Source Code Quality Enhancer

Tristan Percy Read

001151378

Project Supervisors:

Cornelia Boldyreff

John Ewer

BEng (Hons) Software Engineering

University of Greenwich

07/02/2024

Abstract

This paper discusses 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 of software engineering at the University of Greenwich. The motivation for researching in this area was due to an interest in creating tools that aid programmers in creating software as individuals and in a team, so that uniformity can be maintained, increasing the readability of a program that may one day be discontinued or passed onto another developer or set of developers.

Acknowledgements

Contents

1 Introduction **3**
- 1.1 Background . . . 3
- 1.2 Aims and objectives . . . 3
- 1.3 Project overview . . . 3
- 1.4 Justification . . . 3

2 Literature review **4**
- 2.1 Domain research . . . 4
  - 2.1.1 Introduction . . . 4
  - 2.1.2 Problem domain . . . 4
  - 2.1.3 Review . . . 5
  - 2.1.4 Conclusions . . . 5
- 2.2 Methodology . . . 5
  - 2.2.1 Related work . . . 5
  - 2.2.2 Defining software quality . . . 5
  - 2.2.3 Considerations . . . 6
  - 2.2.4 Requirements . . . 6

2.3 Design approach . . . 6
2.3.1 Overview . . . 6
2.3.2 Techinical details . . . 6
2.3.3 Prototypes . . . 7
2.3.4 Reflection . . . 8

3 Project development **9**
3.1 Evaluation . . . 9
3.2 Design philosophy . . . 9
3.2.1 Object oriented programming principles and design patterns . . . 9
3.3 Integration . . . 10
3.3.1 User defined configuration . . . 10
3.3.2 Naming engine . . . 12
3.3.3 Document formatting . . . 13
3.3.4 Language features . . . 14
3.4 Testing . . . 16
3.4.1 Unit testing . . . 16
3.4.2 Integration testing . . . 18
3.4.3 Standalone testing . . . 18
3.5 Product evaluation . . . 18

4 Closing chapters **19**
4.1 Summary . . . 19
4.2 Conclusion . . . 19
4.3 References . . . 19
4.4 Appendices . . . 20

List of Tables

List of Figures

1 Token Breakdown . . . 7
2 YAML Schema . . . 10
3 YAML Configuration . . . 11
4 VSIX Configuration manager . . . 11

1 Introduction

1.1 Background

This project will seek 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 combination with design patterns to create a well structured and higher quality codebase from a pre-existing codebase.

1.2 Aims and objectives

The goals of this project are to create a tool that can: interpret source code; reformat it to a user defined style; support object oriented programming principles; supply integration with existing popular IDEs and provide a user interface for ease of use.

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 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.

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 and thereby cause 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.

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 software quality using papers and resources found primarily on on IEEE Xplore and ACM Digital Library. It had been found that there is a lack of standardization in the way software quality is measured and that there is a demand for high quality code. Additionally, while there were many proposed approaches to measuring software quality, not many of them have not been widely adopted. This project aims to create a solution to these problems by making a tool that can analyze source code and format it to a user-defined schema while making use of object oriented programming principles.

2.1.2 Problem domain

The object oriented programming paradigm has become one of the most popular programming paradigms 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). This can make 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 the problems around the topic 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.

Further research into the area of software quality shows that teachers are struggling to teach students how to write high quality code (Kirk et al. 2020). This can have a knock on effect into the industry as students will be entering the industry with a lack of understanding of how to write high quality code, and so poorly written code may become 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. 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 which is causing issues within the industry.

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.

2.1.4 Conclusions

My 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 to tool will attempt to adhere to.

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:

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

While tools like these exist and can be very helpful in conjunction with each other, given how many tools there are it can be 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 is an important quote as it 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

2.2.4 Requirements

2.3 Design approach

2.3.1 Overview

The task will be 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. In order to do this, use will be made of various static code analysis techniques.

For this task the 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.

In order for the tool to work 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 is "the degree of conformance to explicit or implicit requirements and expectations". This definition for the project by allowing a user to define a schema that sets the requirements and expectations of the codebase. This way 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. In addition to this, 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 is another important metric because it 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, which can then save 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 the links between classes and data within a program. By using a node graph it is possible to see how data is linked between classes and methods. This also gives us 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 be used to detect code smells such as duplicated code, long methods, poor flow control, cyclical dependencies, etc.

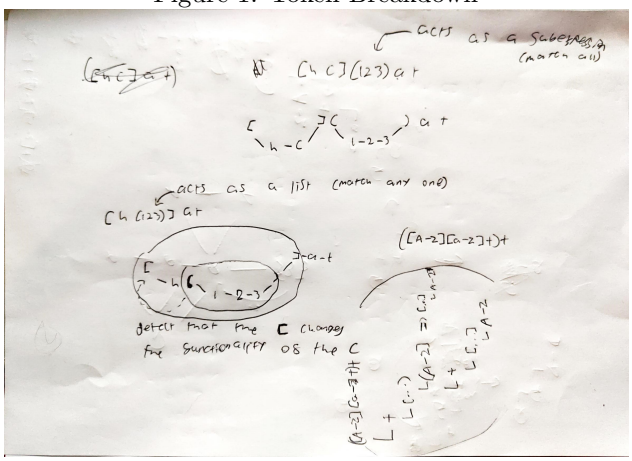
By combining the use of ASTs and node graphs an algorithm can be made to detect the relationship between classes and methods as well as the flow of the program, then the algorithm can decide how to modify and restructure the source code. It is important that both relation and flow are taken into consideration 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 is a good candidate for providing the schema to the tool as it is a human friendly data format, another options is JSON however this format is not as human friendly. 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 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. Regular expressions (regex) are 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 matching to a string and performing simple replacements on matches within a string. 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) was investigated. Theoretical prototypes were created on paper to show how the regex pattern could be deconstructed and reconstructed (see Figure 1). From this syntax, a simplified set was created that would be used for the new regex engine (see Table 2.3.3). This simplified expression set was picked to be simple to implement and easy to read. An advanced language set is not nessecaraly needed either for matching single strings on object names, however there are cases where it could be useful.

Figure 1: Token Breakdown



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

2.3.4 Reflection

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.

3 Project development

3.1 Evaluation

In order to evaluate this tool the following will be assessed:

- Function.
- Comprehensibility.
- 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.

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 to run each. While this tool will not aim to optimize the performance of the code, it should not make it worse.

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.

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;
- Structure - Decomposition & modularization;

My proposal will want to 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 extremely 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 they said "Modularization of code marks better reuse of the code and compilation time".

However it is important that one doesn't abuse these abilities as they can lead to code smells which affect program 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 about the use of these principles and patterns such as abstraction and inheritance and automated comments as they can lead to these undesirable code smells.

3.3 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.

In this section we will discuss 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 discussed in the literature review, due to the lack of standardization in code formatting, the tool should be able to be configured by the user. This is 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 us full permission to use the library in this project.

YAML was chosen as the configuration language due to its human readability, it is 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 2: 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 2).

Like in 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, in the visible snippet the configuration for the custom naming engine is shown. 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 you may want the work environment to be, you can set the severity 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, when the configuration is loaded varies by the implementing program. Due to the limitations of certain platforms, the configuration class is 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, this is because 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, this 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 4).

Figure 4: VSIX Configuration manager

```
        public AConfigManager(string ipcName)
        {
            _sharedMemory = MemoryMappedFile.CreateOrOpen(ipcName + "_memory", MaxBufferSize);
            SharedMemoryAccessor = _sharedMemory.CreateViewAccessor();
        }

        public virtual void Dispose()
        {
            _sharedMemory.Dispose();
        }

        protected string ReadSharedMemory()
        {
            ushort size = SharedMemoryAccessor.ReadUInt16(54);
            byte[] buffer = new byte[size];
            SharedMemoryAccessor.ReadArray(54 + 2, buffer, 0, buffer.Length);
            return Encoding.ASCII.GetString(buffer);
        }

        protected void WriteSharedMemory(string data)
        {
            byte[] Buffer = Encoding.ASCII.GetBytes(data);
            SharedMemoryAccessor.Write(54, (ushort)Buffer.Length);
            SharedMemoryAccessor.WriteArray(54 + 2, Buffer, 0, Buffer.Length);
        }

        public abstract ConfigRoot GetConfiguration();
    }
}
```

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 build-in C# regex engine, if 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.

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 necessitated 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 provided regex pattern 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 we can either navigate up the tree or back one token to find the relevant token or set that needs to be quantified.

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, we more often than not want to remove mismatched literals with a blank. To provide an example of this, say we want to conform the input `__my_variable` to the pattern `([A-Z][a-z]+)+`, which should produce a desired output of `MyVariable`. If mismatched tokens were not dropped the algorithm would fail to conform to the pattern because 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`.

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, 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 API. We have to navigate and interpret the AST as necessary for each of the formatting options.

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 we can determine what line the comment is on, if the user has configured comments to be on a new line then we can navigate backwards through the AST to check if there are any non-whitespace tokens on the same line as the comment, if there are then we can insert a new line token before the comment and add any necessary indentation. We can make some minor performance improvements by not checking tokens after the comment as single-line comments defined by using the trivia \\ will always be the last non-whitespace token a line.

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. This method allows for the tool to determine the correct indentation level for each line in the document. We cannot do a simple text search for the tokens as they may be part of a string or comment, therefore we have to navigate the AST to find the tokens. Additionally the indentation level cannot be 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 our custom analyzers are configured to only be run on syntactically valid C# code.

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 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 **??**).

The brace location works 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 1 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 huge impact on readability and debugging as it contributes massively to reducing the amount of spaghetti code. Spaghetti code is code that is difficult to read and understand due to poor structure, this 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. Like before the sub-module will traverse the abstract syntax tree, 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, from this we 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`, 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. While 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, if the node is a top level node the access modifier will be `internal`, 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 who's 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 you would have to write `var obj = new MyObject();`, with the new syntax you 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, we cannot work on the AST directly as we need context of the compilation to determine the type of the object that is being created, therefore we instead hook to the compilation event provided by the Roslyn API. From this context we check 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 we can check the parent of the object creation expression to see if it is a variable declaration, if it is then we check to see if the expression is being assigned to a `var` keyword or a type, if it is being assigned to a `var` keyword we can determine the type of the object 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 we need to find the definition of the method call with the relevant overload that the object creation expression is being assigned to. Once the method call is found via the compilation context, we must acquire the index of the argument that the object creation expression is being assigned to, from this index we can determine the type of the object that is being created. 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, 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.

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 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.

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 and 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. This 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 reduce 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 3.4.1.

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 we can provide a single source file with the following content:

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

Which the file interpreter will then parse 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. 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.

3.4.2 Integration testing

3.4.3 Standalone testing

3.5 Product evaluation

4 Closing chapters

4.1 Summary

4.2 Conclusion

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