framework provides the basic functionality to compile input source files, run the analyzers and fix providers on the compiled source files and then compare the generated diagnostics and fixes to expected results provided by the test. The work put into making this testing framework more streamlined and easier to use was a significant development in ensuring the reliability of this tool, more detailed information of the workings behind this testing framework can be found in the appendix section 4.4.1.

These unit tests then run the MSBuild diagnostic analyzers and code fix providers with the provided input and expected outputs from the file parser and a reference to the custom modules written for this program. The tests are then run to completion and the results are compared to the expected results. If the tests have a runtime error or the results do not match the expected results the test will fail. Unit tests also allow for rapid testing and development because unit tests are written to be small and so are often fast to run. The results of these tests can also be easily interpreted as they are either pass or fail and most modern IDEs have built-in integration with unit testing frameworks. This project was developed within Visual Studio Community Edition which has built-in support for the MSTest framework which allowed for the tests to be run and the results to be viewed within the IDE itself with inline hints on where tests may have failed (see figure 5a).

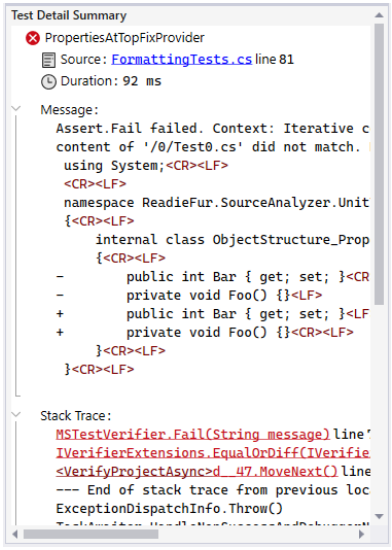
A significant amount of time was spent constantly testing the components of the tool to ensure that they were functioning correctly and as expected. Automated unit test executions were setup so that after each new build of the program, the tests would be re-run. Most of the tests yielded the expected results most of the time, with the exception of a few edge cases that were not major enough to be fixed, in particular any code fix providers that would enter new lines to a document would not always use the correct line ending for the operating system that the tool was running on (see figure 5b). This was not fixed as it was not a major issue and would not affect the functionality of the tool.

Figure 5: Unit Tests

(a) Unit Test Results



(b) Failing Unit Test

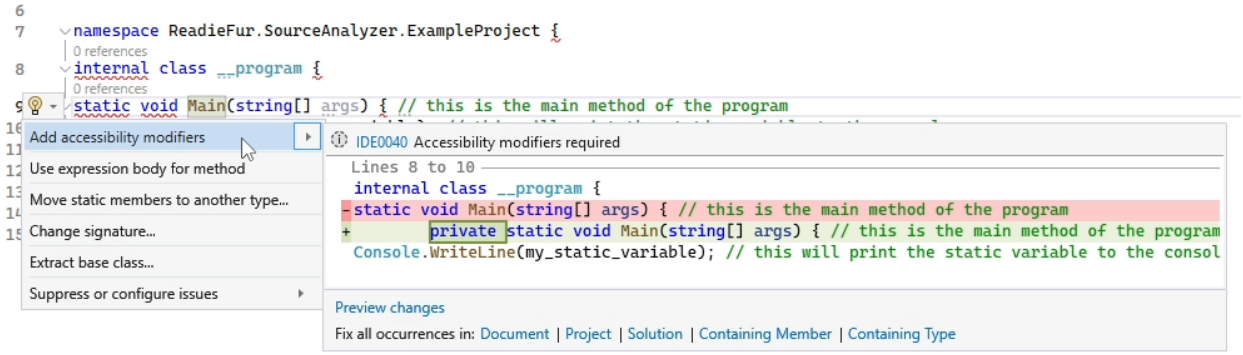


Of the automated tests conducted, fifty-eight were created of which fifty-five were successful. The three tests that failed were due to the line ending issue mentioned above. The tests that failed were not fixed as they were not major issues, would not affect the functionality of the tool and due to a lack of time to resolve the issue. The tests that passed were able to validate the functionality of the tool and ensure that the tool was functioning correctly and as expected. Given the success rate of 94.8%, it can be concluded that the tool is functioning well within a passable margin of error and is ready enough for real-world use.

3.4.2 Integration testing

Using the VSIX build of the project, a plugin compatible with the Visual Studio IDE gets created. This plugin allowed for the tool to be interacted with via IDE hints. In order to examine this a test project was loaded up into Visual Studio with the plugin loaded. After compiling the project the IDE would gather diagnostics from the build tool and display them in the error list and the code editor. It can be seen that amongst the diagnostics generated, the tool has provided hints within the IDE that the user can click on to preview and apply the fix. (see figure 6).

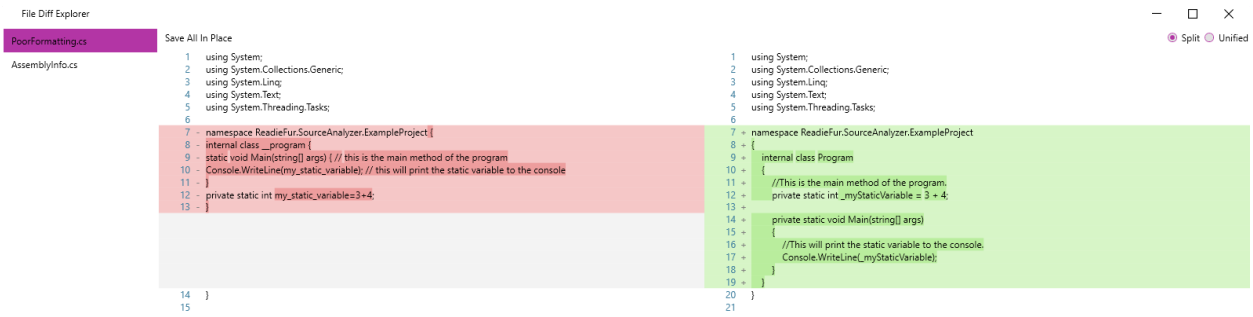
Figure 6: Integrated IDE Hints



3.4.3 Standalone testing

The standalone package was built to provide a way to run the tool without the need for the Visual Studio IDE. The standalone program is simplistic and takes a C# solution file from which the analysis can be run on. Any files that can have fixes applied will be displayed in a graphical window with a file differential view. To validate the results of the standalone tool, checks were made to ensure that it doesn't get stuck in a read-eval loop, which did occur during development. In addition to this, the output files were visually inspected to check that the fixes were applied correctly (see figure 7). Additional, less significant, tests were conducted for the standalone tool which can be read about in the appendix section 4.4.2.

Figure 7: Standalone Tool



3.5 Product evaluation

Throughout the development of the project, the goals that were set out to be solved have, for the most part, been completed. The basis of a tool to restructure source code had successfully been created. The program also shows that being able to easily restructure a programs source would drastically reduce the time required to format documents to a users preferences compared to if it were to be done manually.

The performance of the tool runs fast enough within an integrated environment that it would not hinder the containing process or the development for the end user. With various ways to run the program too, the

user can choose how they would like to run the program and are therefore less prone to vendor lock in.

By limiting the scope of the project to a single language, the project was able to be completed within the time frame set out. however, this has the downside of not being able to restructure code in other languages which is a particularly large limitation of this tool given that in recent years, web-based languages such as Typescript and Javascript as well as Python for AI based tasks, have become increasingly popular, so by not making this tool multi-language compatible, it cannot be used to aid in enforcing coding standards to these other popular languages. If the project had a longer time frame and delivery period, it would've been feasible to such a feature with more resources.

4 Epilogue

4.1 Summary

From the research conducted and the results that were obtained, it can be seen that the project has been successful in achieving the goals that were set out. The project has been able to create a tool that can restructure source code in a way that is easy to use and can be integrated into a user's workflow. The tool has been designed to be fast and efficient, and can be run in a variety of ways to suit the user's needs. The tool has been tested and evaluated, and has been found to perform well in a variety of scenarios. The project has also been able to demonstrate the benefits of using a tool like this, and has shown that it can save time and improve the quality of code. With more time and resources further benefits and efficiencies may have been achievable but given the constraints, the project has been a success and has achieved the goals that were set at the outset.

4.2 Conclusion

The project has been a valuable learning experience, and has provided a good opportunity to develop new skills and knowledge. A better understanding of the principles and patterns of object oriented programming as been gained and provided the opportunity to put these into practice. The project has also allowed the development of skills in software development, and has given me the opportunity to work on a real-world project. While challenging at times, it has been rewarding and has provided a sense of accomplishment, and has been a positive experience.

In the future with more time the aim would be to continue working on the project, to further develop the tool and add new features and functionality. A primary goal I would like to achieve for this tool is to make it multi-language compatible, as this would greatly increase the usefulness of the tool and make it more accessible to a wider range of users. Ideas of how such a change could be implemented is to provide a language schema in the configuration that would then be used to parse to source code into a universal AST that could then be used to restructure the code. This would also allow for a much more flexible tool that could be used on multiple languages and platforms, as currently the tool is limited to only working with C# code via the MSBuild pipeline which is limited to the Windows platform. The integration of a custom intermediatory AST parser would allow for the tool to be used on any platform and with any language. However, these are all developments that would enhance the tool created and this is only possible as the project has demonstrated that the use of a properly developed tool does allow for creating software quality and reformatting source code, which has been achievable here on the constrained project. It is exciting to see this succeed and know how this could further be improved.

4.3 References

Abbes, M., Khomh, F., Gueheneuc, Y.-G. & Antoniol, G. (2011), An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, *in* 'Software maintenance and

reengineering(CSMR), 2011 15th European conference on'.

Ashfaq, Q., Khan, R. & Farooq, S. (2019), A comparative analysis of static code analysis tools that check java code adherence to java coding standards, *in* '2019 2nd International Conference on Communication, Computing and Digital systems (C-CODE)', pp. 98–103.

Dewhurst, R., KirstenS, Bloor, N., Baso, S., Bowie, J., ch, R., EvgeniyRyzhkov, Iberiam, Ann.campbell, Ejohn20, Marcil, J., Schelin, C., Wang, J., Fabian, Achim, Wetter, D. & kingthorin (2021), 'Static code analysis'.
URL: *https://owasp.org/www-community/controls/Static_Code_Analysis*

Grano, G., Palomba, F. & Gall, H. C. (2019), Lightweight assessment of test-case effectiveness using source-code-quality indicators, *in* 'IEEE Transactions on Software Engineering'.

Khomh, F., Penta, M. D., Guéhéneuc, Y.-G. & Antoniol, G. (2012), An exploratory study of the impact of antipatterns on class change-and fault-proneness, *in* 'Empirical Software Engineering'.

Kirk, D., Tempero, E., Luxton-Reilly, A. & Crow, T. (2020), High school teachers' understanding of code style, *in* 'Proceedings of the 20th Koli Calling International Conference on Computing Education Research', Koli Calling '20, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/3428029.3428047*

Kummamuru, S. & Hussaini, S. W. (2015), Designing an organization structure for large and complex it programs using the viable system model(vsm), *in* 'TENCON 2015 - 2015 IEEE Region 10 Conference', pp. 1–5.

Liu, Y. (2019), Jsoptimizer: An extensible framework for javascript program optimization, *in* '2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)', pp. 168–170.

Neto, A., Bezerra, C. & Serafim Martins, J. (2022), Code smell co-occurrences: A systematic mapping, *in* 'Proceedings of the XXXVI Brazilian Symposium on Software Engineering', SBES '22, Association for Computing Machinery, New York, NY, USA, pp. 331–336.
URL: *https://doi.org/10.1145/3555228.3555268*

Palomba, F., Bavota, G., Penta, M. D., Oliveto, F. F. R. & Lucia, A. D. (2017), On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, *in* 'Empirical Software Engineering'.

Pecorelli, F., Palomba, F., Khomh, F. & De Lucia, A. (2020), Developer-driven code smell prioritization, *in* 'Proceedings of the 17th International Conference on Mining Software Repositories', MSR '20, Association for Computing Machinery, New York, NY, USA, pp. 220–231.
URL: *https://doi.org/10.1145/3379597.3387457*

Saraiva, J. (2013), A roadmap for software maintainability measurement, *in* '2013 35th International Conference on Software Engineering (ICSE)', pp. 1453–1455.

Singh, P., Singh, S. & Kaur, J. (2013), 'Tool for generating code metrics for c# source code using abstract syntax tree technique', *SIGSOFT Softw. Eng. Notes* **38**(5), 1–6.
URL: *https://doi.org/10.1145/2507288.2507312*

Stegeman, M., Barendsen, E. & Smetsers, S. (2014), Towards an empirically validated model for assessment of code quality, *in* 'Proceedings of the 14th Koli Calling International Conference on Computing Education Research', Koli Calling '14, Association for Computing Machinery, New York, NY, USA, pp. 99–108.
URL: *https://doi.org/10.1145/2674683.2674702*

Wikipedia contributors (2024), 'Regular expression — Wikipedia, the free encyclopedia', `https://en.wikipedia.org/w/index.php?title=Regular_expression&oldid=1218131380`. [Online; accessed 26-April-2024].

4.4 Appendices

This section contains additional information that is relevant and contains some significance to the project but does not directly relate to the main discussion of the report. This includes detailed and technical documentation such as tables of results, diagrams, program source code, etc.

4.4.1 Custom unit test file parser

The MSBuild testing framework requires a custom format for the test data as described by the documentation on the Microsoft website. The format that the framework desires is not ideal as it produces a lot of duplicated code, so to reduce this a custom source file parser was created to drastically cut down the amount of code that was required to write the tests. In this custom source file parser, the source file can contain code and diagnostic comments. The diagnostic comments are formatted as seen in Table 2.

Table 2: File Parser Syntax

Syntax	Description	Example
`//#<Diagnostic ID>`	Indicates the start of a diagnostic block	`//#RFSA0001`
`//-<Code>`	Indicates code that should be removed from the source file	`//- void Test()`
`//+<Code>`	Indicates code that should be added to the source file	`//+ private void Test()`
`//''<Code>''`	Indicates the area that the diagnostic should occur	`//- return 2''+''3;`

The file interpreter creates three copies of the source file in memory for the analyzer input, code fix input and expected output. The source file is then parsed for diagnostic comments using a regular expression. Once a diagnostic block is found, each copy of the source file is modified according to the syntax required by the MSBuild testing framework. For example, if a single source file had the following content:

```
using System;
//#0007
//- public class ''_class_name_''
//+ public class ClassName
{}
```

The file interpreter will then parse this to produce three files with the required syntax and data for each of MSBuild inputs as well as the diagnostics to be expected from the analyzer. The three files that are produced are as follows:

Expected diagnostic

```
ID: RFSA0007
Message:  Class name should match the pattern
"[A-Z][a-z]+"
Location:  Line 2, Column 13, Span:  12
```

Analyzer input

```
using System;
public class _class_name_
{}
```

Code fix input

```
using System;
public class {{|#0:_class_name_|}}
{}
```

Code fix expected output

```
using System;
public class ClassName
{}
```

Using this system the test data can be written in a more human-readable format with much less code duplication. This system also allows for the easy addition and modification for new and existing tests.

4.4.2 Manual testing

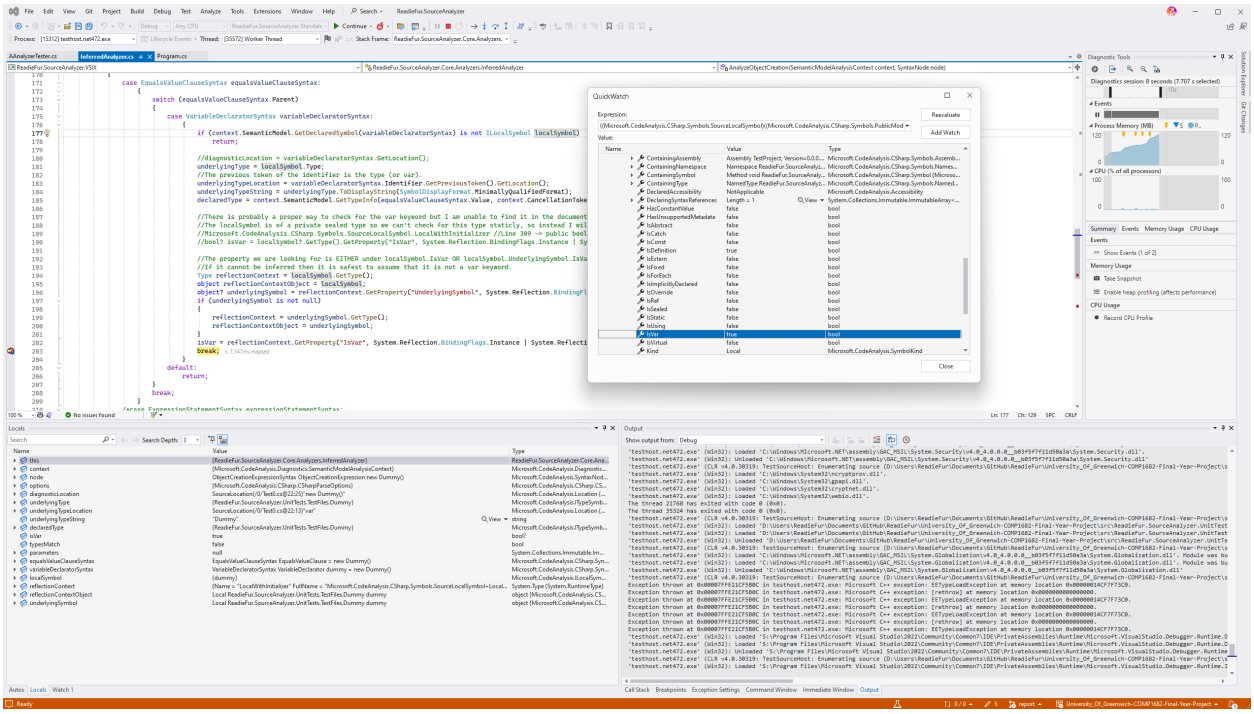
For the manual testing of the program a simple test solution was created, this test solution contained one instance of every error and was used as a worst case scenario example. This worst case scenario example was used to test the reliability of the program and in particular the standalone tool. It was initially feared that if a bug was present or the analyzer would find its code fixes invalid, the potential for a stack overflow could occur as the code would be fixed, re-evaluated and fixed again over and over due to the nature of the analyzers having to run in a read-eval loop to fix the code one by one. Fortunately this was not the case and the program was able to fix all the errors in the test solution without any issues. The program was also tested against the example solutions provided by Microsoft under the MIT license which allows me full permission to use their code as test data for my program, though results for this were not recorded as the test solution was deemed sufficient for testing purposes.

4.4.3 Debugging research process

Throughout the development of specific parts of the program, certain areas of the program were difficult to implement due to the lack of documentation and examples available. So a debugging research process was

created to help with this. The debugging research process is a rather tedious process but when conducted correctly, it can provide a lot of insight into the inner workings of the program and can help to identify and fix bugs in the program. In figure 8, an example is shown of the debugging research process in action where undocumented runtime objects are inspected to see if they hold values of any use to the program. In cases like these useful data from undocumented libraries can be obtained through debugging during runtime or through the use of decompilers like DnSpy and the built-in Visual Studio decompiler in order to see static data and members of a library.

Figure 8: Debugging research process



4.4.4 Additional figures

Figure 9: Project Timeline

(a) Timeline table

ID	Task Name	Duration	Start	Finish	Predecessors	Deadline
1	**Project Proposal**	**262.5 hrs**	**Mon 16/10/23**	**Mon 27/11/23**		**Mon 27/11/23**
2	**Preperation**	**222.5 hrs**	**Mon 16/10/23**	**Mon 20/11/23**		**NA**
3	Domain research	160 hrs	Mon 16/10/23	Thu 09/11/23		NA
4	Structure resaerched information	54.5 hrs	Fri 10/11/23	Mon 20/11/23	3	NA
5	**Writeup**	**46.5 hrs**	**Mon 20/11/23**	**Thu 23/11/23**	**2**	**NA**
6	Introduction	1 hr	Wed 22/11/23	Wed 22/11/23	9	NA
7	Problem domain	3 hrs	Tue 21/11/23	Tue 21/11/23	2	NA
8	Methodology	7 hrs	Tue 21/11/23	Wed 22/11/23	7	NA
9	Evaluation	2 hrs	Wed 22/11/23	Wed 22/11/23	8	NA
10	Peer review	10 hrs	Wed 22/11/23	Thu 23/11/23	6,7,8,9	NA
11	Submit proposal	10 mins	Mon 27/11/23	Mon 27/11/23	5	NA
12	**Project planning**	**70.5 hrs**	**Wed 13/12/23**	**Thu 21/12/23**	**1**	**NA**
13	Goals	24 hrs	Wed 13/12/23	Thu 14/12/23		NA
14	Methodology	12 hrs	Fri 15/12/23	Mon 18/12/23	13	NA
15	Language choice	5 mins	Mon 18/12/23	Mon 18/12/23	14	NA
16	Success metrics	24 hrs	Mon 18/12/23	Thu 21/12/23	15	NA
17	**Initial designs**	**110.5 hrs**	**Tue 02/01/24**	**Wed 17/01/24**	**12**	**NA**
18	Prototyping	96 hrs	Tue 02/01/24	Tue 16/01/24		NA
19	Discoveries	8 hrs	Wed 17/01/24	Wed 17/01/24	18	NA
20	**Contextual report**	78.5 hrs	Wed 17/01/24	Fri 26/01/24	**12,17**	**Mon 29/01/24**
21	Writeup	48 hrs	Wed 17/01/24	Tue 23/01/24		NA
22	Peer review	24 hrs	Wed 24/01/24	Fri 26/01/24	21	NA
23	**Development (CI)**	**502.5 hrs**	**Mon 29/01/24**	**Sun 21/04/24**	**17**	**NA**
24	Research feature	24 hrs	Mon 29/01/24	Tue 30/01/24		NA
25	Impliment feature	96 hrs	Wed 31/01/24	Thu 15/02/24	24	NA
26	Document feature	2 hrs	Fri 16/02/24	Fri 16/02/24	25	NA
27	Create unit tests	2 hrs	Fri 16/02/24	Fri 16/02/24	26	NA
28	Troubleshoot feature	60 hrs	Fri 16/02/24	Tue 27/02/24	27	NA
29	Update goals	5 hrs	Wed 28/02/24	Wed 28/02/24	28	NA
30	Track changes	30 mins	Wed 28/02/24	Wed 28/02/24	29	NA
31	**Final report**	**110.5 hrs**	**Mon 22/04/24**	**Tue 07/05/24**	**23**	**NA**
32	Add previous report content	30 mins	Mon 22/04/24	Mon 22/04/24		NA
33	Add research & discoveries made	5 hrs	Mon 22/04/24	Mon 22/04/24	32	NA
34	Discuss development process	12 hrs	Mon 22/04/24	Wed 24/04/24	33	NA
35	Discuss feature implimentation	12 hrs	Wed 24/04/24	Thu 25/04/24	34	NA
36	Analyze project success	5 hrs	Thu 25/04/24	Fri 26/04/24	35	NA
37	Peer review	24 hrs	Fri 26/04/24	Wed 01/05/24	36	NA
38	Revise sections	24 hrs	Wed 01/05/24	Mon 06/05/24	**37**	NA
39	**Submit project**		**Tue 07/05/24**		**23,31**	**Tue 07/05/24**
40	Archive code	1 hr				NA
41	Render report	10 mins				NA
42	Verify submission contents	1 hr			40,41	NA
43	Upload & submit work	30 mins			42	NA
44	**Presentation**		**Wed 08/05/24**			**Thu 09/05/24**
45	Pick out key features	6 hrs				NA
46	Create script template	6 hrs			45	NA
47	Rehearse	8 hrs			46	NA
48	Prepare for questions	8 hrs			47	NA
49	Record presentation	1 hr			48	NA

(b) Gantt chart

Figure 10: Using GitHub to track changes and issues

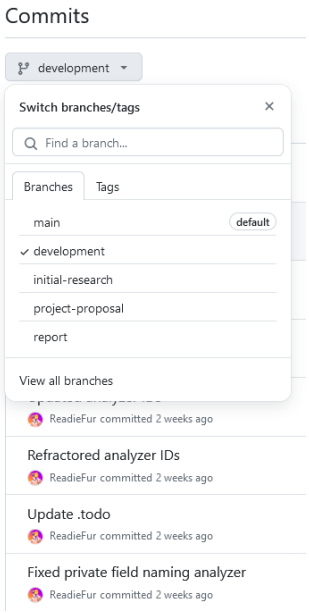


Figure 11: YAML Schema

```
                },
                "object_structure":
                {
                    "type": "object",
                    "description": "The structure of objects in the source code.",
                    "properties":
                    {
                        "severity": { "$ref": "#/definitions/severity" },
                        "properties_at_top":
                        {
                            "type": "boolean",
                            "description": "Whether to place properties at the top of the object."
                        }
                    }
                }
            }
        },
        "implicit":
        {
            "type": "object",
            "description": "Implicit rules for the source code.",
            "properties":
            {
                "this": { "$ref": "#/definitions/inferred_block" },
                "access_modifier": { "$ref": "#/definitions/inferred_block" },
                "constructor": { "$ref": "#/definitions/inferred_block" },
                "type": { "$ref": "#/definitions/inferred_block" }
            }
        }
    }
}
```

Figure 12: Regex Token Recursive Parse

(a) Group Construct Recursive Parse

```
//Iterate over each item in the group until we find the closing parenthesis.
AToken endToken = RecursiveParse(
[
    typeof(GroupConstruct),
    typeof(CharacterClass),
    typeof(MetaSequence),
    typeof(Atom),
], ref consumablePattern, ')');
```

(b) Recursive Token Parser

```
/// <summary>
/// Recursively parses a group of tokens.
/// </summary>
/// <param name="types"></param>
/// <param name="consumablePattern"></param>
/// <param name="returnOnChar"></param>
/// <returns></returns>
/// <exception cref="InvalidOperationException"></exception>
protected AToken RecursiveParse(Type[] types, ref string consumablePattern, char returnOnChar)
{
    //Iterate over each item in the group until we find the closing parenthesis.
    bool isValid = false;
    AToken endToken = this;
    while (consumablePattern.Length > 0)
    {
        if (consumablePattern[0] == returnOnChar)
        {
            isValid = true;
            consumablePattern = consumablePattern.Substring(1);
            break;
        }

        AToken? token = null;
        foreach (Type type in types)
        {
            token = (AToken?)Activator.CreateInstance(type);
            if (token?.CanParse(ref consumablePattern) is not null)
                break;
        }
        if (token is null)
            throw new InvalidOperationException();

        Children.Add(token);
        token.Parent = this;
        token.Previous = endToken;
        endToken.Next = token;
        endToken = token.Parse(ref consumablePattern);
    }
    if (!isValid)
        throw new InvalidOperationException();

    return endToken;
}
```

Figure 13: Character Range Conform

```
public override bool Conform(ConformParameters parameters)
{
    //Consume characters until a letter is found.
    char c;
    do c = Read(parameters.Input, ref parameters.Index, 1)[0];
    while (!char.IsLetter(c) && parameters.Index < parameters.Input.Length);
    if (parameters.Index > parameters.Input.Length)
        return false;

    //The following operation is only applicable to letters.

    //If the character is already in the range then just add it to the output.
    if (c >= _start && c <= _end)
    {
        parameters.Output += c;
        return true;
    }

    //Check if the character is valid by changing the case.
    c = char.IsUpper(c) ? char.ToLower(c) : char.ToUpper(c);
    if (c >= _start && c <= _end)
    {
        parameters.Output += c;
        return true;
    }
    else
    {
        //index--;
        return false;
    }
}
```

Figure 14: Quantifier Conform Parameters

```
//If this is a greedy quantifier then we need to check against the provided options...
if (!isGreedy)
    continue;

char nextChar = parameters.Input[parameters.Index];
char previousChar = parameters.Input[parameters.Index - 1];

//Check if we should be splitting on character case changes.
if ((options.GreedyQuantifiersSplitOnCaseChangeToUpper!.Value || options.GreedyQuantifiersSplitOnCaseChangeToLower!.Value)
    && parameters.LastGreedyQuantifiersSplitOnCaseChange != parameters.Index
    //Make sure that the previous and next characters are letters.
    && char.IsLetter(previousChar)
    && char.IsLetter(nextChar)
    //Check if the case of the characters change and match the conditions.
    && ((options.GreedyQuantifiersSplitOnCaseChangeToUpper!.Value && char.IsLower(previousChar) && char.IsUpper(nextChar))
        || (options.GreedyQuantifiersSplitOnCaseChangeToLower!.Value && char.IsUpper(previousChar) && char.IsLower(nextChar)))
)
{
    //We need to keep track of when we split on case change so that we don't split on the same case change twice as this can occur for nested quantifiers.
    parameters.LastGreedyQuantifiersSplitOnCaseChange = parameters.Index;
    break;
}

if (options.GreedyQuantifiersSplitOnAlphanumericChange!.Value
    && char.IsLetterOrDigit(previousChar) != char.IsLetterOrDigit(nextChar))
    break;

bool encounteredDelimiter = false;
foreach (char delimiter in options.GreedyQuantifierDelimiters!)
{
    if (nextChar != delimiter)
        continue;

    //As the character is a delimiter, we must consume it.
    lastSuccessfulIterationIndex = ++parameters.Index;
    encounteredDelimiter = true;
    break;
}
if (encounteredDelimiter)
    break;
```