A method for improving maintainability in object-oriented software applications

F Khan

orcid.org/0000-0001-9902-7374

Dissertation submitted in fulfilment of the requirements for the degree Master of Engineering in Computer and Electronic Engineering at the North-West University

Supervisor: Dr JC Vosloo

Graduation Ceremony: May 2019

Student number: 29901456
ABSTRACT

Title: A Method for Improving Maintainability in Object-Oriented Software Applications

Author: Faiyaaz Khan

Supervisor: Dr JC Vosloo

Keywords: Maintainability; Software; Rapid Development Environment; Object-Oriented; Code Metrics; Refactoring; Maintainability Assessment; Testing

The prevailing thesis of software development in today’s day and age is that of developing software in rapid development environments. This practice has resulted in software developers achieving higher customer satisfaction rates, faster times to market and a plethora of other benefits. However, Rapid Development Environments do introduce problems. The primary issue that arises is a lack of maintainability. To achieve rapid delivery of software, developers are forced to make decisions to ignore practices that improve software maintainability and make use of crude solutions when implementing software functionality. This is known as introducing technical debt. This thesis aims to improve software maintainability by means of the formulation and testing of a method to improve maintainability via remedying technical debt in object-oriented software applications.

To determine the details of the method to improve maintainability in object-oriented software applications, a literature study is conducted. The literature study is performed to determine techniques for identifying areas of low maintainability, techniques for improving software maintainability and techniques for assessing the maintainability of object-oriented software applications. Findings within the literature study are used to determine refined method requirements as well as formulate the steps and processes used within the method.

The method identifies areas of low maintainability via the usage of code metrics at different application levels and the usage of a third-party tool. Remediying areas of low maintainability consists of two steps: testing and refactoring. Testing is accomplished by ensuring that identified areas are testable and creating automated unit, integration or behavioural tests exercising the identified area. The automated test suite acts as a safety-net preserving existing application functionality while changes are made to the application. Refactoring is accomplished by identifying and remedying anti-patterns, grime and code smells. The refactoring step is the most important part of the method as it is responsible for improving the maintainability of the application. A maintainability assessment is performed using any combination of three distinct methods.

After developing the method to improve maintainability, the method is verified and validated. The method is verified against refined method requirements through a survey of software developers with at the minimum, a degree related to software development and varying ranges of development experience. All responses received
provide positive results in terms of the verification of the method. This led to the conclusion that the method can be applied to a real-world application for validation purposes.

The method is validated by applying it to an Android data collection application. The overall method objective is evaluated for the Android data collection application through the usage of the maintainability index which improved by 9.92%. A more granular look at the change in maintainability index revealed that the method influences the three aspects that affect source code maintainability: the source code control structure; the source code information structure; and the source code typography, naming and commenting.
ACKNOWLEDGEMENT

I would like to extend my gratitude to:

- ETA Operations for funding the research and providing all the data and computational resources.
- Prof. E.H. Mathews and Prof. M. Kleingeld for providing funding and the opportunity to do this work.
- Dr. J.C. Vosloo, Dr. SW van Heerden and Dr Johan du Plessis for their insight and guidance.
- My parents for their endless support throughout my studies and always believing in me.
CONTENTS

1 Introduction ......................................................................................................................... 2
   1.1 Background.................................................................................................................... 2
   1.2 Problems in Rapid Development Environments......................................................... 3
   1.3 Contextualising Technical Debt and Maintainability............................................... 6
   1.4 Technical Debt Types .................................................................................................. 9
   1.5 Object-Oriented Software Applications .................................................................. 10
   1.6 Problem Statement ..................................................................................................... 11
   1.7 Study Objectives ......................................................................................................... 11
   1.8 Study Overview .......................................................................................................... 12

2 Literature Study .................................................................................................................. 15
   2.1 Preamble ...................................................................................................................... 15
   2.2 Identification of Areas for Improving Maintainability in Software Applications .......... 15
       2.2.1 Indicators of Low Maintainability ....................................................................... 15
       2.2.2 Code Metrics ........................................................................................................ 18
       2.2.3 Automated tools .................................................................................................. 24
   2.3 Improving Maintainability in Software Applications .................................................. 25
       2.3.1 Refactoring .......................................................................................................... 25
       2.3.2 Testing .................................................................................................................. 28
   2.4 Assessing the Maintainability of Software Applications ........................................... 32
   2.5 Assessing a Method to Improve Maintainability ....................................................... 36
   2.6 Conclusion .................................................................................................................... 39

3 A Method for Improving Maintainability in Object-Oriented Software Applications .......... 42
   3.1 Preamble ...................................................................................................................... 42
   3.2 Method Overview ........................................................................................................ 43
   3.3 Identification of Areas of Low Maintainability .......................................................... 44
       3.3.1 Overview .............................................................................................................. 44
       3.3.2 Application Quality Assessment ........................................................................... 45
       3.3.3 Sonarqube Assessment ....................................................................................... 47
   3.4 Automated Testing of Identified Areas of Low Maintainability .................................... 52
   3.5 Refactoring Identified Areas of Low Maintainability ................................................... 55
       3.5.1 The Refactoring Process ...................................................................................... 55
       3.5.2 Sonarqube Refactoring Process ......................................................................... 57
   3.6 Maintainability Assessment ........................................................................................... 62
       3.6.1 Overview .............................................................................................................. 62
       3.6.2 Application Quality Assessment ........................................................................... 62
       3.6.3 Sonarqube Assessment ....................................................................................... 63
       3.6.4 ISO 25010 Sub-characteristic Assessment .......................................................... 64
   3.7 Conclusion .................................................................................................................... 67

4 Method Verification, Validation and Analysis of Results ..................................................... 69
   4.1 Preamble ...................................................................................................................... 69
   4.2 Survey Development .................................................................................................... 69
LIST OF FIGURES

Figure 1: MOOD Metrics .................................................................................................................. 19
Figure 2: Martin Packaging Metrics ............................................................................................... 22
Figure 3: CK Metrics ....................................................................................................................... 23
Figure 4: ISO 9126 Product Quality Model .................................................................................... 34
Figure 5: Target Software System Maintainability Hierarchy [103] .................................................. 37
Figure 6: A Method for Improving Maintainability in Object-Oriented Software Applications ......... 43
Figure 7: Identification Process ..................................................................................................... 44
Figure 8: Application Quality Assessment ..................................................................................... 46
Figure 9: Sonarqube Assessment .................................................................................................. 48
Figure 10: Sonarqube Homepage View .......................................................................................... 49
Figure 11: Sonarqube Code View Page (Package/Folder Level) ...................................................... 51
Figure 12: Sonarqube Code View Page (File/Class Level) ................................................................ 51
Figure 13: Testing Process ............................................................................................................ 54
Figure 14: Refactoring Process ..................................................................................................... 56
Figure 15: Sonarqube Code View Page (Single File/Class) ............................................................. 57
Figure 16: Sonarqube Refactoring Process .................................................................................... 58
Figure 17: Sonarqube Issues Page ................................................................................................ 59
Figure 18: Example Issue ............................................................................................................. 61
Figure 19: Sonarqube In-depth Issues Pop-up ............................................................................... 62
Figure 20: General Process for Application Quality Assessment .................................................. 63
Figure 21: ISO 25010 Quality Verification Process ......................................................................... 65
Figure 22: Final Question Responses — Survey Section 1 ............................................................. 73
Figure 23: A Method for Improving Maintainability in Object-Oriented Software Applications ....... 83
Figure 24: Maintainability Index Before and After Method Application ........................................ 95
Figure 25: Change in Cyclomatic Complexity ............................................................................. 96
Figure 26: Change in Lines of Code ............................................................................................. 97
Figure 27: Change in Halstead Volume ......................................................................................... 97
LIST OF TABLES

Table 1: ISO 25010 Quality Characteristics [7] ................................................................. 5
Table 2: Engineering Debt Triggers and Conditions [5] ...................................................... 8
Table 3: SOLID Principles [47] ....................................................................................... 31
Table 4: Maintainability Characteristics and Applicable Code Metrics .......................... 35
Table 5: Code Metric Definitions ..................................................................................... 36
Table 6: Sonarqube Example Issue Details .................................................................. 61
Table 7: Maintainability Characteristics and Applicable Code Metrics .......................... 66
Table 8: Code Metric Definitions ..................................................................................... 67
Table 9: Method Overview — Survey Results ................................................................. 74
Table 10: Identification — Survey Results ..................................................................... 75
Table 11: Application Quality Assessment — Survey Results ....................................... 76
Table 12: Sonarqube Assessment — Survey Results ....................................................... 76
Table 13: Testing — Survey Results ................................................................................. 77
Table 14: The Refactoring Process — Survey Results .................................................... 78
Table 15: Sonarqube Refactoring — Survey Results ...................................................... 78
Table 16: Sonarqube Issue Page — Filter Usefulness ...................................................... 79
Table 17: Maintainability Assessment — Survey Results ............................................... 79
Table 18: Maintainability Assessment Method Combination Popularity ....................... 80
Table 19: Application Quality Assessment Files Ordered by Maintainability ............... 84
Table 20: Sonarqube Assessment Files Ordered by Maintainability .............................. 85
Table 21: Metrics Used for Each Maintainability Sub-characteristic ............................... 91
Table 22: Maintainability Index Before Method Application .......................................... 93
Table 23: Maintainability Index After Method Application .............................................. 94
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDE</td>
<td>Rapid Development Environment</td>
</tr>
<tr>
<td>XP</td>
<td>eXtreme Programming</td>
</tr>
<tr>
<td>TD</td>
<td>Technical Debt</td>
</tr>
<tr>
<td>OOPSLA</td>
<td>Object-Oriented Programming, Systems, Languages and Applications</td>
</tr>
<tr>
<td>ACM</td>
<td>Association for Computing Machinery</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organisation for Standardisation</td>
</tr>
<tr>
<td>Ca</td>
<td>Afferent Coupling</td>
</tr>
<tr>
<td>Ce</td>
<td>Efferent Coupling</td>
</tr>
<tr>
<td>DIT</td>
<td>Depth of Inheritance Tree</td>
</tr>
<tr>
<td>MOOD</td>
<td>Metrics for Object-Oriented Design</td>
</tr>
<tr>
<td>AHF</td>
<td>Attribute Hiding Factor</td>
</tr>
<tr>
<td>AIF</td>
<td>Attribute Inheritance Factor</td>
</tr>
<tr>
<td>MHF</td>
<td>Method Hiding Factor</td>
</tr>
<tr>
<td>MIF</td>
<td>Method Inheritance Factor</td>
</tr>
<tr>
<td>CF</td>
<td>Coupling Factor</td>
</tr>
<tr>
<td>PF</td>
<td>Polymorphism Factor</td>
</tr>
<tr>
<td>CK</td>
<td>Chidamber-Kemerer</td>
</tr>
<tr>
<td>CBO</td>
<td>Coupling Between Objects</td>
</tr>
<tr>
<td>LCOM</td>
<td>Lack of Cohesion of Methods</td>
</tr>
<tr>
<td>NOC</td>
<td>Number of Children</td>
</tr>
<tr>
<td>WMC</td>
<td>Weighted Method Complexity</td>
</tr>
<tr>
<td>CC</td>
<td>Cyclomatic Complexity</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>SRP</td>
<td>Single Responsibility Principle</td>
</tr>
<tr>
<td>OCP</td>
<td>Open/Close Principle</td>
</tr>
<tr>
<td>LSP</td>
<td>Liskov Substitution Principle</td>
</tr>
<tr>
<td>ISP</td>
<td>Integration Segregation Principle</td>
</tr>
<tr>
<td>DIP</td>
<td>Dependency Inversion Principle</td>
</tr>
<tr>
<td>SQALE</td>
<td>Software Quality Assessment based on Lifecycle Expectations</td>
</tr>
<tr>
<td>AQA</td>
<td>Application Quality Assessment</td>
</tr>
<tr>
<td>SA</td>
<td>Sonarqube Assessment</td>
</tr>
<tr>
<td>MI</td>
<td>Maintainability Index</td>
</tr>
</tbody>
</table>
This chapter introduces the need to develop a method for improving maintainability in object-oriented software applications. This need is established by examining problems that occur when developing software in rapid development environments. Insights gained during the examination are used to formulate the problem statement and objectives of this study.
1 Introduction

1.1 Background

Software is ever-present in our everyday lives. People are reliant upon it for responsibilities ranging from menial tasks such as waking up in the morning to managing complex processes such as machine production lines and creating flight plans for thousands of aeroplanes. Ever since the first software application was implemented, the way in which we develop software (software engineering) has evolved. Barry Boehm’s paper: A View of 20th and 21st Century Software Engineering, describes this evolution very well [1]. The paper describes the prevalent engineering practices for each century beginning from the 1950’s and ends with what Boehm speculates may occur by 2020 and beyond. Traditionally, software was developed in a manner that was akin to the way in which hardware is developed: emphasis was placed on formality. This means that the focus was on ensuring that the quality of software applications was high. As time passed, developers and engineers realised that software differs substantially from hardware. Software became something that could be created with a great deal more ease than hardware and could be developed a lot faster. This meant that to be competitive in the market, companies had to develop software quickly.

The prevailing thesis of this current day and age is that of developing software in Rapid Development Environments (RDE). A RDE is an environment where emphasis is placed on the softwares time-to-market. The popularity of software and its relatively low production cost means that there are many competitors within the field and it is usually the first product to market that is widely adapted.

A popular group of methods that is used within RDE’s is called Agile. Examples of methodologies that are classified as Agile development are eXtreme Programming (XP), Adaptive Software Development, Scrum, Dynamic Systems Development, Crystal, and Feature Driven Development. These methodologies adhere to the Agile manifesto which states the following tenets [2]:

- Interactions and individuals over tools and processes.
- Working software over extensive documentation.
- Customer association over contract arbitration.
- Adapting to change over strict planning.

The tenets above are supported by the following principles [3]:

- Customer satisfaction and good working software should be the primary priority for agile practitioners.
- Change should be welcomed by agile practitioners to provide customers with the best possible competitive advantage.
- Working software should be developed quickly with emphasis placed on having the shortest timescale possible.
• Software development personnel and business personnel should work together daily for the duration of the development endeavour.
• Motivated personnel are a must and should be exposed to an environment that provides support and trust.
• Face-to-face interaction is a necessity when conveying information to and within software development teams.
• Progress is measured through the amount of working software that is developed.
• Sustainable development should be a priority to ensure that all stakeholders maintain an indefinite constant pace.
• Attention should be emphasised on technical excellence; encouraging good design that enhances agility.
• Maximisation of work done (simplicity) is essential.
• Self-organising teams produce the best architectures, requirements and designs.
• The software development team should meet regularly to discuss ways that the team can become more effective and suggest changes.

There are numerous examples of software development teams achieving great success by making use of the Agile principles such as the recollection provided within [4]. There is a plethora of issues that may develop into problems when developing in RDE’s.

1.2 Problems in Rapid Development Environments
Developing software quickly results in three categories of problems occurring: engineering; process and management. Engineering related examples of problems that could occur are source code that violates known principles of good object-oriented design, software usability decisions that are made to ease implementation, or software architecture flaws [5]. Engineering problems that arise when developing software in RDE’s are related to the technical aspects of a software application.

Process related problems that arise as a result of developing software too quickly include, but are not limited to, the absence of a software project management framework, the lack of some form of a design methodology being used when developing software, the lack of adequate quality assurance measures, or any other problems that arise from issues with the processes in place within a software development endeavour [5].

Management related problems that arise when developing software quickly occur when priority is placed on schedule or cost rather than developing defect-free software or when the budget at the date of completion is less than estimated. They can also arise when there is an increasing amount of overtime required in order to meet objectives or any problems that are related to the management system in place within a software development endeavour [5].
The cause of the problems that arise when developing software quickly may be attributed to developers making decisions — conscious or otherwise, to find quick solutions to issues rather than implementing well thought-out solutions. In the case of the engineering related problem where code does not adhere to known principles of good design, a developer may have made the decision to implement a feature quickly to meet a release deadline. A practical example of a process related problem occurs when no development methodology is used, so as to avoid the long process of investigating, implementing and enforcing the use of a design methodology with the intention of beginning development as soon as possible. Management related problems also arise when development is rushed: management may decide to forego decisions related to quality assurance, scheduling, and effort estimation to begin development as soon as possible.

An apt metaphor that contextualises the problems that occur when developing software in RDE’s is Technical Debt (TD). TD is a concept coined by Ward Cunningham in an experience report in an OOPSLA (Object-Orientated Programming, Systems, Languages and Applications) experience report during an ACM (Association for Computing Machinery) conference in 1992 [6]. Cunningham’s metaphor uses financial debt as an analogy to describe aspects of software development that cannot usually be seen by the end-user but decrease the overall quality of the application from the developer’s perspective. The original metaphor describes the debt as code that is “not-quite-right”. This means code that does not follow known best practices that lead to good quality software.

The first iteration of the TD metaphor only included engineering related problems as causes of debt, but as more research was conducted on the subject, the metaphor has matured to also include process and management related problems [5]. These three categories of debt are referred to as engineering, process and management related debt.

The result of engineering, process or management debt is a lack of software application quality. Software application quality is defined by the International Organisation for Standardisation (ISO) in standard ISO 25010 by a product quality model that categorises product quality properties into eight characteristics [7]. These are: functional suitability; reliability; usability; performance efficiency; security; compatibility; maintainability; and transferability. See Table 1 for explanations of each of the characteristics.
Table 1: ISO 25010 Quality Characteristics [7]

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functional suitability</strong></td>
<td>The degree to which the product provides functions that meet stated and implied needs when the product is used under specified conditions.</td>
</tr>
<tr>
<td><strong>Reliability</strong></td>
<td>The degree to which a system or component performs specified functions under specified conditions for a specified period.</td>
</tr>
<tr>
<td><strong>Usability</strong></td>
<td>The degree to which the product has attributes that enable it to be understood, learned, used and attractive to the user — when used under specified conditions.</td>
</tr>
<tr>
<td><strong>Performance efficiency</strong></td>
<td>The performance relative to the amount of resources used under stated conditions.</td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td>The degree of protection of information and data so that unauthorized persons or systems cannot read or modify them, and authorized persons or systems are not denied access to them.</td>
</tr>
<tr>
<td><strong>Compatibility</strong></td>
<td>The degree to which two or more systems or components can exchange information and/or perform their required functions while sharing the same hardware or software environment.</td>
</tr>
<tr>
<td><strong>Maintainability</strong></td>
<td>The degree of effectiveness and efficiency with which the product can be modified.</td>
</tr>
<tr>
<td><strong>Transferability</strong></td>
<td>The degree to which a system or component can be effectively and efficiently transferred from one hardware, software or other operational or usage environment to another.</td>
</tr>
</tbody>
</table>

There is a plethora of studies that deal with the management of TD. Some examples can be seen in [8]–[10]. The referenced papers are systematic literature reviews with the aim of determining the state of art of research done on TD. Each review proposes the following research question (with minor alterations): What strategies have been proposed to deal with the management of TD? The answers to the question all propose on-going practices, processes or strategies for facilitating the prevention of TD or present very specific techniques for removing certain types of TD.

Some examples of solutions that make use of management or process related solutions for dealing with TD can be seen in:

- *A Flowchart for Rapid Technical Debt Management Decision Making*, by Yan Wu et al. that proposes a flowchart that can be incorporated into a software development teams management processes in order to determine whether or not TD should be dealt with [11].

- *Improving Software Quality via Code Searching and Mining*, by Madhuri R. Marri et al. which proposes a method for using different search engines for identifying and managing source code in order to deal with TD [12].
• *Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation*, by Vipin Balachandran that proposes making use of static analysis tools in order to aid in the code review step of the software development process [13].

• *Monitoring code quality and development activity by software maps*, by Johannes Bohnet et al. which demonstrates the usage of a type of visualisation called a software map that affords software development teams a visual representation of the code quality of a software application. This visual representation can then be used to identify areas with low quality code and the quality can subsequently be improved [14].

The examples of management and process related solutions for dealing with TD shown above are a small subset of the research done. All solutions mentioned are applicable for usage in situations where reducing future TD is a priority. Of the available management strategies, software development teams may choose to use the strategy that will most easily fit in with the development methodology used within their work environment.

There has been much research done on reducing TD from a practical perspective. Each solution looks at certain causes of TD and proposes a solution for dealing with the TD. Some examples can be seen in:

• *Search based refactoring for improving software maintenance*, by Mark O'Keefe et al. which describes and tests an automated tool that uses search based software engineering techniques for improving software maintenance [15].

• *Java Quality Assurance by detecting code smells*, by Eva van Emden et al. that improves source code quality by detecting and removing code smells for Java based software applications [16].

• *Improving design quality using meta-pattern transformations: a metric-based approach*, by Ladan Tahvildari et al. that improves application source code quality via the detection and removal of anti-patterns [17].

The examples mentioned are a small subset of the available research done on the concept of the removal of TD in software applications. There is a plethora of research available on the subject. This can be substantiated by [8]–[10] — the systematic literature reviews mentioned earlier. The fact that there are so many techniques for removing TD means that developers can become overwhelmed with the decision for what technique would best suit their needs. This means that there is a need to consolidate the techniques into a single method for removing TD.

1.3 **Contextualising Technical Debt and Maintainability**

Maintenance activities on a software application are estimated to account for between fifty and seventy percent of the effort on a typical software project [18]. This percentage varies depending on the source used. Although, the consensus is that maintenance is the most time consuming and costly aspect of software development. Logically, this means that if there is a high amount of TD that affects the maintainability of an application then
there would be an increased amount of effort spent on the most effort intensive aspect of software application development

ISO 25010 defines five sub-characteristics that make up the maintainability of a software application. These are [7]:

- **Modularity**: The level to which an application is composed of distinct entities such that a change in one does not affect another.
- **Reusability**: The level to which entities within the application can be used in more than one application.
- **Analysability**: The degree of effectiveness with which it is possible to identify aspects to be altered, to find insufficiencies or causes of failure or to assess the impact of a change to an entity within the application.
- **Modifiability**: The level to which an application can be altered without introducing adverse effects or decreasing the quality of an application.
- **Testability**: The degree of effectiveness with which an application can be tested.

The five sub-characteristics described are closely related to or affected by the technical aspects of a software application. This means that they are caused by engineering debt. Some triggers and their conditions for introducing engineering debt are listed in Table 2.

To contextualise, TD and Maintainability, an analysis of the triggers and conditions conducive to engineering debt is performed. Engineering debt affects the maintainability of a software application. When there is a lack of domain expertise on a software development team, developers are likely to make decisions to consciously or otherwise, ignore known principles of good design which could affect any of the maintainability sub-characteristics defined by ISO 25010.

Overly complex or inadequate software architecture may affect the maintainability of a software application in various ways. The architecture may not be designed in independent modules resulting in a lack of modularity or it may be overly complex resulting in difficulty in analysing the application.
<table>
<thead>
<tr>
<th>Trigger</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deep Domain Expertise</td>
<td>There is a lack of domain expertise on the project development team.</td>
</tr>
<tr>
<td>Software Architecture</td>
<td>The software architecture is overly complex or is not coupled with the operating system, network services and middleware.</td>
</tr>
<tr>
<td>Requirements Known</td>
<td>Requirements are unclear.</td>
</tr>
<tr>
<td>Technical Risk</td>
<td>There is a lack of knowledge of sources of technical risk.</td>
</tr>
<tr>
<td>Product Size</td>
<td>Product size is a trigger for engineering debt when initial estimates for application size are higher than expected or accounted for.</td>
</tr>
<tr>
<td>Complexity</td>
<td>Complexity is a trigger for engineering debt when the complexity of an application increases as iterations of the product are developed.</td>
</tr>
</tbody>
</table>

When the requirements of an application are unclear, software architects cannot design a good quality software application. There may be additional features that are required that cannot be easily implemented as they may not have been accounted for when design of the application began. To incorporate unplanned for features, software developers may have to introduce engineering debt in the form of low-quality code that affects the maintainability of a software application.

When sources of technical risk cannot be identified while developing a software application, unexpected issues may occur. These issues may result in problems that force developers to introduce source code that is not maintainable.

When a software application has a higher amount of source code than expected, it is very likely that it will be difficult to maintain. This can affect the maintainability of a software application in several ways — the most obvious of which is the analysability of a software application. A high amount of source code is difficult to analyse meaning that developers will not be able to estimate what the effect of adding additional functionality or addressing a defect in the application may result in.

An increase in software complexity as new versions of software are released could affect the maintainability of the application via affecting each of the sub-characteristics that assess the maintainability of a software application.
An increase in software complexity as new versions of software are released could indicate the presence of engineering debt since it implies that developers may have been forced to add complex code to ensure that previous functionality is not hindered. This means that the application is not maintainable as developers could not easily modify or add new functionality. The maintainability sub-characteristic(s) that forces developers to add complex code can be determined by addressing the engineering debt.

Further contextualisation of TD and maintainability is obtained by examining a systematic literature review conducted by Zengyang Li et al., with the goal of gaining a better understanding of TD in software applications [8]. One of the research questions proposed is to determine the aspects of software quality that are affected by TD. The answer to this question reveals that most of the studies examined within the literature review state that maintainability is the primary quality characteristic that is affected by TD.

Using the contextualisation of TD with maintainability, it can be concluded that a method that has the goal of reducing TD in software applications should focus on reducing TD that affects the maintainability of software applications (engineering debt).

1.4 Technical Debt Types

Further expansion on the types of debt discussed earlier is required to fully understand the effect of TD on the maintainability of the application. The types of debt described earlier are management, process, and engineering debt. These types were classified by using the areas that TD affects. Further types of TD are found when analysing their causes. These types of debt are [8]:

- **Design debt**: The debt that arises from poor quality source code that does not follow principles and practices that lead to code that is of a good quality [19], [20].
- **Architecture debt**: The debt that is incurred when the architecture (the structure of the program) has issues. These issues could be in the form of a lack of modular separation, inappropriate consideration of security concerns, etc. The problems that arise from these issues are usually major and will require a great deal of effort to correct them [21], [22].
- **Documentation debt**: This type of debt arises when a software application lacks adequate source code documentation [19].
- **Test debt**: This type of debt is present when there are issues in the testing activities of the software under development. These could be in the form of a lack of tests or tests that do not function correctly [19].
- **Code debt**: This type of debt is similar to design debt but was specifically identified for the purpose of singling out debt that causes the maintainability of the source code to suffer [14].
- **Defect debt**: Known defects in the functionality of the software application cause instances of defect debt. These defects are referred to as bugs in the code and arise during development of the application [23].
• **Requirements debt:** The debt that arises when decisions are made to delay the implementation of functionality that addresses known requirements. This debt can also arise from non-functional requirements such as performance, maintainability, security, etc. [21].

• **Infrastructure debt:** This type of debt is incurred when issues arise with the infrastructure within which software is developed [9]. Infrastructure debt could be in the form of issues with servers that host applications, development environment issues such as a lack of funding or a lack of hardware.

• **People debt:** The debt that is caused by issues with the personnel developing software. An example of this would be the delay of personnel training [9].

• **Process debt:** This refers to inappropriate engineering processes [5], [24].

• **Build debt:** Any activities that delay or cause issues during the build process of an application will incur debt [25].

• **Service debt:** This type of debt only applies to service-based systems. The debt could be caused by a mismatch between the requirements for the system and the services that the system provides, or by decisions that have been made to partially fulfil requirements [26].

• **Usability debt:** The debt that is caused by usability decisions. These types of decisions are usually temporary functionality pertaining to the user interface of the application that will need to be changed in the future [27], [28].

• **Versioning debt:** This type of debt arises when there are issues in the versioning of software source code [29].

For the purposes of this thesis, the only types of TD that are relevant are those that can be considered engineering debt that affects the maintainability of the application. Therefore, the relevant TD debts that are considered are: architectural, design, code and test debt.

### 1.5 Object-Oriented Software Applications

A distinction is made in terms of the types of applications that are considered by this study. There are multiple programming paradigms that are used for the development of software applications. Each paradigm has its own benefits and limitations. When programming languages are developed, creators choose a paradigm that best suits the problems that the creator wishes the programming language to solve.

Some examples of commonly used programming paradigms are aspect-oriented, functional, procedural and object-oriented programming. Software applications that make use of the object-oriented programming paradigm are considered for this investigation. This is done because a preliminary survey of current research done on the subject revealed that the highest amount of emphasis has been placed on object-oriented software applications.

A further reason for choosing to focus on object-oriented software applications is due to the popularity of programming languages that make use of the paradigm. Every year since 2014, IEEE spectrum has created a
software application that allows users to compare the popularity of languages [30]. Using the application, it is concluded that object-oriented languages are the most popular.

1.6 Problem Statement

The prevailing thesis of software development methodology in the current climate is that of developing software in RDE’s. RDE’s provide a plethora of competitive advantages in software development but do introduce problems. These problems are classified as engineering, process or management related. An apt metaphor contextualising the three problem categories is that of TD. The introduction of TD in software applications results in a lack of software quality.

There is a great deal of research done on the concept of reducing engineering related problems within a software development endeavour. One could say that there is almost too much research, meaning that developers may become overwhelmed when determining what method should be used to reduce engineering related TD and improve software quality.

Maintenance activities within a software development endeavour are estimated to account for between fifty to seventy percent of the effort spent on a typical software application. ISO 25010 defines five sub-characteristics that can be evaluated to assess the maintainability of a software application. Each sub-characteristic is affected by different triggers of engineering debt in software applications. This phenomenon results in the conclusion that the improvement of maintainability in software applications can be achieved by reducing engineering debt in software applications.

The types of engineering debt that directly affect the maintainability of software applications are architectural, design, code and test debt.

The object-oriented programming paradigm is the most commonly used paradigm in software development environments. Therefore, the types of applications that will be focussed on are object-oriented software applications.

The above observations led to the conclusion that a method should be developed for improving maintainability in object-oriented software applications.

1.7 Study Objectives

To formulate a method for improving maintainability in object-oriented software applications the following objectives should be achieved in terms of the research that is required:

- Investigate techniques for identifying areas of low maintainability in software applications.
- Investigate techniques for improving maintainability in software applications.
- Investigate techniques for assessing maintainability in software applications.
- Investigate techniques for assessing a method to improve maintainability.
- Formulate refined requirements for the method to improve maintainability in object-oriented software applications.

To address the need for a method to improve maintainability in software applications, the method should be formulated using the following objectives:

- The method should target object-oriented software applications.
- The method should focus on TD that affects the maintainability of a software application.
- The method should encompass or make use of existing techniques for addressing TD (improving maintainability).
- The method should have three high level steps — identification, rectification and assessment.

After the creation of the method to improve maintainability in software applications, the method should be assessed. This is achieved by accomplishing the following objectives:

- The method should be verified by means of determining if requirements have been met.
- Results obtained from the verification of the method to improve maintainability should be analysed to determine if it is feasible to validate the method.
- The method should be validated by applying it to an existing software application and performing all the specified steps.
- Results obtained from both method verification and validation should then be analysed to determine if the study is a success as well as determine recommendations for future research.

1.8 Study Overview

Chapter 1

This chapter introduced the need to develop a method for improving maintainability in object-oriented software application by examining the issues that occur when developing software in rapid development environments. This need is addressed by achieving the objectives listed in the previous section to address the problem statement presented in section 1.6.

Chapter 2

This chapter presents the literature reviewed to accomplish the literature goals discussed in section 1.7. Techniques for identifying areas of low maintainability, improving maintainability, and assessing maintainability in object-oriented software applications are presented and reviewed to determine which techniques are applicable for usage within a method to improve maintainability in object-oriented software applications.
Chapter 3

Refined requirements for the method to improve maintainability in object-oriented software applications are formulated using knowledge gained from the literature study. Using these requirements, the method is presented in full.

Chapter 4

This chapter describes the way the method to improve maintainability is verified before it is applied to a real-world software application. The method is verified utilising a survey of software developers with suitable development experience. The survey presents respondents with the method and postulates questions that relate to the refined requirements presented in chapter 3. The responses from the survey are analysed and the determination of whether the method can be applied to a real-world software application is made.

After determining that the method is verified, this chapter presents the method used for validating the method to improve maintainability in object-oriented software applications. The validation is achieved by performing a case study that applies the method to an android data collection application. The details of the implementation of the method are presented and emphasis is placed on finding insights on the efficacy of the method in terms of how well it addresses its requirements and ways that the method can be improved.

After performing the case study, the method is assessed using a neutral maintainability assessment primarily to determine if the method addresses its primary objective (improving maintainability). This method assessment is also used to provide insights on the way that the method affects software maintainability.

Finally, a discussion is presented with the goal of determining whether the method to improve maintainability addresses its requirements, achieves its objectives and present any information that is relevant when applying the method to improving maintainability to future software applications.

Chapter 6

The study is summarised in this chapter and recommendations for future work are discussed.
CHAPTER 2

LITERATURE STUDY

The previous chapter introduces the objectives that should be achieved to develop a method for improving maintainability in object-oriented software applications. The first set of objectives deal with the research required. These objectives are achieved through a literature study. This chapter presents and reviews existing techniques for identification of areas of low maintainability, techniques for improving maintainability in object-oriented software applications and finally techniques for assessing the maintainability of object-oriented software applications.
2 Literature Study

2.1 Preamble

The previous chapter introduced the need to develop a method for improving maintainability in object-oriented software applications. This need is fulfilled by completing a literature study with the goal of achieving the research objectives stated in Section 1.7.

Techniques for identifying areas of low maintainability will be investigated. This will begin with identifying indicators of TD. Using the indicators, previous work done on identifying areas of low maintainability will be reviewed.

The objective of investigating techniques for improving the maintainability of software applications will be achieved by reviewing previous work done. This is also true for the objective of investigating techniques for assessing maintainability, as well as the objective of determining a way to assess a method for improving maintainability in object-oriented software applications.

2.2 Identification of Areas for Improving Maintainability in Software Applications

2.2.1 Indicators of Low Maintainability

To identify areas of low maintainability, it is necessary to examine what developers should look for. As discussed in Sections 1.3 and 1.4, this thesis will focus on architectural, code, design and test debt which are forms of engineering debt that affect the maintainability of a software application [9]. Using this fact, two papers are very useful in terms of determining indicators of areas of low maintainability. These papers are: Identification and management of technical debt: A systematic mapping study, by Nicolli S. Alves et al. [9]; and A systematic mapping study on technical debt and its management, by Zengyang Li et al. [8]. These papers are systematic literature reviews that provide an insight into the concept of TD.

The first cause of code debt is code without standards. A coding standard relays rules and best practices that developers should follow in order to maintain consistency between the development team members and to ensure that all developers are aware of known practices that ensure high quality code [9]. Coding standards specify details such as: naming conventions for files and variables; file formatting rules (indentations, file organisation, etc.); comment formatting; and other aspects that can be specified as tangible rules [9]. Adhering to a coding standard results in increasing application quality as code becomes more understandable. If code is more understandable, maintenance becomes easier as developers do not need to spend as much time understanding the code [31].

A slow algorithm is an instance of TD that arises when a block or sequence of code takes an overly long amount of time to run [9]. This usually arises because of a lack of developer experience. This decreases the quality of an application, as its performance suffers.
An automatic static analysis issue arises when development teams make use of tools that provide developers with information regarding the quality of an application [9]. Issues arise when the information provided is not useful or is incorrect. The type of information that is provided by automatic static analysis tools appears in the form of ratings for different quality aspects or actual feedback regarding blocks of code that do not follow practices that result in good quality code [9]. These tools are required to be pre-programmed with rules that dictate the feedback that is provided to developers. There is a plethora of causes for issues with static analysis tools and some may even be out of the control of developers that make use of them [9].

Code metrics provide developers with quantitative data that relays information regarding different aspects of a software application. These aspects assess software according to: failures; quality; complexity; change; cohesion; testing; maintainability; thresholds; performance; reliability; coupling; evolution; plagiarism; reusability; inheritance; security; size; usability; responsibility; and understandability [32]. There are over three hundred different code metrics that are within use or have been used in the software industry; each providing insight towards different aspects of a software application. Developers decide what metrics they would like to use based on the aspects of software that are deemed valuable in their context. Based on results of different metrics the health of an application can be assessed. Analysing the health of an application, developers can determine if there is engineering debt that affects the maintainability of a software application.

Code smells are known patterns of code occurrences that result in low quality or problematic code [33]. Developer experience has resulted in the realisation that certain patterns of code result in decreased application quality and may give developers issues in future development. The presence of these patterns within applications means that there could be engineering debt that affects maintainability within the software application. (It is worth noting that not all code smells indicate that there could be a problem with the code). This issue will only become a problem if certain conditions (that depend on the code smell detected) are met. For a full list of known code smells, see [33].

Design debt is a form of TD that results in a lack of code quality, meaning that its causes are relevant when looking for a method to address TD [9]. Two of the indicators listed are code metrics: Afferent/Efferent Couplings (Ca/Ce) and Depth of Inheritance Tree (DIT). Ca/Ce provides information on the coupling between modules and classes within the application and DIT provides information on the usage of inheritance instances within the application [34].

Grime develops within a software application over time. It is the accumulation of additional code within an initial software design [35]. This additional code could be in the form of bug fixes or additional functionality that is written without considering the overall design of the application. Grime affects the maintainability of a software application since it increases application complexity and reduces application understandability.

The final cause of design debt is software design issues. This is deliberately vague as there are many different types of issues that can arise in the design of a software application. Like code smells, common design issues,
called software anti-patterns, have been found to frequently occur in software applications and differ from code smells because they occur at a higher level and always indicate that there is a problem. A list of known anti-patterns can be found at [36].

There are five indicators of architectural debt in software applications: architecture smells; anti-patterns; complex architectural behavioural dependencies; architectural compliance issues; and system-level structure quality issues [8], [9].

All indicators of architecture debt can be considered as extensions of design or code debt [8]. This is because they are simply categorisations of different types of design or code debt. These categorisations are created to emphasise that architectural issues can also be considered as part of relevant types of design or code debt. This can be seen when analysing each of the indicators of architecture debt.

Anti-patterns are code smells at an architecture level rather than a single block of code [8]. Architecture anti-patterns can be considered as a hybrid between design debt and architecture debt. This is because architecture anti-patterns are just a categorisation of anti-patterns that occur at an architecture level.

Complex architectural behavioural dependencies, architectural compliance issues and system-level structure issues are all instances of poor design [8]. This means that they can be considered as part software design issues, meaning they can be considered design debt.

The final type of TD that affects the maintainability of software applications is test debt. [7] and [8] lists six indicators of test debt. These are: low test code coverage; deferring testing; lack of tests; lack of test automation; residual defects not found in tests; and expensive tests. Low test code coverage and a lack of tests can be considered as a single indicator of test debt as low-test code coverage indicates that there is an insufficient number of tests created for the software application.

Deferring testing refers to the situation where developers realise that tests are required but choose not to create them [19]. A lack of test automation is an indicator of TD since a known good practice for developing software tests is that they should not require user input in order to function [37]. Residual defects not found in tests and expensive tests are fundamental errors with the created tests.

In summary, there are four general indicators that developers should look for when trying to identify areas of software applications that decrease maintainability. These are:

- Anti-patterns
- Code Smells
- Grime
- A lack of Tests
Finally, it should also be noted that code metrics can be used as a tool for the identification of the indicators listed above.

2.2.2 Code Metrics

Previous Work on Code Metrics
The usage of source code metrics as a means of identifying areas of low maintainability is an area of research where much effort has been spent.

A study by Ebrahim Bagheri et al. proposes the usage of structural metrics in order to quantify the maintainability of a software application by assessing three sub-characteristics of maintainability — analysability; changeability and understandability [38]. The results of the assessment (values calculated for the structural metrics used) are used to assess where problems lie within the software application.

Bente Ada proposes the usage of structural metrics in conjunction with expert opinion for the assessment of the maintainability of a software application [39]. The assessment results are used to assess where likely causes of problems lie.

S Counsell et al. developed a method for making use of code metrics as means to rank code smells in software applications [40]. The existence of this study indicates that certain code metrics and code smells (a type of engineering debt), are highly related and can be used to indicate the presence of code smells in software applications.

A paper by Frank Simon et al. introduces a method for removing code smells by making use of distance-based software metrics in order to visualise code smells [41]. The visualisation of code metrics using distance-based metrics emphasises how the usage of code metrics as an identification tool can simplify the task of understanding the identification of code smells, as if there is a method to create a tool to visualise the smell, then developers are able to visualise the smell themselves.

Aiko Yamashita et al. conducted a study with the aim of examining the use of code smells as system level indicators of maintainability [42]. The study found that code smells are best used for assessing the health of application maintainability. It also found that code metrics that can be considered size measures are mostly used during the assessment of the maintainability of a software application.

Michael Mohan et al. developed a system capable of automatically detecting and fixing code smells, by making use of search-based software engineering techniques [43]. To determine if the detected code smell should be fixed, the developed tool makes use of code metrics whose values are used as a quantitative measure that (if above or below specified limits) dictates if the code smell should be rectified.

Chen Zhifei et al. conducted a study that makes use of metrics in order to detect code smells in software applications that make use of the Python programming language [44]. They developed a tool that can identify
ten code smells that are commonly encountered within Python applications by making use of code metrics. To determine if a code smell is present, the authors developed a filtering strategy specifying metric thresholds that when exceeded, indicate the presence of a code smell.

There are a great deal of techniques making use of code metrics as a means of detecting areas of low quality code available in literature. However, no technique fully satisfies the needs of a method to improve maintainability in software applications. Therefore, the usage of general metric suites is investigated to satisfy the following requirement: the metric suite used within the method for improving maintainability in software applications should provide developers with feedback regarding the quality of an application on three levels: application level; package/folder level; and class/file level. Ensuring that the metric suite provides insight on these levels means that developers can pinpoint files that suffer from low maintainability.

**Metrics Applicable for Usage within a Method to Improve Maintainability**

At an application level, the Metrics for Object-Oriented Design (MOOD) metric suite is applicable for usage within a method to improve maintainability in software applications. Figure 1 shows the six metrics used in MOOD [45]. These metrics are used to assess the quality of an application from a high level. Each metric is quantified using a percentage that provides developers with an insight regarding the overall quality of the application.

![MOOD Metrics](image)

Figure 1: MOOD Metrics

The first metric used is called the Attribute Hiding Factor (AHF). AHF shows the degree of attribute, or field encapsulation, in a project. This is done by calculating the ratio of hidden attributes to total attributes in the application [46]. AHF can be calculated using (1). The AHF of an application should be as high as possible. The ideal result would be a hundred percent, as the attributes of a class should only be known to the class itself [45].

\[
AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)} = 1 - \frac{\sum_{i=1}^{TC} A_v(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}
\]  

(1)

Where:

- \( A_d = Defined\ Attributes \)
- \( A_v = Visible\ Attributes \)
• \( A_h \) = Hidden Attributes
• \( TC \) = Total Classes
• And: \( A_d(C_i) = A_v(C_i) + A_h(C_i) \)

The second metric used is called the Attribute Inheritance Factor (AIF). This metric provides knowledge on the degree of attribute or field inheritance in a project. It shows the ratio of what percentage of the available fields in a class are due to inheritance [46]. AIF should be within a limit that developers deem appropriate. If it is too high, this would indicate that the member scope of the project is too high. This goes against known good practices for software development [45]. AIF is calculated using (2).

\[
AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_d(C_i)} = 1 - \frac{\sum_{i=1}^{TC} A_d(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}
\]  

(2)

Where:
• \( A_a \) = Available Attributes
• \( A_d \) = Defined Attributes
• \( A_i \) = Inherited Attributes
• And: \( A_a(C_i) = \frac{A_d(C_i) + A_h(C_i)}{A_d(C_i)} = A_h(C_i) + A_o(C_i) \)

Method Hiding Factor (MHF) is the degree of method encapsulation in the software application. It shows the ratio of how many classes an average method or function is visible from other than the class that defines it. A low MHF indicates that the use of abstraction is insufficient. If there is a lack of abstraction within a software application, there is a high probability of errors within the application. A high MHF indicates that the functionality of the application is lacking [46]. MHF can be calculated using (3).

\[
MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)} = 1 - \frac{\sum_{i=1}^{TC} M_v(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}
\]  

(3)

Where:
• \( M_v \) = Visible Methods
• \( M_d \) = Defined Methods
• \( M_h \) = Hidden Methods
• And: \( M_d(C_i) = M_v(C_i) + M_h(C_i) \)

Method Inheritance Factor (MIF) shows the ratio of what percentage of the methods within a class are made available through inheritance rather than declared in the class itself. MIF behaves similarly to AIF, with the exception that its scope is on a method level [46]. The MIF is calculated using (4).
\[
MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)} = 1 - \frac{\sum_{i=1}^{TC} M_d(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}
\]

Where:
- \(M_a = \text{Available Methods}\)
- \(M_d = \text{Defined Methods}\)
- \(M_i = \text{Inherited Methods}\)
- And: \(M_a(C_i) = \frac{M_d(C_i) + M_i(C_i)}{M_d(C_i)} = M_n(C_i) + M_o(C_i)\)

Coupling factor (CF) provides insight into the coupling of an application. Coupling is the degree to which an application module relies upon the other modules within the application. If a module is highly coupled, its maintainability is low [46]. This is because it would be difficult to make changes to the module. The CF is calculated using (5). A class is coupled to another if it calls another class’ methods, or accesses any of its attributes.

\[
CF = \frac{\sum_{i=1}^{TC} \sum_{j=1}^{TC} \text{is\_coupled}(C_i, C_j)}{TC^2 - TC - 2 \times \sum_{i=1}^{TC} DC(C_i)}
\]

Where:
- \(\text{is\_coupled}(C_c, C_s)\) — is \(C_c\) coupled to \(C_s\)
- \(DC = \text{Descendants Count}\)
- \(TC = \text{Total Classes}\)

The Polymorphism Factor (PF) is the degree of polymorphism used within a software application. This is the probability that a method or function will be overridden. PF gives an indication of the complexity of the application. An overly complex application lacks maintainability, since it indicates that the application lacks analysability and modifiability. The metric is calculated using (6).

\[
PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} (M_n(C_i) \times DC(C_i))}
\]

Where:
- \(M_o = \text{Overridden Methods}\)
- \(M_n = \text{New Methods}\)

At a package/folder level, the Martin packaging metrics appear to provide an adequate insight into the maintainability of an application.
The Martin packaging metrics suite provides information regarding the stability and coupling within each application package. They were formulated by Robert Cecil Martin. Figure 2 shows the metrics that are used within the Martin Packaging Metrics. The first metric used is package Abstractness. Abstractness is the ratio of abstract classes and interfaces in a package and is calculated using (7) [47]. Values for the abstractness metric range between zero and one. A value of zero indicates that a package is concrete (all classes have been implemented) and a value of one indicates that all classes are abstract. Abstractness should be used in conjunction with instability; since, if a package is unstable and concrete, this would indicate that the package is not open to change (it will be difficult to perform changes to package functionality).

\[
Abstractness = \frac{\text{number of abstract classes and interfaces in package}}{\text{number of concrete classes and interfaces in package}} \tag{7}
\]

The next two metrics used within the Martin Packaging Metrics suite deal with coupling. Coupling metrics give an indication as to the likelihood of a defect in a single class causing a ripple effect within the system. Efferent coupling (Ce) is the number of classes that a given class depends upon. Afferent coupling (Ca) is the number of classes that depend upon a given class. In the context of the Martin Packaging Metrics, the values calculated for afferent and efferent coupling relay total amount of each coupling type within each package [47].

Instability is a measure of a package’s resilience to change. The metric can be calculated using (8).

\[
Instability = \frac{\text{Efferent Couplings}}{\text{Efferent Couplings + Afferent Couplings}} \tag{8}
\]

The final metric used within the Martin Packaging Metrics suite is the distance from the main sequence. This metric is sometimes referred to as “the perpendicular distance between the idealised line Abstractness + Instability = 1”. “Good packages” would appear on the line that is between the two points where Instability = 0 and Abstractness = 1 and vice versa. The distance from the main sequence can, therefore, be calculated using (9).

The ideal value for the distance to the main sequence is zero; as the higher the abstractness of a package, the more stable it should be since there will be many client classes that depend upon its interfaces. The distance
calculated will be between one and zero and indicates the distance the package is from appearing on the line that constitutes a good package [47].

\[\text{Distance from the Main sequence} = |\text{Abstractness} + \text{Instability} - 1|\]  \hspace{1cm} (9)

At a file/class level, the Chidamber-Kemerer (CK) metrics provide adequate insight into the maintainability of a software application [48]. This is further substantiated by the insight provided by [32]. The paper states that one of the most popular source code metric suites researched is the CK metrics suite.

This CK metrics suite provides insight on a class/file level, and was presented in full by their creators Shyam R. Chidamber and Chris F. Kemmerer in 1994 [48]. Figure 3 shows the metrics used in the CK metric suite. The first metric used is Coupling Between Objects (CBO) which is the total number of afferent and efferent couplings for the class. If a class is highly coupled, its maintainability is low [48].

The next metric used is the Depth of Inheritance Tree (DIT). DIT is the amount of inheritance steps between the class and the basic object class of the object-oriented language (example: java.lang.Object). If class DIT is high, this may indicate that there is an overuse of inheritance.

Lack of Cohesion of Methods (LCOM) is a measure of the interrelation between the methods of a class [48]. Two methods of a class are related if one method calls another, or they share a variable usage. LCOM is the number of relations between all methods within the class. High values for LCOM indicate that a class may have too many responsibilities and should potentially be separated into classes with distinct responsibilities. A value of one (the lowest value possible) indicates that a class is highly cohesive and cannot be easily separated [48].

The Number Of Children (NOC) of a class is the number of classes that directly inherit from the class. Each child class will be susceptible to the TD of its parent class.

The Response For Class (RFC) is the number of possible methods that that can be called when a class is sent a method send. This is the sum of the number of constructors and methods in the class plus the number of

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{ck_metrics.png}
\caption{CK Metrics}
\end{figure}
methods and constructors the class can call directly. A high RFC indicates that a class may be too complex and may need to be restructured.

The final metric in the CK suite is Weighted Method Complexity (WMC). WMC is also called Cyclomatic Complexity (CC) at a method level. CC is defined as the number of linearly independent paths through a program’s source code and is also referred to as the McCabe metric [49]. The McCabe metric can be found at any level of application source code (project, package, class and method). If source code is too complex, this may indicate the presence of TD as the package/project/class/method may be responsible for too much and may contain code that does not follow known best practices. A high CC may also indicate the presence of grime. [32] also states that the CC metric is one of the most commonly used and researched metrics available.

The metrics and metrics suites discussed above are ideal candidates for usage within a method for improving maintainability in object-oriented software applications. An in-depth discussion of each is provided for the purposes of providing clarity in terms of their usage.

2.2.3 Automated tools

Studies that make use of automated tools for the identification and in some cases actual rectification of identified faults, can be separated into two categories: small scale applications and large scale commercially available applications.

A journal article by Ah-Rim Han et al. discusses the operation and creation of an automated tool for identifying a certain known pattern of code that improves software maintainability [50]. The paper finds that the technique greatly facilitates the identification of areas of low maintainability.

Charnpreet Singh et al. discussed an automated method for the identification of duplicated code [51]. Duplicated code is a type of code smell that is considered as bad practice, as it results in multiple instances of code that could potentially introduce the same error within multiple modules of a software application. This automated tool demonstrates that similar tools facilitate the identification of areas of software applications that lack maintainability.

A study by Adnane Ghannem et al. presents an automated identification technique and accompanying tool that makes use of code metrics and design defect rules for identifying refactoring opportunities [52]. The tool addresses the lack of adequate knowledge provided by code metrics alone as a means of identifying low quality code by using an additional technique — in this case, additional rules. The lack of adequacy using code metrics alone demonstrates a need to include additional methods when identifying areas of low maintainability.

Radu Marinescu discussed a technique that makes use of metrics-based rules for identifying design flaws in software applications [53]. The paper implements the technique using an automated tool. The technique and tool demonstrate the inadequacy of code metrics alone, as well as the way that automated tools facilitate the identification of areas of low maintainability.
There is a plethora of examples of automated tools capable of identifying areas of low maintainability. All examples demonstrate the need to include some form of automated identification within a method to improve maintainability in object-oriented software applications.

2.3 Improving Maintainability in Software Applications

2.3.1 Refactoring

Before reviewing previous work done on improving maintainability in software applications, it is necessary to delve into the meaning of the term refactoring. Refactoring is a term that is commonly associated with literature related to improving the quality of software applications. The term refactoring is defined by Martin Fowler in his book: Refactoring: Improving the design of existing code [33]. Its definition changes depending on the way it is used. As a noun it means: A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour [33]. As a verb, the act of refactoring is: to restructure software by applying a series of refactorings without changing its observable behaviour [33].

The refactorings that Fowler refers to are known techniques for addressing code smells. These refactorings appear with guidelines for when they should be used, a motivation for why they should be done and the mechanics of the way they should be carried out. A full explanation of all possible refactorings is out of the scope of this thesis, as the subject is too broad. See [33] for in-depth explanations (with examples) of refactorings.

During the process of finding papers that are relevant to the problem this thesis aims to solve, it is found that authors use the term in ways that do not conform to the original definition provided by Fowler. The term is rather used as a verb that describes the process of restructuring code in order to improve its quality (known refactorings as described in [33] are not the only ways to restructure code without changing its observable behaviour). This is the definition (when used as a verb) that will be used for the rest of this thesis.

Section 2.3.1 identified three indicators of low quality code that require refactoring: code smells; anti-patterns; and grime. Relevant research done pertaining to each indicator is subsequently presented and reviewed.

There are many papers that focus on improving software quality by way of removing code smells. Each paper focusses on improving different aspects of software quality: maintainability; reliability; smaller areas of interests such as the improvement of certain code metrics; transferability; testability; and several others.

Michael Mohan et al. investigated the usage of search based automated refactoring as a means of improving intrinsic code quality [43]. The technique detects code smells which are then refactored using refactorings described by Martin Fowler.
Another paper that makes use of search-based automated refactoring as a technique for improving software maintainability is presented by Selim Kebir et al. [54]. The technique aids developers with the identification of code smells, as well as proposes the sequence that refactorings should be performed.

Research done by Marois Fokaefs et al. investigated the detection of code smells that can be solved by applying a type of refactoring called Extract Class [55]. After determining the location of detected code smells, a custom automated tool is used for applying the Extract Class refactoring.

A study by M. Gatrell et al. investigated the effect of applying known refactorings on the change and fault proneness of software [56]. Change and fault proneness is directly affected by the maintainability of a software application as they are affected by the modifiability of a software application. An application’s modifiability is one of the sub-characteristics of maintainability defined by ISO 25010. The study found that by applying known refactorings via identifying code smells, the change and fault proneness of software applications is reduced, resulting in increased application maintainability.

Anas Mahmoud et al. discuss a method to improve code traceability through making use of refactorings [57]. The improvement of code traceability results in an increase in the analysability of software applications which results in an improvement in software maintainability.

As opposed to the research done on refactorings described by Martin Fowler, exploration done on the concept of anti-pattern refactoring varies greatly. The definitions of anti-patterns are not as well documented as code smells; resulting in much room for interpretation in terms of what sort of code patterns can be considered anti-patterns.

A paper by Nikolaus Tsantalis et al. discusses a method for identifying refactoring opportunities utilising polymorphism as the solution. The paper aims to improve the maintainability of software applications via identifying blocks of state-checking code. State-checking code is considered a type of anti-pattern [58].

An article by Fatimah Mohammed Alghamdi et al. investigated the impact that design patterns have on the maintainability of software applications [59]. Some design patterns can be considered as a solution to the removal of anti-patterns. The article found that software maintainability is improved by making use of design patterns as a refactoring tool.

Further research is done on the concept of making use of design patterns as a means of removing anti-patterns from software applications by Davoud K. Ghourbanpour et al. [60]. The paper investigates the usage of a single design pattern (abstract factory) to remove instances of anti-patterns that it can be applied to.

A book by Joshua Kerevsky provides an in-depth explanation of the usage of design patterns when refactoring software applications [61]. The book explains different anti-patterns that can be found in software applications and methods for using design patterns to improve software application quality.
Yasser A. Khan et al. proposed the improvement of software quality via modifying software application use case models in order to remove any occurrences of anti-patterns [62]. Use case models are a means of documenting the requirements of a software application. Depending on the design methodology used when developing a software application, requirements may be documented at the beginning of the development lifecycle. This means that this approach might not work for all software applications, but this paper does emphasise the usefulness of refactoring anti-patterns when improving software maintainability.

A paper by Rodrigo Morales et al. describes an automated tool capable of identifying and rectifying multiple Java anti-patterns [63]. The tool presented leverages developer context to provide refactoring suggestions. Developer context means that the tool uses the knowledge of where developers are currently working (the tool will provide suggestions for refactoring on files that developers are currently working). The tool only uses developer context since the manageability of the refactoring process will be simpler. When performing a large-scale refactoring of code, developers should account for the fact that it will be easier for them to refactor code that they are familiar with as well as focus on smaller blocks of code.

Research done on the concept of grime within software applications is also called software decay. In terms of remedying instances of grime, research is scarce. This is because grime is a subjective issue that changes depending on the development methodology used within the software development endeavour.

There are several papers that do not directly address code smells, anti-patterns or grime, but do provide insight into the restructuring/refactoring processes that are relevant for a method to improve maintainability in software applications.

During the 27th IEEE/ACM international conference on automated software engineering, Panita Meananeatra presented a paper that proposes that refactorings should be performed in an optimised sequence in order to obtain the best results in terms of the improvement of software quality [64]. This means that the method for improving maintainability in software applications should ensure that the refactoring step should be performed in sequence for better results.

The sequence for applying refactorings to software applications is further discussed by H. Liu et al. [65]. An automated tool is developed that applies refactorings in a sequence that provides an optimal effect on the quality of a software application. The sequence is governed by a conflict matrix of refactorings that describes the inter-relation between refactorings and the conflicts that can arise when applying one or the other.

An article by Ricardo Perez-Castillo et al. discusses the fact that large refactoring sequences have an adverse effect on the sustainability of a software application [66]. The effect of god class refactorings are analysed with respect to the impact they have on the power consumption of the software application. The results found that applications where the refactoring has been applied consume more power, and the sustainability of the application is affected. This fact should be accounted for when applying a method to improve the maintainability of a software application.
Papers that improve software maintainability by remedying the indicators of low maintainability are reviewed, and findings are noted for usage within a method for improving maintainability in object-oriented software applications.

2.3.2 Testing

The Need for Automated Testing

When improving maintainability in software applications, developers should address test debt. The reason for this is twofold: addressing test debt results in an increase in maintainability as testability is a sub-characteristic of maintainability (as specified in ISO 25010); and secondly, the creation of tests simplifies the refactoring process.

Within the book by Martin Fowler et al. — *Refactoring: Improving the Design of Existing Code*, the need for including some form of automated testing when undergoing the refactoring of a software application is explained. These tests should come in the form of unit, integration or behavioural tests. A unit test exercises a small piece of code with a specific functionality. They are developed in order to verify that the code under test performs as expected [37]. Unit tests are usually conducted on specific methods or functions within a class. Integration tests determine if classes function as expected when combined or working together. In the ideal situation, it would only be necessary to create unit tests; but this is not always the case, as some classes may be heavily dependent on others. Classes can be heavily dependent on others as a result of bad design, or because the framework used during development forces developers to do so [67]. Behavioural tests are created to determine if a software application functions as it should. They are usually created when classes interact with elements such as the user interface or databases.

In terms of the reason for developing automated tests before refactoring code, Martin Fowler validates the need for writing tests through a recollection of an instance where he had to refactor code [33]. He then goes on to show the value of writing tests before refactoring by demonstrating an example of refactoring an area of code using unit tests as a bug detecting tool.

Further reasons that testing before performing a refactor is important are:

- They provide a safety-net that prevents developers from introducing regression errors during the refactoring process [68]. This occurs because developers can test any changes made to source code and get feedback easily.
- Tests aid developers in their understanding of the code. This happens because tests can be considered a form of documentation [68]–[70].
- Tests increase the quality of developed code by decreasing method size, decreasing complexity, reducing coupling, and increasing code cohesion [71].
Further validation for the need for the creation of automated test suites is available in the research conducted within the field of improving software maintainability. Melina Mongiovi et al. proposed a method for automatically creating tests when applying refactorings to software applications. The tool creates these tests in order to allow developers to analyse the impact of refactorings on software applications [72]. This analysis ensures that undesired effects do not occur when refactorings are applied.

A paper by Frens Vonken et al. investigated the relation between unit testing and refactoring [73]. The investigation found that unit testing does not improve the time for completing refactorings for the cases where there are no tests already created, but they do reduce the possibility of introducing regression errors. This means that developers should consider whether time is an important factor when determining if tests should be created when performing a refactoring.

An experience report by Sean Stolberg investigated the move-over of a standard development system at a corporate company to a continuous integration agile development environment with emphasis being shown to refactoring and code quality [74]. The recollection indicates a strong need for automated testing via presenting multiple issues that occur when developers do not implement automated tests in a continuous integration environment.

Gustavo Soares et al. conducted a study whereby refactoring engines were tested for validity [75]. Refactoring engines are agents that implement refactorings that were tested via the implementation of unit and behavioural tests. The study found multiple errors and bugs within commercially available refactoring agents. The fact that refactoring engines are validated by making use of automated tests further substantiates the need for the creation of automated tests when undergoing a refactor.

There is a plethora of different sources for reasons why automated tests are required when performing a refactor. These reasons emphasise the importance of including a testing phase within a method to improve maintainability in object-oriented software applications. There are multiple issues that should be considered when aiming to implement automated tests and these issues should be a part of the method to improve maintainability in object-oriented software applications.

Test Creation

The Test Creation Process

Because of the need to incorporate automated tests within a method for improving maintainability, there is a need to research techniques for creating automated tests. There has been a great deal of research conducted on ways to simplify test creation as well as techniques for creating tests. Most of the research (in terms of simplifying the test creation process) has gone into the creation of software testing frameworks. These testing frameworks were created with the goal of automating testing reducing the cost and making tests more reliable and efficient [76]. There are five commonly used testing framework types: package/folder based; data driven; keyword driven; hybrid; and behaviour driven. Package/Folder based testing frameworks make use of the
concept of abstraction. The application under test is viewed as consisting of individual modules that tests are created for. An example of a package/folder based test framework can be found at [77]. This is the most common form of testing framework and is available for most programming languages.

Data-driven testing frameworks were developed to address the need that arose when the application under test is required to perform similar functionality on different data sets. Data-driven testing frameworks assist developers with the storage, management and integration of the data with tests. Examples of data-driven testing frameworks are used in [38] and [39]. Keyword-driven testing frameworks are an extension of data-driven testing frameworks. They provide additional functionality in the handling of data. Two examples of keyword driven frameworks can be found at [40] and [41].

Hybrid testing frameworks consist of different combinations of functionality of package/folder based, data driven or keyword driven testing frameworks. They are developed for special cases of application types and are typically not widely available. An example of a hybrid testing framework can be found at [82].

The final type of commonly used testing framework is behaviour-driven testing frameworks. This type of framework was invented for the specific purpose of verifying application functional requirements. This means that this type of framework typically interacts with an applications user interface or other applications that the application interacts with. An example of a behaviour-driven testing framework can be found at [83].

To make a choice on the type of testing framework to use when creating tests, developers are required to use their own discretion. This is because the choice on testing framework is heavily reliant on application functionality, as well as developer expertise and experience. Developers should perform their own review on available testing frameworks and make an informed decision based on their findings.

Testability

Due to the need to test code before refactoring another concern arises, source code that is to undergo a refactor should be testable. This is a concern because there are certain common occurrences that arise in software development that result in code not being testable. Research done on the concept of improving source code testability is vast. Anthony Alwardt et al. states that there is no golden set of steps that can be followed in order to ensure source code testability [84]. There are only best practices that can be followed. These best practices can be adhered to by following the SOLID principles [85].

Table 3 shows what principle each letter in the SOLID acronym represents. Adhering to the SOLID principles increases the overall quality of the application while also facilitating the process of creating tests. All the SOLID principles make the refactoring process easier in one way or another. The following explanations are formulated using knowledge gained from [85]. A full explanation of the SOLID principles is out of the scope of this thesis, but [86]–[88] can be used for in-depth explanations.
Table 3: SOLID Principles [47]

<table>
<thead>
<tr>
<th>SOLID Letter</th>
<th>Principle Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>The Single-Responsibility Principle (SRP) — There should only be one reason for a class to change.</td>
</tr>
<tr>
<td>O</td>
<td>The Open/Closed Principle (OCP) — Software Components (Classes, methods, modules) should be open for extension but closed to modification.</td>
</tr>
<tr>
<td>L</td>
<td>The Liskov Substitution Principle (LSP) — Subtypes must be substitutable for their base types.</td>
</tr>
<tr>
<td>I</td>
<td>The Interface Segregation Principle (ISP) — Client classes should not be forced to depend on methods they do not use.</td>
</tr>
</tbody>
</table>
| D            | The Dependency-Inversion Principle (DIP):  
|              | 1 — Modules at a high level should not depend on low-level modules. They should both depend on abstractions.  
|              | 2 — Details should depend on abstractions. |

Adhering to the SRP means that class testing becomes easier. This happens because the amount of methods within the class is reduced — reducing the number of tests that are required to fully exercise the functionality of the class. The refactoring process is simplified since classes become less complex. If classes are less complex, it facilitates working with or making changes to them because it is easier to understand the way they work and what their responsibilities are.

Software entities that obey the OCP are written in such a way that it is relatively easy to add functionality. This is because these entities are designed with the goal of not having to make changes to existing code to add functionality to the entity. This goal is reached in using abstraction. Abstraction is the act of defining the essential features of an entity without revealing the details of the entity’s implementation — resulting in the entity being closed to modification [89]. To add functionality to the entity, derivatives of the abstraction are created — resulting in the entity being open for extension. Conforming to the OCP makes code testable by reducing the amount of dependencies between software entities within the application.

The LSP does not directly determine if an application is testable. It does however simplify the testing process. If a sub-class cannot be substituted for a base or abstract class, tests written for the base class will not pass for the sub-class. This means that extra tests would have to be created for the sub-class to test functionality that has already been exercised for the base class. Adhering to the LSP increases application maintainability because application coupling decreases.
The ISP and SRP are closely related. If dependent classes have access to methods that they do not use this could be an indication that the class does not adhere to the SRP. If a class has access to functions or members that it cannot use, tests cannot be created to verify the behaviour of the class when those members are called.

The DIP does not directly affect the testability of a software entity, but it does simplify the refactoring process. The process becomes easier because any refactoring that occurs on low-level modules will not affect high-level modules, meaning that fewer changes will have to be made. This results in a higher amount of changes needing to be made in high-level modules, but the benefit of having to do so is greater. Since high-level modules contain the most important aspects of the way that an application works, any changes that are made will have to be done to ensure that the application meets its requirements.

This section describes how each SOLID principle ensures that software applications are testable, as well as how each principle improves the maintainability of software applications.

2.4 Assessing the Maintainability of Software Applications

The area of research concerned with the assessment of software maintainability that is relevant to a method to improve maintainability in object-oriented software application is separated into two fields. There are techniques for assessing software quality, and techniques for assessing software maintainability only.

There is a plethora of different techniques that can be used to assess software quality. Jean-Louis Letouzey and Thierry Coq present a paper describing the SQALE (Software Quality Assessment based on Lifecycle Expectations) [90]. Usage of the SQALE method allows developers to gain insight on all quality aspects of a software application. It makes use of code metrics and the amount of effort required to remedy errors within a software application to assess different quality characteristics. It is only applicable for usage within certain software applications. The model depends on the ability to implement base measures for certain quality metrics. This method is not applicable for usage within a method to improve maintainability because of its application limitations.

Stefan Wagner et al. developed an operationalised product quality model and assessment tool called the Quacom approach [91]. This model allows developers to obtain insights on both abstract and explicit quality characteristics. The Quacom approach has only been applied to a single software system. This means that it is not applicable for usage within a method to improve maintainability because of the work yet to be completed to ensure its validity.

Miltiadis G. Siavvas et al. developed an integrated framework for generating software quality models that are capable of being customised to user needs [92]. The framework makes use of Fuzzy multi-criteria decision making to model uncertainty imposed by expert judgement and software metrics. This framework is not applicable for usage within a method to improve maintainability because of its complexity, as well as the fact that it assesses all characteristics of software quality.
Meng Yan et al. developed the automated aggregation method for software quality model [93]. This model makes use of low-level code metrics for the determination of the health of different quality characteristics. It makes use of a topic modelling technique to estimate the probabilistic weight of a software metric and determine its effect on software quality characteristics. This process of determining weights makes use of software benchmarks. This technique is not applicable for usage within a method to improve maintainability because of its complexity. It should be noted that low-level software metrics can be used to assess the health of software quality characteristics.

There are multiple techniques for assessing the maintainability of software applications. One such technique is presented by Muhammed Anan et al. who developed an architecture-centric model for assessing software maintainability [94]. At the heart of the model is a metric based on a mathematical model representing a snapshot of the maintainability of the system. This method of assessing maintainability is not applicable for usage within a method to improve maintainability because of its complexity as well as the fact that it only assesses maintainability based on system architecture.

Nejmeddine Tagoug presents a maintainability assessment in object-oriented design [95]. The assessment makes use of a metric that can be calculated by measuring the amount of effort it takes to remedy issues within software applications that affect software maintainability. This technique is not applicable for usage within a method to improve maintainability, as the assessment will be performed after remediying issues. It is worth noting that application maintainability can be measured using the amount of effort to fix issues.

A paper by Fang Zhou et al. presents the findings of a study that constructs and tests multiple software maintainability modelling techniques [31]. The first model makes use of a hierarchical multidimensional assessment. Hierarchical modelling tools are used to categorise characteristics of a software product’s maintainability. Another model uses polynomial regression. Metric attributes are used to assess software maintainability by creating a polynomial equation. Complexity is used to estimate the maintainability of software applications in another of the techniques constructed and tested by Fang Zhou. Two techniques are investigated for quantifying application complexity. Complexity is estimated by the way in which the software application differs from its modules/components. The second technique for quantifying complexity is accomplished via underlying factors. These factors are software metrics quantifying unobservable quantities. The final model that is constructed and tested makes use of a statistical technique called principle components analysis. This analysis reduces the number of components used to construct maintainability models and reduce the collinearity of software metrics. Each modelling technique has advantages and disadvantages and makes use of a statistical technique.

There is a plethora of techniques for assessing software quality, and techniques for assessing software maintainability only, which are not mentioned within this literature study. All techniques are vastly different and provide insight on quality with differing degrees of effectiveness and accuracy. There is no consensus on
the best technique to use. However, there is a consensus that code metrics can be used to determine the health of software quality characteristics.

A paper by Isja Heitlager et al. describes the usage of a model for measuring maintainability based on the maintainability sub-characteristics defined by ISO 9126 (ISO 25010’s predecessor) [96]. The sub-characteristics are assessed using code metrics related to each, which can then be used to obtain a maintainability index. Further work utilising the technique presented by Heitlager is presented by Robert Baggen et al. [97]. This continuation presents the benefits of applying the technique within real-world applications. Results achieved are favourable. This indicates that the technique may be applicable for usage within a method to improve maintainability. The only issue arises from the fact that the technique is based on ISO 9126 instead of the updated ISO 25010.

The technique by Isja Heitlager et al. can be updated using The Compendium of Software Quality Standards and Metrics developed by Rudiger Linke and Welf Lowe in 2007 [34]. It is developed with the purpose of determining which code metrics give an indication of the health of application quality characteristics defined by ISO 9126.

ISO 25010 is a revision of ISO 9126 [98]. ISO 9126 also proposes a product quality model that is very similar to the model proposed by ISO 25010. Figure 4 shows the quality model used by ISO 9126. As can be seen, there are changes made in ISO 25010 to the original product quality model of ISO 9126. The sub-characteristics defined in ISO 25010 and ISO 9126 can be matched to each other meaning that the code metrics defined by Rudiger Lincke et al. can still be used to determine the health of the primary quality characteristics of ISO 25010.

![Figure 4: ISO 9126 Product Quality Model](image)

Table 4 shows the code metrics that can be used to determine the health of each maintainability sub-characteristic defined by ISO 25010. This table was formulated using [32] and [34]. Table 4 uses metric abbreviations found in Table 5. Alberto S. Nuñez-Varela et al. conducted a systematic literature review surveying the research that has been done in the field of code metrics [32]. While conducting the literature review, Alberto S. Nuñez-Varela et al. categorised code metric usage with software quality characteristics.
These code metric usages were compared with the code metric usages defined by Rudiger Lincke et al. and Table 4 was formulated.

**Table 4: Maintainability Characteristics and Applicable Code Metrics**

<table>
<thead>
<tr>
<th>Primary Characteristic</th>
<th>Secondary Characteristic</th>
<th>Applicable Code Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maintainability</strong></td>
<td>Modularity</td>
<td>CBO; CDLC; CDLC; Ce; CF; DAC; I; LD; MPC; PDAC; LCOM; ILCOM; TCC; LOC; SIZE2; NOM; CC; WMC; CC; WMC; RFC; DIT; NOC; Ca; LOD;</td>
</tr>
<tr>
<td>Reusability</td>
<td>SIZE2; NOM; CC; WMC; DIT; NOC; Ca; LOD; LOC; RFC; CBO; CDLC; CDLC; Ce; CF; DAC; I; LD; MPC; PDAC; LCOM; ILCOM;</td>
<td></td>
</tr>
<tr>
<td>Analysability</td>
<td>LOC; SIZE2; NOM; CC; WMC; RFC; DIT; CBO; CDLC; CDLC; Ce; CF; DAC; I; LD; MPC; PDAC; LCOM; ILCOM; TCC; LOD;</td>
<td></td>
</tr>
<tr>
<td>Modifiability</td>
<td>LOC; SIZE2; NOM; CC; WMC; RFC; DIT; NOC; CBO; CDLC; CDLC; Ce; CF; DAC; I; LD; MPC; PDAC; LCOM; ILCOM; TCC; LOD;</td>
<td></td>
</tr>
<tr>
<td>Testability</td>
<td>LOC; SIZE2; NOM; CC; WMC; RFC; DIT; CBO; CDLC; CDLC; Ce; CF; DAC; I; LD; MPC; PDAC; LCOM; ILCOM; TCC; TOD;</td>
<td></td>
</tr>
</tbody>
</table>
Table 5: Code Metric Definitions

<table>
<thead>
<tr>
<th>Label</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>Lines of Code</td>
</tr>
<tr>
<td>SIZE2</td>
<td>Number of Attributes and Methods</td>
</tr>
<tr>
<td>CC</td>
<td>McCabe Cyclomatic Complexity aka Cyclomatic Complexity</td>
</tr>
<tr>
<td>WMC</td>
<td>Weighted Method Count</td>
</tr>
<tr>
<td>RFC</td>
<td>Response For Class</td>
</tr>
<tr>
<td>DIT</td>
<td>Depth of Inheritance Tree</td>
</tr>
<tr>
<td>NOC</td>
<td>Number of Children</td>
</tr>
<tr>
<td>Ca</td>
<td>Afferent Coupling</td>
</tr>
<tr>
<td>CBO</td>
<td>Coupling Between Objects</td>
</tr>
<tr>
<td>CDBC</td>
<td>Change Dependency Between Classes</td>
</tr>
<tr>
<td>CDOC</td>
<td>Change Dependency Of Classes</td>
</tr>
<tr>
<td>Ce</td>
<td>Efferent Coupling</td>
</tr>
<tr>
<td>CF</td>
<td>Coupling Factor</td>
</tr>
<tr>
<td>DAC</td>
<td>Data Abstraction Coupling</td>
</tr>
<tr>
<td>LD</td>
<td>Locality of Data</td>
</tr>
<tr>
<td>MPC</td>
<td>Message Passing Coupling</td>
</tr>
<tr>
<td>PDAC</td>
<td>Package Data Abstraction Coupling</td>
</tr>
<tr>
<td>LCOM</td>
<td>Lack of Cohesion in Methods</td>
</tr>
<tr>
<td>ILCOM</td>
<td>Improvement of LCOM</td>
</tr>
<tr>
<td>TCC</td>
<td>Tight Class Cohesion</td>
</tr>
<tr>
<td>LOD</td>
<td>Lack Of Documentation</td>
</tr>
<tr>
<td>I</td>
<td>Instability</td>
</tr>
</tbody>
</table>

2.5 Assessing a Method to Improve Maintainability

The final objective of the literature study is the determination of a way to assess a method to improve maintainability. This is required to ensure that the method addresses the problem statement it means to solve. A way to determine the effect the method has on the maintainability of a software application is required. The Maintainability Index (MI) is one such method. The MI is a means of determining the relative maintainability of a software application using a single value. The concept of utilising a single index of maintainability was first introduced by Paul Oman and Jack Hagemeister at the Conference for Software Maintenance in 1992 [103].

The authors propose that software maintainability can be thought of as a hierarchy consisting of the management practices being employed within a software development environment; the operational software
and hardware environments involved in working with the software system; and the actual software system being placed under maintenance[103]. For the purposes of creating the MI, the authors only focused on the actual software system. It is found that the maintainability of the actual software system can be thought of as consisting of software maturity attributes, the software’s source code, and the supporting documentation related to the software system.

Software maturity attributes are the physical characteristics and measures related to the age and usage of the target software system [103]. The software’s source code consists of three more sub-categories: the system control structure, the information structure and the systems documentation. System control structure characteristics are those related to the intermodular control attributes, the choice and use of control flow constructs, the manner in which the program or system is decomposed into algorithms, and the method in which those algorithms are implemented [103]. The system’s information structure characteristics are those relating to the information storage and flow in a program or system, including global and intermodular data definition and data flow characteristics, and the system input and output characteristics [103]. System documentation relates to the mapping of the systems documentation to the source code and the physical attributes of the documents set.

For each of the aspects mentioned, there is a matching code metric that can be used to ascertain the health of different sub-aspects. For example, a sub-aspect of the systems control structure is the systems complexity. The Cyclomatic complexity of the system can be used to determine the systems complexity.

![Target Software System Maintainability Hierarchy](image)

Figure 5: Target Software System Maintainability Hierarchy [103]
Figure 5 shows a diagram describing the hierarchy of aspects that relate to the maintainability of the target software system. The health of each aspect shown can be determined using a code-metric related to each [103]. To assess the maintainability of the target software system, the hierarchy shown above is thought of as a tree structure and the maintainability of the software system can be calculated using the following formula:

\[
\prod_{i=1}^{m} W_{D_i} \left( \frac{\sum_{j=1}^{n} W_{A_j} M_{A_j}}{n} \right)
\]

(10)

Where:

- \( W_{D_i} = \text{Weight of influence of Maintainability Dimension} \)
- \( W_{A_j} = \text{Weight of influence of Maintainability Attribute} \)
- \( M_{A_j} = \text{Metric of influence of Maintainability Attribute} \)

The simplification of the above formula was presented by Paul Oman and Jack Hagemeister within [104]. This is achieved utilising a polynomial regression model for each of the terms available that arise through the usage of each of the above formula. The polynomial regression model resulted in the derivation of equation (11).

\[
MI = 171 - 3.42 \times \ln(E_{Avg}) - 0.23 \times C_{Avg} - 16.2 \times \ln(LOC_{Avg}) + 0.99 \times CM_{Avg}
\]

(11)

Where:

- \( E_{Avg} = \text{Average Halstead Effort per package/folder} \)
- \( C_{Avg} = \text{Average Cyclomatic Complexity per package/folder} \)
- \( LOC_{Avg} = \text{Average Lines of Code per package/folder} \)
- \( CM_{Avg} = \text{Average Number of Comments per package/folder} \)

Successful usage of the MI is shown within [105]. [105] also presents a refined formula for calculating the MI utilising insights gained during the testing of the formula, as well as expert opinion. The refined formula is shown in (12).

\[
MI = 171 - 5.2 \times \ln(V_{Avg}) - 0.23 \times C_{Avg} - 16.2 \times \ln(LOC_{Avg}) + 50 \times \sin \sqrt{2.46 \times CM_{per}}
\]

(12)

Where:

- \( V_{Avg} = \text{Average Halstead Volume per package/folder} \)
- \( C_{Avg} = \text{Average Cyclomatic Complexity per package/folder} \)
- \( LOC_{Avg} = \text{Average Lines of Code per package/folder} \)
- \( CM_{Avg} = \text{Percentage of Comments per package/folder} \)
The creators of the MI have suggested that the term related to the percentage of comments can be neglected when calculating the maintainability index for a system. This is applicable when developers place a lower weight on the importance of documentation in terms of the maintainability of the software system. Values calculated for the MI using (13) range between 0 and 171. The interpretation of the calculated values can be achieved by instead representing the value as a percentage using (13).

\[
MI = \text{MAX}(0, 171 - 5.2 \times \ln(V_{AVG}) - 0.23 \times C_{AVG} - 16.2 \times \ln(LOC_{AVG}) \times \frac{100}{171})
\] (13)

A further requirement for a method to improve maintainability in object-oriented software applications is found through the understanding of the MI. The MI represents an index of the three aspects that represent or influence the maintainability of a software applications source code. These are: the source code control structure; the source code information structure; and the source code typography, naming and commenting. This means that a method to improve maintainability should ensure that the identified areas of low-quality source code should consist of defects in the three influencers of source code maintainability.

The finding above is accounted for within the investigation of Section 2.2.2. Section 2.2.2 concludes that the indicators of low maintainability that should be the focus of a method to improve maintainability are: anti-patterns; code smells; grime and a lack of tests.

The existence of anti-patterns and grime within a software application affects the control and information structure of an applications source code. A lack of tests and the presence of grime within the software application influences the code typography, naming and commenting of the application source code. The conclusions reached for each of the indicators of low maintainability are reached by examining the definitions of each.

This section presented the MI as a means of assessing the effect a method to improve maintainability has on the maintainability of a software application. This was required to ensure that there is a means of assessing if this study accomplishes its objective of creating a method to improve maintainability in object-oriented software applications. It is also found that a further requirement of a method to improve maintainability is that it should positively influence the three indicators of source code maintainability (the source code control structure; the source code information structure; and the source code typography, naming and commenting).

### 2.6 Conclusion

A literature study is completed to achieve the research objectives stated in Section 1.7. Techniques for identifying areas of low maintainability in software applications are investigated by determining what the indicators of low maintainability are. These indicators are identified by understanding what the indicators of code, test, architectural and design debt are. Research revealed that indicators of areas of low maintainability are anti-patterns, code smells, grime and a lack of tests. These indicators are identified using code metrics and automated tools. Multiple papers that make use of code metrics for the identification of areas of low
maintainability are reviewed, and code metrics that are applicable for usage within a method to improve maintainability are thoroughly investigated. Research completed on the concept of using automated tools for the identification of low maintainability is reviewed and it is concluded that a method to improve maintainability should make use of an automated tool for the identification of areas of low maintainability.

Techniques for improving software maintainability are reviewed to achieve the next literature objective. The concept of refactoring software applications is examined in detail followed by a review of papers that refactor software applications to improve software maintainability. While investigating the concept of refactoring software applications, the need for including automated tests while performing a software refactor is identified. This need is investigated, and the results of the investigation are presented. Because automated tests are required to accurately perform a software refactor, the processes for creating automated tests as well as ensuring software testability are reviewed.

Penultimately, techniques for assessing the maintainability of software applications are presented. Papers that assess the general software quality, as well as papers that assess the maintainability of software applications are reviewed. It is concluded that techniques for assessing software application quality make use of source code metrics.

Finally, the MI is explained as a means of determining the effect a method to improve maintainability has on the maintainability of a software application. This was required to ensure that the method addresses its problem statement.

The accomplishment of all research objectives leads to the conclusion that the method to improve maintainability in object-oriented software applications can be developed. The full method is presented in Chapter 3.
The previous chapter presented findings of a literature study conducted to determine the knowledge required to develop a method for improving maintainability in object-oriented software applications. Knowledge gained is used to formulate refined requirements for the method. The method consists of four high-level steps: identification; testing; refactoring; and maintainability assessment. This chapter presents the processes that are followed within each step of the method to improve maintainability in object-oriented software applications.
3 A Method for Improving Maintainability in Object-Oriented Software Applications

3.1 Preamble

This chapter presents a method for improving maintainability in object-oriented software applications. The method is formulated using techniques discussed in the literature study and has the goal of accomplishing the objectives listed in Section 1.7.

Before formulating the method, refined requirements are formulated using insights and knowledge gained from the literature study:

1. The method should consist of four high-level processes or steps: identification; testing; refactoring; and a maintainability assessment.

2. The identification process should:
   a. provide developers with the knowledge required to determine which classes or files should be refactored.
   b. result in developers obtaining a list of files ordered by maintainability — facilitating requirement 2a.
   c. use more than one method for obtaining lists of files ordered by maintainability — facilitating requirement 2a, as well as ensuring validity.

3. The testing process should:
   a. provide developers with instructions for the creation of an automated test suite.
   b. provide developers with a safety-net, preventing the addition of regression errors when modifying application source code.
   c. improve maintainability via reducing test debt within the software application.

4. The refactoring process should:
   a. remedy instances of grime within the software application.
   b. remedy instance of anti-patterns within the software application.
   c. remedy instances of code smells within the software application.

5. The maintainability assessment step/process should:
   a. be an optional step that can be performed at varying levels of completeness dependent on individual needs.
   b. make use of techniques used within the identification step of the method to satisfy minimal requirements for maintainability assessment completeness.
   c. make use of a formal method for defining the maintainability of the software application to fulfil in-depth requirements for maintainability assessment accuracy and completeness.
The method is presented by first providing a high-level overview of the method, then proceeding to provide detailed descriptions of each high-level step. These detailed descriptions correspond to a numbered requirement listed above.

### 3.2 Method Overview

<table>
<thead>
<tr>
<th>Identify</th>
<th>Test</th>
<th>Refactor</th>
<th>Assess</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Quality Assessment</td>
<td>Testability</td>
<td>Test</td>
<td>ISO 25010 maintainability sub-characteristics</td>
</tr>
<tr>
<td>Sonarqube Assessment</td>
<td>Sonarqube</td>
<td>Anti-patterns and grime</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6: A Method for Improving Maintainability in Object-Oriented Software Applications**

The method to improve maintainability in object-oriented software applications consists of four distinct steps: identification; testing; refactoring; and a maintainability assessment.

The identification step is broken down in two further steps: performing an application quality assessment; and performing a Sonarqube assessment of application source code. These steps can be performed in the order that the developer chooses. Further details of the identification step are discussed in Section 3.3.

The testing step consists of two sub-steps that ensure that the refactoring process does not affect application functionality, while also facilitating the process of improving maintainability. These sub-steps are: ensuring that identified areas of code are testable; and creating automated unit, integration or behavioural tests for identified areas of code. An in-depth discussion of the testing step of the method to improve maintainability is available in Section 3.4.

The refactoring step of the method to improve maintainability is the most important step, as it is responsible for remedying source code that suffers from a lack of maintainability. This step is broken down in two further sub-steps: manually identifying and addressing any instances of grime or anti-patterns found in the file that is to be refactored; and performing recommendations suggested by Sonarqube. The refactoring process is discussed in detail in Section 3.5.

The final step involved in the method to improve maintainability is the assessment of software application maintainability. This step is performed to determine the effect the method has on application maintainability.

The maintainability assessment is optional and can be performed in varying degrees of detail. The sub-steps involved are: re-performing an application quality assessment outlined in the identification process; re-
performing a Sonarqube analysis of application source code and ensuring that refactored files do not contain instances of code smells, bugs or vulnerabilities; and assessing application quality using ISO 25010 maintainability sub-characteristics. A discussion of the processes involved in the maintainability assessment is available in Section 3.5.

3.3 Identification of Areas of Low Maintainability

3.3.1 Overview
To improve software maintainability, there is a need to know where problem areas lie within the software application. This means that we need to identify areas of low maintainability and determine if these identified areas should be refactored or not.

The identification process of the method to improve maintainability can be broken down into three steps. Figure 7 shows the steps involved in this process. The first two steps can be performed in any order. The first of these steps is to assess the quality of the application. This is accomplished using code metrics; see Section 3.3.2 for an in-depth discussion of the Application Quality Assessment (AQA).

The next step is to analyse the application using Sonarqube. Sonarqube is a web-based application that provides insight into the quality of a software application; see Section 3.3.3 for the details of this step of the identification process. Both steps mentioned result in developers obtaining a list of files or classes that appear to be the least maintainable.

After obtaining the lists of files or classes from the Application Quality Assessment and the Sonarqube Assessment, these lists should be compared to determine which files or classes should be refactored. When making the decision on which classes should be refactored, developers should consider the following:

- For a file, where does it lie in each ordered list? If a file is high on both lists, then it should be considered a high priority for refactoring.
• For a file, what is the likelihood that changes will be made in the future? If there is a high chance of changes being made, then the file should be considered a high priority for refactoring since the amount of time to make any changes to the file will be decreased because of the refactoring.

• Are there external management aspects that may affect the refactoring process? These management factors depend on the type of environment the application is developed in.

When the classes or files that should be refactored have been determined, developers can move on to the testing phase of the method to improve maintainability.

3.3.2 Application Quality Assessment

The first step involved in the identification process is to assess the quality of the application source code. This is accomplished through the usage of code metrics. Code metrics provide quantitative data that allows developers to assess the health of an application [32]. There are different categories of metrics — each giving an insight towards different aspects of application health. The category of metrics that are used to identify a lack of code quality is code quality metrics. Quality metrics provide varying degrees of depth in terms of where code lacks quality (application/project, package/folder, class and method). These metrics also are only applicable for Object-Oriented programming languages. Detailed explanations of the metrics used when performing an AQA can be found within the literature study.

Figure 7 shows a flow diagram that describes the steps involved in the AQA. Squares indicate that developers are required to complete a certain task, and diamonds indicate that a decision is to be made.

The first step is to perform a MOOD metrics assessment. The MOOD metrics suite provides developers with information regarding application quality from a project/module level. After performing the MOOD metrics assessment, developers are required to answer the question: Do MOOD metrics indicate that application maintainability is lacking? This question is answered using insights gained from each of the metrics within the MOOD metrics suite.

If developers find evidence that there is a lack of maintainability at an application level, a Martin Packaging Metrics assessment and a CC metric at a package/folder level should be conducted. This provides developers with application quality information on a package/folder level. For each package/folder, developers should answer the question: Do Martin Packaging Metrics and CC metric indicate that the package lacks maintainability? To answer this question developers, use calculated values for each metric.

For each package that developers deem to lack maintainability, a CK metrics suite assessment and a CC metric assessment should be performed at a file/class level. Information regarding source code quality on a file/class level is obtained after completing these assessments. Using the values obtained for each metric, developers can answer the question: Do CK and CC metrics indicate that the file lacks maintainability?
Developers should add each file that appears to lack maintainability to a list sorted by files that receive the worst results for metrics within the CK metrics suite and the CC metric. This list is used when determining where focus is placed when determining which files should be refactored.

![Flowchart: Application Quality Assessment]

**Figure 8: Application Quality Assessment**
3.3.3 Sonarqube Assessment

This Sonarqube Assessment (SA) step of the identification step of the method to improve maintainability in software applications involves assessing the application source code using Sonarqube. Sonarqube is a web based open source quality management platform. It provides information on application code quality that includes the existence of bugs, vulnerabilities, code smells, duplications, source code size and languages used [99]. The goal of the SA is to obtain a list of classes or files that appear to lack maintainability.

Figure 8 shows a flow diagram describing the steps involved in the SA step of the method to address TD. Sonarqube has a plethora of pages or views that provide developers with different insights into application quality. For the purposes of the Sonarqube assessment, the views and features that are required to find a list of classes/files that appear to lack maintainability are discussed. For a full specification of all the features of Sonarqube, see the documentation which can be found at [99]. The steps relayed by Figure 8 should be considered as a guideline for one possible method that developers can follow to obtain the final list of classes. Developers can use any combination of Sonarqubes features, so long as a list of files ordered by maintainability is obtained.
Figure 9: Sonarqube Assessment
The first step of the Sonarqube assessment is for developers to view the Sonarqube Homepage. The details provided by this page allow developers to answer the question: Do details shown on the Sonarqube Homepage indicate that the application lacks maintainability?

Figure 10 shows the Sonarqube Homepage View for an example application. This page provides a high-level view of the application quality. It shows the total amount of bugs, vulnerabilities, code smells, duplicated blocks and lines of code.

A bug is an error or fault in an application that may cause the application to behave differently than expected.

A vulnerability is a block or line of code that decreases the security of the application. Software security is defined as the idea that an application will continue to function when malicious attacks occur [100].

Code smells are known structures in code that can cause the maintainability of an application to suffer [33].

Duplicated blocks are the number of repeated blocks of code within the application. They are a type of anti-pattern that is known to cause many errors when application maintainability is concerned.

The number of lines of code is the number of executable lines of code within the application.

Bugs, vulnerabilities and code smells are referred to as issues in Sonarqube and are rated by five severity levels [101]. These levels indicate the risk to productivity and the risk the issues pose to the application. The five levels of severity that are used are [101]:

- **Blocker** — the issue is a security or operational risk that could make the entire application unstable.
• Critical — the issue is a security or operational risk that may lead to unexpected behaviour without affecting the whole application.
• Major — the issue may have a high impact on developer productivity.
• Minor — the issue may have minimal impact on developer productivity.
• Info — the issue impact cannot be classified because not enough information is known.

In terms of the software quality model defined by ISO 25010 (see Table 1), bugs affect the reliability of an application. The “E” rating above the number of bugs (982) shown on Figure 10 is the Sonarqube reliability rating for the application. The reliability rating is calculated using the following scale:

• A — zero bugs
• B — at least one minor bug
• C — at least one major bug
• D — at least one critical bug
• E — at least one blocker bug

Sonarqube assesses application and file security using a rating that is based on the amount of vulnerabilities in an application. The “E” rating above the number of vulnerabilities (“35”) shown on the Sonarqube Homepage View indicates the rating Sonarqube gives for application security. The following scale is used to rate the security of the application [101]:

• A — zero vulnerabilities
• B — at least one minor vulnerability
• C — at least one major vulnerability
• D — at least one critical vulnerability
• E — at least one blocker vulnerability

After determining that the software application may lack maintainability, developers access the Sonarqube Code View page (by clicking “Code” tab on the Homepage). Figure 11 shows the Code view page at a package/folder level. This page shows a list of all the packages/folders in the application with the same details shown on the Homepage at a package level rather than a project/application level. The information relayed by the Sonarqube Code view page at a package level allows developers to answer the question: Do details shown on the Sonarqube Code view page indicate that a package lacks maintainability?

After determining that there are packages that lack maintainability, developers should click the name of each package/folder. This action reloads the Sonarqube Code view page with a list of all the files within the relevant package/folder.
Figure 11: Sonarqube Code View Page (Package/Folder Level)

Figure 11 shows an example of the Sonarqube Code view page at a file/class level. This page shows information regarding the amount of issues within each file. Using this information, developers can answer the question: Does file or class appear to lack maintainability? This question should be answered for each file within all packages that appear to lack maintainability. Files that are deemed to lack maintainability should be added to a list of files that lack maintainability. This list should be sorted by files that appear to have the most problem areas — this decision is left to the discretion of the developer performing the Sonarqube assessment. An example of a sorting mechanism would be to sum the amount of issues (bugs, vulnerabilities and code smells), then sort by number of issues, then duplication percentage and, finally, lines of code.

Figure 12: Sonarqube Code View Page (File/Class Level)
3.4 Automated Testing of Identified Areas of Low Maintainability

Before starting with the refactor of areas identified via the identification process, it is necessary to create integration or behavioural tests for identified classes — if they have not been created already. Automated tests provide developers with a safety-net that prevents the addition of regression errors, as well as aids in the understanding of the area of code that is to be refactored. The testing process can be repeated for each identified area of low quality code, then developers can move on to the refactoring step, or developers can repeat the testing and refactoring steps in sequence for each identified area of code.

Figure 13 shows a flow diagram describing the steps involved in the testing phase of the method to improve maintainability in software applications. The first question that developers should ask is: Has an adequate test suite been created? This question can be answered by answering the following sub-questions:

- What type of test framework is used?
- What types of tests have been created?
- Does the created test suite adequately exercise identified areas in such a manner that the developer can, with relative confidence, state that the likelihood of regression errors being introduced without detection during the refactoring process is minimal?

If developers conclude that automated tests are required before beginning with the refactoring process, developers should ensure that the file or class in question is testable. Developers can determine if a class is testable by ensuring that the answer to the following questions is yes:

- Does the file or class adhere to the SOLID principles?
- Does the file or class contain any examples of non-testable code found in [102]?
- Are developers certain that there will be minimal difficulty in creating tests for the file/class in question?

After ensuring that the file/class is testable, developers should determine the type or types of tests that should be created. The possible types of tests that could be created are unit, integration or behavioural tests. To determine the type(s) of tests required, developers should answer the following questions:

- Does file or class interact with the user interface or external applications?
  - If it does, then behavioural tests should be created.
- Does the file or class interact with a database, the external file system or any entities that cannot be isolated from the class/file that is currently in question?
  - If it does, then integration tests are required.
- Can the file or class be isolated from all entities that it interacts with?
  - If it can be, then unit tests should be created.
After answering the questions above, developers have the knowledge required to determine the type of test framework that is required. Developers can determine the required test framework type by using the knowledge presented in Section 2.3.2, as well as the types of tests required and consideration of the language the application under development is created in.

When the required test framework type has been determined, developers can move on to creating their chosen test types(s). After creating tests, developers should ensure that the following statement is true before moving on to the refactoring step of the method:

- Developers are confident that any changes made to the class or file in question will not introduce any regression errors that cannot be detected.
Has an adequate test suite been created?

Ensure that file/class is testable

Is file/class testable?

Determine type(s) of tests required

Have required test type(s) been determined?

Determine what test framework is required to implement required test type(s)

Has an adequate test framework been found?

Implement required tests

Figure 13: Testing Process
3.5 Refactoring Identified Areas of Low Maintainability

3.5.1 The Refactoring Process

The refactoring process consists of three steps: anti-pattern refactoring; grime refactoring and Sonarqube refactoring. Figure 14 shows a flow diagram describing the process. The first step that developers should complete is anti-pattern refactoring. This step consists of developers manually analysing a file or class and determining if there are any anti-pattern instances. After determining the presence of any anti-patterns, developers should refactor the code to remove them. Possible anti-patterns and their solutions can be found at [36]. Developers can move on to grime refactoring by ensuring that the answer to the following question is yes: Have identified anti-pattern instances been remedied?

The grime refactoring step consists of remedying instances of grime within identified areas. Grime consists of source code instances that do not adhere to a development methodology when developing the software application or source code instances that do not adhere to the coding standard used during application development. Developers should ensure that they recognise the following considerations when identifying instances of grime:

- Is there a design methodology used for the software application?
- Is there a coding standard, and does it specify any rules that relate to the software design methodology?
- Are there bug fixes that appear to be temporary solutions to software behaviour mishaps?

After addressing any instances of grime and anti-patterns, developers should move on to the Sonarqube refactoring process. Details of the steps involved in the Sonarqube refactoring process can be found in Section 3.5.2.

When the Sonarqube refactoring process is complete, developers should ask themselves one final question before assessing the impact the method for improving maintainability in object-oriented software applications has on application maintainability: Will a sufficient increase in application maintainability be achieved? This question is included to ensure that developers always take their maintainability goals into consideration when completing any of the refactoring processes.

The steps involved in the refactoring process are formulated in a manner such that if the steps are followed correctly, a satisfactory increase in maintainability should always be achieved.
Start

Anti-pattern refactoring

Identify instances of anti-patterns

Remedy identified anti-pattern instances

Have identified anti-pattern instances been remedied?

Grime refactoring

Yes

Identify instances of grime

Address identified instances of grime

Have identified instances of grime been addressed?

Yes

Perform Sonarqube refactoring

Will a sufficient increase in application maintainability be achieved?

Yes

End

No

No

No

No
3.5.2 Sonarqube Refactoring Process

The process for remedying bugs, vulnerabilities and code smells identified by Sonarqube is facilitated by the Sonarqube user interface. Figure 16 shows a flow diagram describing the steps involved in performing a Sonarqube refactoring. The first step that developers should complete is to re-run a Sonarqube analysis on the application after the changes specified by the anti-pattern and grime refactoring processes have been performed.

After re-running the Sonarqube analysis, there are two paths that can be followed to address issues highlighted by Sonarqube. The first path indicates that developers can view the Sonarqube Code View page for a single file and view all file issues that way. Figure 15 shows an example of the Sonarqube Code View page for a single file. Developers can view the contents of the file, with information on the severity and the location of issues. In addition to this, Sonarqube also shows users duplicated code — this duplicated code may appear in other files as well. Users can scroll up and down the page to view all issues for the selected file. This method for viewing file or class issues may overload the developer with too many issues and does not allow developers to filter issues. Developers should rather click the “Issues” link (the number above “Issues”). This will lead to the Sonarqube Issues page.

Figure 15: Sonarqube Code View Page (Single File/Class)
The second path that developers can follow to view file or class issues is to make use of the Sonarqube Issues page. Figure 17 shows the Sonarqube Issues page. This page allows users to view all project issues. Issues can
be filtered by: type; resolution; severity; status; new issues (day, week, month, year); rule; tag; project; module; directory (package or folder); file; assignee (the person assigned to the issue); author (author of the section of code); and language. Using available filters, developers can narrow down issues by concerns that are deemed high priority.

Using available filters, developers can narrow down issues by concerns that are deemed high priority.

Figure 17: Sonarqube Issues Page

In terms of the decision process for determining issue priority, developers should use the following definitions:

- **Issue Type**: These can be bugs, vulnerabilities or code smells. Developers should use the definitions for each issue type to make the decision on which type is deemed higher on the priority list of issues to address.

- **Resolution**: An issue resolution status can be: unresolved; false positive; removed; fixed; or won’t fix. This filter is useful for issue management within Sonarqube and allows developers to categorise issues that they do not deem valuable.

- **Severity**: The options for issue severity levels were described earlier. Developers should always ensure that issues at a blocker and critical severity level are addressed. Any issues at a lower severity level can be addressed if it is deemed feasible by developers.

- **Status**: The issue status can be: open; reopened; confirmed; resolved; or closed. The status can be used for issue management to allow developers to communicate and categorise issues that should be addressed based on if they are deemed feasible for refactoring.

- **New Issues**: This filter allows developers to filter issues by creation/detection date. If new issues are deemed to be of higher importance than old or vice versa; developers can use this knowledge as a distinguishing factor in the decision of where an issue lies on the priority list.

- **Rule**: The options for this filter are the rules that dictate if a pattern of code is constituted as an issue. These options are sorted by the amount of times they are violated. Developers can set a priority for
each of the rules that are in place for the application and use them as a distinguishing factor in the decision of where an issue lies on the priority list.

- **Tag**: Issue tags categorise issue rules. These tags provide users with more information regarding a rule and can be used to aid in decisions regarding the priority of a rule.

- **Project**: More than one project can be scanned for a Sonarqube analysis. This filter allows developers to narrow issues down to the project being refactored.

- **Package/Folder**: A module is a folder within a project that has been scanned by Sonarqube. This filter allows developers to narrow issues down to folders within the project.

- **File**: This filter allows developers to narrow issues down to files within the project. This filter is important in the context of the method, as it should ideally be set to the file/class that is currently being refactored.

After isolating issues that should be remedied for a file or class, developers are required to address each issue; the Sonarqube user interface facilitates this process.

Figure 18 shows an example of an issue. Table 6 describes what each of the elements shown on Figure 18 represents. The information described for an issue provides developers with all the information required to address the issue. In-depth issue details can be accessed by clicking the in-depth issues button (the ellipsis). This will reveal a view like Figure 19. The view provides an explanation of why the issue may be a problem and an example of code that does not comply with the rule, as well as a compliant example. Using the example, developers can address the issue.

Using the information described by the previous paragraph, developers can address each issue within the file or class being refactored. Developers can consider the Sonarqube refactoring process to be complete after ensuring that the answer to the following question is yes: Have all issues at appropriate filter levels been remedied?
### Table 6: Sonarqube Example Issue Details

<table>
<thead>
<tr>
<th>Detail</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Describes the issue details</td>
<td>Reduce this &quot;switch/case&quot; number of lines from 38 to at most 10, for example by extracting code into function.</td>
</tr>
<tr>
<td>Type</td>
<td>Bug/Vulnerability/Code Smell</td>
<td>Code Smell</td>
</tr>
<tr>
<td>Severity</td>
<td>Blocker/Critical/Major/Minor/Info</td>
<td>Major</td>
</tr>
<tr>
<td>Status</td>
<td>Open/Reopened/Confirmed/Resolved/Closed</td>
<td>Open</td>
</tr>
<tr>
<td>Assignee</td>
<td>Sonarqube user name</td>
<td>Not assigned</td>
</tr>
<tr>
<td>Estimated</td>
<td>Estimated time to rectify issue</td>
<td>5min effort</td>
</tr>
<tr>
<td>Effort</td>
<td>Reveals in-depth issue details (see Figure 18)</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>The amount of time since the issue was created</td>
<td>A month ago</td>
</tr>
<tr>
<td>Line Number</td>
<td>The line number that the issue begins at</td>
<td>L145</td>
</tr>
<tr>
<td>Number</td>
<td>Open issue in new tab</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>Identiﬁes the type of issue</td>
<td>Brain-overload</td>
</tr>
<tr>
<td></td>
<td>Filter similar issues button</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Expands all issues in file revealing file contents and location of the clicked issue</td>
<td></td>
</tr>
</tbody>
</table>

Figure 18: Example Issue
3.6 Maintainability Assessment

3.6.1 Overview

The maintainability assessment is performed to assess the effect the method for improving maintainability in object-oriented software applications has on application maintainability.

This step is an optional one that can be performed in a range of detail in terms of the completeness. Developers can rerun a Sonarqube analysis on the code after completing the refactoring step; they can perform an AQA in reverse; they can assess the quality of newly refactored code against the maintainability sub-characteristics defined by ISO 25010; or any combination of the three assessment possibilities.

3.6.2 Application Quality Assessment

An AQA can be used as a means of determining the effect the method has on application maintainability with a few minor modifications. Succinctly, developers can perform the AQA in reverse for areas of code that have been refactored.

Figure 20 shows a flow diagram representing the general process that should be followed when performing an AQA in reverse. The steps shown in the diagram should be performed at a file/class, package/folder level, and application level. This means developers will have an insight into the change in maintainability at different levels of resolution.

Developers should use their own preferences and priorities when determining if they are satisfied with the changes in values in metrics used within the AQA. If developers are not satisfied with the change in metric value, developers should determine if improving the metric will result in a change in another metrics’ value. If it does result in a change, developers should move to the next metric. If no change will occur, then developers
should try to improve the metric value using the tools presented in the refactoring step of the method to improve maintainability.

3.6.3 Sonarqube Assessment
If developers choose to make use of a SA to assess the effect the method for improving maintainability has on application maintainability, the following questions should be answered by making use of several Sonarqube views:

- Using the Sonarqube Homepage view:
  o Has the refactoring process introduced any new issues?
  o Has the number of issues decreased?
  o Have the maintainability, reliability and security ratings changed?

- Using the Sonarqube Code View page for each file that has been refactored or the Sonarqube Issues page:
  o Have all issues at pre-determined filter levels been addressed?
  o Have any new issues been introduced?
  o Has the number of issues decreased?

Figure 20: General Process for Application Quality Assessment
Using the answers to the questions above, developers can determine if they are satisfied with the effects the changes dictated by the method have had on the application in terms of metrics provided by Sonarqube. If developers are not satisfied with the changes found, they should try to remedy the metric values using tools provided by the refactoring step of the method to improve maintainability.

3.6.4 ISO 25010 Sub-characteristic Assessment

The final maintainability assessment method is accomplished by assessing the maintainability of an application using maintainability sub-characteristics defined by ISO 25010. This assessment type is suited to users of the method that require insights that allow developers to determine the effect the testing and refactoring steps of the method have had on the application, in terms of maintainability characteristics defined by ISO 25010.

Figure 21 shows a flow diagram describing the steps involved. Developers make use of code metrics as a means of assessing the health of each maintainability sub-characteristic. Tables 7 and 9 show the code metrics that are applicable for usage when determining the health of each maintainability sub-characteristic.

Developers should determine if the ISO 25010 sub-characteristic assessment is to be used as a means of assessing maintainability before beginning with the refactoring process. This is because developers should choose metrics that are used for assessing the health of each sub-characteristic and calculate their values before modifying application source code. These values are used as a benchmark when assessing the health of each maintainability sub-characteristic.

After performing the testing and refactoring step of the method to improve maintainability, developers should assess the health of each maintainability sub-characteristic. This is achieved by calculating the values of each chosen metric (applicable for the sub-characteristic under assessment) and determining if the developer is satisfied with the changes that were dictated by the method. For example, metrics related to application analysability may indicate that analysability has deteriorated in the case where analysability is a priority.
Determine what metrics are going to be used to assess the health of each maintainability sub-characteristic defined by ISO 25010.

Calculate values for each decided upon metric.

Complete Testing and Refactoring steps of method.

Calculate chosen metric values.

For each ISO 25010 sub-characteristic

Satisfied with change?

Yes

End

Yes

No

Try to improve metric value using tools presented in refactoring step of method.

Will remedying this change affect another metric?

Yes

No

Figure 21: ISO 25010 Quality Verification Process
<table>
<thead>
<tr>
<th>Primary Characteristic</th>
<th>Secondary Characteristic</th>
<th>Applicable Code Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>Modularity</td>
<td>CBO; CDBC; CDOC; Ce; CF; DAC; I; LD; MPC; PDAC; LCOM; ILCOM; TCC; LOC; SIZE2; NOM; CC; WMC; CC; WMC; RFC; DIT; NOC; Ca; LOD;</td>
</tr>
<tr>
<td></td>
<td>Reusability</td>
<td>SIZE2; NOM; CC; WMC; DIT; NOC; Ca; LOD; LOC; RFC; CBO; CDBC; CDOC; Ce; CF; DAC; I; LD; MPC; PDAC; LCOM; ILCOM; TCC;</td>
</tr>
<tr>
<td></td>
<td>Analysability</td>
<td>LOC; SIZE2; NOM; CC; WMC; RFC; DIT; CBO; CDBC; CDOC; Ce; CF; DAC; I; LD; MPC; PDAC; LCOM; ILCOM; TCC; LOD;</td>
</tr>
<tr>
<td></td>
<td>Modifiability</td>
<td>LOC; SIZE2; NOM; CC; WMC; RFC; DIT; NOC; CBO; CDBC; CDOC; Ce; CF; DAC; I; LD; MPC; PDAC; LCOM; ILCOM; TCC; LOD;</td>
</tr>
<tr>
<td></td>
<td>Testability</td>
<td>LOC; SIZE2; NOM; CC; WMC; RFC; DIT; CBO; CDBC; CDOC; Ce; CF; DAC; I; LD; MPC; PDAC; LCOM; ILCOM; TCC; TOD;</td>
</tr>
</tbody>
</table>
Table 8: Code Metric Definitions

<table>
<thead>
<tr>
<th>Label</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>Lines of Code</td>
</tr>
<tr>
<td>SIZE2</td>
<td>Number of Attributes and Methods</td>
</tr>
<tr>
<td>CC</td>
<td>McCabe Cyclomatic Complexity aka Cyclomatic Complexity</td>
</tr>
<tr>
<td>WMC</td>
<td>Weighted Method Count</td>
</tr>
<tr>
<td>RFC</td>
<td>Response for Class</td>
</tr>
<tr>
<td>DIT</td>
<td>Depth of Inheritance Tree</td>
</tr>
<tr>
<td>NOC</td>
<td>Number of Children</td>
</tr>
<tr>
<td>Ca</td>
<td>Afferent Coupling</td>
</tr>
<tr>
<td>CBO</td>
<td>Coupling Between Objects</td>
</tr>
<tr>
<td>CDBC</td>
<td>Change Dependency Between Classes</td>
</tr>
<tr>
<td>CDOC</td>
<td>Change Dependency Of Classes</td>
</tr>
<tr>
<td>Ce</td>
<td>Efferent Coupling</td>
</tr>
<tr>
<td>CF</td>
<td>Coupling Factor</td>
</tr>
<tr>
<td>DAC</td>
<td>Data Abstraction Coupling</td>
</tr>
<tr>
<td>LD</td>
<td>Locality of Data</td>
</tr>
<tr>
<td>MPC</td>
<td>Message Passing Coupling</td>
</tr>
<tr>
<td>PDAC</td>
<td>Package Data Abstraction Coupling</td>
</tr>
<tr>
<td>LCOM</td>
<td>Lack of Cohesion in Methods</td>
</tr>
<tr>
<td>ILCOM</td>
<td>Improvement of LCOM</td>
</tr>
<tr>
<td>TCC</td>
<td>Tight Class Cohesion</td>
</tr>
<tr>
<td>LOD</td>
<td>Lack Of Documentation</td>
</tr>
<tr>
<td>I</td>
<td>Instability</td>
</tr>
</tbody>
</table>

3.7 Conclusion

A method for improving maintainability in object-oriented software applications is presented. The method consists of four high-level steps: identification; testing; refactoring and maintainability assessment. The details of performing each step were presented. The next step of this study is to determine if all method requirements have been met, and if the method can be applied to a real-world software application without the need for further work to improve the method.
CHAPTER 4

METHOD VERIFICATION, VALIDATION AND ANALYSIS OF RESULTS

The previous chapter introduced the processes and steps followed within the method. Study objectives state that the method should be verified, ensuring that method requirements are met before applying the method to a real-world software application. Verification of the method is achieved through the usage of a survey. This chapter presents an overview of the survey and analyses responses received to determine if the method can be considered verified. After verifying the method, this chapter presents a case study used to validate the method and ends with a conclusion regarding if the method achieves its requirements and by relation if the study achieves its objectives.
4 Method Verification, Validation and Analysis of Results

4.1 Preamble

This chapter presents the method used to verify the method to improve maintainability in object-oriented software applications. The verification of the method is achieved through a survey created with the goal of determining if the requirements for each step of the method are addressed.

After verifying the method, this chapter will present the method used for validation. The validation is achieved by performing a case study that applies the method to a suitable software application. The details of the implementation of the method will be presented and emphasis is placed on finding insights into the efficacy of the method in terms of how well it addresses its requirements and ways that the method can be improved.

After performing the case study, the method is assessed using a neutral maintainability assessment primarily to determine if the method addresses its primary objective (improving maintainability). This method assessment is also used to provide insights into the way that the method affects software maintainability.

Finally, a discussion is presented with the goal of determining if the method to improve maintainability addresses its requirements, achieves its objectives and presents any information that is relevant when applying the method to improving maintainability to future software applications.

4.2 Survey Development

To ensure that the method is complete enough to justify applying it to a real-world application, the method needs to be verified for completeness. This means that the verification of requirements outlined in Section 3.1 is addressed by the method to improve maintainability in object-oriented software applications.

The verification of the requirements is achieved through the usage of a survey aiming to verify each of the requirements. The survey was created making use of Google Forms for ease of completion, as well as results analysis and compilation. Each potential survey respondent was sent an invite via email for participation in the survey.

An example of the survey used to verify the efficacy of a method to improve maintainability in object-oriented software applications is available in Appendix A. The survey is separated in ten sections:

- Section 1 — Introduction
  - Aims to determine the developers experience, programming paradigm exposure and their interest in software maintainability.
- Section 2 — Method Overview
• Provides developers with an overview of the method with high-level explanations of each of the steps/processes involved for providing developers with an adequate idea of the method. (The explanation is supplemented by the usage of Figure 6).

• Aims to obtain feedback regarding individual techniques used within each of the steps within the method.

• Intends to obtain feedback regarding whether the formulation of the order of steps/processes involved in the method will result in developers achieving their goals for maintainability improvement.

• Primarily intends to verify requirement 1 while also obtaining initial feedback on requirements 2 through 5.

• Section 3 — Identification
  • Provides survey respondents with an overview of the identification process of the method via Figure 7 as well as an explanation.
  • Intends to obtain feedback regarding the usage of the combination of the AQA as well as Sonarqube for the identification of areas that are refactoring candidates.
  • Aims to verify requirements 2a through 2d.

• Section 4 — Application Quality Assessment
  • Provides developers with a detailed explanation of the AQA making use of Figure 7.
  • Aims to obtain feedback regarding the perceived efficacy of the metrics used within the AQA.
  • Intends to ascertain if the steps involved in the AQA allows developers to achieve the goal of obtaining a list of files ordered by maintainability.
  • Intends to achieve further verification of requirements 2a through 2d.

• Section 5 — Sonarqube Assessment
  • Provides developers with a detailed explanation of the steps involved in the Sonarqube Assessment via the usage of Figure 8 and an explanation like Section 3.3.3.
  • Intends to obtain feedback regarding the efficacy of each Sonarqube view used within the method.
  • Aims to verify whether the steps proposed by the SA allows developers to accomplish the goal of obtaining a list of files ordered by maintainability.
  • Aims to obtain further verification of requirements 2a through 2d.

• Section 6 — Testing
  • Provides developers with a detailed explanation of the testing process via the usage of Figure 13 and an explanation like Section 3.4.
  • Aims to obtain feedback regarding whether the testing process improves maintainability, aids developer understanding in application source code and facilitates the creation of an automated test suite.
Intends to determine if the creation of an automated test suite will reduce the introduction of regression errors resulting in the simplification of the refactoring process.

Directly verifies requirements 3a through 3c.

Section 7 — The Refactoring Process

Provides a detailed explanation of the refactoring process through the usage of Figure 14 and an explanation like Section 3.5.1.

Intends to determine if the removal of anti-patterns will result in the improvement of software application maintainability.

Aims to ascertain if the removal of grime within software applications will result in improving maintainability.

Intends to verify if the removal of code smells improves maintainability in software applications.

Verifies requirements 4a and 4b.

Section 8 — Sonarqube Refactoring Process

Makes use of Figure 16 and an explanation like 3.5.2 for providing an in-depth explanation of the Sonarqube refactoring process.

Intends to ascertain if the steps proposed by the Sonarqube assessment process result in improved software maintainability.

Specifically aims to obtain feedback on Sonarqube features. This is done to determine Sonarqube’s usefulness as a tool for addressing code smells.

Verifies requirement 4c.

Section 9 — Maintainability Assessment

Provides an explanation of all the options for assessing maintainability provided by the method via making use of an explanation like Section 3.6, as well as Figure 20 and Figure 21.

Intends to determine if developers can determine if goals for maintainability improvement are met.

Aims to determine if performing an AQA as a means of assessment is sufficient.

Aims to determine if making use of Sonarqube’s different views are applicable for usage as a means of maintainability assessment.

Intends to ascertain whether ISO 25010 is an adequate means of assessing software maintainability.

Aims to determine what developers preferred means of assessing maintainability is.

Verifies requirements 5a through 5d.

Section 10 — Final Remarks

Aims to obtain developers final thoughts on the method after gaining knowledge on the method in its entirety.
There are a few threats to the validity of the results of the survey that required addressing while developing the survey.

Each section of the survey starts with a question utilizing a Likert scale as the answer with the purpose of determining if the explanation was sufficient for the survey respondent to grasp what the process is about. A Likert scale starts from one and ends at five, with one being strongly disagree and five being strongly agree. This is done to place weight on the rest of the responses for the section. If the answer is unfavourable, then the rest of the sections answers cannot be considered useful as the developer did not grasp what the process should do.

Another concern for the validity of results obtained from the survey is its length. Due to the high amount of explanation that is required to fully understand the survey, the length could not be reduced any further than it already was. This concern is addressed by means of ensuring that each question of the survey is requires an answer; the completion of the survey is incentivized; and survey respondents were chosen that would have a great deal of interest in the subject matter. Validity was further established by ensuring that candidates who were picked have at the least, a degree related to software development.

An overview of the survey used to verify that the method for improving maintainability in object-oriented software applications is provided. The results obtained for each section of the survey will be presented in the sections that follow and the determination of whether the requirement related to the survey section has been met will be made.

4.3 Survey Results

4.3.1 Survey Section 1 — Introduction

Twenty-two developers of varying experience were approached as survey participants. Eleven responses were received. Of the eleven developers, 18.2% of respondents had less than six months of software development experience, 45.5% had between six months and two years of experience and 36.4% of developers had more than two years’ experience.

The relatively high variability in the experience of survey respondents helps to validate the results obtained for the survey as many different perspectives are represented. The lack of survey completions can be attributed to the length of the survey and was expected when the survey was developed.

Survey question two asked developers to provide an indication on what programming paradigm they have been exposed to the most. 81.8% of survey respondents had the most exposure to the object-oriented programming paradigm, meaning that their input can be considered accurate in terms of the verification of the method. The additional respondents lack of experience is not an issue, as another perspective may prove useful.
An open-ended question was postulated regarding what software maintainability means to survey respondents. The detail of responses varied depending on the experience software developers had. Developers with greater experience gave answers closer to what ISO 25010 considers software maintainability. The full definition, including all the sub-characteristics defined by ISO 25010 was not prevalent in any of the responses. This indicates that the concept of maintainability in software applications is not fully understood.

The next question utilises a Likert scale (a scale from one to five with one being “not at all important” and five being “extremely important”). The question asks: how important is software maintainability to you? Four developers rated maintainability at a four, and the rest at five. This is a good indication that developers understand the importance of software maintainability although they may not fully understand it.

The results of the final question of survey section 1 is shown in Figure 22. Research done shows that the correct answer to this question is approximately 75%. 27.3% of respondents did not answer this correctly. Respondents that did not answer the question correctly may not have a grasp on the usefulness of the method. This should be considered when analysing survey responses.

In conclusion, the responses obtained for the questions of section 1 indicate that the responses for the rest of the survey can be considered as accurate verification of the method to improve maintainability in object-oriented software applications.

**4.3.2 Survey Section 2 — Method Overview**

Section 2 provides developers with an overall view of the method from a high level to obtain their first impressions. One survey respondent deemed the method explanation as insufficient while the rest of the respondents agreed that the explanation was adequate. The response for the developer that did not deem the explanation sufficient is discarded as part of the analysis.
The questions within this section all utilize Likert scales to gain insight into each of the processes and sub-processes of the method. Table 9 shows the results obtained for each question/process in survey section 2. The weights of the answers on the Likert scale can be considered a rating survey respondent’s have given for each of the processes within the method.

The ratings received for all the processes are very positive. All the considered responses either agree or strongly agree that each of the processes within the method will achieve their respective goals.

<table>
<thead>
<tr>
<th>Process</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>AQA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>SA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Testability</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Test Suite Creation</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Grime</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Anti-patterns</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Code Smells</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Maintainability Assessment</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Overall Method Rating</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

A final open-ended question was asked: Do you have any recommendations for the way that the method could be improved? Two of the responses had to do with the implementation of such a method within a real-world software development team. They were concerned with how well it would integrate. This is a valid point that should be considered if large scale adoption of the method is to be achieved. The rest of the responses either had nothing to add or were concerns that are addressed in subsequent detailed explanations of each process.

In terms of the requirements this section aims to verify, requirement 1 can be considered as verified. This is because there were no serious concerns with the process the method uses for improving maintainability in object-oriented software applications. The initial responses obtained for each of the sub-processes (identification, testing, refactoring and maintainability assessment) are all favourable, indicating that good results are expected for the verification of requirements dealing with these processes individually.

4.3.3 Survey Section 3 — Identification

This section of the survey explained the high-level steps of the identification process of the method (not much detail was provided on the steps involved in the AQA and SA). The explanation of the identification process was deemed adequate by all survey respondents.
Table 10: Identification — Survey Results

<table>
<thead>
<tr>
<th>Paraphrased Statement</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>The AQA and SA provide sufficient insight into locations of areas of low maintainability.</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Two lists of files ordered by maintainability allow developers to decide what areas should be refactored.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Two lists of files ordered by maintainability allow developers to formulate maintainability goals.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>The identification process accomplishes its goal.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 10 shows a paraphrased version of each of the statements within survey section 3, as well as the number of people that marked the respective rating. All results, except for the first statement, appear to be favourable. This can be attributed to the fact that the explanation may not alleviate concerns developers may have with the AQA and SA, and further analysis is required within subsequent sections for these processes.

Survey section 3 aims to verify requirements 2a through 2d. The results of the survey indicate that all four sub-requirements can be considered verified. This is useful as an initial conclusion, but further verification is required, since the questions posed are at a high-level and developers do not have a full picture of the identification process since the AQA and SA have not been presented in full.

4.3.4 Survey Section 4 — Application Quality Assessment

This section provides a detailed explanation of steps involved in the AQA. The explanation was deemed adequate by all survey respondents.
Table 11: Application Quality Assessment — Survey Results

<table>
<thead>
<tr>
<th>Metric Suite</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOOD Metrics</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Martin Package and CC Metrics</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>CK Metrics and CC Metrics</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Overall AQA Metric Suite and Process</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 11 shows the results of section 4 of the survey. Questions utilizing Likert scales were postulated regarding the effectiveness of the metric suites used at an application, package and file level, as well as a final question regarding whether the AQA process will result in developers being able to formulate a list of files sorted by maintainability.

Overall, there were no concerns with any of the metric suites used or the process. Four of the respondents indicated a neutral response to the metric suites used. This could be attributed to the fact that the developers may not have any experience regarding the metrics involved and may not feel comfortable committing to an answer without having first-hand experience using the metric suites. The fact that developers may not have experience using the metric suites should be considered during the validation process, since the favourable responses may not consider practical metric usage.

In terms of the requirements this section of the survey aims to verify, further verification of requirements 2a and 2b is achieved. The AQA provides developers with the knowledge required to determine what classes or files should be refactored and will result in developers obtaining a list of files ordered by maintainability.

4.3.5 Survey Section 5 — Sonarqube Assessment

This survey section provides developers with an in-depth explanation of the SA process. No concerns were conveyed regarding the explanation of the process.

Table 12: Sonarqube Assessment — Survey Results

<table>
<thead>
<tr>
<th>Sonarqube Page</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homepage</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Code View page (folder/package level)</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Code View page (file/class level)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Sonarqube Assessment Process</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Table 12 shows the results obtained for survey Section 5. All responses were either favourable or neutral. The presence of neutral responses as well as the lack of commitment to fully agreeing with the statements posed, could be because developers do not have first-hand experience with Sonarqube. Experience is required in the usage of a tool to fully endorse its usefulness.

Survey section 5 aims to further verify requirements 2a and 2b. The combination of favourable results of survey sections 4 and 5 further verifies that requirements 2b and 2d have been met. This can be inferred since survey results acknowledge that both the AQA and SA allows developers to achieve the overall goals of obtaining a list of files ordered by maintainability.

4.3.6 Survey Section 6 — Testing
Section 6 of the survey aims to verify the testing process of the method to improve maintainability in object-oriented software applications. The explanation of the process was deemed satisfactory by all survey respondents.

<table>
<thead>
<tr>
<th>Test Sub-process</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testability</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Automated Test Suite Creation</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Overall Method Test Process</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 13 shows the total number of people that selected a response for each question within survey Section 6. Responses received within Section 6 were all favourable, with a single response being neutral regarding ensuring that application source code is testable. The single neutral response can be attributed to a developer that lacks experience in terms of creating automated tests for software applications.

In terms of the requirements that are the subject of verification for this section of the survey, all requirements can be considered verified. Requirements 3a through 3c can tentatively be considered verified. The verification is considered tentative as there are a lack of questions related to this requirement. This means that extra attention should be given when the validation of the test process is done.

4.3.7 Survey Section 7 — The Refactoring Process
Survey section 7 was created with the intention of verifying the refactoring process of the method to improve maintainability in object-oriented software applications. No concerns were conveyed regarding the explanation of the process.
Table 14: The Refactoring Process — Survey Results

<table>
<thead>
<tr>
<th>Refactoring Process Step</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-Pattern Refactoring</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Grime Refactoring</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Overall Process</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 14 shows responses received for the questions within survey Section 7. All responses received can be considered favourable except for a single neutral response regarding the overall process involved in the refactoring step of the method. The single neutral response can be attributed to the lack of an explanation of the Sonarqube refactoring step of the refactoring process.

Section 7 of the survey was formulated with the intention of verifying requirements 4a and 4b. Requirement 4a can be considered verified, since responses received for the grime refactoring step of the method to improve maintainability are favourable. The same applies for requirement 4b regarding anti-patterns.

4.3.8 Survey Section 8 — Sonarqube Refactoring

This section of the survey provides respondents with a detailed explanation of the steps involved in the Sonarqube refactoring step of the refactoring process. The first question of the survey revealed no reservations regarding the explanation of the Sonarqube refactoring process.

Table 15: Sonarqube Refactoring — Survey Results

<table>
<thead>
<tr>
<th>Sonarqube Page</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code View page (Single File)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Issues page</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Overall Process</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 15 shows a summary of the responses obtained for survey Section 8. All results are favourable. The intention of the questions regarding the Sonarqube Code View page and the Issues page was to draw a comparison between both views and determine where preference was shown. The favourable results for both show that both pages are useful, but a line share of survey respondents indicated that they strongly agree with the usage of the issues page.

The information obtained from survey Section 9 verifies that requirement 4c is achieved. This is because Sonarqube issues include code smells.
A supplementary question is included within Section 8 intended to aid in the validation of the method to improve maintainability in object-oriented software applications. The question presented developers with each of the filters provided by the Sonarqube Issues page and asked respondents to rate their usefulness on a Likert scale. Table 16 shows a summary of the ratings provided by respondents and is ordered by usefulness. This information is used within the validation of the method during the refactoring process.

<table>
<thead>
<tr>
<th>Issue Page Filter Option</th>
<th>Useless</th>
<th>Somewhat useless</th>
<th>Neutral</th>
<th>Somewhat useful</th>
<th>Useful</th>
</tr>
</thead>
<tbody>
<tr>
<td>Severity</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>Status</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>File</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Issue Type</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Rule</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Package/Folder</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Resolution</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Project</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Tag</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>New Issues</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

4.3.9 Survey Section 9 — Maintainability Assessment
Survey section 9 provides respondents with an explanation of the options that the method provides for the assessment of maintainability after performing the refactoring process. The explanation of options provided were deemed satisfactory by survey respondents.

<table>
<thead>
<tr>
<th>Assessment Type</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>AQA</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>SA</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>ISO 25010 Maintainability Assessment</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Combination of Assessments</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>

All methods proposed for obtaining a gauge on the maintainability of the software application after performing the refactoring step of the method are rated highly by survey respondents. Making use of a combination of the methods proposed (as stated within the method) is rated extremely high. The neutral responses received could be attributed to a lack of knowledge on the tools used within each assessment type.
An additional question was postulated regarding what combination of methods respondents deem the most satisfactory to assess software maintainability. Table 18 shows the combinations of methods selected by survey respondents, as well as the number of respondents that selected the combination. The most popular choices for assessment are a combination of the AQA and SA or the usage of all three assessment types. The popularity of the AQA and SA combination can be attributed to the fact that it requires the usage of previously used techniques and allows developers to readily move towards re-performing the refactoring step of the method. The inclusion of the ISO 25010 assessment, along with the other methods, allows developers to gain a formal specification of application maintainability, and was chosen by developers with the most experience.

<table>
<thead>
<tr>
<th>Application Quality Assessment</th>
<th>Sonarqube Assessment</th>
<th>ISO 25010 Quality Assessment</th>
<th>Number of selections</th>
</tr>
</thead>
<tbody>
<tr>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>5</td>
</tr>
<tr>
<td>✅</td>
<td>✅</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>✅</td>
<td></td>
<td>✅</td>
<td>1</td>
</tr>
<tr>
<td>✅</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

In terms of the requirements that this section of the method aims to verify, requirement 5a, c and d can be considered verified. This conclusion is drawn by analysing the results within Table 17. Requirement 5b can be considered verified by drawing a conclusion from the fact that survey respondents chose varying combinations of maintainability assessment methods.

4.3.10 Survey Section 10 — Final Remarks

The final section of the survey to verify the efficacy of a method to improve maintainability in object-oriented software applications presents three questions, created with the intention of obtaining respondents’ final remarks on the method after obtaining a full understanding.

Question one states: The steps involved in the method are formulated in a manner that appears to facilitate the accomplishment of the goal to improve maintainability in software applications. The respondent is presented with a Likert scale rated between strongly disagree to strongly agree. This question was created with the intention of comparing respondents’ thoughts before and after gaining a full understanding of the method.

Seven respondents strongly agreed with the statement presented in question one and four respondents agreed with the statement. Comparing this result with the same question within section 2 of the survey, five respondents agreed, and five respondents strongly agreed. Interestingly, the results obtained for this question after the presentation of the method did not drastically increase.
Question two is an open-ended question asking if respondents have any final thoughts on ways the method could be improved. The only useful response was the suggestion that the method consists of a great deal of steps. The respondent suggested that the method could rather be composed of various paths that developers can take that allow them to tweak the method according to individual needs. This suggestion indicates that the way the method is presented makes it appear as if there is no flexibility in terms of the way that the process can be carried out. This should be considered when presenting the method to developers for the purposes of applying the method to a software application.

The final question of survey section 10 asked developers if they would be willing to make use of the method. Ten developers answered yes, and one developer answered no. The single refusal was interesting as the respondent did not provide a reason for this. The overwhelming agreement to the usage of the method provides further substantiation of the conclusion that the method can be considered verified.

4.4 Survey Summary

Twenty-two developers were approached to complete a survey with the aim of determining the efficacy of the method to improve maintainability in object-oriented software applications. Eleven responses were received. The responses are analysed and the determination of whether each step of the method achieves its goals is performed.

Survey section 1 was created with the aim of determining participant background information which was required to ensure the validity of responses received. The responses received for this section indicated that all responses received could be considered when determining whether the method addresses its requirements.

Survey section 2 presents a high-level overview of the method to improve maintainability. The responses for this section revealed that requirement 1 (stated in section 3.1) can be considered verified. This section also provided a preliminary indication of whether the method achieves each of its sub-requirements (requirements related to each of the steps within the method).

Survey sections 3,4 and 5 were created for verifying requirement 2. This requirement dictates the objective of the identification step of the method to improve maintainability. All responses for questions appearing in the aforementioned sections were either neutral or favourable. This leads to the conclusion that the identification step of the method achieves its objective — identifying areas of low-quality code that become the subject of the next three steps (testing, refactoring and maintainability assessment).

Survey section 6 was created with the objective of determining whether the testing step of the method to improve maintainability achieves its objective of creating an automated test suite that serves as a safety-net, while also improving software maintainability via improving source code testability. All responses received for questions within this section were either neutral or favourable. This led to the conclusion that the testing step of the method achieves its objective and can be considered verified.
Survey sections 7 and 8 were formulated to determine if the refactoring step of the method achieves its objective of improving application maintainability via remedying instances of anti-patterns and grime and performing a Sonarqube assessment. All responses for questions within these sections were favourable or neutral. This led to the conclusion that the refactoring step of the method suitably addresses the related requirement.

Survey section 9 was created for determining if the maintainability assessment step achieves its objective of providing users’ insights on the effect the method had on the maintainability of the software application. All responses received were either neutral or favourable, leading to the conclusion that the maintainability assessment step of the method can be can be considered verified.

In conclusion, the responses received from the survey substantiate the fact that the method to improve maintainability can be considered verified, and it can be applied to a real-world software application without the need for any further improvement.

4.5 Method Implementation

4.5.1 Introduction to Case Study

The final phase of the study begins with the implementation and validation of the method to improve maintainability in object-oriented software applications. Validation is achieved through applying it to a suitable software application.

The application picked to validate the method is a generic data collection application developed for usage within multiple industries. It was developed to alleviate risks associated with manual data collection via the simplification of data ingress, as well as the verification of data entered. It is implemented for the Android operating system. The application provides multiple data ingress interfaces and can communicate with a central server for data synchronisation.

The application is an adequate candidate as the subject of the method for several reasons. Development of the application began in October 2015, with little to no emphasis placed on the internal quality characteristics (maintainability, reliability, performance efficiency) of the software application. Several occasions during the development lifecycle revealed insights leading to the conclusion that application maintainability was lacking.

Further development of the application occurs while the method was applied. This means that when assessing the affect the method has on the maintainability of the software, application changes that are not a result of applying the method should be disregarded or accounted for.

The objective of performing the case study is to determine if the method achieves its requirements, and by association, the objectives related to each requirement have been met.
Subsequent sections will continue by explaining the process of applying the method to improve maintainability in object-oriented software applications. These sections follow the high level-process diagram describing the method available in Figure 23.

<table>
<thead>
<tr>
<th>Identify</th>
<th>Test</th>
<th>Refactor</th>
<th>Assess</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Quality Assessment</td>
<td>Testability</td>
<td>Test</td>
<td>Anti-patterns and grime</td>
</tr>
<tr>
<td>Sonarqube Assessment</td>
<td></td>
<td></td>
<td>Application Quality Assessment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sonarqube Assessment</td>
<td>ISO 25010 maintainability sub-characteristics</td>
</tr>
<tr>
<td>Begin</td>
<td></td>
<td></td>
<td>End</td>
</tr>
</tbody>
</table>

Figure 23: A Method for Improving Maintainability in Object-Oriented Software Applications

4.5.2 Identification of Areas of Low Maintainability

The first step of the method to improve maintainability in object-oriented software applications is the identification of areas of low maintainability. This is accomplished by performing an AQA and SA. Both the AQA and SA result in obtaining a list of classes ordered by maintainability. These lists are compared, and a final determination of the classes that are the subject of the testing and refactoring processes of the method is made. This section explains the identification steps as they are applied to the Android data collection application.

The AQA is performed by making use of instructions specified in Section 3.3.2 with emphasis placed on Figure 7. A MOOD, Martin Packaging and CK and CC metrics assessment is performed to obtain a list of files ordered by maintainability. Appendix B shows the values calculated for each metric assessment. Table 19 shows the final list of files obtained from performing the AQA.
Table 19: Application Quality Assessment Files Ordered by Maintainability

<table>
<thead>
<tr>
<th>Number</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Database Manager (Database)</td>
</tr>
<tr>
<td>2.</td>
<td>Service Manager classes</td>
</tr>
<tr>
<td>3.</td>
<td>Service classes</td>
</tr>
<tr>
<td>4.</td>
<td>Data Input Selector (User Interface)</td>
</tr>
<tr>
<td>5.</td>
<td>Data Input Summary (User Interface)</td>
</tr>
<tr>
<td>6.</td>
<td>Homepage (User Interface)</td>
</tr>
<tr>
<td>7.</td>
<td>Welcome Page (User Interface)</td>
</tr>
<tr>
<td>8.</td>
<td>Management Page (User Interface)</td>
</tr>
<tr>
<td>9.</td>
<td>Synchronisation Page (User Interface)</td>
</tr>
<tr>
<td>10.</td>
<td>Review Page (User Interface)</td>
</tr>
<tr>
<td>11.</td>
<td>Exit Page (User Interface)</td>
</tr>
<tr>
<td>12.</td>
<td>Constants (Database)</td>
</tr>
</tbody>
</table>

The next step specified by the Identification process of the method to improve maintainability is the SA. A SA is performed by making use of the explanation provided in Section 3.3.3. The android application is analysed utilising the Sonarqube Gradle plugin and Sonarqube version 5.6.6. Appendix B shows screenshots of the views used for the determination of a final list of files ordered by maintainability. Table 20 shows the final list of files obtained as result of applying the SA.
Table 20: Sonarqube Assessment Files Ordered by Maintainability

<table>
<thead>
<tr>
<th>Number</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Database Manager (Database)</td>
</tr>
<tr>
<td>2.</td>
<td>Constants (Database)</td>
</tr>
<tr>
<td>3.</td>
<td>Service classes</td>
</tr>
<tr>
<td>4.</td>
<td>Service Manager classes</td>
</tr>
<tr>
<td>5.</td>
<td>Data Input Selector (User Interface)</td>
</tr>
<tr>
<td>6.</td>
<td>Data Input Summary (User Interface)</td>
</tr>
<tr>
<td>7.</td>
<td>Homepage (User Interface)</td>
</tr>
<tr>
<td>8.</td>
<td>Management Page (User Interface)</td>
</tr>
<tr>
<td>9.</td>
<td>Welcome Page (User Interface)</td>
</tr>
<tr>
<td>10.</td>
<td>Synchronisation Page (User Interface)</td>
</tr>
<tr>
<td>11.</td>
<td>Review Page (User Interface)</td>
</tr>
<tr>
<td>12.</td>
<td>Exit Page (User Interface)</td>
</tr>
</tbody>
</table>

The determination of files that should be tested and refactored is completed utilising the explanation within Section 3.3.1. Each file appearing in the lists of Table 19 and Table 20 are compared, and the areas that should be refactored are determined.

The Database Manager class appears at the top of both lists. This file accounts for most of the issues within the application according to the SA, and the analysis of the metrics associated with the class resulted in the file appearing as the least maintainable. These facts alone are not sufficient to decide whether it should be refactored. External factors such as the likelihood of change, amount of effort required to fix all issues, etc., are considered.

The Database Manager class is not considered as a candidate for refactor, since the high line of code count (6800) and the fact that the application database as well as associated functionality, is not susceptible to change for the foreseeable future.

The files within the Service and Service Manager packages consist of functionality that is heavily dependent on matching classes within each package. This, coupled with the fact that they both appear highly within both lists of files ordered by maintainability and their high susceptibility to change, resulted in the conclusion that all the files within both packages should be refactored.

A deeper analysis on the results received from the AQA and SA for files within the User Interface package revealed that the file that accounts for most of the TD is the Data Input Selector class. Other files within the package do not appear to lack maintainability on a scale that compares with the rest of the classes that have been decided as candidates for a refactor and will not be refactored. A further reason for choosing the Data
Input Selector class as a candidate for refactor is because it is the starting point for entering data and references or makes use of a great deal of other classes within the application. This means that its susceptibility to change is high.

The final list of files that are tested and refactored is:

1. Service classes
2. Service Manager classes
3. Data Input Selector

The application of the identification step of the method to improve maintainability in object-oriented software applications as described above provides sufficient information to determine if the identification step achieves its objectives in terms of the requirements outlined by Section 2.1. Analysis and generalisation of these findings is performed in Section 5.4.

4.5.3 Automated Testing of Identified Areas of Low Maintainability

The testing step of the method to improve maintainability is performed for classes within the Service Manager and Service packages as well as the Data Input Selector class. This is achieved utilising instructions provided by Figure 13 and the explanation provided in Section 3.4. The testing step is included within the method to ensure that there is a safety-net preserving existing functionality for the software application, as well as to improve maintainability via improving application testability. This section continues by describing the processes of the method as they are applied to the Android data collection application.

No tests were created for the entire Android application, meaning that the next step followed is to ensure that classes identified within the identification step are testable. The process is facilitated by the explanation of Section 3.4. No issues or noteworthy occurrences were observed during the process of ensuring that any of the classes were testable.

After ensuring testability, the next step completed is the determination of the types of tests required to fully exercise the functionality for each class. This is achieved using the explanation provided by Section 3.4. Behavioural tests are created for the Data Input Selector class, as it is part of the user interface of the Android application. Hybrid tests are required for classes/files within the Service package, as these classes interact with the application user interface and database. No tests were required for classes within the Service Manager package, as they are management classes with a small amount of functionality.

Using the constraints specified by the types of automated tests required, a test framework for implementing these tests was decided upon. The Android development platform provides a test framework with all the tools required for the implementation of behavioural and hybrid tests. The decision on the framework that is used to implement the automated tests is a simple one for this application. This may not be the case for other
applications of the method. This should be considered when determining the types of tests required for future applications that are subjected to the method. This will be discussed further within Section 5.4.

After deciding on the test framework for implementing automated tests, required tests are implemented. These tests are performed throughout the refactoring process of the method. They will fail whenever errors are introduced or when changes are made that do not preserve existing application functionality. This ensures that existing application functionality is preserved.

Following the steps specified by Figure 13, the first decision made is that there is a need to implement automated tests for classes identified within the identification step of the method. Before implementing automated tests, each class was made testable. After ensuring testability, the decision to implement behavioural and hybrid tests was made based on the classes in question. An adequate test framework capable of implementing required tests was determined. The chosen test framework is the native Android test development framework. This framework was used to implement automated tests performed throughout the refactoring step of the method for improving maintainability in object-oriented software applications.

The application of the testing step of the method to improve maintainability as described above provides enough information for the determination of whether the testing step achieves its requirements as stated in Section 3.1. Analysis of these findings is performed in Section 5.4.

4.5.4 Refactoring of Identified Areas of Low Maintainability

The refactoring process of the method for improving maintainability in object-oriented software applications entails following the steps specified by Figure 14 and is facilitated using the explanation provided by Section 3.5.1. This process is the most important step of the method as it is the primary means of improving application maintainability. As specified by the figure mentioned, the anti-pattern, grime and Sonarqube refactoring processes are completed for each file identified within the Android data collection application.

The first step completed for all files is anti-pattern refactoring. This consists of identifying and addressing all occurrences of anti-patterns. This was achieved by following the explanation provided by Section 3.5.1. A single issue occurred while performing this process; some anti-patterns found affect multiple files. This means that the fix for the anti-pattern is not able to be remedied in isolation — as the method implies. This issue did not affect an affirmative answer to the question: Have identified anti-pattern instances been addressed?

Grime refactoring is the next step of the method. This consists of remedying all instances of grime for identified files. This means that instances of code that do not adhere to an overall development methodology are changed to comply. Instances of code that do not adhere to a coding standard are also changed to comply with the standard.
During the process of ensuring that code adheres to a coding standard for the Android application, no issues were encountered. The only noteworthy finding during this process was that the fixes were simple and would be greatly facilitated with some form of tool that allows for specifying coding standards.

No development methodology was used during the development of the Android application. This means that no instances of this type of grime could be remedied, resulting in this aspect of the method to improve maintainability not being validated.

Afterremedying instances of grime and anti-patterns within the Android application, the next step specified by Figure 14 is to perform the Sonarqube refactoring process. This was accomplished by following the steps specified by Figure 16, and using the explanation within Section 3.5.2. The process was performed for each class that was chosen as a candidate for refactoring.

The first step of the Sonarqube refactoring process is re-performing a Sonarqube analysis after completing the anti-pattern and grime refactoring processes. When the analysis is complete, the next step is to identify Sonarqube Issues for each of the classes. The method provides two options for achieving this. Utilising the Sonarqube Issue page is the recommended path, resulting in this path being used primarily when addressing issues within the Android application. For the sake of complete validation of the method, the unrecommended Sonarqube Code View page, at a file level, is also used.

The recommended path for Issue identification is followed for all but five files within the Service Manager package. The issues within five files within the Service Manager package are identified using the unrecommended path. Issues are filtered using the most popular filters found within Section 8 of the survey verifying the method to improve maintainability. Only the top five filters are used: severity; status; file; issue type; and rule. After identifying and filtering issues for each file, the fixes suggested by Sonarqube are implemented.

While utilising the unrecommended path for identifying issues for five files within the Service Manager package, an issue is encountered. There was difficulty answering the question: Have all Issues at appropriate filter levels been addressed? The choice of only subjecting five files to the unrecommended path was made since this path was tested first and its impracticality in answering this question is recognised early into the Sonarqube refactoring process.

Fixes for issues suggested by Sonarqube are only implemented if they satisfy certain constraints. For example, an issue is only addressed if its severity level is blocker, critical or major. Using the Sonarqube Code View page at a file level means that the decision of whether an issue should be remedied must be made for every issue within the file.

A final noteworthy observation encountered during the Sonarqube refactoring process is that all files that are refactored do not contain Bugs. The Sonarqube Assessment (within the identification step) revealed that the
application contains 304 Bugs. This is interesting, as it means that most of the Bugs within the application are in the Database Manager class and were not remedied. This is relevant to validation of the method to improve maintainability since it means that Sonarqubes Bug fixes are not validated.

After completing the anti-pattern, grime and Sonarqube refactoring processes for all identified files, a noteworthy occurrence is identified. All refactorings/fixes for files within the Service Manager and Service packages are similar within their respective package. This results in the prediction that results obtained for each file will be like other files within the same package.

The final step of the refactoring step of the method is answering the question: Are you satisfied that a sufficient increase in maintainability will be achieved? This question is answered using the final questions within each refactoring type. These questions are:

- Anti-pattern refactoring: Have identified anti-pattern instances been addressed?
- Grime refactoring: Have identified grime instances been addressed?
- Sonarqube refactoring: Have all Issues at appropriate filter levels been addressed?

The answers to these questions for each file is yes. This means that the answer to the final question posed by Figure 14 is also yes. The final question is included within the refactoring step of the method to ensure there is some form of reflection on work done and the results the work will achieve.

To summarise, the steps outlined by Figure 14 are followed for the completion of the refactoring step of the method. Anti-pattern and grime refactoring are completed for all files chosen as candidates for testing and refactoring. After completing these refactorings, a Sonarqube refactoring is performed following the steps outlined by Figure 16. The first step of this refactoring type is re-performing a Sonarqube analysis. After the analysis, two paths are followed for identifying Sonarqube issues. Identified issues satisfying predetermined filter specifications are remedied using Sonarqubes recommendations. The completion of the Sonarqube refactoring process is followed by the answer to a final question meant to encourage reflection on work done. After determining that the answer to this question is positive, the maintainability assessment step of the method can be performed to determine the effect the method has on application maintainability.

The information presented regarding the application of the refactoring step of the method to improve maintainability provides sufficient information to determine whether this step addresses the requirements outlined in Section 3.1. Analysis of this information is provided within Section 5.4.

4.5.5 Maintainability Assessment

All three options provided by the method for improving maintainability within object-oriented software applications are performed for the Android data collection application. This is done to ensure that the method is fully validated. The explanations and figures provided within Section 3.6 are used when performing each assessment.
This section presents the findings obtained when applying the AQA, SA and ISO 25010 assessments to the android data collection application. Appendix C provides explanations and figures supporting how the conclusions presented have been reached.

The AQA is used as a means of maintainability assessment by following the instructions given in Section 3.6.2. The instructions state that the AQA should be performed in reverse and the results obtained before and after testing and refactoring should be compared to determine what effect the method has on application maintainability. Appendix B shows values calculated for the CK, CC, Martin package and MOOD metrics for files that have been refactored after the testing, and refactoring steps of the method to improve maintainability have been performed. Metric values before and after method application are compared and it is determined that no further work is required.

An issue that challenges the effectiveness of the AQA as a means of assessing maintainability is the fact that determining if whether further work is required is very subjective. This means that the conclusion reached can change depending on developer preferences. More details on the way that the above conclusions have been reached can be found in Appendix C.

Despite the issues that arose while using the AQA as a maintainability assessment, the conclusion that no further work is required in terms of improving application maintainability is reached.

The SA is used as a means of maintainability assessment by following the instructions specified by Figure 8 and the explanation provided by Section 3.6.3. The metrics shown by Sonarqube at different software application levels can be compared before and after the testing and refactoring processes and the determination of whether further work is required can be made. An explanation of the way the conclusions reached regarding the maintainability of the software application is provided within Appendix C.

The analysis of the effect of the method on the maintainability of the Android data collection application utilising the SA revealed that no further work is required.

The final method provided for assessing maintainability is the usage of the ISO 25010 assessment. The ISO 25010 Assessment consists of following the steps outlined within Section 3.6.4. It is like the AQA in that it makes use of code metrics, with the exception that insights are gained on maintainability via maintainability sub-characteristics defined by ISO 25010. The first step — as specified by Figure 21, is to determine what metrics are used to assess the effects the method has on maintainability for each sub-characteristic. This assessment type is suited to users of the method that require insights on application maintainability, in terms of maintainability sub-characteristics defined by ISO 25010.
Table 21: Metrics Used for Each Maintainability Sub-characteristic

<table>
<thead>
<tr>
<th>ISO 25010 Sub-characteristic</th>
<th>Metrics Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modularity</td>
<td>Coupling Between Objects; Instability; Coupling Factor; Lack of Cohesion of Methods; Efferent Coupling; Lines of Code; Cyclomatic Complexity; Number of Children; Afferent Coupling;</td>
</tr>
<tr>
<td>Reusability</td>
<td>Lines of Code; Cyclomatic Complexity; Depth of Inheritance Tree; Efferent Coupling;</td>
</tr>
<tr>
<td>Analysability</td>
<td>Lines of Code; Cyclomatic Complexity; Depth of Inheritance Tree; Efferent Coupling;</td>
</tr>
<tr>
<td>Modifiability</td>
<td>Lines of Code; Cyclomatic Complexity; Response For Class; Depth of Inheritance Tree;</td>
</tr>
<tr>
<td>Testability</td>
<td>Lines of Code; Cyclomatic Complexity; Response For Class; Coupling Between Objects; Depth of Inheritance Tree;</td>
</tr>
</tbody>
</table>

Table 21 shows the metrics used to determine the effect application of the method has on each ISO 25010 maintainability sub-characteristic. These metrics are chosen using Table 7 and Table 8, as well as the explanations for what each of the metrics mean within Section 2.2.2. The metrics chosen for assessing each maintainability sub-characteristic are applicable for usage at different levels of the software application.

The results of the metrics before and after the method to improve maintainability that were applied for the ISO 25010 assessment revealed that no further work was required.

The three options provided by the method to improve maintainability are applied to the Android data collection application and all three conclude with the finding that no further work is required on the Android data collection application.

4.5.6 Summary
The method to improve maintainability in object-oriented software applications was successfully applied to an android data collection application. Insights were gained on the effectiveness of each of the steps within the method. Along with these insights, knowledge is gained describing ways that the method can be improved, as well as a plethora of concerns that should be accounted for when applying the method to further software applications.

4.6 Method Assessment
The purpose of the method assessment is determining the effect the method to improve maintainability has on the maintainability of the mobile application from a neutral stand-point. This is achieved using a neutral
method of assessing the maintainability of software applications. The chosen method of assessment is the Maintainability Index (MI).

The MI is used to assess if the overall objective of the method to improve maintainability in object-oriented software applications (improving maintainability) is reached. Firstly, the MI is calculated for the Android data collection application before the method is applied to the application. The values for the average Halstead Volume, Cyclomatic Complexity and Lines of Code are for each application package/folder were calculated and a final value for the MI was found using (13). Table 22 shows the calculated values and the resultant MI. It is found that the MI of the Android data collection application was 61.11% before the method to improve maintainability was applied.

Table 23 shows the MI after the method to improve maintainability in object-oriented software applications was applied to the Android data collection application. The final value for the MI was calculated as 71.02%. This shows a 9.92% increase to the maintainability of the Android data collection application.
Table 22: Maintainability Index Before Method Application

<table>
<thead>
<tr>
<th>Package/Folder</th>
<th>$V_{Avg}$</th>
<th>$LOC_{Avg}$</th>
<th>$C_{Avg}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package/Folder 1</td>
<td>178.67</td>
<td>2608.83</td>
<td>20.00</td>
</tr>
<tr>
<td>Package/Folder 2</td>
<td>167.00</td>
<td>1566.65</td>
<td>18.12</td>
</tr>
<tr>
<td>Package/Folder 3</td>
<td>39.88</td>
<td>164.00</td>
<td>7.41</td>
</tr>
<tr>
<td>Package/Folder 4</td>
<td>103.60</td>
<td>927.87</td>
<td>6.60</td>
</tr>
<tr>
<td>Package/Folder 5</td>
<td>41.11</td>
<td>479.67</td>
<td>5.78</td>
</tr>
<tr>
<td>Package/Folder 6</td>
<td>154.45</td>
<td>1729.82</td>
<td>16.82</td>
</tr>
<tr>
<td>Package/Folder 7</td>
<td>101.69</td>
<td>705.77</td>
<td>20.31</td>
</tr>
<tr>
<td>Package/Folder 8</td>
<td>78.00</td>
<td>591.33</td>
<td>15.33</td>
</tr>
<tr>
<td>Package/Folder 9</td>
<td>126.22</td>
<td>1082.11</td>
<td>23.00</td>
</tr>
<tr>
<td>Package/Folder 10</td>
<td>177.17</td>
<td>1383.50</td>
<td>35.83</td>
</tr>
<tr>
<td>Package/Folder 11</td>
<td>9619.00</td>
<td>162816.00</td>
<td>1227.00</td>
</tr>
<tr>
<td>Package/Folder 12</td>
<td>179.43</td>
<td>2117.00</td>
<td>19.14</td>
</tr>
<tr>
<td>Package/Folder 13</td>
<td>256.13</td>
<td>3141.53</td>
<td>28.40</td>
</tr>
<tr>
<td>Package/Folder 14</td>
<td>178.67</td>
<td>1023.33</td>
<td>19.33</td>
</tr>
<tr>
<td>Package/Folder 15</td>
<td>735.50</td>
<td>9672.50</td>
<td>86.25</td>
</tr>
<tr>
<td>Package/Folder 16</td>
<td>82.89</td>
<td>658.89</td>
<td>14.89</td>
</tr>
<tr>
<td>Package/Folder 17</td>
<td>84.23</td>
<td>544.77</td>
<td>16.31</td>
</tr>
<tr>
<td>Package/Folder 18</td>
<td>63.00</td>
<td>739.63</td>
<td>7.25</td>
</tr>
<tr>
<td>Package/Folder 19</td>
<td>330.00</td>
<td>4640.25</td>
<td>36.75</td>
</tr>
<tr>
<td>Package/Folder 20</td>
<td>128.07</td>
<td>1082.40</td>
<td>13.87</td>
</tr>
<tr>
<td>Package/Folder 21</td>
<td>479.13</td>
<td>6843.75</td>
<td>51.75</td>
</tr>
<tr>
<td>Total</td>
<td>13303.84</td>
<td>204519.60</td>
<td>1690.14</td>
</tr>
<tr>
<td>Average</td>
<td>633.52</td>
<td>9739.03</td>
<td>80.48</td>
</tr>
<tr>
<td>Maintainability Index</td>
<td></td>
<td></td>
<td>61.11 %</td>
</tr>
</tbody>
</table>
Since development on the application occurred that was not part of the application of the method to improve maintainability, the difference of 9.92% cannot be considered as accurate. For the sake of determining the effect the method has on maintainability, the MI is calculated for packages/folders that were directly changed because of applying the method to improve maintainability. Figure 24 shows the MI for the packages/folders that were affected by the method. MI was calculated using values within Tables 22 and 23. The Service package/folder experienced an improvement of 3.13% to maintainability; the Service Manager package/folder experienced a deterioration of 0.23% to maintainability; and the User Interface package/folder experienced an
improvement to maintainability of 1.29%. The MI values calculated indicate that the changes introduced by
the method do not make up a bulk of the improvement to maintainability obtained when calculating the MI at
an application level. This does not, however, invalidate the conclusion that the method does result in an
improvement to maintainability.

The decrease in maintainability of the Service Manager package/folder can be explained by noting the changes
that were made when performing the refactoring step of the method. The Service Manager and Service
packages/folders are highly related. To improve the maintainability of the Service package/folder (which has
a lower maintainability rating), the decision was made to decrease the maintainability of the Service Manager
package/folder. This was a valid decision as the improvement of 3.23% to the MI of the Service package/folder
outweighs the 0.23% decrease experienced by the Service package/folder.

![Maintainability Index Before and After Method Application](image)

**Figure 24: Maintainability Index Before and After Method Application**

More granular insight on the effect the method has on the maintainability of the Android data collection
application can be obtained by analysing each of the terms within the MI. The MI makes use of three metrics
that represent the source code control structure, information structure and code typography, naming and
commenting aspects of software maintainability. The cyclomatic complexity metric represents the software
control structure. Figure 25 shows the change in cyclomatic complexity for the packages/folders affected by
the method to improve maintainability. Cyclomatic complexity improved for the packages/folders where the
MI improved (cyclomatic complexity decreased). This means that the method has a direct effect on the
software control structure. The methods picked for improving maintainability in this instance of applying the method, reduced software complexity.

![Change in Cyclomatic Complexity](image_url)

**Figure 25: Change in Cyclomatic Complexity**

Figure 26 shows the change in lines of code for the packages/folders affected by the method to improve maintainability. As with the cyclomatic complexity, packages/folders that experience an improvement to maintainability also experience an improvement to the number of lines of code (a decrease in the number of lines of code). This means that the code typography of the Android data collection application is affected by the method.

Figure 27 shows the change in Halstead volume for packages/folders that are affected by the method to improve maintainability. Halstead volume represents the information structure of the software application. The information structure of the android data collection application improved for all packages/folders where the MI improved.
The MI of the Android data collection application improved from 61.12% to 71.02%. This improvement was not only because of applying the method as external development independent of the method occurred while the method was applied. This was addressed by calculating the MI for packages/folders that were directly affected by the method. It was found that the MI improved for the Service and User Interface packages/folders.
while the MI decreased for the Service Manager package/folder. Further insight was gained on the effect the method has on the maintainability of the Android data collection application by analysing the change that resulted for each of the code metrics used within the formula for calculating the MI of a software application.

4.7 Result Discussion

Each high-level step within the method to improve maintainability is assessed using results received from the verification and validation of the method. Using the results for each step, a conclusion is reached regarding whether requirements related to each have been met. Improvements to each step are determined using results from method verification and validation. This section continues by analysing each of the high-level steps of the method for improving maintainability and concludes by using the conclusions reached for each step to determine if the problem statement within Section 1.6 has been solved.

4.7.1 Identification

The identification step of the method to improve maintainability in object-oriented software applications is assessed using the results and insights gained from the verification and validation of the method. Survey results pertaining to this step of the method are favourable. Responses for questions related to this step of the method all either agreed or strongly agreed that it would allow developers to achieve the goal of obtaining a list of files that will be the subject of the testing and refactoring steps of the method.

Results obtained from applying the identification process to an Android data collection application are all favourable. The AQA and SA results in obtaining a list of files ordered by maintainability without any issues or noteworthy occurrences. Finally, a list of files is obtained that will be the subject of the testing and refactoring steps of the method using the instructions provided without issue.

The favourable results received for the verification and validation of the identification step of the method to improve maintainability result in the conclusion that requirement 2 (defined in Section 3.1) has been met. This means that there is no need to improve or rectify any of the sub-steps within the identification step of the method to improve maintainability in object-oriented software applications.

4.7.2 Testing

The testing step of the method is assessed using results and insights gained from the verification and validation of the method. Survey results received from both the overview and testing sections of the survey are favourable. All respondents agreed or strongly agreed that the testing step of the method will achieve its goal of the creation of an automated test suite.

The validation of the testing step of the method reveals that the process allows developers to achieve the goal of creating an automated test suite that will serve as a safety-net while performing the refactoring step. A single issue is encountered when deciding upon a suitable test framework capable of implementing decided upon test
types. This decision is simple for Android software applications. This may not be the case for future applications of the method and should be considered.

The results obtained from both the verification and validation of the testing step of the method to improve maintainability are favourable. This leads to the conclusion that requirement 3 (defined in Section 3.1) has been met. The testing step needs only to be improved by accounting for the concern pertaining to the decision on the test framework used to implement required test types.

4.7.3 Refactoring
The refactoring step of the method to improve maintainability is assessed using the results obtained from the verification and validation of the method. Results obtained from the method overview section of the survey are favourable. Respondents agreed or strongly agreed that the anti-pattern, grime and code smells (Sonarqube) steps of the refactoring process will result in an improvement to application maintainability. Results obtained from the refactoring and Sonarqube sections of the survey reaffirmed that the refactoring step of the method to improve maintainability will result in the improvement of application maintainability.

Results received from applying the method to an Android software application brought attention to many issues within the refactoring step of the method to improve maintainability. The explanation of the anti-pattern refactoring step does not consider anti-patterns that span multiple files. The explanation should be updated to allow for this possibility.

No instances of grime that do not adhere to an overall development methodology are remedied within the validation of the method because the Android data collection application was not developed using a methodology. This can be remedied by applying the method to a software application that is developed using a methodology.

Another issue is discovered within the grime refactoring process that deals with addressing grime instances that are caused by source code that does not comply with a coding standard. The process of remedying these grime instances will be greatly faciliated through the usage of an automated tool for identifying and remedying them.

Two issues arose when performing the Sonarqube refactoring process. The unrecommended path for identifying Sonarqube issues results in having to determine if all issues within a class should be remedied. This path should be removed from the method since the recommended path provides a much better alternative. Finally, no Sonarqube issues that are considered bugs are remedied during the validation of the method. This type of issue and its remedy should be validated in a future application of the method.

The issues encountered during the validation of the refactoring step of the method did not hinder the accomplishment of the goal to rectify all instances of grime, anti-patterns and Sonarqube issues within the Android data collection application.
Results obtained from the verification and validation of the refactoring step of the method to improve maintainability are favourable. It is concluded that requirement 3 (defined in Section 3.1) has been met. The issues found within the validation process should be rectified to improve the processes involved in the refactoring step of the method to improve maintainability.

4.7.4 Maintainability Assessment

The maintainability assessment step of the method to improve maintainability is assessed utilising results obtained from verification and validation of the method. Survey results pertaining to the maintainability assessment step of the method are all favourable. All respondents agreed or strongly agreed that the maintainability assessment allows users of the method to determine if further work is required.

The process of completing the AQA, SA and ISO 25010 assessments for the Android data collection application revealed no issues with completing the steps outlined for each. The results obtained from the verification and validation of the method to improve maintainability led to the conclusion that requirement 5 (defined in Section 3.1) has been met.

4.7.5 Method Assessment

The method to improve maintainability in object-oriented software applications was assessed using the maintainability index. The maintainability index was evaluated at an application level before and after the method was applied to an Android data collection application. The evaluation revealed that the method improved maintainability by 9.92%. External development occurred while the method was applied to the application. This meant that a more granular assessment of the application was required.

This granular assessment revealed that two of the three packages/folders experienced an improvement to maintainability. The granular assessment also reveals that all three aspects that make up source code maintainability are affected by the method. The fact that all three aspects are affected by the method reveals that the method has the ability to improve maintainability for all aspects required to successfully improve maintainability.

4.8 Conclusion

The method to improve maintainability in object-oriented software applications was verified through the usage of a survey. The responses were analysed, and it was concluded that each step of the method achieves its requirements, without the need for further work to be done.

After verifying that the method to improve maintainability achieves its goals, the method was applied to an Android data collection application. Each step of the method was applied and findings for each are noted. The method was applied successfully. This conclusion was reached via determining of the requirements and objectives (by association) are achieved by each step of the method.
After the method was applied to the Android data collection application, the overall method objective was assessed utilising the maintainability index. The maintainability index improved by 9.92%. A more granular look at the change in MI revealed that the method influences the three aspects that affect the maintainability of a software applications source code — the source code control structure; the source code information structure; and the source code typography, naming and commenting.

Finally, the results obtained from the verification and validation of the method are analysed to determine whether the requirements and objectives (by association) of the method are achieved. It is concluded that the method addresses all its requirements, and some improvements can be made.
CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

This chapter provides a conclusion to the study. A summary of work completed, and findings of previous chapters, is presented. Finally, recommendations for additional work that should be completed concerning the method for improving maintainability in object-oriented software applications is discussed.
5 Conclusion and Recommendations

5.1 Study Summary

The study begins by introducing the need to develop a method for improving maintainability in object-oriented software applications. This need is established by explaining the problems that occur when developing software in RDE’s. Problems that arise because of developing software in RDE’s can be classified as TD, which causes a lack of software application quality. A high amount of research has been performed on the concept of removing TD from software applications. This high amount of research results in a plethora of different ways for removing TD.

Maintenance activities on a software application account for most work completed during the development of an application. TD that causes a lack of maintainability is a major issue. Architectural, design, code and test debt are the focus of this study. Finally, a method for improving maintainability in software applications developed using any programming paradigm is not feasible. The method only focuses on applications that are developed using the object-oriented programming paradigm, as it is the most used.

Findings from the discussion above are used to develop a problem statement describing the need to develop a method for improving maintainability in object-oriented software applications. The following literature objectives are achieved in Chapter 2:

- Investigate techniques for identifying areas of low maintainability in software applications.
- Investigate techniques for improving maintainability in software applications.
- Investigate techniques for assessing maintainability in software applications.
- Formulate refined requirements for the method to improve maintainability in object-oriented software applications.

Findings from the literature study are used to formulate refined requirements for a method to improve maintainability in object-oriented software applications. These requirements govern what the method should achieve, as well as the steps it should follow. The refined requirements (found in section 3.1) are considered an extension of the following objectives:

- The method should target object-oriented software applications.
- The method should focus on TD that affects the maintainability of a software application.
- The method should encompass or make use of existing techniques for addressing TD (improving maintainability).
- The method should have three high level steps — identification, rectification and assessment.

The method consists of four high-level steps: identification; testing; refactoring; and maintainability assessment. A full explanation of the method is available in chapter 3.
The goal of the identification step is to obtain a list of classes that are candidates to be subjected to the testing and refactoring steps of the method to improve maintainability. This list is obtained by comparing two lists of classes obtained from completing the AQA and SA. The AQA makes use of code metrics to gain insight into application maintainability at a class, package and application level. The SA makes use of functionality provided by the Sonarqube web application to gain insight into application maintainability at levels matching the AQA.

The goal of the testing step of the method is to develop an automated test suite that is used during the refactoring step. The automated test suite acts as a safety-net preserving existing application functionality while changes are completed. An additional effect introduced by the testing step is the improvement of maintainability by improving application testability.

The refactoring step of the method to improve maintainability is the most important step of the method, as its goal is the removal of instances of anti-patterns, grime and code smells. These are indicators of low maintainability that, if remedied, drastically improve the maintainability of software applications. The steps involved in achieving the remedy of these fixes is outlined by the anti-pattern, grime and Sonarqube refactoring processes.

The final step of the method is to perform a maintainability assessment. The goal of this step is to determine the effect the method has on the maintainability of the software application. The method provides three options for performing a maintainability assessment: the AQA can be performed in reverse, and metrics before and after can be analysed for the determination of the effect on maintainability; the SA can be re-performed and the information available on pages provided by Sonarqube can be analysed for the determination of the effect on application maintainability; an ISO 25010 assessment can be performed for the determination of application maintainability via maintainability sub-characteristics defined by ISO 25010.

After developing the method to improve maintainability, the following objectives are addressed:

- The method should be verified by means of determining if the requirements have been met.
- Results obtained from the verification of the method to improve maintainability should be analysed to determine whether it is feasible to validate the method.
- The method should be validated by applying it to an existing software application and performing all the specified steps.
- Results obtained from both method verification and validation should then be analysed to determine if the study is a success as well as determining recommendations for future research.

The method is verified using a survey of software developers with at the minimum, a degree related to software development and varying ranges of development experience. The survey verifies the method against the requirements of the method. Eleven survey responses are received that provide positive results in terms of the
verification of the method. This leads to the conclusion that the method can be applied to a real-world application for validation purposes. Details on the verification process appear in section 4.1, 4.2, 4.3 and 4.4.

The method is validated by applying it to an Android data collection application. All steps involved in the method are explained and shortcomings are noted. If these shortcomings are rectified, the method to improve maintainability will be improved. Implementation details are found in section 4.5.

The determination of whether the method to improve maintainability accomplishes its overall objective is performed utilising the maintainability index. The maintainability index is calculated for the Android software application before and after the method is applied to the android data collection application. Overall application maintainability is found to have improved. The method assessment appears in section 4.6.

An assessment of the method is performed using the results obtained from its verification and validation. The assessment found that the method solves the problem stated in Section 1.6, but can be improved using the recommendations found from the experiences encountered during the application of the method to an Android data collection application.

5.2 Recommendations for Future Work

There are several possibilities for future work regarding a method to improve maintainability in object-oriented software applications.

First and foremost, the recommendations for method improvement discussed within Section 5.4 should be implemented. This will result in a greater improvement to application maintainability and improve the usability of the method. These recommendations are: the refactoring step of the method should be modified to account for anti-patterns that span multiple files; and the Sonarqube refactoring step of the refactoring process of the method should be modified to not include the unrecommended path for identifying issues (see section 3.3.3)

The method should be applied to object-oriented software applications that are developed using architectures other than Android. The same process outlined in section 4.5 can be followed. All shortcomings and ways that the method can be improved should be noted and added to the instructions provide in chapter 3.

A study can be conducted with the aim of gathering feedback from software developers with varying degrees of experience, and with exposure to different software development architectures. The method can be provided to a suitable sample of developers that apply the method to an object-oriented software application. Feedback can then be obtained regarding the way the method is presented, the ease of understanding the instructions provided for each step of the method, the efficacy with which developers are able to achieve the goals dictated by each step of the method, etc.

The method can be applied to a software application that is within the maintenance phase. Time spent on maintenance related activities can be compared before and after applying the method. This will reveal how
effective the method is at reducing time spent on maintenance and provide an easier means of demonstrating the usefulness of the method to management stakeholders.

The method should be integrated for usage within a RDE at regular intervals. A study should be conducted on how this process can be performed, and instructions for doing so should be added to the method for improving maintainability in object-oriented software applications.

A study can be conducted on ways the method should be modified to be applicable for usage with software applications that do not make use of the object-oriented programming paradigm. The high-level steps (identification, testing, refactoring and maintainability assessment) would remain the same.

Finally, the method can be improved as research on the maintainability of software applications matures. There may be newer strategies that are developed for each of the steps involved in the method, and the method can be modified to use these methods instead of, or in conjunction with, the methods currently specified.
REFERENCE LIST


[66] R. Perez-Castillo and M. Piattini, “Analyzing the harmful effect of god class refactoring on power


APPENDIX A

A SURVEY VERIFYING A METHOD FOR IMPROVING MAINTAINABILITY IN OBJECT-ORIENTED SOFTWARE APPLICATIONS
1. Introduction

This appendix shows an example of a survey used to verify a method for improving maintainability in object-oriented software applications.

A Survey Regarding the Efficacy of a Method to Improve Maintainability in Object Oriented Software Applications

This survey aims to verify the efficacy of a method to improve maintainability in software applications.

The method to improve maintainability in software applications is aimed at developers of long running object oriented software applications. The method relays information regarding all the steps a developer must follow in order to transform a software application from one that lacks maintainability to one that is as maintainable as the developer chooses.

The high level steps of the method to improve maintainability are: Identification of areas of code that lack maintainability; refactoring or restructuring of areas that lack maintainability; and finally assessing the maintainability of the application in order to determine if maintainability goals have been met.

The method is an amalgamation of verified tools and techniques. This means the primary priority of the survey is to obtain information regarding the processes and steps postulated rather than information about the techniques and tools used.

The survey begins by postulating questions regarding developer experience in the field of software development as well as software maintainability.

Section 2 consists of an explanation of the high-level steps involved in the method and questions regarding the perceived efficacy of the method from a high level stand point.

Section 3 begins with an explanation of the high level steps involved in the Identification process and ends with questions postulated with the goal of gaining insight into the efficacy of the Identification process.

Section 4 starts with an explanation of the steps involved in the Application Quality Assessment step of the Identification process and finishes with questions regarding the efficacy of the assessment.

Section 5 explains the steps involved in the SonarQube Assessment step of the Identification process and presents questions regarding the perceived effectiveness of the assessment.

Section 6 consists of an explanation of the steps involved in the Testing process of the method to improve maintainability in software applications and concludes with questions on the effectiveness of the Testing process.

Section 7 presents the Refactoring process of the method to improve maintainability in software applications.

Section 8 demonstrates the SonarQube Refactoring process.

Section 9 introduces the process for the verification of an increase in maintainability via a maintainability assessment and presents questions regarding the effectiveness of the process.

Section 10 postulates questions regarding the method given the fact that the survey respondent has knowledge on the method to improve maintainability in software applications as a whole.

*Required
   1. Email address *
2. **Software Development Experience** *
   *Mark only one oval.*
   - Less than 6 months
   - Between 6 months and 2 years
   - Greater than 2 years

3. **What is the programming paradigm you have been exposed to the most?** *
   *Mark only one oval.*
   - Object-Oriented
   - Aspect Oriented
   - Feature Driven
   - Functional
   - Other: ____________________________

4. **What does software maintainability mean to you?** *
   __________________________________________
   __________________________________________
   __________________________________________
   __________________________________________
   __________________________________________

5. **How important is software maintainability to you?** *
   *Mark only one oval.*
6. How much time and effort is spent on the maintenance of software applications? *

Mark only one oval.

☐ Less than 10%
☐ 25%
☐ 50%
☐ 75%
☐ More than 75%

Start this form over.

Method Overview
The goal of this section is to present an overview of the method and obtain feedback regarding the high level steps involved in the method to improve maintainability in software applications.

A Method to Improve Maintainability in Object-Oriented Software Applications
A Method for Improving Maintainability in Object-Oriented Software Applications

<table>
<thead>
<tr>
<th>Identify</th>
<th>Test</th>
<th>Refactor</th>
<th>Assess</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Quality Assessment</td>
<td>Testability</td>
<td>Anti-patterns and prime</td>
<td>Sonarqube recommendations</td>
</tr>
<tr>
<td>Sonarqube Assessment</td>
<td>Fast</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>End</td>
<td></td>
<td>ISO 25010 maintainability sub-characteristics</td>
</tr>
</tbody>
</table>

Method Summary

The method to improve maintainability in software applications consists of four high level steps: Identification; Testing; Refactoring and Assessment.

Identification

The goal of the identification step of the method to improve maintainability is to allow developers to gauge what areas of a software application lack maintainability. The step consists of two sub-steps (An application quality assessment and a Sonarqube Assessment) that can be performed in any order.

The Application Quality Assessment (AQA) consists of making use of a purpose designed code metric suite that provides developers with information on the maintainability of the software application under study at an application, package/folder and file/class level. The result of performing an AQA is a list of files sorted by maintainability.

The Sonarqube Assessment (SA) requires developers to make use of an application called Sonarqube. Sonarqube is a web based open source quality management platform. It provides information on application code quality that includes the existence of bugs, vulnerabilities, code smells, duplications, source code size and languages used. The ultimate goal of the Sonarqube Assessment is to obtain a list of classes or files appear to lack maintainability.

The lists obtained from the AQA and SA are then compared and developers can make a decision on what areas should be focused on.

Testing

The testing step consists of two sub-steps that ensure that the refactoring process does not affect application functionality while also facilitating the process of improving maintainability. These sub-steps are: ensuring that identified areas of code are testable; and creating unit, integration or behavioural tests for identified areas of code.

Improving the testability of source code also results in an increase in maintainability while also allowing for the creation of automated tests.

Refactoring

As a noun refactoring means: A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour. As a verb, the act of refactoring is: to restructure software by applying a series of refactorings without changing its observable behaviour. A refactoring is a known code pattern or snippet that is used to remove a code smell.

Code smells are known patterns of code occurrences that result in low quality or problematic code. Developer experience has resulted in the realisation that certain patterns of code result in decreased application quality and may provide developers with issues in future development. The presence of
these patterns within applications means that there could be difficulty in the maintenance of the software application.

In terms of the method to improve maintainability refactoring is a verb that describes the process of improving maintainability. The actual refactoring process of refactoring the method to improve maintainability consists of two steps: manually identifying and addressing any instances of grime or anti-patterns found in the file that is to be refactored; and performing recommendations suggested by Sonarqube.

Grime develops within a software application with time. It is the accumulation of additional code within an initial software design. This additional code could be in the form of bug fixes or additional functionality that is written without taking into account the overall design of the application.

Anti-patterns are code smells at an architecture level rather than a single block of code. Anti-patterns always indicate the presence of problems and should be dealt with as soon as they are identified. More information on the different anti-patterns that may be found at www.sourcemaking.com/antipatterns.

Recommendations by Sonarqube come in the form of the removal of code smells, code duplications and any bugs or vulnerabilities in a software application.

Assessment

The maintainability assessment is performed in order to ensure that developers reach goals set for application maintainability increases.

The maintainability assessment is performed after refactoring the identified instances of areas of code with low maintainability. This step is optional and can be performed in varying degrees of detail. The sub-steps involved are: re-performing an application quality assessment outlined in the identification process; re-performing a Sonarqube analysis of application source code and ensuring that refactored files do not contain instances of code smells, bugs or vulnerabilities; and assessing application quality using ISO 25010 maintainability sub-characteristics.

7. The explanation of the high level steps involved in the method to improve maintainability is sufficient.*

Mark only one oval.
8. The usage of code metrics as a means of identifying areas of low maintainability in software applications is useful. *
   *Mark only one oval.*

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly Disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strongly Agree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9. The usage of Sonarqube as a means of identifying areas of low maintainability in software applications is useful. *
   *Mark only one oval.*

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly Disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strongly Agree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10. Ensuring that source code is testable before explicitly refactoring allows developers to gain a better understanding of source code as well as improve the maintainability of source code. *
    *Mark only one oval.*

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly Disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strongly Agree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11. The creation of automated tests will facilitate the process of making changes to application source code while preserving functionality. *
    *Mark only one oval.*
12. The removal of instances of grime will improve maintainability in identified areas of low maintainability. *
   *Mark only one oval.*

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strongly Disagree</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

13. The removal of any anti-pattern instances will improve maintainability in identified areas of low maintainability. *
   *Mark only one oval.*

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strongly Disagree</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

14. The removal of code smells, code duplications and any bugs or vulnerabilities will improve maintainability in identified areas of low maintainability. *
   *Mark only one oval.*

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strongly Disagree</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

15. The assessment of maintainability after performing the identification and refactoring steps will ensure that maintainability goals are met. *
   *Mark only one oval.*

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strongly Disagree</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

16. The steps involved in the method are formulated in a manner that appears to facilitate the accomplishment of the goal to improve maintainability in software applications. *
   *Mark only one oval.*

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strongly Disagree</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Identification
The goal of this section is to present more detail on the steps involved in the Identification process of the method to improve maintainability of software applications and gain insight into its efficacy.

In order to improve software maintainability, there is a need to know where problem areas lie within the software application. This means that we need to identify areas of low maintainability and determine whether or not these identified areas should be refactored or not.

Identification Process

Identification Process explanation

The Identification process of the method to improve maintainability can be broken down into three steps. The figure above shows the steps involved in this process. The first two steps can be performed in any order. The first of these steps is to assess the quality of the application. This is accomplished through the use of code metrics; details on the Application Quality Assessment will be provided later.

The next step is to analyse the application using Sonarqube. Sonarqube is a web based application that provides insight into the quality of a software application; details on the Sonarqube Assessment will be provided later. Both steps mentioned result in developers obtaining a list of files or classes that appear to be the least maintainable (files or classes that appear to have the lowest quality).

After obtaining the lists of files or classes from the Application Quality Assessment and the Sonarqube Assessment, these lists should be compared in order to determine which files or classes should be refactored. When making the decision on which classes should be refactored, developers should consider the following:
• For a particular file, where does it lie in each ordered list? If a file is high on both lists, then it should be considered a high priority for refactoring.
• For a particular file, what is the likelihood that changes will be made in the future? If there is a high chance of changes being made, then the file should be considered a high priority for refactoring since this the amount of time to make any changes to the file will be decreased as a result of the refactor.
• Are there external management aspects that may affect the refactoring process? These management factors depend on the type of environment the application is developed in.

When the classes or files that should be refactored have been determined, developers can move on to the testing phase of the method to improve maintainability.

18. The explanation of the high level steps involved in the Identification Process is sufficient. *
   
   Mark only one oval.

   1 2 3 4 5

   Strongly Disagree Strongly Agree

19. The usage of two methods for identifying areas of low maintainability will provide a sufficient insight into where major areas of low quality lie. *

   Mark only one oval.

   1 2 3 4 5

   Strongly Disagree Strongly Agree

20. Using the two lists of files ordered by maintainability as well as the explanation provided, developers will be able to make informed decisions about where focus should be placed in terms of the improvement of maintainability. *

   Mark only one oval.

   1 2 3 4 5

   Strongly Disagree Strongly Agree

21. Using the two lists of files ordered by maintainability as well as the explanation provided, developers will be able to make informed decisions regarding the formulation of maintainability goals. *

   Mark only one oval.

   1 2 3 4 5

   Strongly Disagree Strongly Agree

22. The Identification Process will accomplish its goal of determining what files or classes should be refactored. *

   Mark only one oval.

   1 2 3 4 5

   Strongly Disagree Strongly Agree
23. Do you have any recommendations for ways in which the Identification Process can be improved upon? (Please elaborate) *

Application Quality Assessment

The goal of this section is to present more detail on the steps involved in the Application Quality Assessment process of the method to improve maintainability of software applications and gain insight into its efficacy.

The first step involved in the identification process is to assess the quality of the application source code. This is accomplished through the usage of code metrics. Code metrics provide quantitative data that allows developers to assess the health of an application. There are different categories of metrics — each giving an insight towards different aspects of application health. The category of metrics that are used to identify a lack of code quality is code quality metrics. Quality metrics provide varying degrees of depth in terms of where code lacks quality (Application/project, package/folder, class/file and method). These metrics also are only applicable for Object-Oriented programming languages. The goal of the Application Quality Assessment is to obtain a list of files sorted by maintainability.
MOOD Metrics

The Metrics for Object-Oriented Design (MOOD) metrics suite provides developers with information regarding application quality from a project/module level. Each metric is quantified using a percentage that provides developers with an insight towards the overall quality of the application. Formulase for each of the metrics used within the suite can be found at [http://www.alvocito.com/project/tejp.pm-opmood.html](http://www.alvocito.com/project/tejp.pm-opmood.html)

The first metric used is called the Attribute Hiding Factor (AHF). AHF shows the degree of attribute or field encapsulation in a project. This is done by calculating the ratio of hidden attributes to total attributes in the application. The AHF of an application should be as high as possible. The ideal would be a hundred percent as the attributes of a class should only be known to the class itself.

The second metric used is called the Attribute Inheritance Factor (AIF). This metric provides knowledge on the degree of attribute or field inheritance in a project. It shows the ratio of what percentage of the available fields in a class are due to inheritance. AIF should be within a reasonable limit. If it is too high, this would indicate that the member scope of the project is too high. This goes against known good practices for software development.

Method Hiding Factor (MHF) is the degree of method encapsulation in the project. It shows the ratio of how many classes an average method or function is visible from other than the class that defines it. A low MHF indicates that the use of abstraction is insufficient. If there is a lack of abstraction within a software application, there is a high probability that there may be errors within the application. A high
MHF indicates that the functionality of the application is lacking.

Method Inheritance Factor (MIF) shows the ratio of what percentage of the methods within a class are made available through inheritance rather than declared in the class itself. MIF behaves similarly to AIF, with the exception that its scope is on a method level.

Coupling factor (CF) provides an insight towards the coupling of an application as a whole. Coupling is the degree to which an application module relies upon the other modules within the application. If a module is highly coupled its maintainability is low. This is because it would be difficult to make changes to the module. A class is coupled to another if it calls another class’ methods or accesses any of its attributes.

The Polymorphism Factor (PF) is the degree of polymorphism on the entire application. This is the probability that a method or function will be overridden. PF gives an indication on the complexity of the application. An overly complex application lacks maintainability since it indicates that the application lacks analysability and modifiability.

**Martin Packaging Metrics**

The Martin packaging metrics suite provides information regarding the stability and coupling within each application package or folder.

The first metric used is package Abstractness. Abstractness is the ratio of abstract classes and interfaces in a package. Values for the abstractness metric range between zero and one. A value of zero indicates that a package is concrete (all classes have been implemented) and a value of one indicates that all classes are abstract. Abstractness should be used in conjunction with Instability since if a package is unstable and concrete this would indicate that the package is not open to change (it will be difficult to perform changes to package functionality).

The next two metrics used within the Martin packaging metrics deal with coupling. Coupling metrics give an indication as to the likelihood of a defect in a single class causing a rippling effect within the system. Efferent coupling is the number of classes that a given class depends upon. Afferent coupling is the number of classes that depend on a given class. In the context of the Martin packaging metrics, the values calculated for afferent and efferent coupling relay total amount of the different coupling types in each package.

Instability is a measure of a packages resilience to change. The metric is the ratio of efferent couplings to the total amount (efferent and afferent) couplings in the package.

The final metric used by the Martin packaging metrics is the Distance from the main sequence. This metric is sometimes referred to as “the perpendicular distance between the idealised line Abstractness + Instability = 1”. “Good packages” would appear on the line that is between the two points where Instability = 0 and Abstractness = 1 and vice versa.

Thus, the distance from the main sequence can be calculated using: \[\text{Distance from the Main sequence} = \text{Abstractness} + \text{Instability} - 1\].

The ideal value for the distance to the main sequence is zero since the higher the abstractness of a package, the more stable it should be since there will be many client classes/packages that depend upon its interfaces. The distance calculated will be between one and zero and indicates the distance the package is from appearing on the line that constitutes a good package.

**Chidamber-Kemerer Metrics**
Chidamber-Kemerer Metrics

The CK metrics suite provides insight on an applications quality at a class/file level.

The first metric used is Coupling Between Objects (CBO). CBO is the total number of afferent couplings and efferent couplings for the class. If a class is highly coupled its maintainability is low.

The next metric used is the Depth of Inheritance Tree (DIT). DIT is the amount of inheritance steps between the class and the basic object class of the OO language (example: java.lang.Object). If class DIT is high, this may indicate that there is an overuse of inheritance.

Lack of Cohesion of Methods (LCOM) is a measure of the interrelation between the methods of a class. Two methods of a class are related if one method calls another or they share a variable usage. LCOM is the number of relations between all methods within the class. High values for LCOM indicate that a class may have too many responsibilities and should potentially be separated into classes with distinct responsibilities. A value of one — the lowest value possible, indicates that a class is highly cohesive and cannot be easily separated.

The Number Of Children (NOC) of a class is the number of classes that directly inherit from the class. Each child class will be susceptible to the TD of the parent class.

The Response For Class (RFC) is the number of possible methods that can be called when a class is sent a method send. This is the sum of the number of constructors and methods in the class plus the number of methods and constructors the class can call directly. A high RFC indicates that a class may be too complex and may need to be restructured.

The final metric in the CK suite is Weighted Method Count (WMC). WMC is defined as the number of methods that are declared in a class. If a class has a high amount of methods, this may indicate that it may have too many responsibilities.

Cyclomatic Complexity

A further metric that is used within a method to improve maintainability in software applications is Cyclomatic Complexity (CC). CC is defined as the number of linearly independent paths through a programs source code and is also referred to as the McCabe metric. The McCabe metric can be found at any level of application source code (package/project, package, class and method). If source code is too complex, this may indicate the presence of TD as the package/project/class/method may be responsible for too much and may contain code that does not follow known best practices. A high CC may also indicate the presence of grime. More information on the CC metric and its usage can be found at https://www.guru99.com/cyclomatic-complexity.html

24. The explanation of the Application Quality Assessment is sufficient. *

*Mark only one oval.

1 2 3 4 5

Strongly Disagree □ □ □ □ □ Strongly Agree

25. The MOOD metrics will allow developers to make an informed decision regarding whether or not an application lacks maintainability. *

*Mark only one oval.
26. The usage of Martin Package metrics in combination with the CC metric will allow developers to make an informed decision regarding whether or not a package lacks maintainability.

*Mark only one oval.*
27. The usage of CK metrics in combination with the CC metric will allow developers to make an informed decision regarding whether or not a file lacks maintainability. *
Mark only one oval.

28. The sequence of steps proposed within the Application Quality Assessment will result in developers accomplishing the goal of obtaining a list of files sorted by maintainability. *
Mark only one oval.

29. Do you have any recommendations for ways in which the Application Quality Assessment can be improved upon? (Please elaborate) *

________________________________________
________________________________________
________________________________________
________________________________________

Sonarqube Assessment
The goal of this section is to present more detail on the steps involved in the Sonarqube Assessment process of the method to improve maintainability of software applications.

This Sonarqube Assessment step of the identification step of the method to improve maintainability in software applications involves assessing the application source code using Sonarqube. Sonarqube is a web based open source quality management platform. It provides information on application code quality that includes the existence of bugs, vulnerabilities, code smells, duplications, source code size and languages used. The ultimate goal of the Sonarqube Assessment is to obtain a list of classes or files appear to lack maintainability.

Sonarqube has a plethora of pages or views that provides developers with different insights into application quality. For the purposes of the Sonarqube assessment, the views and features that are required to find a list of classes/files that appear to lack maintainability are discussed. For a full specification of all the features of Sonarqube, see the documentation which can be found at https://docs.sonarqube.org/.

The steps relayed by the figure shown below should be considered as a guideline for one possible method that developers can follow in order to obtain the final list of files/classes — developers can use any combination of Sonarqubes features as long as a list of files sorted by maintainability is obtained.

Sonarqube Assessment
Homepage usage

The first step of the Sonarqube assessment is for developers to view the Sonarqube Homepage. The details provided by this page allow developers to answer the question: Do details shown on the Sonarqube Homepage indicate that the application lacks maintainability?

The Sonarqube Homepage provides a high level view of the application quality. It shows the total amount of bugs, vulnerabilities, code smells, duplicated blocks and lines of code.

A bug is an error or fault in an application that may cause the application to behave differently than expected.

A vulnerability is a block or line of code that decreases the security of the application. Software security is defined as the idea that an application will continue to function when malicious attacks occur.

Duplicated blocks are the number of repeated blocks of code within the application. They are a type of anti-pattern that is known to cause many errors when it comes to maintaining the application.

The number of lines of code is the number of executable lines of code within the application.

Bugs, vulnerabilities and code smells are referred to as issues in Sonarqube and are rated by five severity levels. These levels indicate the risk to productivity and the risk the issues pose to the application. The five levels of severity that are used are:

- Blocker — the issue is a security or operational risk that could make the entire application unstable.
- Critical — the issue is a security or operational risk that may lead to unexpected behaviour without affecting the whole application.
- Major — the issue may have a high impact on developer productivity.
- Minor — the issue may have minimal impact on developer productivity.
- Info — the issue impact cannot be classified because not enough information is known.

Bugs affect the reliability of an application. The “E” rating above the number of bugs (982) shown on the Homepage is the Sonarqube reliability rating for the application. The reliability rating is calculated using the following scale:

- A — zero bugs
- B — at least one minor bug
- C — at least one major bug
- D — at least one critical bug
- E — at least one blocker bug

Sonarqube assesses application and file security using a rating that is based on the amount of vulnerabilities in an application. The “E” rating above the number of vulnerabilities (35) shown on the Sonarqube Homepage view. The following scale is used to rate the security of the application:

- A — zero vulnerabilities
- B — at least one minor vulnerability
• C — at least one major vulnerability
• D — at least one critical vulnerability
• E — at least one blocker vulnerability

After determining that the software application may lack maintainability, developers access the Sonarqube Code View page (by clicking “Code” tab on the Homepage).

**Sonarqube Code view page (package/folder level)**

**Code view page (package/folder level) usage**

This page shows a list of all the packages/ folders in the application with the same details shown on the Homepage at a package level rather than a project/application level. The information relayed by the Sonarqube Code view page at a package level allows developers to answer the question: Do details shown on Sonarqube Code view page indicate that a package lacks maintainability?

After determining that there are packages/folders that contain TD, developers should click the name of each package/folder. This action reloads the Sonarqube Code view page with a list of all the files within the clicked package/folder.

**Sonarqube Code view page (file/class level)**

**Code view page (file/class level) usage**

xix
This page shows information regarding the amount of issues within each file. Using this information, developers can answer the question: Does file or class appear to lack maintainability? This question should be answered for each file within all packages that appear to lack maintainability. Files that are deemed to lack maintainability should be added to a list of files that lack maintainability. This list should be sorted by files that appear to have the most problem areas — this decision is left to the discretion of the developer performing the Sonarqube assessment. An example of a sorting mechanism would be to sum the amount of issues (bugs, vulnerabilities and code smells) then sort by number of issues, then duplication percentage and finally lines of code.

30. The explanation of the Sonarqube Assessment is sufficient. *
   
   Mark only one oval.

   1  2  3  4  5

   Strongly Disagree  ○  ○  ○  ○  ○  Strongly Agree

31. The usage of the Sonarqube Homepage view will allow developers to make an informed decision regarding whether or not an application lacks maintainability. *
   
   Mark only one oval.

   1  2  3  4  5

   Strongly Disagree  ○  ○  ○  ○  ○  Strongly Agree

32. The usage of the Sonarqube Code View page at a folder/package level will allow developers to make an informed decision regarding whether or not a folder/package lacks maintainability. *
   
   Mark only one oval.

   1  2  3  4  5

   Strongly Disagree  ○  ○  ○  ○  ○  Strongly Agree

33. The usage of the Sonarqube Code View page at a file/class level will allow developers to make an informed decision regarding whether or not a file/class lacks maintainability. *
   
   Mark only one oval.

   1  2  3  4  5

   Strongly Disagree  ○  ○  ○  ○  ○  Strongly Agree

34. The sequence of steps proposed within the Sonarqube Assessment will result in developers accomplishing the goal of obtaining a list of files sorted by maintainability. *
   
   Mark only one oval.

   1  2  3  4  5

   Strongly Disagree  ○  ○  ○  ○  ○  Strongly Agree
Testing
The goal of this section is to present more detail on the steps involved in the Testing process of the method to improve maintainability of software applications and gain insight into its efficacy.

Before beginning the refactor of areas identified via the identification process, it is necessary to create, integration or behavioural tests on the identified classes — if they have not been created already. Automated tests provide developers with a safety net that prevents the addition of regression errors as well as aids in the understanding of the area of code that is to be refactored. The testing process can be repeated for each identified area of low quality code, then developers can move on to the refactoring step or developers can repeat the testing and refactoring steps in sequence for each identified area of code.

The Testing Process
Testing Process explanation

The above diagram describes the steps involved in the testing phase of the method to improve maintainability is software applications. The first question that developers should ask is: Has an adequate test suite been created? This question can be answered by answering the following sub-questions:

- What type of test framework is used?
- What types of tests have been created?
- Does the created test suite adequately exercise identified area of low quality code in such a manner that the developer can with relative confidence state that the likelihood of regression errors being introduced without detection during the refactoring process is minimal?

If developers conclude that automated tests are required before beginning with the refactoring process, developers should ensure that the file or class in question is testable. Developers can determine whether or not a class is testable by ensuring that the answer to the following questions is yes:

- Does the file or class adhere to the SOLID principles (More information can be found on the SOLID principles be found at https://medium.com/@cramiroz92/a-o-i-d-the-first-5-principles-of-object-oriented-design-with-javascript-790f6ac9b9fa)?
- Are developers certain that there will be minimal difficulty in creating tests for the file/class in question?

After ensuring that file/class is testable, developers should determine the type or types of tests that should be created. The possible types of tests that are could be created are unit, integration or behavioural tests. In order to answer the question: Have required test types(s) been determined? The determination of the types of tests that should be created should be considered by answering the following questions:

- Does file or class interact with the user interface or external applications?
  - If it does, then behavioural tests should be created.
- Does the file or class interact with a database, the external file system or any entities that cannot
be isolated from the class/file that is currently in question?
  o If it does, then integration tests are required.
• Can the file or class be isolated from all entities that it interacts with?
  o If it can be, then unit tests should be created.

There are five commonly used testing framework types: module based; data driven; keyword driven; hybrid; and behaviour driven. Module based testing frameworks make use of the concept of abstraction. The application under test is viewed as being consisted of individual modules that tests are individually created for. This is the most common form of testing framework and is available for most programming languages.

Data driven testing frameworks were developed in order to address the need that arose when the application under test is required to perform similar functionality on different data sets. Data driven testing frameworks assist developers with the storage, management and integration of the data with tests.

Key word driven testing frameworks are an extension of data driven testing frameworks. They provide additional functionality in the handling of data.

Hybrid testing frameworks consist of different combinations of the functionality of module based, data driven or key word driven testing frameworks. They are developed for special cases of application types and typically are not widely available.

The final type of commonly used testing framework is behaviour driven testing frameworks. This type of framework was invented for the specific purpose of verifying application functional requirements. This means that this type of framework typically interacts with an applications user interface or other applications that the application under test interacts with.

Whence the required test framework type has been determined, developers can move on to actually creating their chosen test type(s). After creating tests, developers should ensure that the following statement is true before moving on to the refactoring step of the method to improve maintainability in software applications:
• Developers are confident that any changes made to the class or file in question will not introduce any regression errors that cannot be detected.

36. The explanation of the Testing Process is sufficient. *
Mark only one oval.

1 2 3 4 5
Strongly Disagree    Strongly Agree

37. Ensuring that source code is testable increases maintainability, aids developers understanding, as well as facilitates the creation of automated tests. *
Mark only one oval.

1 2 3 4 5
Strongly Disagree    Strongly Agree

38. The creation of automated unit, integration or behavioural tests increases maintainability and provides developers with a safety net that prevents the addition of regression errors when making changes to application source code. *
Mark only one oval.
39. The steps involved in the Testing Process facilitate the creation of automated tests. *  

Mark only one oval.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Strongly Disagree   Strongly Agree
40. Do you have any recommendations for ways in which the Testing Process can be improved upon? (Please elaborate) *

The Refactoring Process
The goal of this section is to present more detail on the steps involved in the Refactoring Process of the method to improve maintainability in software applications as well as assess its efficacy.

The Refactoring Process
41. The explanation of the Refactoring Process is sufficient. *

Mark only one oval.

---

**The Refactoring Process explanation**

The first step that developers should complete is anti-pattern refactoring. This step consists of developers manually analysing a file or class and determining whether or not there are any anti-pattern instances. After determining the presence of any anti-patterns, developers should refactor the code in order to remove them. Possible anti-patterns and their solutions can be found at [https://sourcemaking.com/antipatterns](https://sourcemaking.com/antipatterns).

Developers can move on to grime refactoring by ensuring that the answer to the following question is yes: Have identified anti-pattern instances been remedied?

The grime refactoring steps consists of developers analysing a file or class and determining if the overall design of the class adheres to the software applications design methodology. This step varies on a case-by-case basis. Developers should ensure that they recognise the following considerations when identifying instances of grime:

- Is there a design methodology used for the software application?
- Is there a coding standard and does it specify any rules that relate to the software design methodology?
- Are there bug fixes that appear to be temporary solutions to software behaviour mishaps?

After addressing any instances of grime and anti-patterns, developers should move on to the Sonarqube refactoring process. Details of the steps involved in the Sonarqube refactoring process will be discussed in the next section of the survey.

Whence the Sonarqube refactoring process is complete, developers should ask themselves one final question before moving on to the verification step of the method to improve maintainability in software applications: Will a sufficient increase in application maintainability be achieved?

The steps involved in the refactoring process are formulated in a manner such that if the steps are followed correctly, a satisfactory increase in maintainability should always be achieved. The postulated question is included in order to ensure that developers always take their quality goals into consideration when completing any of the refactoring processes.
42. The removal of anti-patterns within identified areas of low maintainability improves the maintainability of software applications. *

Mark only one oval.

1 2 3 4 5

Strongly Disagree  ○ ○ ○ ○ ○  Strongly Agree

43. The removal of grime within identified areas of low maintainability improves the maintainability of software applications. *

Mark only one oval.

1 2 3 4 5

Strongly Disagree  ○ ○ ○ ○ ○  Strongly Agree

44. The steps involved in the Refactoring Process allows developers to improve maintainability in software applications. *

Mark only one oval.

1 2 3 4 5

Strongly Disagree  ○ ○ ○ ○ ○  Strongly Agree

45. Do you have any recommendations for ways in which the Refactoring Process can be improved upon? (Please elaborate) *

________________________________________________________

________________________________________________________

________________________________________________________

Sonarqube Refactoring
The goal of this section is to present more detail on the steps involved in the Sonarqube refactoring step of the Refactoring Process of the method to improve maintainability in software applications as well as assess its efficacy.

Sonarqube Refactoring Process
Sonarqube Code View Page (Single file)
Sonarqube Refactoring Process explanation (Part 1)

The first step that developers should complete is to re-run a Sonarqube analysis on the application after the changes specified by the anti-pattern and grime refactoring processes have been performed. After re-running the Sonarqube analysis, there are two paths that can be followed in order to address issues highlighted by Sonarqube.

The first path indicates that developers can view the Sonarqube Code View page for a single file and view all file issues that way. The Sonarqube Code View page for a single file allows developers to view the contents of the file, with information on the severity and the location of issues. In addition to this, Sonarqube also shows users duplicated code — this duplicated code may appear in other files as well. Users can scroll up and down the page to view all issues for the selected file.

This method for viewing file or class issues may overload the developer with too many issues and does not allow developers to filter issues; developers should rather click the “Issues” link (the number above “Issues”). This will lead to the Sonarqube Issues page.

Sonarqube Issues Page

Sonarqube Refactoring Process explanation (Part 2)

The second path that developers can follow in order to view file or class issues is to make use of the Sonarqube Issues page. This page allows users to view all project issues. Issues can be filtered by: type; resolution; severity; status; new issues (day, week, month, year); rule; tag; project; module;
directory (package or folder); file; assignee (the person assigned to the issue; author (author of the section of code); and language. Using available filters developers can narrow down issues by concerns that are deemed high priority.

In terms of the decision process for determining issue priority, developers should use the following definitions:

- **Issue Type**: These can be bugs, vulnerabilities or code smells. Developers should use the definitions for each of the issue types in order to make the decision on which type is deemed higher on the priority list of issues to address.
- **Resolution**: An issue resolution status can be: unresolved; false positive; removed; fixed; or won’t fix. This filter is useful for issue management within Sonarqube and allows developers to categorise issues that they do not deem valuable.
- **Severity**: The options for issue severity level were discussed in section 3.2.3. Developers should always ensure that issues at a blocker and critical severity level are addressed. Any issues at a lower severity level can be addressed if it is deemed feasible by developers.
- **Status**: The issue status can be: open; reopened; confirmed; resolved; or closed. The status can be used for issue management in order to allow developers to communicate and categorise issues that should be addressed based on whether or not they are deemed feasible for refactor.
- **New Issues**: This filter allows developers to filter issues by creation/detection date. If new issues are deemed to be of higher importance than old or vice versa; developers can use this knowledge as a distinguishing factor in the decision of where an issue lies on the priority list.
- **Rule**: The options for this filter are the rules that dictate whether or not a pattern of code is constituted as an issue. These options are sorted by the amount of times they are violated. Developers can set a priority for each of the rules that are in place for the application and use them as a distinguishing factor in the decision of where an issue lies on the priority list.
- **Tag**: Issue tags categorise issue rules. These tags provide users with more information regarding a rule and can be used to aid in developers decisions about the priority for a rule.
- **Project**: More than one project can be scanned for a Sonarqube analysis. This filter allows developers to narrow issues down to the project being refactored.
- **Module**: A module is a folder within a project that has been scanned by Sonarqube. This filter allows developers to narrow issues down to folders within the project.
- **File**: This filter allows developers to narrow issues down to files within the project. This filter is important in the context of the method to address TD in software applications as it should ideally be set to the file/class that is currently being refactored.

After isolating issues that should be remedied for a file or class, developers are required to address each issue.

**Sonarqube Example Issue**

**Sonarqube Example Issue Explanations Table**
<table>
<thead>
<tr>
<th>Detail</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Describes the issue details</td>
<td>Reduce this &quot;switch/case&quot; number of lines from 38 to at most 10, for example by extracting code into function.</td>
</tr>
<tr>
<td>Type</td>
<td>Bug/Vulnerability/Code Smell</td>
<td>Code Smell</td>
</tr>
<tr>
<td>Severity</td>
<td>Blocker/Critical/Major/Minor/Info</td>
<td>Major</td>
</tr>
<tr>
<td>Status</td>
<td>Open/Reopened/Confirmed/Resolved/Closed</td>
<td>Open</td>
</tr>
<tr>
<td>Assignee</td>
<td>Sonarqube user name</td>
<td>Not assigned</td>
</tr>
<tr>
<td>Estimated Effort</td>
<td>Estimated time to rectify issue</td>
<td>5min effort</td>
</tr>
<tr>
<td>Time Created</td>
<td>The amount of time since the issue was created</td>
<td>A month ago</td>
</tr>
<tr>
<td>Line Number</td>
<td>The line number that the issue begins at</td>
<td>L145</td>
</tr>
<tr>
<td>Tag</td>
<td>Identifies the type of issue</td>
<td>Brain-overload</td>
</tr>
<tr>
<td></td>
<td>Expands all issues in file revealing file contents and location of the clicked issue</td>
<td></td>
</tr>
</tbody>
</table>

**Sonarqube example in-depth issues pop-up**
Sonarqube Refactoring Process explanation (Part 3)

The first image displayed is a Sonarqube Example Issue. The Sonarqube Example Issue Explanations Table describes what each of the elements shown on a Sonarqube Example Issue represents. The information described for an issue provides developers with all the information required in order to address the issue. In-depth issue details can be accessed by clicking the in-depth issues button (the ellipsis). This will reveal the Sonarqube example in-depth issues pop-up. This popup provides an explanation of why the issue may be a problem and an example of code that does not comply with the rule as well as a compliant example. Using the code example developers can address the issue.

Using the information described above, developers can address each issue within the file or class that being refactored. Developers can consider the Sonarqube refactoring process to be complete after ensuring that the answer to the following question is yes: Have all issues at appropriate filter levels been remedied?

46. The explanation of the Sonarqube Refactoring Process is sufficient. *

Mark only one oval.

1  2  3  4  5

Strongly Disagree 〇 〇 〇 〇 〇  Strongly Agree

47. Using the Sonarqube Code View Page for a single file allows developers to address issues in an efficient manner. *

Mark only one oval.
48. Using the Issues page with filters set at appropriate levels allows developers to address issues in an efficient manner. *

*Mark only one oval.

49. Rate issue filter option usefulness: *

*Tick all that apply.

<table>
<thead>
<tr>
<th>Issue Type</th>
<th>Useless</th>
<th>Somewhat useless</th>
<th>Neutral</th>
<th>Somewhat useful</th>
<th>Useful</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resolution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Severity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New Issues</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

50. Addressing Sonarqube issues improves application maintainability. *

*Mark only one oval.

51. Do you have any recommendations for ways in which the Sonarqube Refactoring Process can be improved upon? (Please elaborate) *

______________________________________________
______________________________________________
______________________________________________

Maintainability Assessment
The maintainability assessment is performed in order to ensure that developers reach goals set for application maintainability increases.

This step of the method to improve maintainability in software applications is an optional one that can be performed in a range of detail in terms of the completeness. Developers can rerun a Sonarqube analysis on the code after completing the refactoring step; they can perform an application quality assessment; they can assess the quality of newly refactored code against the maintainability sub-characteristics defined by ISO 25010; or any combination of the three assessment possibilities.

Application Quality Assessment Verification Process
**Application Quality Assessment Verification Process**

**Explanation**

An application quality assessment can be used as a means of verification of an increase in maintainability with a few minor modifications. Succinctly, developers can perform an application quality assessment in reverse for areas of code that have been refactored.

The image above shows a flow diagram representing the general process that should be followed when performing an application quality assessment in reverse. The steps shown in the diagram should be performed at a file/class, package level and application level. This means developers will have an insight into the change in maintainability at different levels of resolution resulting in the determination of whether or not maintainability goals have been met.

**Sonarqube Assessment Verification Explanation**

If developers choose to make use of a Sonarqube assessment in order to verify an increase in maintainability, the following questions should be answered by making use of several Sonarqube views:

- Using the Sonarqube Homepage view:
  - Has the refactoring process introduced any new issues?
  - Has the number of issues decreased?
  - Have the maintainability, reliability and security ratings changed?

- Using the Sonarqube Code View page for each file that has been refactored or the Sonarqube Issues page:
  - Have all issues at the pre-determined filter level been addressed?
  - Have any new issues been introduced?
  - Has the number of issues decreased?

Using the answers to the questions above, developers can determine whether or not maintainability has increased. If the answers to the questions above are generally favourable, then there was an increase in maintainability.
ISO 25010 sub-characteristic Assessment

Start

Determine what metrics are going to be used to assess the health of each maintainability sub-characteristic defined by ISO 25010

Calculate values for each decided upon metric

Complete Testing and Refactoring steps of method

Calculate chosen metric values

For each ISO 25010 sub-characteristic

For each chosen metric

Has metric value improved when comparing values before and after refactor?

Yes

Characteristic Improved

No

Characteristic unchanged or deteriorated

ISO 25010 maintainability sub-characteristic health

Results for each ISO 25010 maintainability sub-characteristic

Has maintainability increased?
altered, to find
insufficiencies or causes of failure or to assess the impact of a change to an entity within the
application.
  • Modifiability: The level to which an application can be altered without introducing adverse effects or decreasing the
    quality of an application.
  • Testability: The degree of effectiveness with which an application can be tested.

The final method for the verification of an increase in maintainability consists of making use of the
sub-characteristics defined for maintainability by ISO 25010 as well as code metrics. Developers
make use of code metrics as a means of assessing the health of each maintainability sub-
characteristic. The ISO Maintainability characteristics and Applicable Metrics table and Code Metrics
Definitions table show the code metrics that are applicable for usage for each maintainability sub-
characteristic.

Developers should determine whether or not the ISO 25010 sub-characteristic assessment is to be
used as a means of verification before beginning with the refactoring process. This is because
developers should choose metrics that are going to be used for assessing the health of each sub-
characteristic and calculate their values before modifying application source code. These values are
used as reference when assessing the health of each maintainability sub-characteristic.

After performing the refactoring step of the method to improve maintainability, developers should
assess the health of each maintainability sub-characteristic. This is achieved by calculating the values
of each chosen metric (applicable for the sub-characteristic under assessment) and answering the
question: Has metric value improved when comparing values before and after refactoring? The answer to
this question for each metric will reveal whether or not the characteristic under assessment has
improved or deteriorated.

Using the results for each maintainability sub-characteristic developers can determine whether or not
software application maintainability increased as well as understand whether or not maintainability
goals have been met.

52. The explanation of the Maintainability Assessment is sufficient. *

  Mark only one oval.

  1  2  3  4  5

  Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

53. Re-performing an Application Quality Assessment allows developers to determine if
maintainability goals are met. *

  Mark only one oval.

  1  2  3  4  5

  Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

54. Making use of Sonarqube’s various views allows developers to determine if maintainability
goals have been met. *

  Mark only one oval.

  1  2  3  4  5

  Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

xxxvi
55. Performing a maintainability assessment using a combination of ISO 25010 maintainability sub-characteristics and code metrics allows developers to determine if maintainability goals have been met. *
   Mark only one oval.

   1  2  3  4  5

   Strongly Disagree  ○  ○  ○  ○  ○  Strongly Agree

56. Using a combination of the assessments mentioned will allow developers to determine if maintainability goals have been met. *
   Mark only one oval.

   1  2  3  4  5

   Strongly Disagree  ○  ○  ○  ○  ○  Strongly Agree

57. If you agree to the above question, what combination of methods would be best suited for the determination of whether or not maintainability goals have been met?
   Tick all that apply.
Use Assessment

- Application Quality Assessment
- Sonarqube Assessment
- ISO 25010 Assessment

58. **Do you have any recommendations for ways in which process for the verification of an increase in maintainability can be improved upon? (Please elaborate)** *

---

**Final Remarks**

59. **The steps involved in the method are formulated in a manner that appears to facilitate the accomplishment of the goal to improve maintainability in software applications.** *

*Mark only one oval.*

1  2  3  4  5

| Strongly Disagree | | | | | Strongly Agree |

60. **Do you have any recommendations for ways in which the method to improve maintainability can be improved (please elaborate)?** *
61. I would be willing to make use of the method in order to improve maintainability in a personal software application. 
Mark only one oval.

☐ Yes
☐ No
☐ Other: __________________________

You’ve made it to the end.

I'd like to thank you for taking the time to complete the survey.
APPENDIX B

SUPPLEMENTARY VALUES FOR A METHOD FOR IMPROVING MAINTAINABILITY IN OBJECT-ORIENTED SOFTWARE APPLICATIONS
1. Introduction

This document serves as supplementary material to A Method for Improving Maintainability in Object-Oriented Software Applications.

2. AQA - MOOD Metrics and Martin Packaging Metric Values

Table 1: MOOD Metrics Assessment Values

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute Hiding Factor (AHF)</td>
<td>64.67%</td>
</tr>
<tr>
<td>Attribute Inheritance Factor (AIF)</td>
<td>82.72%</td>
</tr>
<tr>
<td>Coupling Factor (CF)</td>
<td>6.03%</td>
</tr>
<tr>
<td>Method Hiding Factor (MHF)</td>
<td>33.39%</td>
</tr>
<tr>
<td>Method Inheritance Factor (MIF)</td>
<td>11.91%</td>
</tr>
<tr>
<td>Polymorphism Factor (PF)</td>
<td>196.42%</td>
</tr>
</tbody>
</table>

Table 2: Martin Package and Cyclomatic Complexity Metrics

<table>
<thead>
<tr>
<th>Package</th>
<th>Abstractness</th>
<th>Afferent</th>
<th>Coupling</th>
<th>Efferent</th>
<th>Coupling</th>
<th>Distance from Main Sequence</th>
<th>Instability</th>
<th>Average Complexity</th>
<th>Total Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>0</td>
<td>21</td>
<td>13</td>
<td>0.62</td>
<td>0.38</td>
<td>3.11</td>
<td>1413</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Service Manager</td>
<td>0.07</td>
<td>44</td>
<td>48</td>
<td>0.41</td>
<td>0.53</td>
<td>2.3</td>
<td>115</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Service</td>
<td>0.17</td>
<td>23</td>
<td>101</td>
<td>0.02</td>
<td>0.83</td>
<td>3.62</td>
<td>333</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User Interface</td>
<td>0</td>
<td>14</td>
<td>69</td>
<td>0.17</td>
<td>0.83</td>
<td>4.85</td>
<td>417</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. AQA - Chidamber-Kemerer and Cyclomatic Complexity Metric Values (Identification)

The following tables list the calculated results for the Chidamber-Kemerer and Cyclomatic Complexity metrics for the packages within an android software application for the identification step of a method to improve maintainability.
<table>
<thead>
<tr>
<th>Class/ File</th>
<th>Coupling Between Objects (CBO)</th>
<th>Depth Of Inheritance Tree (DIT)</th>
<th>Lack of Cohesion Of Methods (LCOM)</th>
<th>Number Of Children (NOC)</th>
<th>Response For Class (RFC)</th>
<th>Weighted Method Complexity (WMC)</th>
<th>Average Cyclomatic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Manager File 1</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>23</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Service Manager File 2</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>25</td>
<td>7</td>
<td>3.5</td>
</tr>
<tr>
<td>Service Manager File 3</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>41</td>
<td>6</td>
<td>1.5</td>
</tr>
<tr>
<td>Service Manager File 4</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>43</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>Service Manager File 5</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>38</td>
<td>10</td>
<td>3.33</td>
</tr>
<tr>
<td>Service Manager File 6</td>
<td>10</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>27</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Service Manager File 7</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>19</td>
<td>5</td>
<td>1.67</td>
</tr>
<tr>
<td>Service Manager File 8</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>38</td>
<td>10</td>
<td>3.33</td>
</tr>
<tr>
<td>Service Manager File 9</td>
<td>23</td>
<td>4</td>
<td>4</td>
<td>13</td>
<td>9</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Service Manager File 10</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Service Manager File 11</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>16</td>
<td>3</td>
<td>1.5</td>
</tr>
<tr>
<td>Service Manager File 12</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>25</td>
<td>5</td>
<td>1.67</td>
</tr>
<tr>
<td>Service Manager File 13</td>
<td>11</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>43</td>
<td>7</td>
<td>1.75</td>
</tr>
<tr>
<td>Service Manager File 14</td>
<td>14</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>46</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Service Manager File 15</td>
<td>13</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>43</td>
<td>7</td>
<td>1.75</td>
</tr>
</tbody>
</table>
### Table 4: Service Package Chidamber-Kemerer and Cyclomatic Complexity Metric Results

<table>
<thead>
<tr>
<th>Class/ File</th>
<th>Coupling Between Objects (CBO)</th>
<th>Depth Of Inheritance Tree (DIT)</th>
<th>Lack of Cohesion Of Methods (LCOM)</th>
<th>Number Of Children (NOC)</th>
<th>Response For Class (RFC)</th>
<th>Weighted Method Complexity (WMC)</th>
<th>Average Cyclomatic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service File 1</td>
<td>37</td>
<td>2</td>
<td>2</td>
<td>13</td>
<td>40</td>
<td>27</td>
<td>1.8</td>
</tr>
<tr>
<td>Service File 2</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3.86</td>
</tr>
<tr>
<td>Service File 3</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Service File 4</td>
<td>39</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>34</td>
<td>23</td>
<td>3.83</td>
</tr>
<tr>
<td>Service File 5</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>9</td>
<td>6.17</td>
</tr>
<tr>
<td>Service File 6</td>
<td>25</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>75</td>
<td>37</td>
<td>3</td>
</tr>
<tr>
<td>Service File 7</td>
<td>13</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>54</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>Service File 8</td>
<td>13</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>54</td>
<td>21</td>
<td>2.67</td>
</tr>
<tr>
<td>Service File 9</td>
<td>16</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>55</td>
<td>16</td>
<td>5.5</td>
</tr>
<tr>
<td>Service File 10</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>15</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>Service File 11</td>
<td>12</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>27</td>
<td>10</td>
<td>2.2</td>
</tr>
<tr>
<td>Service File 12</td>
<td>12</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>42</td>
<td>11</td>
<td>7.83</td>
</tr>
<tr>
<td>Service File 13</td>
<td>28</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>88</td>
<td>47</td>
<td>1.8</td>
</tr>
<tr>
<td>Service File 14</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>22</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>Service File 15</td>
<td>12</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>25</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>Service File 16</td>
<td>14</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>45</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>Service File 17</td>
<td>15</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>35</td>
<td>18</td>
<td>2.67</td>
</tr>
</tbody>
</table>

### Table 5: Database package Chidamber-Kemerer and Cyclomatic Complexity metric results

<table>
<thead>
<tr>
<th>Class/ File</th>
<th>Coupling Between Objects (CBO)</th>
<th>Depth Of Inheritance Tree (DIT)</th>
<th>Lack of Cohesion Of Methods (LCOM)</th>
<th>Number Of Children (NOC)</th>
<th>Response For Class (RFC)</th>
<th>Weighted Method Complexity (WMC)</th>
<th>Average Cyclomatic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database Manager</td>
<td>122</td>
<td>2</td>
<td>75</td>
<td>0</td>
<td>650</td>
<td>1227</td>
<td>3.28</td>
</tr>
<tr>
<td>Constants</td>
<td>14</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### Table 6: User Interface Package Chidamber-Kemerer and Cyclomatic Complexity Metric Results

<table>
<thead>
<tr>
<th>Class/ File</th>
<th>Coupling Between Objects (CBO)</th>
<th>Depth Of Inheritance Tree (DIT)</th>
<th>Lack of Cohesion Of Methods (LCOM)</th>
<th>Number Of Children (NOC)</th>
<th>Response For Class (RFC)</th>
<th>Weighted Method Complexity (WMC)</th>
<th>Average Cyclomatic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Input Selector</td>
<td>27</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>171</td>
<td>150</td>
<td>4.69</td>
</tr>
<tr>
<td>Exit Page</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>1.5</td>
</tr>
<tr>
<td>Homepage</td>
<td>24</td>
<td>11</td>
<td>2</td>
<td>0</td>
<td>104</td>
<td>55</td>
<td>2.39</td>
</tr>
<tr>
<td>Management Page</td>
<td>13</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>42</td>
<td>14</td>
<td>1.75</td>
</tr>
<tr>
<td>Welcome Page</td>
<td>13</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>59</td>
<td>40</td>
<td>3.33</td>
</tr>
<tr>
<td>Synchronisation Page</td>
<td>22</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>50</td>
<td>36</td>
<td>2.12</td>
</tr>
<tr>
<td>Review Page</td>
<td>21</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>82</td>
<td>72</td>
<td>2.1</td>
</tr>
<tr>
<td>Data Input Summary</td>
<td>18</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>79</td>
<td>44</td>
<td>3.6</td>
</tr>
</tbody>
</table>

4. Sonarqube Homepage screenshot and Summary of Sonarqube Code View Page at a package level

![Sonarqube Homepage Screenshot](image)

Figure 1: Sonarqube Homepage Screenshot
Table 7: Sonarqube Code View Page Package Metric Values

<table>
<thead>
<tr>
<th>Package/Folder</th>
<th>Lines of Code</th>
<th>Bugs</th>
<th>Vulnerabilities</th>
<th>Code Smells</th>
<th>Duplications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>7700</td>
<td>150</td>
<td>418</td>
<td>1300</td>
<td>3.5%</td>
</tr>
<tr>
<td>Service Manager</td>
<td>1300</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>34%</td>
</tr>
<tr>
<td>Service</td>
<td>2400</td>
<td>2</td>
<td>7</td>
<td>168</td>
<td>32.5%</td>
</tr>
<tr>
<td>User Interface</td>
<td>3500</td>
<td>3</td>
<td>11</td>
<td>439</td>
<td>10.5%</td>
</tr>
</tbody>
</table>

5. Sonarqube Code View Page Screenshots (Identification)

The following figures show the Sonarqube Code View Page for each of the packages that were deemed to lack maintainability for the identification step of the method to improve maintainability.

Figure 2: Service Manager Package Sonarqube Code View Page

Figure 3: Service Package Sonarqube Code View Page
6. Application Quality Assessment — Maintainability Assessment Metric Results After Refactor

The following tables list the calculated results for the Chidamber-Kemerer and Cyclomatic Complexity metrics for the packages within an android software application for the maintainability assessment step of a method to improve maintainability.
<table>
<thead>
<tr>
<th>Class/ File</th>
<th>Coupling Between Objects (CBO)</th>
<th>Depth Of Inheritance Tree (DIT)</th>
<th>Lack of Cohesion Of Methods (LCOM)</th>
<th>Number Of Children (NOC)</th>
<th>Response For Class (RFC)</th>
<th>Weighted Method Complexity (WMC)</th>
<th>Average Cyclomatic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Manager File 1</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>24</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Service Manager File 2</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>26</td>
<td>7</td>
<td>3.5</td>
</tr>
<tr>
<td>Service Manager File 3</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>42</td>
<td>6</td>
<td>1.5</td>
</tr>
<tr>
<td>Service Manager File 4</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>44</td>
<td>13</td>
<td>3.25</td>
</tr>
<tr>
<td>Service Manager File 5</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>39</td>
<td>10</td>
<td>3.33</td>
</tr>
<tr>
<td>Service Manager File 6</td>
<td>9</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>30</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Service Manager File 7</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>20</td>
<td>5</td>
<td>1.67</td>
</tr>
<tr>
<td>Service Manager File 8</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>39</td>
<td>10</td>
<td>3.33</td>
</tr>
<tr>
<td>Service Manager File 9</td>
<td>23</td>
<td>4</td>
<td>4</td>
<td>13</td>
<td>9</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Service Manager File 10</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Service Manager File 11</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>17</td>
<td>3</td>
<td>1.5</td>
</tr>
<tr>
<td>Service Manager File 12</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>26</td>
<td>5</td>
<td>1.67</td>
</tr>
<tr>
<td>Service Manager File 13</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>44</td>
<td>7</td>
<td>1.75</td>
</tr>
<tr>
<td>Service Manager File 14</td>
<td>13</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>47</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Service Manager File 15</td>
<td>11</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>44</td>
<td>7</td>
<td>1.75</td>
</tr>
</tbody>
</table>
### Table 9: Service Package Chidamber-Kemerer and Cyclomatic Complexity Metric Results

<table>
<thead>
<tr>
<th>Class/ File</th>
<th>Coupling Between Objects (CBO)</th>
<th>Depth Of Inheritance Tree (DIT)</th>
<th>Lack of Cohesion Of Methods (LCOM)</th>
<th>Number Of Children (NOC)</th>
<th>Response For Class (RFC)</th>
<th>Weighted Method Complexity (WMC)</th>
<th>Average Cyclomatic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service File 1</td>
<td>11</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>17</td>
<td>5</td>
<td>1.67</td>
</tr>
<tr>
<td>Service File 2</td>
<td>42</td>
<td>2</td>
<td>2</td>
<td>13</td>
<td>45</td>
<td>32</td>
<td>2.67</td>
</tr>
<tr>
<td>Service File 3</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Service File 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3.83</td>
</tr>
<tr>
<td>Service File 5</td>
<td>43</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>34</td>
<td>23</td>
<td>5</td>
</tr>
<tr>
<td>Service File 6</td>
<td>27</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>47</td>
<td>20</td>
<td>3.4</td>
</tr>
<tr>
<td>Service File 7</td>
<td>15</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>46</td>
<td>17</td>
<td>3.6</td>
</tr>
<tr>
<td>Service File 8</td>
<td>15</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>46</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>Service File 9</td>
<td>18</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>44</td>
<td>12</td>
<td>6.8</td>
</tr>
<tr>
<td>Service File 10</td>
<td>33</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>64</td>
<td>34</td>
<td>2</td>
</tr>
<tr>
<td>Service File 11</td>
<td>14</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>17</td>
<td>6</td>
<td>2.33</td>
</tr>
<tr>
<td>Service File 12</td>
<td>14</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>32</td>
<td>7</td>
<td>1.8</td>
</tr>
<tr>
<td>Service File 13</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>14</td>
<td>9</td>
<td>1.67</td>
</tr>
<tr>
<td>Service File 14</td>
<td>11</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>14</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Service File 15</td>
<td>13</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>18</td>
<td>6</td>
<td>3.4</td>
</tr>
<tr>
<td>Service File 16</td>
<td>15</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>37</td>
<td>17</td>
<td>3.5</td>
</tr>
<tr>
<td>Service File 17</td>
<td>16</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>28</td>
<td>14</td>
<td>3</td>
</tr>
</tbody>
</table>

### Table 10: Data Input Selector Chidamber-Kemerer and Cyclomatic Complexity Metric Results

<table>
<thead>
<tr>
<th>Class/ File</th>
<th>Coupling Between Objects (CBO)</th>
<th>Depth Of Inheritance Tree (DIT)</th>
<th>Lack of Cohesion Of Methods (LCOM)</th>
<th>Number Of Children (NOC)</th>
<th>Response For Class (RFC)</th>
<th>Weighted Method Complexity (WMC)</th>
<th>Average Cyclomatic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Input Selector</td>
<td>30</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>172</td>
<td>114</td>
<td>3.8</td>
</tr>
</tbody>
</table>
APPENDIX C

DETAILED EXPLANATIONS ON THE MAINTAINABILITY ASSESSMENT STEP OF A METHOD TO IMPROVE MAINTAINABILITY IN OBJECT-ORIENTED SOFTWARE APPLICATIONS
1. Introduction

This appendix serves to provide a detailed explanation of how the conclusions were reached for the maintainability assessment performed for an android data collection application. The maintainability assessment was performed using instructions provided by the method to improve maintainability in object-oriented software applications.

2. Application Quality Assessment

The AQA is used as a means of maintainability assessment by following the instructions given in Section 3.6.2 of a method to improve maintainability in object-oriented software applications. The instructions state that the AQA should be performed in reverse and the results obtained before and after testing and refactoring should be compared to determine the effect the method has on the maintainability of the software application. This section will continue by comparing the metrics used by the AQA at a class, package and application level and conclude if further work is required on an Android data collection application.

Table 1 shows a comparison of the results obtained for the CK and CC metrics before and after completing the testing and refactoring step of the method to improve maintainability for a select few files.

Table 1: Selected File Chidamber-Kemerer and Cyclomatic Complexity Metric Results

<table>
<thead>
<tr>
<th>Class/ File</th>
<th>Before CBO</th>
<th>After CBO</th>
<th>Before DIT</th>
<th>After DIT</th>
<th>Before LCOM</th>
<th>After LCOM</th>
<th>Before NOC</th>
<th>After NOC</th>
<th>Before RFC</th>
<th>After RFC</th>
<th>Before WMC</th>
<th>After WMC</th>
<th>Before ACC</th>
<th>After ACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Input Selector</td>
<td>27</td>
<td>30</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>171</td>
<td>172</td>
<td>150</td>
<td>114</td>
<td>4.69</td>
<td>3.8</td>
</tr>
<tr>
<td>Service Manager Class</td>
<td>14</td>
<td>13</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>43</td>
<td>47</td>
<td>7</td>
<td>7</td>
<td>1.75</td>
<td>2</td>
</tr>
<tr>
<td>Service Class</td>
<td>14</td>
<td>15</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>45</td>
<td>37</td>
<td>21</td>
<td>17</td>
<td>3.5</td>
<td>3.5</td>
</tr>
</tbody>
</table>

The first class shown is the Data Input Selector found within the User Interface package. The CBO of the class increased from 27 to 30. This means that the class depends on three more classes then it did before it was refactored. This indicates a higher coupling which means that class maintainability has decreased. The RFC for the class increased by one. This value is not a good barometer of the classes maintainability since the class is responsible for a high amount of functionality within the application. The WMC and ACC of the class decreased significantly. This provides an explanation for the increase in the CBO since functionality for the
class needed to be simplified. This was achieved by moving class functionality to another to reduce the amount of functionality it is responsible for. The high decrease in class complexity makes up for the increase in CBO and it is concluded that no further work is required.

Fixes for the classes within the Service Manager package were similar, this means that results obtained for the CK and CC metrics for the classes revealed similar differences before and after. The CK and CC metrics for each of the classes within the package can be found in Appendix B, Section 4 of a method for improving maintainability in object-oriented software applications.

Table 1 shows the results comparison for the metrics calculated for a class within the Service Manager package. The CBO of the class decreased by one; the DIT, LCOM, NOC and WMC did not change; and RFC and ACC increased. It cannot be known if the improvement to the CBO outweighs the increase in RFC and ACC. Similar results were obtained for the other classes within the package.

Like the Service Manager package, fixes for classes within the Service package were similar. This means that results obtained for the CK and CC metrics for each class revealed similar differences before and after. The CK and CC metrics for each class within the Service package can be found in Appendix B, Section 4 of a method for improving maintainability in object-oriented software applications.

Table 1 shows the results comparison for metrics calculated for a class within the Service package. The CBO and ACC increased; the DIT, LCOM and NOC did not change; and RFC and WMC decreased. The fact that the CBO increased indicates that application maintainability has decreased as the object relies upon more objects. The reduction in the RFC indicates that the class is responsible for less functionality meaning its maintainability has increased. The decrease in CC indicates that class functionality has increased and outweighs the fact that its ACC has increased.

Analysing the conclusions made for all the classes that have been refactored, the conclusion can be made that AQA results at a package and application level will conclude that no further work is required. The confirmation of this conclusion starts by analysing the Martin packaging metrics for each of the packages before and the testing and refactoring steps of the method to improve maintainability were performed.
Table 2: Martin Packaging and Cyclomatic Complexity Metrics Before and After Refactor

<table>
<thead>
<tr>
<th>Package/Folder</th>
<th>Abstractness (A)</th>
<th>Afferent Coupling (Ca)</th>
<th>Efferent Coupling (Ce)</th>
<th>Depth of Inheritance Tree (D)</th>
<th>Instability (I)</th>
<th>Cyclomatic Complexity (CC)</th>
<th>Average Cyclomatic Complexity (ACC)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
<td>Before</td>
<td>After</td>
<td>Before</td>
<td>After</td>
<td>Before</td>
</tr>
<tr>
<td>User Interface</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>16</td>
<td>69</td>
<td>71</td>
<td>.17</td>
</tr>
<tr>
<td>Service Manager</td>
<td>.07</td>
<td>.07</td>
<td>44</td>
<td>32</td>
<td>48</td>
<td>48</td>
<td>.41</td>
</tr>
<tr>
<td>Service</td>
<td>.17</td>
<td>.1</td>
<td>23</td>
<td>29</td>
<td>101</td>
<td>106</td>
<td>.02</td>
</tr>
</tbody>
</table>

Table 2 shows a comparison of the Martin Packaging metrics before and after the refactoring and testing steps were completed. Four metric values for the User Interface package increased while the remaining values increased by minimal amounts or remained the same. Service Manager package metrics that did improve, did not improve significantly and metric values that deteriorated, deteriorated quite significantly. Service package metrics improved. No further work is required based on the Martin package values for each class.

Method for Object-Oriented Design Metrics Before and After Refactor

Figure 1 shows a bar graph describing the MOOD metric results before and after the testing and refactoring processes were completed. The colour of the bars that represent the metric values after the testing and refactoring processes were completed, indicate if maintainability has improved or deteriorated. This
determination was made using the explanation provided by Section 2.2.2 of a method for improving maintainability in object-oriented software applications. The AHF, AIF, CF and PF all improved. The MHF and MIF metrics results were concluded to be conducive to a deterioration in maintainability. The fact that more metrics improved rather than deteriorated indicates that there is no need to perform any further work.

In summary, the CK and CC metrics were compared before and after for each of the files that have been tested and refactored. The Data Input Selector class, and files within the Service package were deemed to have their maintainability levels improved while the effect on the classes within the Service Manager package could not be determined. Further maintainability insights were gained by comparing values for the Martin Packaging and CC metrics before and after the testing and refactoring processes were completed. It was concluded that the maintainability of the Data Input Selector class and files within the Service package has improved. The maintainability of the Service Manager package deteriorated. Lastly, the maintainability of the entire application was assessed by comparing MOOD metrics before and after refactor. It is concluded that no further work is required on the Android data collection application.

3. Sonarqube Assessment

The SA is used as a means of maintainability assessment by following the instructions specified by Figure 8 and the explanation provided by Section 3.6.3 of a method for improving maintainability in object-oriented software applications. The metrics shown by Sonarqube at different software application levels can be compared before and after the testing and refactoring processes and the determination of whether further work should be done is made. This section will continue by presenting and analysing metrics shown on the Sonarqube Homepage and determining if further work should be completed. Finally, package maintainability will be determined by making use of Sonarqubes Code View page at a package level.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugs</td>
<td>304</td>
<td>307</td>
</tr>
<tr>
<td>Vulnerabilities</td>
<td>495</td>
<td>501</td>
</tr>
<tr>
<td>Code Smells</td>
<td>4600</td>
<td>4500</td>
</tr>
<tr>
<td>Technical Debt</td>
<td>60 days</td>
<td>60 days</td>
</tr>
<tr>
<td>Duplications</td>
<td>15.20%</td>
<td>13.00%</td>
</tr>
<tr>
<td>Duplicated Blocks</td>
<td>499</td>
<td>457</td>
</tr>
</tbody>
</table>

Table 3 shows the Sonarqube metrics that can be viewed on the Sonarqube Homepage view before and after the testing and refactoring steps of the method to improve maintainability were performed. The determination
of the effect the testing and refactoring steps has on the maintainability of the software application is made using the explanations provided within Section 3.3.3 of a method for improving maintainability in object-oriented software applications.

The number of bugs and vulnerabilities increased; the number or duplications and code smells reduced; and the amount of days estimated to remove the TD within the software application did not change. The decrease in the number of code smells and duplications within the software application indicates that the maintainability of the application has improved.

The increase in the number of bugs and vulnerabilities both result in the conclusion that application maintainability did not improve. Vulnerabilities within the software application affect the security of a software application and is not useful in determining the effect the testing and refactoring steps have on the maintainability of the software application. The increase in the number of bugs within the software application implies that software application maintainability decreased. This cannot be taken at face value as the severity of the bugs determine its true effect on maintainability. Further investigation is required at a lower application level to determine the real effect of the introduced bugs on maintainability. The lack of change in the amount of TD is attributed to the fact that the Database Manager class accounts for most of the work required by Sonarqube. This means that it overshadows results achieved by the testing and refactoring processes since the class was not subjected to the processes.

Table 4: Sonarqube Metrics Before and After Refactor (Package/Folder Level)

<table>
<thead>
<tr>
<th>Package/Folder</th>
<th>Lines of Code (LOC)</th>
<th>Bugs</th>
<th>Code Smells</th>
<th>Duplication Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Interface</td>
<td>3500 3300</td>
<td>3 2</td>
<td>439 342</td>
<td>10.5% 4.9%</td>
</tr>
<tr>
<td>Service Manager</td>
<td>1300 1300</td>
<td>0 0</td>
<td>111 107</td>
<td>34% 26.2%</td>
</tr>
<tr>
<td>Service</td>
<td>2400 2200</td>
<td>2 4</td>
<td>168 157</td>
<td>32.5% 19.6%</td>
</tr>
</tbody>
</table>

Table 4 shows the Sonarqube metrics shown on the Sonarqube Code View page at a package level. Vulnerabilities are not listed as they do not affect maintainability. All packages affected by the testing and refactoring steps are deemed to have undergone an improvement to maintainability.
The LOC for each package either reduced or remained the same. This indicates that maintainability improved or remained the same. There is a lack of bugs within packages even before the testing and refactoring steps were completed. This means that the effect of introducing or removing any bugs does not greatly affect the maintainability of the application. Code smells and Duplications within the packages were greatly reduced.

The fact that all packages affected by the testing and refactoring step of the method experienced an improvement to maintainability lead to the conclusion that application maintainability has improved. This is further substantiated by the analysis of the Sonarqube Homepage view.

The next step specified by Section 3.6.3 of a method for improving maintainability in object-oriented software applications is to make use of the Sonarqube Code View page at a file level to determine the effect that the method has at a class level. This is not necessary as the effect of the testing and refactoring steps on the application has been determined sufficiently. It is therefore concluded that no further work is required on the Android data collection application.

In summary, different views provided by Sonarqube are used to determine the effect the method has on application maintainability. The usage of the features provided by these views was determined using the explanation provided within Section 3.6.3 of a method for improving maintainability in object-oriented software applications. Application maintainability at a package level was determined to have improved for all packages within the application. No further assessment was required as the conclusion that no further work is required has been sufficiently validated. It is therefore concluded that no further work is required on the Android data collection application.

4. ISO 25010 Assessment

Introduction

The ISO 25010 Assessment consists of following the steps outlined within Section 3.6.4 of a method for improving maintainability in object-oriented software applications. It is like the AQA in that it makes use of code metrics with the exception that insights are gained on maintainability via maintainability sub-characteristics defined by ISO 25010. The first step — as specified by Figure 21 (of a method for improving maintainability in object-oriented software applications), is to determine what metrics are used to assess the effects the method has on maintainability for each sub-characteristic. This assessment type is suited to users of the method that require insights on application maintainability in terms of maintainability sub-characteristics defined by ISO 25010. This section will continue by explaining the steps as they were performed for the Android data collection application.
Table 5: Metrics Used for Each Maintainability Sub-characteristic

<table>
<thead>
<tr>
<th>ISO 25010 Sub-characteristic</th>
<th>Metrics Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modularity</td>
<td>Coupling Between Objects; Instability; Coupling Factor; Lack of Cohesion of Methods; Efferent Coupling; Lines of Code; Cyclomatic Complexity; Number of Children; Afferent Coupling;</td>
</tr>
<tr>
<td>Reusability</td>
<td>Lines of Code; Cyclomatic Complexity; Depth of Inheritance Tree; Efferent Coupling;</td>
</tr>
<tr>
<td>Analysability</td>
<td>Lines of Code; Cyclomatic Complexity; Depth of Inheritance Tree;</td>
</tr>
<tr>
<td>Modifiability</td>
<td>Lines of Code; Cyclomatic Complexity; Response For Class; Depth of Inheritance Tree;</td>
</tr>
<tr>
<td>Testability</td>
<td>Lines of Code; Cyclomatic Complexity; Response For Class; Coupling Between Objects; Depth of Inheritance Tree;</td>
</tr>
</tbody>
</table>

Table 5 shows the metrics used to determine the effect application of the method has on each ISO 25010 maintainability sub-characteristic. These metrics are chosen using Table 7 and Table 8 as well as the explanations for what each of the metrics mean within Section 2.2.2 of a method for improving maintainability in object-oriented software applications. The metrics chosen for assessing each maintainability sub-characteristic are applicable for usage at different levels of the software application. This means that the determination of the effect that the method has on each sub-characteristic must be determined at each level.

Modularity

Table 6: Modularity Metrics Before and After Refactor (File/Class Level)

<table>
<thead>
<tr>
<th>Class/ File</th>
<th>Coupling Between Objects (CBO)</th>
<th>Lack of Cohesion of Methods (LCOM)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>Data Input Selector</td>
<td>27</td>
<td>30</td>
</tr>
<tr>
<td>Service Manager Class</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>Service Class</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
Table 6 shows the metrics chosen to assess the health of the modularity of the application at a class level before and after the testing and refactoring steps of the method to improve maintainability were performed. The first result shows metrics for the Data Input Selector class. The CBO increased indicating a decrease in modularity while the LCOM decreased indicating an improvement to modularity. The higher increase in CBO compared to the decrease in LCOM indicates that the modularity of the Data Input Selector class decreased.

Fixes for the classes within the Service Manager package were similar, this means that results obtained for the metrics chosen for the determination of the modularity of the package have similar results. Table 6 shows the metric results for one of the classes within the package. The CBO decreased indicating an improvement in modularity while the LCOM did not change. This means that the modularity of the class improved. Similar results for all the classes within the Service Manager package were obtained meaning that the modularity of all classes within the package has improved.

Fixes for classes within the Service package were similar, this means that results obtained for the metrics chosen for the determination of the modularity of the application have similar results. Table 6 shows the metrics picked for one of the classes within the package. The CBO increased indicating a decrease to modularity while the LCOM did not change. This indicates that the modularity of the class deteriorated. Similar results were achieved for each of the classes within the Service package meaning that the modularity of all classes within the package deteriorated.

Fixes for classes within the Service Manager package were similar, this means that results obtained for the metrics chosen for the determination of the modularity of the package have similar results. Table 6 shows the metric results for one of the classes within the package. The CBO decreased indicating an improvement in modularity while the LCOM did not change. This means that the modularity of the class improved. Similar results for all the classes within the Service Manager package were obtained meaning that the modularity of all classes within the package has improved.

Fixes for classes within the Service package were similar, this means that results obtained for the metrics chosen for the determination of the modularity of the application have similar results. Table 6 shows the metrics picked for one of the classes within the package. The CBO increased indicating a decrease to modularity while the LCOM did not change. This indicates that the modularity of the class deteriorated. Similar results were achieved for each of the classes within the Service package meaning that the modularity of all classes within the package deteriorated.

Table 7: Modularity Metrics Before and After Refactor (Package/Folder Level)

<table>
<thead>
<tr>
<th>Class/ File</th>
<th>Afferent Coupling (Ca)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
</tr>
<tr>
<td>User Interface</td>
<td>14</td>
</tr>
<tr>
<td>Service Manager</td>
<td>44</td>
</tr>
<tr>
<td>Service</td>
<td>23</td>
</tr>
</tbody>
</table>

Afferent Coupling (Ca) is a metric chosen to determine the modularity of the application at a package level. Table 7 shows Ca results for each of the packages that were affected by the testing and refactoring process. The Ca of the User Interface package increased, indicating a reduction in modularity; the Ca of the Service Manager package decreased, indicating an improvement in modularity; the Ca of the Service package
increased, indicating a reduction in modularity. These results are expected based on results for modularity metrics at a class level.

The final metric chosen at an application level for determining the health of the modularity of the application is the Coupling Factor (CF). The CF decreased from 6.03% to 5.84%. This indicates that application modularity improved, meaning that the improvement to modularity achieved by the changes within classes in the Service Manager package outweighed the negative effects of the changes within classes in the User Interface and Service packages.

**Reusability**

Table 8: Reusability Metrics Before and After Refactor (File/Class Level)

<table>
<thead>
<tr>
<th>File/Class</th>
<th>Lines of Code (LOC)</th>
<th>Number of Children (NOC)</th>
<th>Cyclomatic Complexity (CC)</th>
<th>Average Cyclomatic Complexity (ACC)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>Data Input Selector</td>
<td>1200</td>
<td>914</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Service Manager Class</td>
<td>131</td>
<td>132</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Service Class</td>
<td>131</td>
<td>114</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8 shows the metrics picked for the determination of reusability at a class level before and after the testing and refactoring processes were completed. The first results shown are the values calculated for the Data Input Selector class. All metrics decreased except for the NOC which did not change. This indicates that class reusability has improved.

Fixes for the classes within the Service Manager package were similar, this means that results obtained for the metrics chosen for the determination of the reusability of the package have similar results. Table 8 shows the metrics picked for one of the classes within the package. All metric values either increased or remained the same. This results in the conclusion that the class is less reusable. The same conclusion is reached for all classes within the package based on similar metric values.

Fixes for classes within the Service package were similar, this means that results obtained for metrics chosen for the determination of the reusability of the package have similar results. Table 8 shows metrics picked for one of the classes within the package. All metrics decreased except for the NOC—which did not change. This
indicates that class reusability has improved. The same conclusion is reached for all classes within the package based on similar metric values.

Table 9: Reusability Metrics Before and After Refactor (Package/Folder Level)

<table>
<thead>
<tr>
<th>File/Class</th>
<th>Lines of Code (LOC)</th>
<th>Cyclomatic Complexity (CC)</th>
<th>Average Cyclomatic Complexity (ACC)</th>
<th>Afferent Coupling (Ca)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>After</td>
<td>Before</td>
<td>After</td>
<td>Before</td>
</tr>
<tr>
<td>User Interface</td>
<td>3500</td>
<td>3300</td>
<td>417</td>
<td>378</td>
</tr>
<tr>
<td>Service Manager</td>
<td>1300</td>
<td>1300</td>
<td>115</td>
<td>116</td>
</tr>
<tr>
<td>Service</td>
<td>2400</td>
<td>2200</td>
<td>333</td>
<td>274</td>
</tr>
</tbody>
</table>

Table 9 shows the results obtained for the metrics chosen to assess reusability at a package level. The reusability of the User Interface and Service packages improved. This conclusion is reached since all values improved except for Ca. Only the Ca of the Service Manager package improved while other metrics increased or remained the same. This results in the conclusion that package reusability did not improve.

Table 10 shows the values calculated for the metrics chosen for assessing reusability at an application level. The LOC increased; the CC increased; and the ACC decreased. This indicates that the reusability of the application has decreased. Typically, when code is reused, we only reuse certain parts of a software application. This means that even though the reusability of the application has deteriorated, the reusability of the individual classes and folders/packages is a more valuable gauge of reusability.

Table 10: Reusability Metrics Before and After Refactor (Application Level)

<table>
<thead>
<tr>
<th>Lines of Code (LOC)</th>
<th>Cyclomatic Complexity (CC)</th>
<th>Average Cyclomatic Complexity (ACC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>After</td>
<td>Before</td>
</tr>
<tr>
<td>Application</td>
<td>36000</td>
<td>37000</td>
</tr>
</tbody>
</table>
Analysability

Table 11: Analysability Metrics Before and After Refactor (File/Class Level)

<table>
<thead>
<tr>
<th>File/Class</th>
<th>Lines of Code (LOC)</th>
<th>Depth of Inheritance Tree (DIT)</th>
<th>Cyclomatic Complexity (CC)</th>
<th>Average Cyclomatic Complexity (ACC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>After</td>
<td>Before</td>
<td>After</td>
<td>Before</td>
</tr>
<tr>
<td>Data Input Selector</td>
<td>1200</td>
<td>914</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Service Manager Class</td>
<td>131</td>
<td>132</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Service Class</td>
<td>131</td>
<td>114</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 11 shows metric values for metrics chosen to assess class analysability before and after the testing and refactoring processes were completed. The first results shown are the values calculated for the Data Input Selector class. All metric values decreased or remained the same. This results in the conclusion that the analysability of the class has improved.

The next set of values are shown for a class within the Service Manager package. Fixes for all classes within the Service Manager package were similar resulting in similar metric values for each. This means that the analysability of the classes within the Service Manager package has not improved. All metrics chosen for the determination of class analysability have either increased or remained the same.

The final set of values shown are the metrics for a class within the Service package. All metrics for this class have improved or remained the same, indicating an improvement to the analysability of the class. Fixes for all classes within the Service package were similar resulting in the conclusion that the analysability of all the classes have improved.
Table 12: Analysability Metrics Before and After Refactor (Package/Folder Level)

<table>
<thead>
<tr>
<th>File/Class</th>
<th>Lines of Code (LOC)</th>
<th>Cyclomatic Complexity (CC)</th>
<th>Average Cyclomatic Complexity (ACC)</th>
<th>Efferent Coupling (Ce)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>User Interface</td>
<td>3500</td>
<td>3300</td>
<td>417</td>
<td>378</td>
</tr>
<tr>
<td>Service Manager</td>
<td>1300</td>
<td>1300</td>
<td>115</td>
<td>116</td>
</tr>
<tr>
<td>Service</td>
<td>2400</td>
<td>2200</td>
<td>333</td>
<td>274</td>
</tr>
</tbody>
</table>

Table 12 shows results obtained for the metrics chosen to assess the analysability of the software application at a package level. All values decreased after completing the testing and refactoring steps except for Ce (which decreased) for the User Interface package. This results in the conclusion that the analysability of the package improved. Only the Ce of the Service Manager package improved while other metrics increased or remained the same. This results in the conclusion that package analysability did not improve. All values decreased after completing the testing and refactoring steps except for Ce for the Service package. This results in the conclusion that the analysability of the package improved.

Table 13: Analysability Metrics Before and After Refactor (Application Level)

<table>
<thead>
<tr>
<th>Lines of Code (LOC)</th>
<th>Cyclomatic Complexity (CC)</th>
<th>Average Cyclomatic Complexity (ACC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>After</td>
<td>Before</td>
</tr>
<tr>
<td>Application</td>
<td>36000</td>
<td>37000</td>
</tr>
</tbody>
</table>

Table 13 shows the metrics used to assess the health of application analysability. The LOC and CC increased while the ACC decreased. The fact that the amount of source code increased indicates that application analysability decreased. CC increased while ACC decreased, this implies that the analysability of the application decreased while analysability at lower levels of the application improved.
Table 14: Modifiability Metrics Before and After Refactor (File/Class Level)

<table>
<thead>
<tr>
<th>Class/ File</th>
<th>Before LOC</th>
<th>After LOC</th>
<th>Before DIT</th>
<th>After DIT</th>
<th>Before RFC</th>
<th>After RFC</th>
<th>Before WMC</th>
<th>After WMC</th>
<th>Before ACC</th>
<th>After ACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Input Selector</td>
<td>1200</td>
<td>914</td>
<td>5</td>
<td>5</td>
<td>171</td>
<td>172</td>
<td>150</td>
<td>114</td>
<td>4.69</td>
<td>3.8</td>
</tr>
<tr>
<td>Service Manager Class</td>
<td>131</td>
<td>132</td>
<td>5</td>
<td>5</td>
<td>43</td>
<td>47</td>
<td>7</td>
<td>7</td>
<td>1.75</td>
<td>2</td>
</tr>
<tr>
<td>Service Class</td>
<td>131</td>
<td>114</td>
<td>3</td>
<td>3</td>
<td>45</td>
<td>37</td>
<td>21</td>
<td>17</td>
<td>3</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Table 14 shows the values calculated for the metrics chosen to determine the effect the method to improve maintainability has on the modifiability of the Android application. The first results are shown for the Data Input Selector class. LOC, CC and ACC decreased; RFC increased by one; and DIT did not change. The decrease in LOC, CC and ACC indicate that there is a lower amount of source code that is less complex. This outweighs the effect of the slight increase to RFC and means that changes to the class will be easier. This ease of implementing changes implies modifiability improved.

The values for all metrics calculated for a file within the Service Manager package increased or remained the same. This results in the conclusion that class modifiability deteriorated. Metric values for other classes within the Service Manager package are similar implying that all classes within the package also suffered from a deterioration to modifiability.

Values for metrics before and after the testing and refactoring steps of the method, for classes within the Service package are favourable. All values increased except for DIT which remained the same. This means that the modifiability for each file improved. Fixes implemented for all classes within the Service package are similar resulting in similar results for all files. This implies that all the classes within the package have experienced an improvement to modifiability.
Table 15: Modifiability Metrics Before and After Refactor (Package/Folder Level)

<table>
<thead>
<tr>
<th>File/Class</th>
<th>Lines of Code (LOC)</th>
<th>Cyclomatic Complexity (CC)</th>
<th>Average Cyclomatic Complexity (ACC)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
<td>Before</td>
</tr>
<tr>
<td>User Interface</td>
<td>3500</td>
<td>3300</td>
<td>417</td>
</tr>
<tr>
<td>Service Manager</td>
<td>1300</td>
<td>1300</td>
<td>115</td>
</tr>
<tr>
<td>Service</td>
<td>2400</td>
<td>2200</td>
<td>333</td>
</tr>
</tbody>
</table>

Table 15 shows values for metrics chosen to assess the health of package modifiability before and after the testing and refactoring steps of the method are performed. The first set of results shown is for the User Interface package. All values decrease, implying that the modifiability of the package improved.

Results obtained for the Service Manager package are not favourable. The CC increased, and remaining package metrics remained the same. This results in the conclusion that the modifiability of the package deteriorated.

Metric values calculated for the Service package all decreased after the refactoring and testing steps of the method are complete. This means that package modifiability has improved.

Table 16: Modifiability Metrics Before and After Refactor (Application Level)

<table>
<thead>
<tr>
<th>Lines of Code (LOC)</th>
<th>Cyclomatic Complexity (CC)</th>
<th>Average Cyclomatic Complexity (ACC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>After</td>
<td>Before</td>
</tr>
<tr>
<td>Application</td>
<td>36000</td>
<td>37000</td>
</tr>
</tbody>
</table>

Table 16 shows the results obtained for metrics picked to assess application modifiability before and after the testing and refactoring steps of the method are complete. LOC and CC increased. This means that the Android
application is larger and more complex. The ACC of the application decreased implying that the complexity of lower levels of the application has decreased. Application modifiability has decreased but the modifiability of the application for areas of the application that have been tested and refactored has improved.

**Testability**

Table 17: Testability Metrics Before and After Refactor (File/Class Level)

<table>
<thead>
<tr>
<th>Class/ File</th>
<th>Coupling Between Objects (CBO)</th>
<th>Depth Of Inheritance Tree (DIT)</th>
<th>Number Of Children (NOC)</th>
<th>Response For Class (RFC)</th>
<th>Weighted Method Complexity (WMC)</th>
<th>Average Cyclomatic Complexity (ACC)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
<td>Before</td>
<td>After</td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>Data Input Selector</td>
<td>27</td>
<td>30</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Service Manager Class</td>
<td>14</td>
<td>13</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Service Class</td>
<td>14</td>
<td>15</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 17 shows the metric values picked to assess class testability before and after the testing and refactoring steps of the method are complete. CC is defined as the number of distinct execution paths that can be followed through each function/method within a class. This means that it can be considered as the number of tests required to fully exercise the functionality of a class — making it the most useful gauge of testability. The RFC is an indication of the number of behavioural tests required to fully exercise class behaviour. The CBO is an indication of the number of additional classes required when implementing automated tests.

Results for the Data Input Selector class are shown within Table 17. The fact that LOC, CC and ACC decreased implies that there is less source code to test and a lower number of tests required. Both the RFC and CBO increased resulting in more behavioural tests required as well as more set-up when performing automated tests. Comparing the increases and decreases to the metrics in question, it is concluded that the testability of the class has improved.

All metrics — except for CBO, have increased or remained the same for a class within the Service Manager package. This means that the file is larger, requires more automated tests but requires less objects during set-up when performing automated tests. A conclusion is made that class testability has decreased. All classes within the Service Manager package underwent similar changes. This means that results for metrics associated with testability are also similar. It is concluded that all files within the Service package have experienced a reduction to testability.
Table 18: Testability Metrics Before and After Refactor (Package/Folder Level)

<table>
<thead>
<tr>
<th>File/Class</th>
<th>Lines of Code (LOC)</th>
<th>Cyclomatic Complexity (CC)</th>
<th>Average Cyclomatic Complexity (ACC)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
<td>Before</td>
</tr>
<tr>
<td>User Interface</td>
<td>3500</td>
<td>3300</td>
<td>417</td>
</tr>
<tr>
<td>Service Manager</td>
<td>1300</td>
<td>1300</td>
<td>115</td>
</tr>
<tr>
<td>Service</td>
<td>2400</td>
<td>2200</td>
<td>333</td>
</tr>
</tbody>
</table>

Table 18 shows metrics associated with the testability of packages within the Android software application before and after the testing and refactoring steps of the method are performed. As expected from results at a class level, the testability of the User Interface and Service packages have improved while the testability of the Service Manager package deteriorated. These conclusions are reached by noting that all metrics have improved after the testing and refactoring processes were completed for the User Interface and Service packages while metric values have deteriorated or remained the same for the Service Manager package.

Table 19: Testability Metrics Before and After Refactor (Application Level)

<table>
<thead>
<tr>
<th>Lines of Code (LOC)</th>
<th>Cyclomatic Complexity (CC)</th>
<th>Average Cyclomatic Complexity (ACC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>After</td>
<td>Before</td>
</tr>
<tr>
<td>Application</td>
<td>36000</td>
<td>37000</td>
</tr>
</tbody>
</table>

Table 19 shows the metrics picked to assess application testability before and after the testing and refactoring steps of the method to improve maintainability is completed. The LOC of code within the application has increased. CC has increased whereas ACC has decreased. This results in the conclusion that application testability has decreased because of its size and complexity. It should be noted that the testability of all files (all files that are refactored) except for those within the Service Manager package has improved.
Summary

The ISO 25010 assessment is completed using the instructions provided by Figure 21 of a method to improve maintainability in object-oriented software applications. This assessment type is suited to users of the method that require insights on application maintainability in terms of maintainability sub-characteristics defined by ISO 25010. Specific code metrics are utilised to assess the effect the method has on each sub-characteristic. The health of each sub-characteristic is assessed at a class, package and application level.

Modularity for classes within the Service package as well as the Data Input Selector class deteriorated. Files within the Service Manager package experienced an improvement to modularity. Modularity is further assessed at a package level. The User Interface and Service packages experienced a deterioration to modularity while modularity of the Service Manager package improved. Overall application modularity improved, implying that the increase in modularity to the Service Manager package outweighs the decrease experienced by the User Interface and Service packages.

Reusability for classes within the Service package as well as for the Data Input Selector class improved. Files within the Service Manager package experienced a reduction in reusability. Reusability at a package level improved for the User Interface and Service packages while the reusability of the Service Manager package improved. The final level that reusability was assessed at, is at an application level. Application reusability decreased, implying that the reduction within the Service Manager package was greater than the improvement within the User Interface and Service packages.

Analysability of classes within the Service Manager package deteriorated while the analysability of classes within the Service package and the Data Input Selector class improved. Analysis of metrics picked to assess analysability at a package revealed that the analysability of the Data Input Selector class and classes within the Service package improved while the analysability of the Service Manager package deteriorated. Application analysability decreased implying that the reduction within the Service Manager package was greater than the improvement within the User Interface and Service packages.

Modifiability for classes within the Service package as well as for the Data Input Selector class improved. Files within the Service Manager package experienced a reduction in modifiability. Modifiability at a package level improved for the User Interface and Service packages while the modifiability of the Service Manager package improved. The final level that was assessed for modifiability is for the application. Application modifiability decreased, implying that the deterioration of the Service Manager package was greater than the improvement within the User Interface and Service packages.

Testability of classes within the Service Manager package deteriorated while classes within the Service package and the Data Input Selector class improved. Analysis of metrics picked to assess testability at a package revealed that the testability of the Data Input Selector class and classes within the Service package improved while the testability of the Service Manager package deteriorated. Application testability decreased...
implying that the deterioration of the Service Manager package was greater than the improvement within the User Interface and Service packages.

It is concluded that the maintainability of the application when analysing it at an application level can be considered to have decreased. This conclusion is reached by noting that testability, modifiability, analysability, reusability has decreased while modularity at an application level improved. The fact that the health of four of the five maintainability sub-characterises deteriorated at an application level leads to the conclusion that application maintainability has deteriorated.

Analysis of the results obtained at a package and file level within the ISO 25010 Assessment lead to the conclusion that areas of code that a subjected to the testing and refactoring processes do not require further work.

5. Conclusion

The Application Quality Assessment, Sonarqube Assessment and ISO 25010 Assessment are performed for an android data collection application. The conclusions in terms of whether maintainability improved or deteriorated are presented for each assessment type. The assessments serve primarily as a means of obtaining a granular view of the effect the method to improve maintainability has on object-oriented software applications.