

# 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**

**Preface**

**Acknowledgements**

## **Contents**

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Background . . . . .	3
1.2	Aims and objectives . . . . .	3
1.3	Project overview . . . . .	3
1.4	Justification . . . . .	3
<b>2</b>	<b>Literature review</b>	<b>4</b>
2.1	Introduction . . . . .	4
2.2	Problem domain . . . . .	4
2.3	Review . . . . .	5
2.4	Conclusions . . . . .	5
<b>3</b>	<b>Methodology</b>	<b>6</b>
3.1	Related work . . . . .	6
3.1.1	Defining software quality . . . . .	6
3.2	Considerations . . . . .	6
3.3	Requirements . . . . .	6
<b>4</b>	<b>Design approach</b>	<b>7</b>
4.1	Overview . . . . .	7
4.2	Technical details . . . . .	7
4.3	Prototypes . . . . .	8
4.4	Reflection . . . . .	8
4.4.1	Further research . . . . .	8
<b>5</b>	<b>Project management</b>	<b>9</b>
5.1	Evaluation . . . . .	9
5.2	Design philosophy . . . . .	9
5.2.1	Object oriented programming principles and design patterns . . . . .	9
<b>6</b>	<b>Integration and testing</b>	<b>11</b>
6.1	Integration . . . . .	11
6.2	Testing . . . . .	11

6.3	Product evaluation . . . . .	11
<b>7</b>	<b>Closing chapters</b>	<b>12</b>
7.1	Summary . . . . .	12
7.2	Conclusion . . . . .	12
7.3	References . . . . .	12
7.4	Appendices . . . . .	13

## List of Tables

## List of Figures

# **1 Introduction**

## **1.1 Background**

## **1.2 Aims and objectives**

## **1.3 Project overview**

## **1.4 Justification**

## 2 Literature review

### 2.1 Introduction

This project proposal 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, and 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. It was also found that while there are many proposed approaches to measuring software quality, not many of them have not been widely adopted. The proposed solution is to create a tool that can analyze source code and format it to a schema while making use of object oriented programming principles.

### 2.2 Problem domain

The object oriented programming paradigm has become one of the most popular programming paradigms in the industry (Saraiva 2013). The wide adoption 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 (Kumnamuru & 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, Kumnamuru & 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.

## **2.3 Review**

The proposed 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.

## **2.4 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 the tool will attempt to adhere to.

## 3 Methodology

### 3.1 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).

#### 3.1.1 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".

### 3.2 Considerations

### 3.3 Requirements

## 4 Design approach

### 4.1 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.

### 4.2 Technical 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.

### **4.3 Prototypes**

### **4.4 Reflection**

#### **4.4.1 Further research**

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.

## 5 Project management

### 5.1 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.

### 5.2 Design philosophy

#### 5.2.1 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 inheritance 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, Abbas 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.

## 6 Integration and testing

### 6.1 Integration

### 6.2 Testing

### 6.3 Product evaluation

## 7 Closing chapters

### 7.1 Summary

### 7.2 Conclusion

### 7.3 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](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>
- Kumnamuru, 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, p. 99–108.  
**URL:** <https://doi.org/10.1145/2674683.2674702>

## 7.4 Appendices