Julius-Maximilians-Universität Würzburg
Computer Science VIII

THE RELATIONSHIP BETWEEN SOFTWARE COMPLICACY AND SOFTWARE RELIABILITY

Dissertation / *Dissertation*

for the doctoral degree / *zur Erlangung des Doktorgrads*

Doctor rerum naturalium (Dr. rer. nat.)

submitted by *vorgelegt von*

Michael Dorin

from *aus*

Eden Prairie, Minnesota, United States

Supervisor: Dr. Sergio Montenegro

July 11, 2022

Copyright

Statutory declaration

I declare that I have authored this work independently, that I have not used sources or resources other than those declared, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Date: ____________________ Signature ____________________

Acknowledgments

I want to acknowledge the most important person who supported this project. Despite all the time I abandoned her while writing and researching, despite my occasional bad moods when things were not going well, and despite all my travel, my wife Judith was always supportive of my work. Her support was and is immeasurable, including providing critical insights for the linguistic aspects of this dissertation. Considering all of this happened during her battle with cancer, I cannot even think of the right words to describe my appreciation.

Other participants supporting my project process include my son, my daughter, and my son-in-law. Patrick Dorin and Anders Koskinen spent hours of what would have been free time reviewing my work. My daughter Rebecca Koskinen provided support in ways too numerous to mention here and provided essential ideas for the construction of this thesis.

I wish to thank Dr. Sergio Montenegro for accepting me into his lab and supporting my work with his research direction. Professor Montenegro has supported my work throughout all of the paths. This project has been exciting, and I sincerely appreciate the opportunity given to me.

I wish to thank Dr. Brahama Dathan for accepting the challenge of participating on my committee.

I wish to thank Magda Montenegro for her efforts translating my abstract in to German.

I wish to thank my dean, Dr. Bhabani Misra, for all the support he provided. This process would have been immensely more difficult without his ongoing support. I wish to thank my students from the University of St. Thomas who helped with this work. Sarija Janardhanan, Rajkumar Kolakarluri, Isha Agarwal, Jayashree Pawar, and Trang Le. I very much appreciate all of the support provided by the University of St. Thomas.

I want to thank Juan Machuca of the University of Lima and Dr. Mario Chong of the University of the Pacific for their assistance and help with research for this project. I wish to thank all my friends and family in Peru for their support and permitting all my visits to present papers.

So many people have helped me during these years that I have likely forgotten some names. That is not intentional, and I sincerely appreciate their help as well.

Author's Publications

This work contains information from previously published material. The following papers, from the author, are considered for this thesis. Work from these papers is used directly, often verbatim, in the chapters of this thesis.

0.1 Journal Publications

1. M. Dorin and S. Montenegro, "Coding standards and human nature," International Journal of Performability Engineering, vol. 14, no. 6, June, 2018.
2. M. Dorin and S. Montenegro, "Designing uncomplicated software," Interfases, no. 11, pp. 73–86, 2018.
3. M. Dorin, "Building instant-up real-time operating systems," Embedded Systems Design, vol. 21, no. 5, p. 18, 2008.
4. M. Dorin and S. Montenegro, "Linguistic Economy Applied to Programming Language Identifiers," Scientific Research Publishing, Special Issue "Software Quality and Software Metrics", 2021.

0.2 Conference Publications

1. M. Dorin, S. Janardhanan and S. Montenegro, "Software Streamlining: Reducing Software to Essentials," 2021 IEEE International Conference on Automation/XXIV

Congress of the Chilean Association of Automatic Control (ICA-ACCA), 2021.

2. M. Dorin and S. Montenegro, “Ethical Lapses Create Complicated and Problematic Software,” 2021 IEEE/ACM 2nd International Workshop on Ethics in Software Engineering Research and Practice (SEthics). IEEE, 2021.

3. M. Dorin, H. Mortensen, and S. Montenegro, “Open Source Medical Device Safety: Loop Artificial Pancreas Case Report,” 2020 IEEE Symposium on Product Compliance Engineering-(SPCE Portland). IEEE, 2020.

4. M. Dorin and S. Montenegro, “Eliminating software caused mission failures,” 2019 IEEE Aerospace Conference, pp. 1–4, IEEE, 2019.

5. M. Dorin and S. Montenegro, “Metrics to Understand Future Maintenance Effort Required of Complicated Source Code,” Actas del II Congreso Internacional de Ingeniería de Sistemas, September 2019.

6. M. Dorin and S. Montenegro, “A life cycle for creating an uncomplicated software,” Actas del I Congreso Internacional de Ingeniería de Sistemas (pp. 129-140), Lima, 13 y 14 de septiembre del 2018.

7. M. Doirn, “Coding for Inspections and Reviews,” Proceedings of the 19th International Conference on Agile Software Development: Companion. ACM, 2018.

8. M. Dorin, J Machuca, S. Montenegro, “The Suitability of Java for Satellite Applications,” International Conference on CubeSat Technologiy, Arequipa, Peru, 2018.

9. M. Dorin, J. M. Machuca, and S. Montenegro, “Teaching software engineering to career-changers,” in 2019 IEEE World Conference on Engineering Education (EDUNINE), pp. 1–5, IEEE, 2019.

10. M. Dorin, T. Le, R. Kolakalur, and S. Montenegro, “Using Maching Learning Image Recognition for Code Review,” ACITY, London, November 2020.

0.3 Presentations and Posters

1. M. Dorin, "Implementing Dependable Software for Satellites," Workshop Pico- and Nano-Satellites, ILA Berlin, Berlin, Germany, April 2019.

2. M. Dorin, S. Montenegro, "Instant-Rodos: An Instant-Up Version of the Rodos Operating System," 12th Pico- and Nano-Satellite Workshop on "Techonologies for Small Satellite Research", Wurzburg, Germany, September 2019.

3. M.Dorin, S. Montenegro, "Software Design: It's Complicated," Simplicities and Complexities Conference, Bonn, Germany, May 2019

4. M. Dorin, H. Mortensen, and S. Montenegro, "Complicated Source Code and Medical Devices," MedFuse, Uniting Connected Medical Innovation Pioneers, Minneapolis, October 2019 .

5. M. Dorin, H. Mortensen and S. Montenegro, "Open Source Software & Personal Medical Devices: Interpreting risk through an evaluation of software testing," Congreso Internacional de Ingeniería de Sistemas, September 2019.

6. M.Dorin, S. Montenegro, "Applying the Linguistic Economy Principle to Programming Languages," III Congreso Internacional de Lingüística Computacional y de Corpus, October 2020.

7. M. Dorin, H. Mortensen, S. Montenegro, "Open Medical Device Safety:Loop Artificial Pancreas Case Report," SPCE, Portland, November 2020.

0.4 Other Related Authored Publications[1]

1. J. Machuca, M. Dorin, and A. Garcia-Yi, “Evaluacion experimental de un modelo de programacion lineal para el problema de ruteo de vehículos (vrp),” Interfases, no. 011, pp. 103–117, 2018.

2. I Agarwal, R Kolakaluri, M. Dorin, and Chong M., “TensorFlow for Doctors,” SIMBig 2019, Lima, Peru, Auagust 21–23, 2019, Proceedings, Vol. 1070. Springer, 2019.

[1] The publications listed here were not directly used in this thesis, however they were part of the overall research effort of defining software complicacy as well as validating image recognition tools.

Table of Contents

III Software Applied Linguistics 46

IV Machine Learning and Code “Tumor” Identification 60

V Preparation and Planning for Avoiding Complicacy 72

VII Conclusion 124

List of Tables

List of Figures

Abstract - English

An enduring engineering problem is the creation of unreliable software leading to unreliable systems. One reason for this is source code is written in a complicated manner making it too hard for humans to review and understand. Complicated code leads to other issues beyond dependability, such as expanded development efforts and ongoing difficulties with maintenance, ultimately costing developers and users more money.

There are many ideas regarding where blame lies in the creation of buggy and unreliable systems. One prevalent idea is the selected life cycle model is to blame. The oft-maligned "waterfall" life cycle model is a particularly popular recipient of blame. In response, many organizations changed their life cycle model in hopes of addressing these issues. Agile life cycle models have become very popular, and they promote communication between team members and end users. In theory, this communication leads to fewer misunderstandings and should lead to less complicated and more reliable code.

Changing the life cycle model can indeed address communications issues, which can resolve many problems with understanding requirements. However, most life cycle models do not specifically address coding practices or software architecture. Since lifecycle models do not address the structure of the code, they are often ineffective at addressing problems related to code complicacy.

This dissertation answers several research questions concerning software complicacy, beginning with an investigation of traditional metrics and static analysis to evaluate their usefulness as measurement tools. This dissertation also establishes a new concept in applied linguistics by creating a measurement of software complicacy based on linguistic economy. Linguistic economy describes the efficiencies of speech, and this thesis shows the applicability of linguistic economy to software. Embedded in each topic is a discussion of the ramifications of overly complicated software, including the relationship of complicacy to software faults. Image recognition using machine learning is also investigated as a potential method of identifying problematic source code.

The central part of the work focuses on analyzing the source code of hundreds of different projects from different areas. A static analysis was performed on the source code of each project, and traditional software metrics were calculated. Programs were also analyzed using techniques developed by linguists to measure expression and statement complicacy and identifier complicacy. Professional software engineers were also directly surveyed to understand mainstream perspectives.

This work shows it is possible to use traditional metrics as indicators of potential project bugginess. This work also discovered it is possible to use image recognition to identify problematic pieces of source code. Finally, this work discovered it is possible to use linguistic methods to determine which statements and expressions are least desirable and more complicated for programmers.

This work's principle conclusion is that there are multiple ways to discover traits indicating a project or a piece of source code has characteristics of being buggy. Traditional metrics and static analysis can be used to gain some understanding of software complicacy and bugginess potential. Linguistic economy demonstrates a new tool for measuring software complicacy, and machine learning can predict where bugs may lie in source code. The significant implication of this work is developers can recognize when a project is becoming buggy and take practical steps to avoid creating buggy projects.

Abstract - German

Ein nach wie vor ungelöstes technisches Problem ist das Erstellen unzuverlässiger Software, was zu unzuverlässigen Systemen führt. Eine der Ursachen ist, dass Quellcode auf zu komplizierte Weise geschrieben wird, so dass es für Menschen zu schwierig wird, ihn zu überprüfen und zu verstehen. Komplizierter Code führt über die Zuverlässigkeit hinaus zu weiteren Problemen, wie z. B. erweiterte Entwicklungsanstrengungen und anhaltende Schwierigkeiten bei der Wartung, was Entwickler und Benutzer letztendlich mehr Geld kostet. Vielfach schiebt man die Schuld an der Entwicklung von BuggySystemen auf das gewählte Lebenszyklusmodell. Das oft geschmähte "Wasserfall"Modell wird besonders häufig beschuldigt. Als Reaktion darauf änderten viele Organisationen ihr Lebenszyklusmodell in der Hoffnung, diese Probleme zu beheben. Agile Lebenszyklusmodelle sind sehr beliebt und fördern die Kommunikation zwischen Entwicklungsteam und Endnutzern. Theoretisch führt diese Kommunikation zu weniger Missverständnissen, und ein besseres Verständnis sollte zu weniger kompliziertem und zuverlässigerem Code führen. Eine Änderung des Lebenszyklusmodells kann tatsächlich Kommunikationsprobleme lösen, insbesondere beim Verständnis der Anforderungen. Die meisten Lebenszyklusmodelle selbst befassen sich jedoch nicht speziell mit Codierungspraktiken oder Softwarearchitekturen. Da Lebenszyklusmodelle aber nicht auf die Struktur des eigentlichen Codes eingehen, sind sie bei der Lösung von Problemen im Zusammenhang mit Codekompliziertheit wenig hilfreich. Diese Dissertation behandelt mehrere Forschungsfragen zur Software-Kompliziertheit, beginnend mit einer Untersuchung traditioneller Metriken und statischer

Analyse, um ihre Nützlichkeit als Messwerkzeug zu bewerten. Diese Dissertation etabliert auch ein wesentliches neues Konzept der angewandten Linguistik, indem sie auf der Basis der linguistischen Ökonomie ein Maß für Software-Kompliziertheit erstellt. Die linguistische Ökonomie beschreibt die Effizienz von Sprache, und diese Arbeit zeigt ihre Anwendbarkeit auf Software. Darin eingeschlossen ist eine Diskussion der Auswirkungen übermäßig komplizierter Software sowie des Zusammenhangs zwischen Kompliziertheit und Softwarefehlern. Als potenzielle Methode zur Identifizierung von problematischem Quellcode wird auch Bilderkennung mittels maschinellen Lernens untersucht. Der zentrale Teil der Arbeit konzentriert sich auf die Analyse des Quellcodes hunderter verschiedener Projekte aus unterschiedlichen Bereichen. Zuerst wird eine statische Analyse des Quellcodes jedes Projekts durchgeführt und traditionelle Softwaremetriken berechnet. Programme werden auch unter Verwendung linguistischenrTechniken analysiert, um die Kompliziertheit von Ausdrücken und Aussagen sowie die Kompliziertheit von Identifikatoren zu messen. Professionelle Software-Ingenieure wurden auch direkt befragt, um Mainstream-Perspektiven zu verstehen. Diese Arbeit zeigt, dass es möglich ist, traditionelle Metriken als Indikatoren für potenzielle Projektfehler zu verwenden. Sie belegt auch die Möglichkeit, vermittels Bilderkennung problematische Teile im Quellcode zu identifizieren. Schließlich beschreibt diese Arbeit die Entdeckung linguistischer Verfahren als neue Methode, Anweisungen und Ausdrücke zu identifizieren, die für Programmierer am wenigsten wünschenswert, da zu kompliziert sind. Die Hauptschlussfolgerung dieser Arbeit ist: Es gibt mehrere Möglichkeiten, Merkmale zu finden, die darauf hindeuten, dass ein Projekt oder ein Stück Quellcode fehlerbehaftet ist. Herkömmliche Metriken und statische Analysen können verwendet werden, um ein Verständnis für Kompliziertheit und Fehlerpotenziale von Software zu erlangen. Die linguistische Ökonomie demonstriert ein neues Werkzeug zur Messung von Softwarekompliziertheit, und maschinelles Lernen kann vorhersagen, wo potenzielle Fehler im Quellcode liegen könnten. Das wesentliche Ergebnis dieser Arbeit ist, Entwicklern Werkzeuge zur Verfügung zu stellen, mit denen sie erkennen können, dass ein Projekt

fehlerhaft wird. So können sie praktische Schritte unternehmen, um fehlerhafte Projekte zu vermeiden.

Preface[†]

0.1 Why This Work is Important

When a mistake is made during programming it often results in the creation of a flaw within the code being written. These coding flaws, known as "bugs" or faults, result in failures when the code is executed [151]. It is obvious to software engineers that software faults are more than simply problematic, they cause unreliable and undependable systems. The unhappy truth is software development practices do not always produce bug-free programs. Despite a strong desire to avoid bugs, developers, for many reasons, do not always deliver perfect work. Buggy software is always a problem, but bugs are particularly problematic in mission-critical applications. In this preface, a presentation of important bugs is made to emphasize the impact they had on human lives and important human endeavors. Many dramatic and tragic failures that occurred in mission critical systems, such as medical devices and aerospace systems, were caused by software bugs. This thesis ultimately shows a relationship between software bugs and software complicacy (the state of being complicated).

It is not difficult to find stories of software failures. For example, Simson Garfinkel in his paper "History's Worst Software Bugs" provides an interesting description of several bugs [70]. Kristine Pinedo describes four prominent bugs in another article [132]. Doing a simple Google search for the terms "worst software bug" returns 8.6 million results [73]. The sheer volume of bug analysis reports demonstrates a large underlying concern regard-

[†] This text is substantially from Conference Publication #2, "Ethical Lapses Create Complicated and Problematic Software."

ing problems caused by faulty code. While these types of articles are very informative, many times they do not address the full impact faulty code has on human lives or scientific exploration.

0.2 Complicacy and Space Exploration

Space exploration has a very long history of mission failures caused by software bugs. Many missions were ruined or nearly ruined solely because of software. For example, shortly after takeoff, the rocket carrying Mariner 1 responded improperly to commands from the guidance systems on the ground. The improper responses caused an apparently software-related guidance system failure [70]. A second example is the classic failure of the Ariane 5 rocket, which is famous enough to be taught in software engineering textbooks [19]. This failure was caused through the reuse of software thought to have been proven correct in the Ariane 4 rocket but not adequately revisited for Ariane 5.

With so many historical software failures in mind, it might be thought such problems have since been solved for space systems. Sadly, software bugs caused failures in additional missions, such as the Mars Climate Orbiter (1999) [123], the Mars Polar Lander (1999) [39], and the Mars Global Surveyor (2006) [204]. Finally, though many times it is impossible to know for sure, many engineers suspect software is to blame for many nano- and pico- satellite failures as well [98].

0.3 Complicacy and Aviation Systems

Aviation systems also have a history of problems related to bugs. In 1994 a Chinook helicopter crashed, killing 25 people in Northern Ireland. In this accident, it was suggested software problems might have kept the pilot from properly controlling the engines [147]. Also in the early 1990s, software was at least partly to blame for the crash of a Swedish JAS 39 fighter plane [26]. Fortunately there were no deaths in this accident, but the crash might have been avoided, as it is reported a similar software issue was apparent in the crash of a prototype JAS 39 in 1989 [69].

Software requirement issues in the 737-MAX resulted in a tremendous loss of life. The software in the 737-MAX flight computer was not designed to detect and cope with conflicting sensor readings. Although the 737-MAX has redundant flight computers, each monitored a different inclination sensor. There was no way for the system to know if there were conflicting sensor readings and, worse yet, the software also did not allow the pilot to override the software and take control [85]. The 737-MAX software is a clear example of a very complicated software system that might have benefited from additional review.

0.4 Complicacy and Medical Devices

One of the earliest, and likely the most famous, buggy medical system was the Therac-25 radiation therapy device made by Atomic Energy of Canada Limited [131]. The problems with this system were discovered during the 1980s and these bugs manifested themselves in the administration of excessive quantities of radiation during cancer treatments. The Therac-25 problems were responsible for the deaths of many patients, and hundreds more were injured [131].

While the Therac-25 device example is profound, it might be suggested that due to its age and fame these software problems are no longer relevant in the medical device world, and a path to eliminating such ethical dilemmas has been found in the intervening time. It might also be believed that since the medical device community is strongly regulated, new bugs of this type are rare.

However, many modern life affecting bugs have been found and new bugs are still reported. One recent example bug was found within the 8100 series of CareFusion Alaris infusion pump, causing the manufacturer to recall the devices in February 2020. It was reported this pump would not properly delay an infusion, as requested through the “Delay

Until" or "Multi-dose" features, leading to severe adverse health events [63]. According to the American Food and Drug Administration (FDA), 55 injuries and one death were reportedly caused by this bug [63]. Even more medical device examples exist. According to the database of device recalls maintained by the FDA, hundreds of device recalls were issued over software design problems in the past 16 years. In the year 2020 alone, more than 150 devices were recalled because of software bugs [60].

0.5 Complicacy and Nuclear Energy

Even nuclear power safety has been impacted by software problems. According to a report by Robert Brill of the U.S. Nuclear Regulatory Commission covering problem reports from 1994 to 1999, nearly a third of problems were classified as software related [27]. Software was also identified in a March 2020 report from the Nuclear Energy Agency analyzing common problems arising after plant modifications [1].

0.6 Other Relevant Considerations

Complicated software impacts teams in many ways. For example, it has been shown productivity drops of up to 50 percent are possible when software gets more complicated [167]. Staff turnover also increases when complicated software is part of a work environment [167]. Complicated software is said to contribute up to 25% of a software project's maintenance costs and up to 17% of the overall cost of the development effort [16]. A white paper by McCabe software describes how complicated problems potentially cause security issues as well as being harder to understand and test [158].

It has also been shown how "Cognitive Bias" and "Illogical Reasoning" can affect software engineers' design decisions [171]. One might conclude similar psychological factors play out to some extent in a review process, making software complicacy a possible impediment to a proper review. If code is difficult to review, it is possible it will only be superficially reviewed, or not reviewed at all.

0.7 Preface Conclusion

In 1967, NATO formed a study group that coined the term ‘software engineering’ as they believed software could be engineered using standard practices from other engineering disciplines. The NATO group thought applying standard engineering processes to software would be the solution to what they termed the ‘Software Crisis,’ which had manifested in the 1960s. Development projects were and continue to be delivered late, over budget, and full of bugs. It is interesting to note that the identification of the crisis and the original goal of correcting it occurred more than 50 years ago, and yet software problems continue. It is apparent there is no single way to address the problems of quality and reliability, and various approaches should be studied. This thesis examines the usage of classic methods and introduces new and novel methods to measure complicacy. Ideas are presented for reducing complicacy to support the important goal of creating more dependable software.

Part I

Overview

Chapter 1

Background and Introduction

1.1 Reliability and Dependability

Reliability is generally considered to be a measure of the frequency and severity of failures. A failure is generally considered to be an unacceptable behavior when the application is operating in acceptable conditions [151]. There is more to consider for mission-critical applications than the frequency of failures, as a single failure could destroy an entire mission. Dependability describes the trustworthiness of the system to provide necessary services when required [12]. Over the years, many strategies have been created to improve reliability and dependability in mission-critical applications. This is evident by the many standards created for reliability and dependability for the aerospace [28], railroad [24], medical device [86], and automotive [83] industries.

Redundancy of both hardware and software is a commonly required feature in these standards. In an engineering application, having redundancy means extra components are included to be used in case of a failure [163]. Redundancy has proven to be a critical factor in several successful aerospace missions [62]. However, as powerful a tool as redundancy can be, it is not all good news. Professor Nancy Leveson of the Massachusetts Institute of Technology writes some software issues have actually been made worse by redundancy. She describes how added complicacy caused by the complexity introduced with redundancy has resulted in failures that otherwise might not have occurred. Leveson points out that software risks are often misunderstood, as software failures are usually the result of dysfunctional interactions among modules, not module failure itself. She provides abundant examples to support her assertions [102].

Levenson is not the only expert to discuss software problems with respect to dependability. A 2016 paper on CubeSat reliability indicates for satellites that fail during the first six months of operation, most engineers believe the probability the failure was caused by software is at least 30% [98]. In fact, there are a staggering number of papers discussing problematic software failures [100] [7] [33] [74] [169]. Considering the extensive collection of works existing in this area, it is easy to conclude elimination of software faults (bugs) is the key to having dependable applications, including the key to creating successful redundancy systems.

1.2 Focus and Scope

As described in this introduction, software complicacy caused failures in systems with redundancy. Also previously described is the role software bugs have played in mission failure. One obvious conclusion directs software developers to reduce software complicacy and create less buggy software. For these reasons, in this thesis, I present new methods to produce dependable software that is less complicated and less buggy.

Metrics and Static Analysis - It is abundantly clear bugs play a significant role in all aspects of software dependability. It is essential for developers to know if their projects have a high propensity of being buggy. I believe any software metric that is too complicated for a developer to understand is not practical for regular use as such in this thesis I take a new and innovative look at traditional metrics and static analysis and show how these tools can be used to identify buggy projects. In addition, I also create a new method, Sheficom, for measuring efferent coupling. Chapter 2 extensively describes the application of both metrics and static analysis.

Preparation and Planning - One commonality between the creation of dependable code and dangerous military missions is proper preparation and planning. An intriguing aspect of military planning and preparation is the concept of the importance of rehearsal for missing understanding. Rehearsals are seldom if ever, done in software engineering. In Chapter 8, Section 8, I describe a new and innovative approach to software architecture creation using rehearsals.

Continuing with the concept of preparation and planning, in Chapter 9, I propose a new and innovative style of coding standard which makes it easier for software engineers to comply with organizational coding practices. In addition, in Chapter 6, I describe the innovative procedure of using machine learning coupled with image recognition to identify problematic parts of source code and thereby automate code reviews.

Software Complicacy - Software complicacy has been shown to cause problems, even in systems designed to recover from problems. Addressing the complicated state of source code is abundantly necessary. During the first half of the 1900s. George Zipf described a way to measure the effort required for speech. He showed how complicated words require more effort and are used less often in daily speech [206]. In this thesis, I show how these concepts from spoken languages can be applied to programming languages to determine the complicacy of source code. This novel and innovative approach to complicacy measurement is described in detail in Chapter 5.

1.3 Aims and Objectives

The following research questions are addressed in this thesis:

1. How can traditional metrics and static analysis be used to foretell project bugginess

and dependability?

2. How can software engineers write software with a lower probability of being buggy?
3. How can principles from linguistics be applied to measure software complicacy?
4. How can machine learning be applied to detect complicated parts of source code?
5. What steps can developers take to avoid complicacy and improve dependability when performing analysis and design workflows?
6. How can knowledge discovered by this thesis be used to analyze existing works?

1.4 Important Contributions

This thesis focuses on software reliability as influenced by complicacy. Within the thesis, new metrics are created, and new practical methodologies are defined. I now summarize the most important contributions.

1. **Creation of the Sheficom tool for coupling measurement.** As part of the exploration of research questions 1 and 2, it was discovered coupling impacts software dependability. However, many tools for measuring coupling are expensive or difficult to understand. To fill this void, a new method of measuring efferent coupling called Sheficom was invented. Sheficom is both simple to use and easy to understand. This is described in detail in Part 2 of this thesis.

2. **Creation of software applied linguistics.** Though many software metrics exist, previous to this thesis, no metric has measured software complicacy strictly from the perspective of what makes software complicated for humans to understand. The use of applied linguistics fills this void. This contribution explores research question 3, and part 3 of this thesis explores the concept.

3. **Use of Image Recognition for Code Reviews.** People often scan source code searching for visual triggers of possible trouble spots when reviewing code. Part 4 of this thesis, in the exploration of research question 4, demonstrates using machine learning to spot trouble spots in source code automatically.

4. **Introduction of narrations as a method to establish architecture.** Good software starts with a design so efforts need to be made early in the development process. This concept explores research questions 2 and 5 and is covered in detail in Part 5.

5. **Creation of a practical coding standard.** Reliable software requires proper coding be completed. This concept is covered in detail in Part 5, research question 5.

1.5 Structure of This Document

This document is divided into six parts, each containing one or more chapters. Each chapter presents background information along with the research objective. As this thesis covers a wide variety of topics related to software complicacy, each chapter has a section reviewing existing literature related to the topic being covered.

Part 1 contains the introduction and problem description.
Part 2 addresses research questions 1 and 2.
Part 3 addresses research question 3.
Part 4 addresses research question 4.
Part 5 addresses research question 5.
Part 6 addresses research question 6.
Part 7 presents the conclusion.

Part II

Software Metrics and Static Analysis

Chapter 2

Measuring Software Complicacy

2.1 Introduction and Background

"Perseverance lands safely on Mars" was the fortunate headline in many technical blogs and journals. Mission controllers' joy was justified, as historically more missions to Mars have failed than have succeeded [23]. Many space mission failures can be attributed to software problems[84]. Projects related to space exploration demonstrate one type of project that requires long-lived, bug-free software. Buggy software does not write itself; humans write buggy software. This section investigates the characteristics of buggy software.

Over the years many theories have attempted to identify things that might indicate a buggy project. Metrics have been created, promoted, defended, and criticized. The overall research performed here is intended to analyze different metrics and measurements and their relationship to project bugginess (and reliability). The importance of understanding the likelihood of a buggy project becomes more critical depending on the sophistication of the project being created. It goes without saying the stress of possibly creating a buggy checkers game is less than the stress of potentially creating buggy aerospace or medical device applications. The ramifications of this information provide insight into the creation and organization of projects with fewer bugs and more reliable software.

2.1.1 Existing Literature

Using metrics in an effort to predict bugs has been happening for decades. During the 1970's, Fumio Akiyama devised a method to predict a bug count based on the number of the total lines of code in a source file [68]. More recently, Steve McConnell suggested a potential bug count range estimated per KLOC[111].

In addition to traditional metrics, my thesis also looks at the number of authors who participated in the development of a project. There have been studies specifically analyzing how the number of authors participating on a project impacts its quality. A good example is Linus's Law. Linus's Law is described by Eric S. Raymond is his book *"The Cathedral and* *the Bazaar"* that stipulates "given enough eyeballs, all bugs are shallow" [139]. Linus's law would seem to imply a positive relationship between more authors and a reduction in bugs. That is to say, more authors do produce better code. There have been several attempts to determine the validity of Linus's Law. For example, in their paper, "Secure Open Source Collaboration: An Empirical Study of Linus' Law" Andrew Meneely and Laurie Williams show that files with many developers have more security faults than files with fewer developers [113]. Adding confusion to this topic, work done by Elaine Weyuker et al. did not see a significant correlation between the number of authors and the number of bugs [190]. However, the Weyuker paper was based on a small collection of projects analyzing bugs and authors over a number of years, giving the conclusion a variance of authors on the same project does influence bug counts.

Definitions of Software Metrics

Only easily understood metrics having available and reliable tools producing consistent results were used in this thesis. The metrics named below were extensively analyzed across many projects, all of which are listed in Appendix C. More information on each metric, including discovered minimum, maximum, average, and median values, can be found in Appendix 15.3.

Lines of Code - As mentioned, probably the oldest metric is a tally of the lines of code (LOC) contained in a project. Often counted as thousands of lines of code (KLOC), many people still consider it a good measure of software complicacy [112]. Counting lines, however, lacks granularity and cannot specify precisely where a complicated portion of code may lie. This thesis used both the Lizard tool (lizard) [197] and the Succinct Code Counter (Scc) tool [25] for measuring KLOC. The faster performance of Scc on huge collections of files made it preferable to work with.

Cyclomatic Complexity - In 1976, Thomas McCabe developed Cyclomatic Complexity, which measures the number of linearly independent paths through a program [109]. Intuitively, this metric seems useful for measuring software complicacy, as more decisions in source code likely mean the program is more difficult to understand. Though several tools were tried, finding a tool that worked over many projects of many sizes proved difficult. **Decision complexity** is an approximation of cyclomatic complexity and the Scc tool gives fast, and reliable calculations on all the sample projects. As with LOC calculation, the Lizard tool can measure cyclomatic complexity. Because of the faster performance of Scc, except where specifically stated, decision complexity will be reported instead of cyclomatic complexity.

Efferent coupling - In software engineering, efferent coupling measures the number of classes and data types known to a module. As with cyclomatic complexity, finding a tool for reliably measuring efferent coupling over various large projects proved difficult. So in this thesis, a straightforward metric called Sheficom was created to measure efferent coupling. The Sheficom metric computes external coupling by counting the number of headers included in a module. More detailed information on the creation of Sheficom can be found in Chapter 3.

Number of authors - This is a count of how many authors participated in writing a collection of source code. As mentioned, the nature of being complicated implies the difficulty of understanding. Authors were counted using the log option of the git tool [160].

Blank Spaces and Comments - It has long been believed blank spaces and code comments impact code quality, so they too are analyzed in this thesis. I measured the proportion of blank lines and comments also using the Scc tool [25]. I did not find a strong relation to bugginess from either blank lines or comments.

Halstead Measurements - Halstead Measurements were created in 1977 as part of an effort to establish software development as an empirical science. These measurements touched many areas, including measuring program difficulty and estimating development effort. All formulas for Halstead measurements are based on total and distinct operators and operands. For example, program difficulty is calculated by dividing the number of distinct operators by two and then multiplying by the total number of operands divided by the number of distinct operands [80]. Superficially, these measurements seem straightforward and easy to implement, and since they were developed in the 1970s, they certainly qualify as classic. As such, Halstead measurements were explored as part of this work. However, finding good and reliable tools for all projects proved difficult. Measurements made using the available open source tools were not adequate to indicate bug-related tendencies for these projects, so further investigation on Halstead was not performed.

2.1.2 Research Objective

As shown in section 2.1.1, there are many different existing studies provide interesting information. However, the contrasting approaches of these works does not provide overall clarity with respect to how to identify projects with the potential of low quality. To establish clarity, I have analyzed different metric measurements versus bug counts of hundreds of open source projects downloaded from GitHub [72]. Refer to Appendix C for the complete list. The log files for each project were scanned and various metrics were tabulated, including project size, number of authors, various metrics, and number of bugs per project. This information provides insight on the creation and organization of projects with fewer bugs and leading to more reliable software.

2.2 Large Projects Metric Studies[*]

To initiate the research of software metrics, a study of five large projects was performed, analyzing several years of their history to gauge the connections between software metrics and bugginess. The five projects' source code came from their active Git repositories. The Lizard tool was used to provide cyclomatic complexity measurements as well as the measurement of LOC [197]. Scanning project git logs was done to tally the bug and author counts. Commits were counted as bugs when bug-related keywords were found in their commit messages. Author count was based on tracking the email address of the committer. More information on both bug and author computing will be provided later in this chapter. Coupling measurements were made using the Sheficom tool. The results from these case studies lead to further investigation of prominent metrics.

2.2.1 Linux Study

Linux is a very well established operating system, and for this study, metrics measurements were made from archives spanning ten years, with the first year being 2008. See Table 2.1 for a summary of the results. Improvements in average module coupling and slight improvements to average cyclomatic complexity are apparent. The size of the project grows consistently over time. As the code grows, so does the percentage of veteran authors working on the code. Bug-related commits seem pretty consistent, not deviating much from the year-to-year average of 41.5 percent. One may suspect, considering the massive growth of the product, that having many veteran authors has contributed to maintaining a steady effort of bug fixes.

[*] The text in this section is substantially taken from Conference Publication #5, "Metrics to Understand Future Maintenance Effort Required of Complicated Source Code."

Table 2.1: Year-Over-Year Metric Measurements for Linux

Metric	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018
Sheficom (coupling)	12.55	12.36	12.6	12.78	12.58	12.42	12.26	12.16	12.1	12.15	11.94
Cyclomatic Complexity	4.5	4.5	4.4	4.4	4.4	4.3	4.3	4.3	4.3	4.3	4.3
KLOC	5646	7045	7966	8673	9521	10459	11090	11941	12938	14094	14519
Veteran Authors[1]	48%	51%	54%	55%	58%	60%	57%	59%	62%	61%	64%
Bug Commits[2]	45%	43%	42%	39%	39%	39%	40%	41%	43%	43%	43%

[1] Veteran Authors reflects the percent of authors who remained on the project year over year

[2] Bug Commits reflects the percentage of project commits tagged as bugs

2.2.2 Apache Study

The results of the Apache analysis are listed in Table 2.2. As far as significant projects go, Apache does an outstanding job with the level of effort dedicated to bug fixes versus improvements (approximately 32 percent for bug fixes). A positive aspect of Apache is how many authors remain from release to release (approximately 85 percent), which means the author understanding of the code base remains high, thus reducing the impact of complicated code. We can also see improvements to cyclomatic complexity.

Table 2.2: Year-Over-Year Metric Measurements for Apache

Metric	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018
Sheficom	10.7	10.65	10.74	10.99	10.92	11.03	11.27	11.56	11.88	11.81	12.02
Cyclomatic Complexity	7.57	6.74	6.75	6.68	6.68	6.63	6.64	6.5	6.44	6.37	6.23
KLOC	143	125	131	139	151	162	170	178	189	196	209
Veteran Authors[1]	81%	85%	85%	91%	75%	92%	78%	77%	86%	94%	93%
Bug Commits[2]	14%	43%	37%	41%	98%	28%	9%	18%	21%	26%	26%

[1] Veteran Authors reflects the percent of authors who remained on the project year over year

[2] Bug Commits reflects the percent of project commits tagged as bugs

2.2.3 MySQL Study

Results for MySQL are listed in Table 2.3. MySQL has dramatically increased in the number of lines of code. Note the growth of the codebase between 2015 and 2018, where a large amount of code was added. Along with the dramatic increase in the number of lines of code, the coupling (Sheficom) has also increased. The amount of effort spent on fixing bugs has also increased over time, but not dramatically so. The number of veteran authors is very high compared to other projects, contributing to controlling the amount of effort required for bug fixing.

Table 2.3: Year-Over-Year Metric Measurements for MySQL

Metric	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018
Sheficom	6.02	6.07	6.79	7.04	7.76	8.06	8.18	8.66	9.1	13.07	13.67
Cyclomatic Complexity	4.54	4.55	4.54	4.38	4.11	4.09	4.03	3.9	3.6	3.52	3.68
KLOC	904	923	1160	1204	1493	1571	1674	1833	2066	2581	2715
Veteran Authors[1]	46%	58%	66%	71%	66%	73%	73%	67%	82%	77%	92%
Bug Commits[2]	33%	37%	34%	30%	35%	36%	40%	37%	41%	42%	42%

[1] Veteran Authors reflect the percent of authors who remained on the project year over year

[2] Bug Commits reflects the percent of project commits tagged as bugs

2.2.4 PHP Metrics Study

PHP results are shown in Table 2.4. Over time, the number of lines of code has grown. Average Sheficom (coupling) and cyclomatic complexity have gotten worse over the years. The number of veteran authors has fluctuated but still maintains a high percent year after year. A high cyclomatic complexity suggests complicated code in this product. Interestingly PHP bug commits actually improve over the years, bucking the trend of other projects.

Table 2.4: Year-Over-Year Metrics Measurements for PHP

Metric	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018
Sheficom (Coupling)	6.47	6.95	6.86	7.25	7.39	7.34	7.31	7.29	7.22	7.18	7.18
Cyclomatic Complexity	7.68	7.65	7.69	7.65	7.74	7.82	7.77	8.33	8.27	8.23	8.19
KLOC	600	789	733	745	812	836	900	954	1036	1183	1234
Veteran Authors[1]	59%	83%	80%	84%	49%	45%	53%	36%	42%	45%	64%
Bug Commits[2]	54%	61%	59%	59%	41%	26%	37%	34%	30%	25%	24%

[1] Veteran Authors reflects the percent of authors who remained on the project year over year

[2] Bug Commits reflects the percent of project commits tagged as bugs

2.2.5 ImageMagick Study

The ImageMagick project was also analyzed, and the results are shown in Table 2.5. Measurements such as cyclomatic complexity are not superb. However, note the strong connection between veteran authors and bug changes. After 2014 a decrease in existing authors coincides with a dramatic increase in bug efforts. The amount of energy spent on bugs impacted the amount of effort s[emt]on new features as new authors arrived.

Table 2.5: Year-Over-Year Metrics Measurements for ImageMagick

Metric	**2010**	**2011**	**2012**	**2013**	**2014**	**2015**	**2016**	**2017**	**2018**
Sheficom	17.24	18.13	17.93	18.77	19.05	19.2	19.96	20.04	19.5
Cyclomatic Complexity	9.1	9.4	9.4	9.4	9.1	9.2	9.3	9.4	9.5
KLOC	287	297	314	315	335	341	342	346	353
Veteran Authors[1]	75%	100%	100%	80%	100%	44%	14%	12%	11%
Bug Commits[2]	5%	3%	27%	5%	13%	17%	27%	34%	47%

[1] Veteran Authors reflect the percent of authors who remained on the project year over year

[2] Bug Commits reflects the percent of project commits tagged as bugs

2.2.6 Discussion of Metrics Case Study Results

Though the results from these projects do not show any conclusive evidence the measurements made foretell project bugginess, there are some trends warranting further investigation. For example, especially apparent with the ImageMagick project, the number of authors in a project makes an important contribution to efforts spent on bugs.

2.3 Traditional Metrics Application to Project Bugginess

2.3.1 Methods

Included Projects

As mentioned, data was gathered from a collection of several hundred projects found on GitHub [72] and listed in Appendix C. The projects were written in both C and C++ and have various sizes and lifetimes. As it is impossible to count bugs that have not yet been reported, only mature projects were selected. Projects that were less than seven years old were discarded. Also, small projects with less than 1000 lines of code were also discarded.

Design and Procedure

The research design for this section analyzes the relationship between the various software project metrics and the number of bugs produced in those projects. Descriptive information on the metrics used can be found in Section 2.1.1. As many abbreviations exist in this section, Table 2.6 gives definitions.

Bug Count -The first step was to determine the number of bugs contained in each selected project. Because of the informal nature of many open source projects, getting a good understanding of project bugginess is challenging. The process of determining the exact number of bugs in a project is not straightforward. The method used to identify and count bugs is based on a technique developed by Idan Amit, and Dror Feitelson [9]. Bugs are identified using commit text comments in the git-log. In this chapter, bug counts were normalized to bugs per KLOC (BKLOC). Bugs were considered ‘bad’ if the description contained a term such as ‘crash’ or ‘exception.’ Bad bugs were also normalized per KLOC (BBKLOC).

Author Count -The next step required was identifying how many authors participated in each project. In this step, a tally of the different email addresses of those committing files was made. There was no attempt to filter out multiple email addresses used by a single person. Author count was also normalized based on KLOC (AKLOC).

Decision Complexity - It did not prove easy to find reliable tools for measuring cyclomatic complexity. In place of cyclomatic complexity, decision complexity (a count of logical decisions) measurements were taken. This measurement was performed using the open-source tool named Succinct Code Counter (Scc) [25].

Efferent Coupling - Measurements for efferent coupling were made using Sheficom. The Python source code for the tool used to measure efferent coupling is shown in Appendix F.

Lines of Code - Metrics related to estimating project lines of code were made using the Scc tool [25].

Central Tendencies - The central tendencies of several descriptive features, such as thousands of lines of code per project (KLOC), age, and the number of authors, were calculated. The measurements were made with Python and Linux shell scripts. These are shown in Table 2.7.

Outlier Removal - When necessary, outlier removal was performed using the interquartile range (IQR) method. Anything not in the range of (First Quartile - 1.5 * IQR) and (Third Quartile + 1.5 * IQR) was considered an outlier and was removed.

Table 2.6: Descriptions and Abbreviations of Metrics Used

Description	Abbreviation
Thousands of Lines of Code	KLOC
Authors per KLOC	AKLOC
Bugs per KLOC	BKLOC
Bad Bugs per KLOC	BBKLOC
Complexity per KLOC	SCCKLOC
Efferent Coupling per KLOC	EKLOC

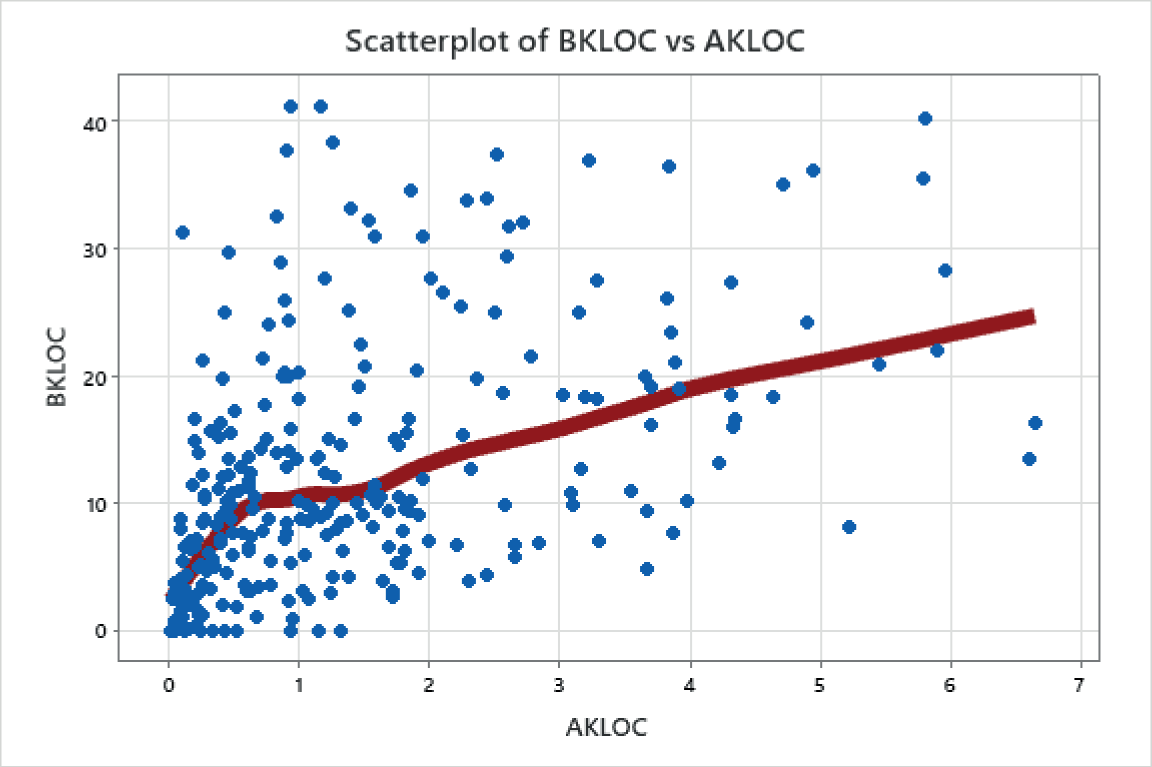


Figure 2.1: Authors Per KLOC (AKLOC) vs. Bugs Per KLOC (BKLOC)

2.3.2 Project Bugginess Results

The overall calculations of the selected measurements are shown in Table 2.7. As a reminder, for this project, the mean age was about seven years, so only projects older than seven years were evaluated. Mature projects do not necessarily have more bugs than immature projects. Instead, they have had more opportunities for bugs to be exposed and reported by users. Overall the BKLOC for all projects was about 14. The overall BBKLOC was nearly one-third.

There were a few metrics found to be significant. For example, the overall AKLOC was 1.5 authors per thousand lines of code. Projects with more AKLOC have a tendency to have more BKLOC. This is visible in Figure 2.1.

Efferent coupling was another metric that indicated project bugginess. Measurements of EKLOC using Sheficom show that projects with larger efferent coupling values tend to have more bugs than those with lower. Figure 2.2 shows the tendency of the BKLOC to

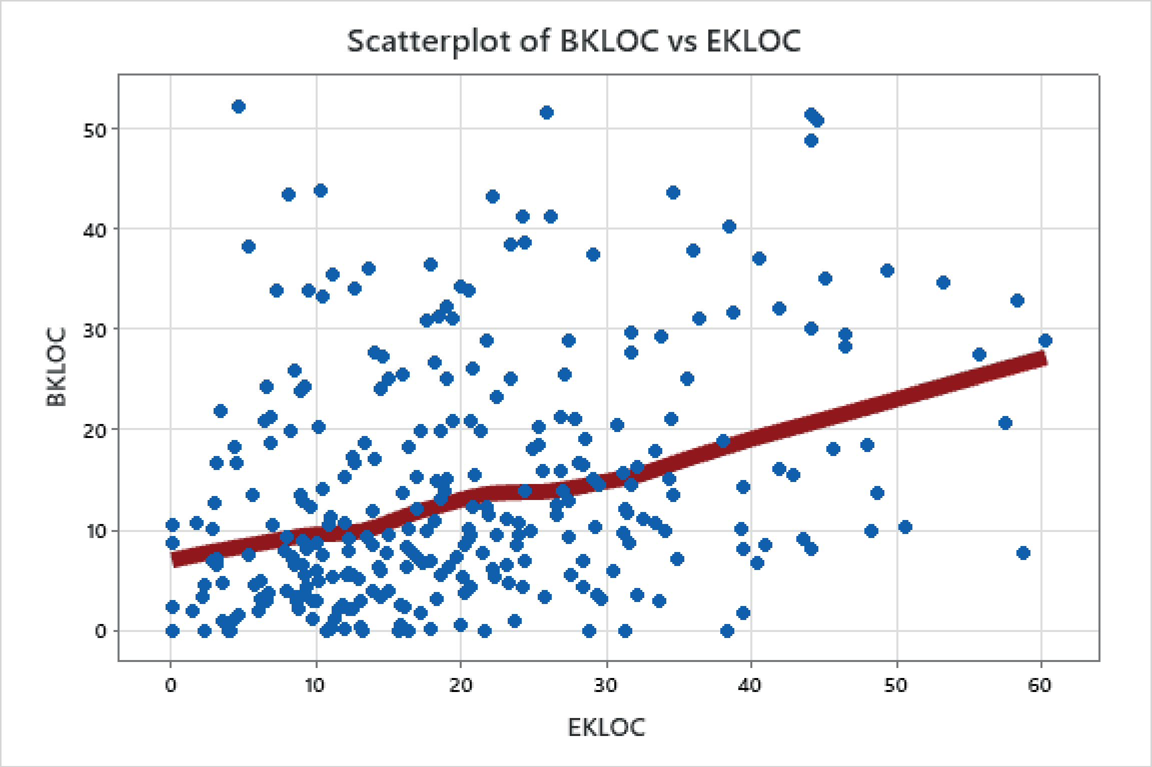


Figure 2.2: Efferent Coupling Per KLOC (EKLOC) vs. Bugs Per KLOC (BKLOC)

increase as the EKLOC increases. Sheficom and coupling are discussed additionally in Chapter 3.

Decision complexity, approximating cyclomatic complexity, demonstrated projects having files with fewer logical decisions were less buggy than those with more decisions. This is shown in Figure 2.3.

Finally, likely a crucial metric. As a reminder, bad bugs are bugs that were found to have phrases such as “crash” in their description. Projects with fewer bad bugs also had fewer overall bugs. This can be seen in Figure 2.4.

2.3.3 Discussion

As mentioned, Figure 2.1 shows the tendency that having more authors working on the same code base seems to lead to more bugs. It might be assumed more authors produce more lines of code and that by itself presents more opportunities for bugs. However, Figure

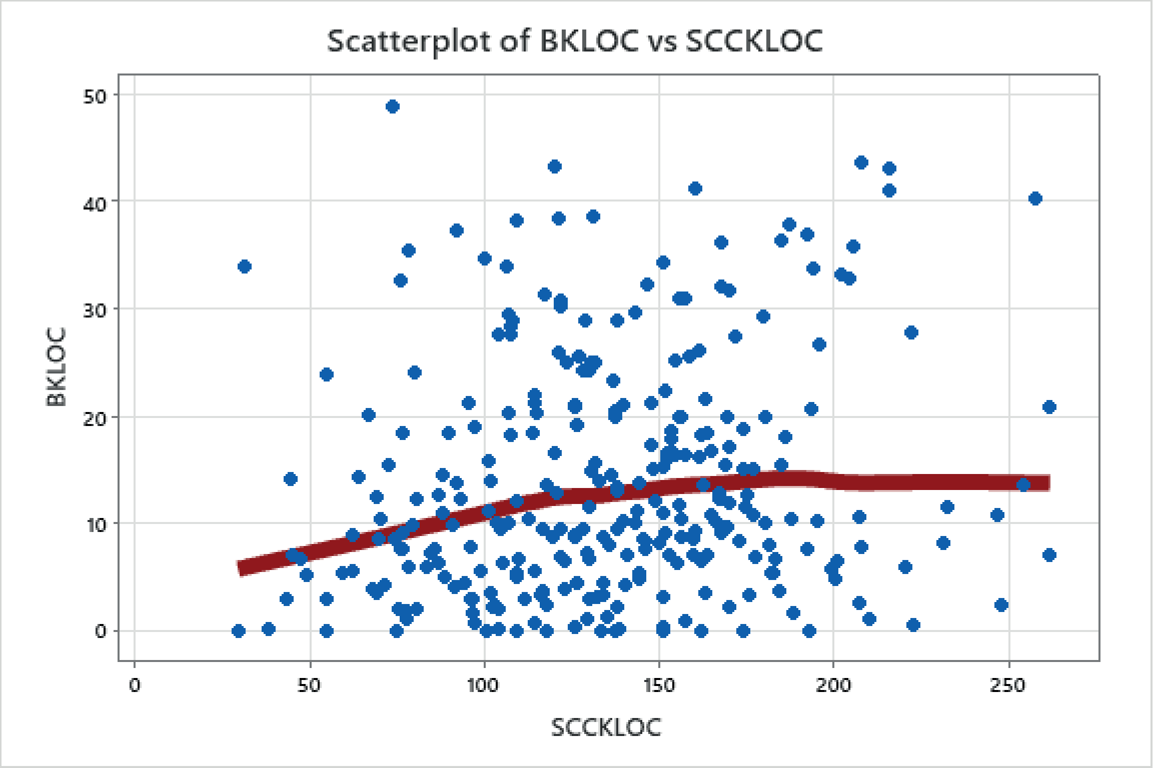


Figure 2.3: Decisions Per Kloc (SCCKLOC) vs. Bugs Per KLOC (BKLOC)

Table 2.7: Tendencies of Software Metrics

Metric[1]	**Mean Value**
Authors (AKLOC)	1.5
Bugs (BKLOC)	14.17
Bad Bugs (BBKLOC)	0.35
Decision Complexity (SCCKLOC)	136.8
Efferent Coupling (Sheficom)	20.73
Average KLOC per Project	291.1

[1] Metrics are measured per project per KLOC

2.5, shows that the tendency of BKLOC actually lowers for projects with higher than average KLOC. This suggests the increasing BKLOC is not simply explained by more code being written by more authors.

In their work on technical debt identification, Nico Zazworka et al. identified modularity problems and coupling issues as solid indicators of technical debt, with modularity

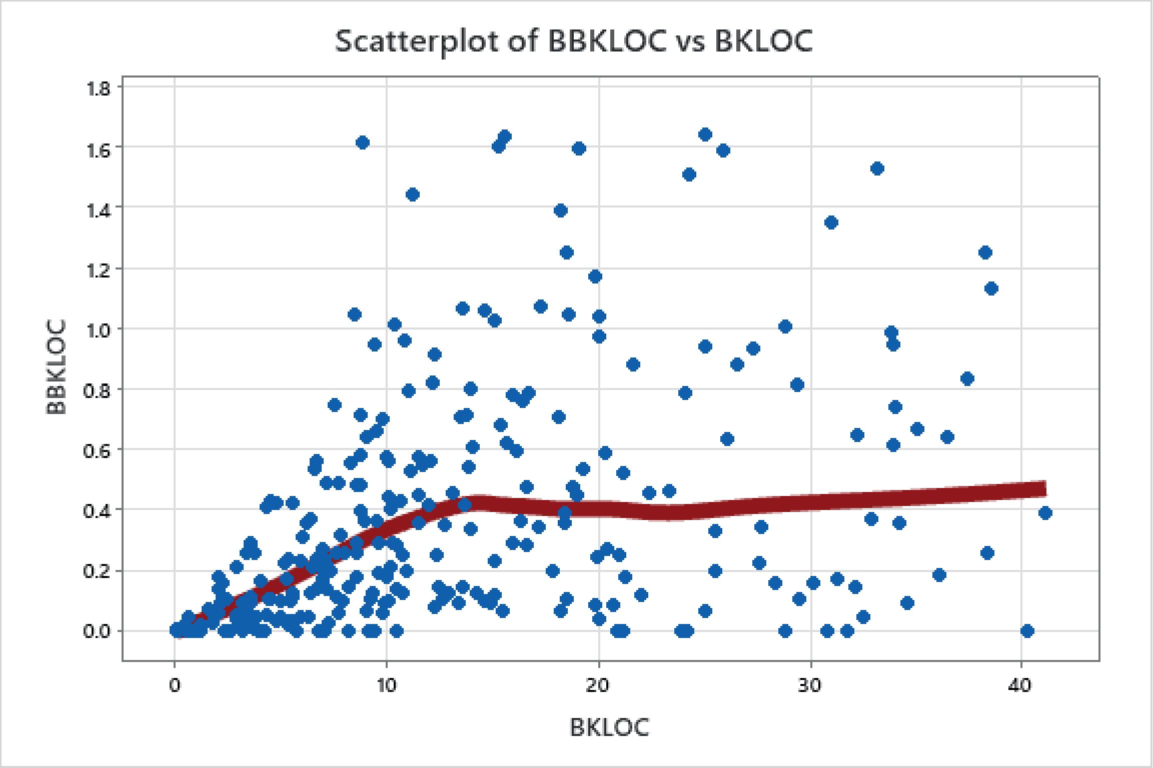


Figure 2.4: Bad Bugs per KLOC (BBKLOC) vs. Bugs Per KLOC (BKLOC)

issues leading to the highest likelihood of change[200]. Additional analysis as part of this work as shown in Figure 13.3 shows buggy projects do have higher coupling. The results would indicate a reduction in coupling would be to the benefit of a project.

Decision complexity also appears to be an indicator of a propensity for bugs, and reducing the number of decisions made in the code can improve code quality.

2.3.4 Conclusion

Although software engineers have a desire to create quality projects programmers inevitably write code with bugs. Avoiding bugs becomes more critical when a software failure can have dire consequences.

The idiom “too many cooks spoil the broth” appears to apply to software development. The number of authors can be used as a method in prognosticating the bugginess of code in a project.

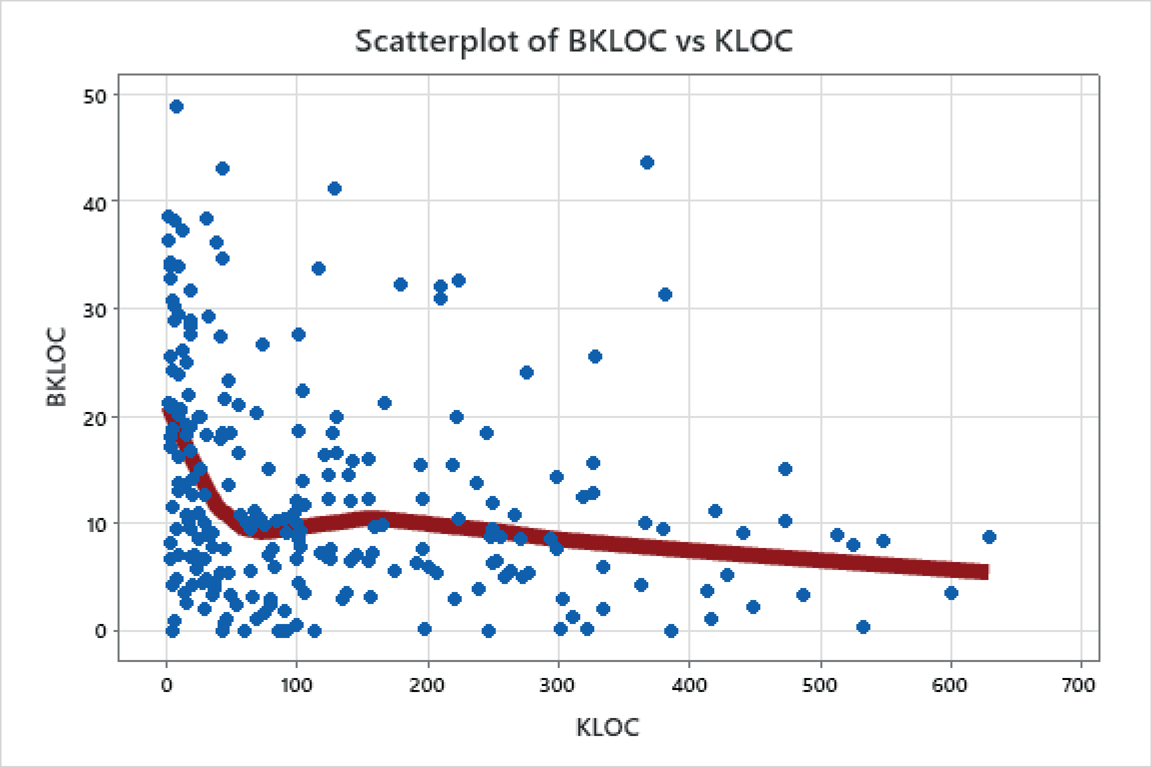


Figure 2.5: Lines of Code (KLOC) vs. Bugs Per KLOC (BKLOC)

For approaching modularity, developers need more than a tacit understanding of coupling and cohesion. This is not always covered adequately by software engineering courses. There is a tendency to leave a course remembering the slogan, 'high cohesion low coupling,' but without an adequate understanding of how to identify coupling and cohesion problems correctly.

This research shows how buggy projects also exhibit other characteristics, measured using traditional metrics, that are not shown in projects with fewer bugs. Traditional metrics are indicators of what may make software source code more difficult to understand and more complicated.

Chapter 3

Sheficom - The Easy Way to Estimate Efferent Coupling

3.1 Introduction and Background

Coupling, which measures the dependence modules have on one another, is a vital notion of software engineering [137]. Simply following Miller's Law, which describes the human ability to think about multiple concepts simultaneously, it is evident modules with higher coupling will be more challenging to read and understand [116]. It can then be expected harder to read and understand modules will be more troublesome and have more faults. This makes coupling measurement critical.

This chapter introduces the Sheficom metric, which estimates software coupling. Sheficom simplifies coupling measurement by only counting included headers. This results in an uncomplicated and easily implemented utility for many programming languages. To demonstrate the utility of Sheficom, measurements were made on source files from various open-source projects, including the projects used in Chapter 2. Files having the term 'crash' reported in their git-log were selected. A strong association between files with reporting crashes and the Sheficom measured coupling can be seen. This strong association demonstrates coupling issues can be identified and can indicate bug potential in modules. Having early knowledge that files may be faulty can allow for required refactoring during software deployment.

3.1.1 Existing Literature

Many articles suggest a connection between coupling and program faults. For example, Syed Nadeem Ahsan and Franz Wotawa find a correlation between logical-coupling metrics and the number of bugs [2]. Gihan Ubayawardana and Damith Karunaratna describe coupling problems and bug detection in their paper, "Bug prediction model using code smells [183]". Mustafa Efendioglu, Alper Sen, and Yavuz Koroglu even apply machine learning to coupling in their paper "Bug prediction of systemic models using machine learning" [55].

Metric Standards in general are discussed in the paper "Software metric selection methods." Zubaidah Bukhari, Jamaiah Yahaya, and Aziz Deraman describe several important attributes of software metrics, including independence, automation, simplicity, and accuracy. [31]. Prabhjot Kaur, in his paper "A Review of Software Metric and Measurement" explains that the goals of metrics should include better software quality, and should improve software inspection [87].

3.2 Methods

3.2.1 Step 1 - Gather Source Code to Measure

Source files were gathered from different origins. For example, some source comes from textbooks, such as the XINU operating system by Douglas Comer [37] the Minix operating system by Andrew Tanenbaum [170], and "Operating Systems: Three Easy Pieces by Andrea and Remzi (OSTEP) Arpaci-Dusseau" [11]. Since the source code described in this paragraph originated as part of college textbooks, it is expected that Sheficom will indicate low complicacy.

Also, some source files from the various open-source projects used in Chapter 2, are included. In order to perform measurements based on complicacy, it is assumed files reporting more crashes are likely more complicated, so the files were grouped based on the reported number of crashes. To ensure sufficient time had passed for crashes to be reported, only files from projects older than seven years were selected. File crash counts were discovered by scanning the git-log of each file.

3.2.2 Step 2 - Enumerate Requirements and Create Tool

For the metric to be useful, it needs to be consistent and simple to understand. With this in mind the following requirements are stipulated for the metric.

1. Each “include” file from each source file shall be counted.

2. If an “include” file has the same name as a ‘C’ or ‘CPP’ source file, a total count of includes from both the include file and the source file shall be made.

3. If a particular “include” file is discovered to only have further includes in it, with no other code or function headers, it shall be declared a cheater and a count of those “include” files shall be added to the total count of includes for the measured source file.

4. Standard library “include” files shall be treated as a single module.

Based on the requirements in step 2, a small Python utility was written.

3.2.3 Step 3 - Measure Against Test Files

First, the Sheficom tool was run against the sample files within their original groups. For example, files reporting one crash were grouped with others reporting one crash. Source code from textbooks was grouped by the originating book. The Sheficom value for each group was then calculated. This was followed by an effort to determine if a statistically significant relationship existed among the data sets. Statistical ANOVA testing was run, followed by Tukey testing against the calculated means.

3.3 Results

3.3.1 Results from Step 1

Files for XINU, Minux, and OSTEP were downloaded. Files were also collected from previously mentioned open-source projects. More than 120,000 "C" and "CPP" files older than seven years were found for classification.

3.3.2 Results from Step 2

The Sheficom tool is approximately 350 lines of Python code, including comments. It has an open-source license and is checked into GitHub [72], (`https://github.com/mikedorin/sheficom`). For operation, Sheficom requires the Python libraries "os", "defaultdict" library, and "sys." In addition, Sheficom requires the "re" library for removing comments from source code. Sheficom runs from the command line and can be automated.

3.3.3 Results From Step 3

For crash files reporting zero to ten crashes, almost none of the groups were found to have a statistically significant relationship. However groups with 11 to 456 reported crashes were statistically related. At this point the one-Way ANOVA testing was run against the group reporting zero crashes vs. all other crash reporting groups. One-way ANOVA was also run against the group reporting zero crashes and the large group reporting 11 to 456 crashes.

For the group of files reporting zero crashes compared to all the crash reporting groups,

Table 3.1: Results From Sheficom Calulation of Files

Source	Sheficom Measurement
Minix	3.6
OSTEP	5.6
XINU	30.1
Group of Files Reporting 0 Crashes	7.8
Group of Files Reporting 1 Crash	12.8
Group of Files Reporting 2 Crashes	17.9
Group of Files Reporting 3 Crashes	22.5
Group of Files Reporting 4 Crashes	23.8
Group of Files Reporting 5 Crashes	25.0
Group of Files Reporting 6 Crashes	30.0
Group of Files Reporting 7 Crashes	26.9
Group of Files Reporting 8 Crashes	30.7
Group of Files Reporting 9 Crashes	28.5
Group of Files Reporting 10 Crashes	30.6
Group of Files Reporting more than 10 Crashes	47.0

the ANOVA did not show a significant relationship. The F-stat result was 9204.289 and a p-value of 0. Neither did ANOVA testing show a statistically significant relationship between the zero crash group and the combination of groups with eleven or more reported crashes. The F-stat of 5336.767 with a p-value of 0.

As mentioned, the Tukey test on crash reporting from groups with between zero and ten reported crashes shows most groups are independent. There was some dependence shown for groups reporting six and eight crashes, six and nine crashes, and six and ten crashes. There was some dependence shown between the seven and nine crash reporting groups. There was some dependence shown in groups reporting eight and ten crashes as well as reporting nine and crashes.

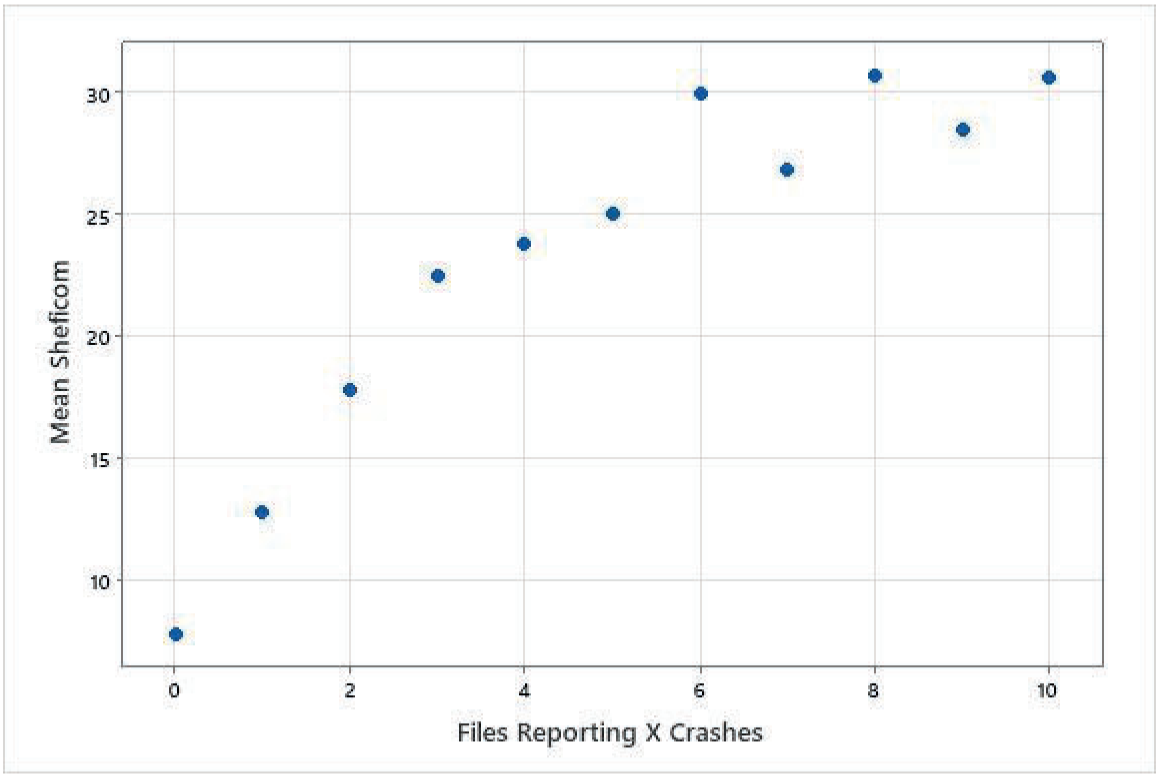


Figure 3.1: Sheficom vs. Crash Reporting Files

3.4 Discussion

The scatter plot shown in Figure 3.1 shows the association of files reporting crashes to their mean Sheficom value. It was assumed files reporting more crashes would be more complicated, and their Sheficom scores would increase as the number of reported crashes increased, and this appears to be the case. Interestingly, the relationship between reported crashes and the Sheficom measurement is satisfactory until the groups are no longer statistically independent. Because statistical independence did not exist in files reporting a considerable number of crashes, it was decided to present an aggregate total and represent these files as a group (see Table 3.1).

The textbook source code was expected to have small Sheficom values, as this code is written to be easily built for student understanding. As shown in Table 3.1, this is generally the case as two out of three textbook-based projects have Sheficom values of less than six. However, one result is noteworthy. The XINU project has a very high Sheficom score. Upon further exploration, it was determined this was due to a “cheater” file, “xinu.h.” For developers’ convenience, XINU has a header that includes all the XINU-related “include” files. Cheater files may be convenient, but they are also seen as a poor programming practice, as cheater-files can create unnecessary coupling. Suppose someone wants to use a file or utility from XINU in another application. In that case, extra effort is required to decouple and sort out the needed headers.

3.5 Conclusion

Sheficom has demonstrated its utility as a quick and straightforward estimate of efferent coupling. Sheficom follows a growth trend as source code complicacy grows. In addition, though “cheaters” can drive up the Sheficom result, Sheficom identifies the practice. Even if action is not taken to address the coupling issue, developers are still aware of it. Sheficom is a small script that is easy to automate and can measure many projects. It is also generalizable and could easily be modified to other programming languages. Since crash reporting files appear to associate satisfactorily with the reported coupling score, measuring with Sheficom allows authors to identify parts of their work that share characteristics with crashing code, giving them time to make improvements before deployment. Sheficom is a positive contribution to the practice of estimating coupling.

Chapter 4

Measuring Complicacy With Static Analysis*

4.1 Introduction and Background

The next part of the dissertation measures how beneficial static analysis may be in understanding software complicacy. It is widely believed that static analysis is a powerful tool to find defects in programs [56]. As Nathaniel Ayewah et al. describe, static analysis is used to examine source code without input data and without actually running the program [13].

4.1.1 Existing Literature

A considerable amount of research has already been performed on static analysis. Brian Chess et al. write about using static analysis to create secure programs [34]. As previously mentioned, searching for bugs is an everyday use of static analysis as these tools are even built into compilers [14]. Al Bessey et al., in their paper "A few billion lines of code later: using static analysis to find bugs in the real world" describe how one organization has actually made a business static analysis and bug finding [20].

*The text in this section is substantially taken from Conference Publication #4, "Eliminating software caused mission failures."

Table 4.1: Coding Issues Impacting Source Code Desirability

Coding Issue Description
There should be space around operators
Do not write over 120 columns per line
Have short functions
Indent blocks inside of a function
Put matching braces in the same column
Use less than five parameters for a function
Do not use the question keyword
Avoid deeply nested blocks
Use braces even for one statement

4.1.2 Research Objective

Although static analysis has been explored from many perspectives, a novel aspect inviting further investigation relates to using static analysis to identify source code confusing for people to read. In this chapter, I explore this avenue. In this chapter, I explore this avenue.

The primary research was conducted by surveying software developers with the objective of classifying the characteristics of complicated code and measuring the impact complicated code has on software development. In this process, two surveys were performed at different times. In conjunction with the survey analysis, a code review was undertaken using the information found in the surveys to associate the knowledge learned in the survey with the actual technical debt caused by complicated code.

4.1.3 Programmer Surveys

Source File Survey - This was the first survey I performed, and it was done with the intention of gaining an understanding of what source code traits software developers find undesirable. These traits can easily be measured using static analysis tools, such as N'SIQ CppStyle [198]. Further information on the source files used and raw data collected can be found in Appendix B. In this survey, programmers were shown a source file and asked to immediately and intuitively identify whether they found it unpleasant to review. The objective of this survey was to determine the significant characteristics of source code developers feel are complicated. Based on the assumption humans consider complicated code unpleasant to review, it may also be assumed code that is unpleasant to review is likely complicated. This survey was conducted in early 2018, and detailed results were published in the XP Conference Companion [48]. Another way of looking at it is software is complicated if the person looking at that code thinks it is complicated. Programming style is a significant contributor to complicated code and technical debt.

More than 400 participants completed the survey, reviewing 10 randomly selected C and C++ source files. Volunteers provided the source files or the files were found in open source repositories. When the survey was completed, the source files were analyzed using N'SIQ CppStyle to determine the characteristics of the programs which were considered unpleasant to review [198]. The most prominent characteristics of source code considered unpleasant to review are displayed in Table 4.1.

Statements Survey - In the second survey, volunteers where shown complicated code, as well as the same functionality, implemented using an easier to understand approach. The volunteers were asked to evaluate the code and describe the anticipated output if the code were run. This survey was intended to reflect how difficult it is to review complicated code properly and also to identify a complicated code penalty. An example of one of the questions with its code listing is shown in Listing 4.1. Put another way; the second survey attempted to determine if there is an actual cost associated with the characteristics of complicated software. The survey measured how long statements took to review as well as the accuracy of the volunteer's evaluation.

Listing 4.1: Sample Survey Question and Code Listing

```
What is the numerical output of the following code?

#define MAX_USB_IO_CTL_SIZE 61
#define CONFIG_USB_VENDODR_1 1

void myFunction(int temp) {
#ifdef CONFIG_USB_VENDOR
unsigned char  *tmp_buf;
unsigned char buf_index=4;
#else // use stack memory
unsigned char tmp_buf[MAX_USB_IO_CTL_SIZE];
unsigned char buf_index=3;
#endif
#ifdef CONFIG_CONCURRENT_MODE
if(padapter->adapter_type > PRIMARY_ADAPTER)
{
  padapter = padapter->pbuddy_adapter;
  pdvobjpriv = adapter_to_dvobj(padapter);
  udev = pdvobjpriv->pusbdev;
  buf_index = 6;
}
#endif
  printf(”buf_index␣%d␣\n”, buf_index);
}

int main()
{
 myFunction(1);
}
```

In total 140 people participated in the code statements survey, with 75 participants completing the entire survey. The survey allowed volunteers to evaluate as much code as they desired, with the complete survey lasting 10 questions. As mentioned previously, this survey displayed both complicated and uncomplicated constructs, and volunteers were asked to evaluate the results of the code snippet. The correctness of the volunteer's code evaluation, as well as the time taken to do the evaluation, were recorded. The results of this survey are shown in Table 4.2.

Table 4.2: Program Statements Survey Results

Undesirable Statement Question Topic	**Number of Reviewers**	**Average Seconds Taken by Reviewer**	**Evaluation Success Rate**
Embedded Comment	85	52.45	10.59%
Badly Indented ‘if’	86	27.01	34.88%
Yes space around operators	86	73.43	45.35%
No space around operators	79	62.58	51.90%
#ifdef block	82	66.91	59.76%
Max 120 column lines	83	75.0	60.24%
Yes ternary operator	86	38.08	94.19%
no #ifdef block	88	31.93	95.45%
Properly Indented ‘if’	89	28.98	95.51%
No Indentation ‘if’	89	33.94	96.63%
K and R (Java Style) Braces	87	28.06	97.70%
No ternary operator	84	24.01	98.10%

4.1.4 Evaluation Considering Open Source Projects

Finally, a cursory evaluation of actual projects was performed to see if the results of the surveys have merit. Source code from three relatively popular open source repositories was analyzed. The three projects were randomly selected from published Bugzilla supported projects [30]. The three projects selected were GCC [172], LyX [173], and Sudo [174]. All three projects have public bug tracking spanning many years, and all three projects are in active use. the program N’SIQ CppStyle [198] was configured to only scan for non-conformance with those rules listed in Table 4.1. Each project had a collection of milestones selected for evaluation and was then scanned. As each project is different in size and scope, development effort was estimated based on individual project characteristics.

GCC Results Code Review- GCC is the largest of the three evaluated projects, with more than four million lines of code and many participating developers. The effort was measured based on the number of comments per reported bug and the bugs themselves. Four releases of GCC were scanned and conformance measurements were taken. A Bugzilla report was also generated showing the number of bugs and how many replies each bug had. In the analysis, it was assumed more replies meant more effort spent on that particular release. Results of the evaluation of GCC are shown in Table 4.3. It is important to note that 8.x is still an active release. At the time of this writing, the current release is 8.2.

Table 4.3: Gnu C (GCC) Evaluation of Coding Style Conformance

Major Release	**Bugzilla Entries**	**Bugs Per Release**	**Conformance to Style Rules**
5.x	5339	1042	58.50%
6.x	5994	1206	57.27%
7.x	7822	1510	53.52%
8.x	6149	1418	53.50%

LyX Results Code Review- LyX is a favorite Latex formatting tool. Though we found LyX through Bugzilla, the LyX project ironically did not use Bugzilla for bug tracking. According to the LyX website, the LyX project was started in 1995 with the most recent release dated September, 2018. LyX has more than 400 thousand lines of C, C++, and header file source code. For LyX, the number of reported critical bugs on their external bug tracking was used as an indication of effort. The results are shown in Table 4.4. A linear relationship is shown in Figure 4.1.

Sudo Code Review- Sudo is a popular tool on Linux and claims it: "allows a system administrator to delegate authority to give certain users (or groups of users) the ability to run some (or all) commands as root or another user while providing an audit trail of the commands and their arguments [174]." Sudo is the smallest project reviewed with about 100 thousand lines of code, and it has been in active use and development since 1994. Since the number of errors reported was relatively small compared to the other projects reviewed,

Table 4.4: LyX Evaluation of Coding Style Conformance

Release	Critical Bugs	Coding Style Conformance
1.1	3	70.2%
1.2	49	78.04%
1.3	50	68.65%
1.4	118	66.62%
1.5	209	50.03%
1.6	263	42.11%

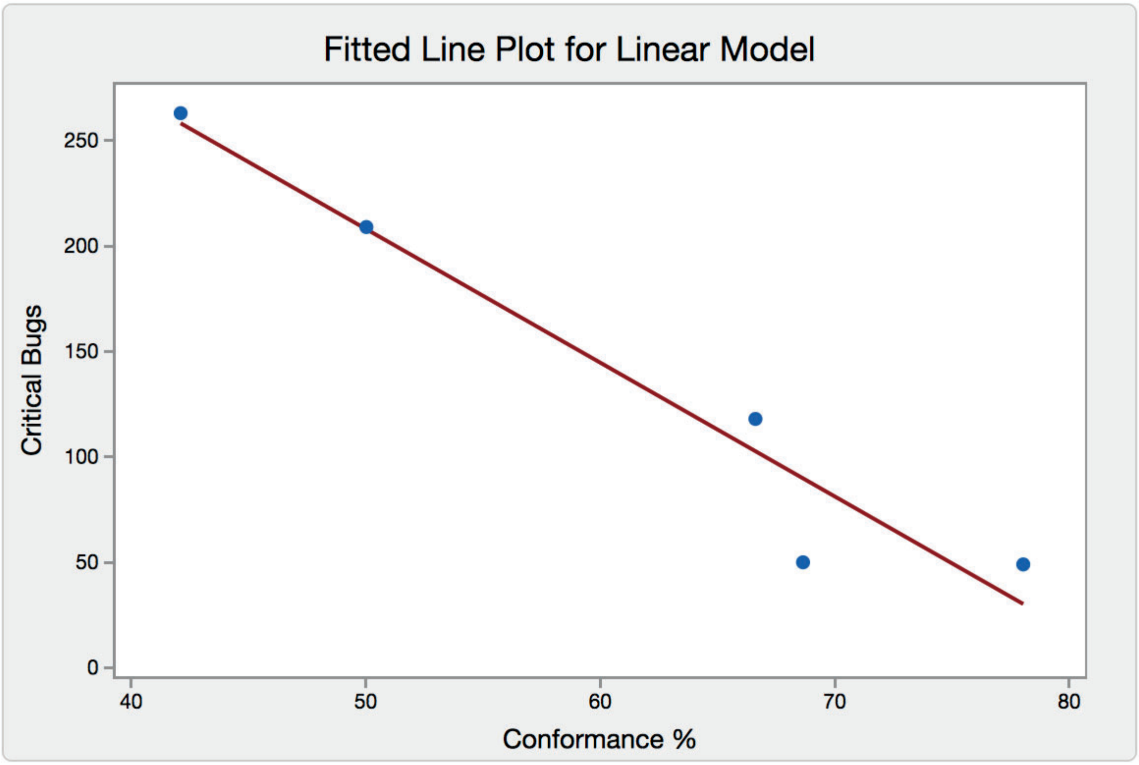


Figure 4.1: LyX Critical Bugs Versus Coding Style Conformance

this investigation made for a simple comparison of the number of reported bugs from their external bug-tracking as a measure of effort. The results are shown in Table 4.5. Note Sudo is in active production and the current stable release, at the time of writing, is Sudo 1.8.25p1 [174].

Table 4.5: Sudo Utility Evaluation of Coding Style Conformance

Release Range	**Reported Bugs**	**Years**	**Average Coding Style Conformance**
1.6.3-1.7.6	21	2000-2012	50.68%
1.8.0-1.8.25	44	2011-2018	23.08%

4.1.5 Discussion

Complicated lines of code were expected to take longer to review, and the Software Statements Survey results shown in Table 4.2 confirm this. In Table 4.6, we can see that complicated statements took about 40 percent more time on average to review than uncomplicated statements. A dramatic and somewhat unexpected result was how often the result of the complicated code review was incorrect. While we may consider a successful evaluation to have occurred when results are higher than 90 percent, on average, less than 50 percent of the complicated code was evaluated correctly. Stephen Schach suggests code reviews are the least expensive way to find faults in software [151]. One can conclude at the very least that the benefit of code reviews is lost or not taken advantage of to its maximum potential when complicated constructs are used. Considering the low success rate of evaluations of complicated code, one can imagine faults slipping into final applications. We also observe that lousy indentation is far more problematic than no indentation at all. Fortunately, indentation is something development tools can manage. Space around operators did not seem to help evaluations of complicated formulas as they were poorly done with and without spaces. The ternary operator success rate was not very different from alternative coding in the successful evaluation. However, it seems it takes more time to digest a ternary statement than code built with “if” statements.

Table 4.6: Time Required for Successful Code Snippet Evaluations

Success Rate	Question Count	Average Time Required
Correct >90%	6*	37.15 seconds
Correct <90%	6*	59.57 seconds
*Six questions were answered correctly more than 90% of the time. *Six questions were answered incorrectly more than 90% of the time.		

In the GCC code review, it was interesting to see that GCC maintainers must be enforcing a coding standard. The conformance of GCC concerning items unpleasant to review only marginally varied over many years. Further examination of GCC found two popular issues of conformance were braces for even one statement and space around operators, which some may argue are not significant issues. Looking at the GCC results, we see the number of bugs per release is relatively stable. Interestingly, when the conformance dropped marginally, the number of bugs and amount of effort changed as well.

A review of the LyX code also showed interesting results. There was no single moment where conformance changed suddenly, but a gradual change of code conformance occurred as shown in Table 4.4. The obvious outlier in the table is due to release 1.1 of LyX having just moved from Bugzilla to Trac, so it is possible not all of the release 1.1 bugs were captured by Trac [138]. The main takeaway is that as conformance dropped, the number of reported critical bugs increased linearly. See Figure 4.1.

The final project reviewed was Sudo. As mentioned, Sudo is a smaller project than both GCC and LyX, with a smaller number of bugs reported on their bug reporting website. Sudo was not analyzed in great detail, but what made their results interesting is how starting with version 1.8, conformance fell from 50 percent to 23 percent, basically falling by half, but at the same time, the number of reported bugs increased two-fold.

4.1.6 Conclusion

The results of these surveys and code reviews show a promising connection between complicated software and software faults and development efforts. Creating software is generally an expensive endeavor. Software failure not only causes financial losses but can also have negative effects on people's lives. Understanding what makes software complicated is an essential part of avoiding faulty, complicated software. The results of the conducted survey indicate software considered unpleasant to review is likely overly complicated. It is possible for an organization to employ readily available tools to measure the reviewability of their software, adjust, and improve their software process.

These findings imply that the probability of success of software systems improves with the reduction of complicacy. This is supported by the idea that a better understanding of the code will permit better quality tests to be created. Better quality tests will allow for better testing and debugging. Knowing what programming features make a complicated system helps create practical programming guidelines for programmers to follow. These guidelines are further discussed in Chapter 9. This information will help programmers create less complicated source code, resulting in higher success rates for deployed software systems.

Part III

Software Applied Linguistics

Chapter 5

Software Applied Linguistics[*]

5.1 Introduction

Traditional software metrics were created based on a concept of software development as analogous to a mathematical practice or an engineering discipline. Consider that a pioneer of software metrics, Maurice Halstead, coined the term “software physics” in his efforts to establish an empirical science of software measurement[80]. Static analysis and metrics do a good job measuring source code mathematically, but do not incorporate human behavior in the calculations. This chapter views software development as more analogous to an artistic practice, such as book authorship or music composition, rather than an empirical science that is easily measured.

5.2 Existing Literature

Previous investigations have been performed considering the human aspects of complicated projects. For example, Dan Sturtevant does excellent work in his thesis for MIT describing the impact of software architecture design, connecting it to employee productivity and staff

[*] The text in this section is substantially taken from conference presentation number 6. “Applying the Linguistic Economy Principle to Programming Languages.”

turnover. Sturtevant worked with a professional organization, and his work includes a case study of a development project showing that software structure impacts productivity and employee retention. Sturtevant suggests further research on different types of projects is warranted and that there are many opportunities for this [168].

Human considerations were also explored in a 2015 paper from Matthieu Foucault et al. researching the impact developer turnover has on software quality. This paper covers open-source software and demonstrates a link between developer turnover and the number of bugs found in the project [65]. Interestingly, Foucault and Sturtevant do not entirely agree on whether poorly designed software drives developers away, or if the turnover creates the poorly designed software. Nachiappan Nagappan et al. have suggested turnover of developers might also be a reason for code churn (a measure of how much code has changed) as described in their paper "Use of Relative Code Churn Measurements to Predict System Defect Density" [119]. This is shown as an indicator of faults, and it is suggested that more recent code is more faulty than original code [119].

This chapter employs the work of George Zipf, particularly Zipf's Law of Vocabulary balance. This "law" predicts an orderly distribution between word size and word usage in spoken languages. Zipf described how words used in speech are subject to two opposing forces. One force from the speaker endeavors to economize efforts by reducing their vocabulary to the smallest vocabulary necessary to convey all meanings. Opposing this force is the need for precise communication, which encourages an ever-increasing vocabulary until there is a word for every meaning [206]. The result is humans try to use the least complicated words and phrases required to communicate. The work "Économie des changements phonétiques" by André Martinet supports Zipf's ideas by describing these opposing forces of communication needs and the natural human tendency of wanting stability, as linguistic economy. [107].

Before continuing, reviewing existing literature regarding Zipf's work is essential. Although the work of Yu et al. is supportive of Zipf and shows the applicability of Zipf's work to fifty different languages [199], it would seem there is no universal agreement on the validity of Zipf's ideas. Duarte describes how Zipf's concepts have been studied and reviewed by many scholars, and their conclusions are not all supportive [129]. Even in 1965, George Miller wrote in the introduction to the reprint of Zipf's "The Psycho-Biology of Language: An Introduction to Dynamic Philology" a collection of monkeys with typewriters would get the same word size and frequency distribution as indicated by the Law of Vocabulary Balance [205]. Duarte seemed to focus on the concept of Linguistic Economy as an application of laziness [129]. However, a close reading of Zipf and Martinet shows the focus is not on laziness but instead on efficiency. As development and operational efficiency are important in software, Linguistic Economy should not be discounted from use in software development. It must also be remembered even Zipf described this phenomenon as an observational study. With this caveat in mind, I believe it is not unreasonable to recognize Linguistic Economy patterns in humans' software development work.

5.3 Background

Zipf discovered vocabulary balance by analyzing word usage from James Joyce's novel *Ulysses*, where he noticed word popularity (i.e., its rank (r)) multiplied by how frequently the word is used (f) yields a constant (C) (Equation 5.1) [206].

(For example, the tenth-ranked word in *Ulysses* was used 2,653 times in the book, giving a constant of 26,530. The 20th ranked word multiplied by its frequency of use of 1,311 gives a constant of 26,220. Notice the resulting constants from both examples are very close in value.)

$$r \times f \approx C \tag{5.1}$$

Using algebra, the equation can be modified as follows in Equation 5.2:

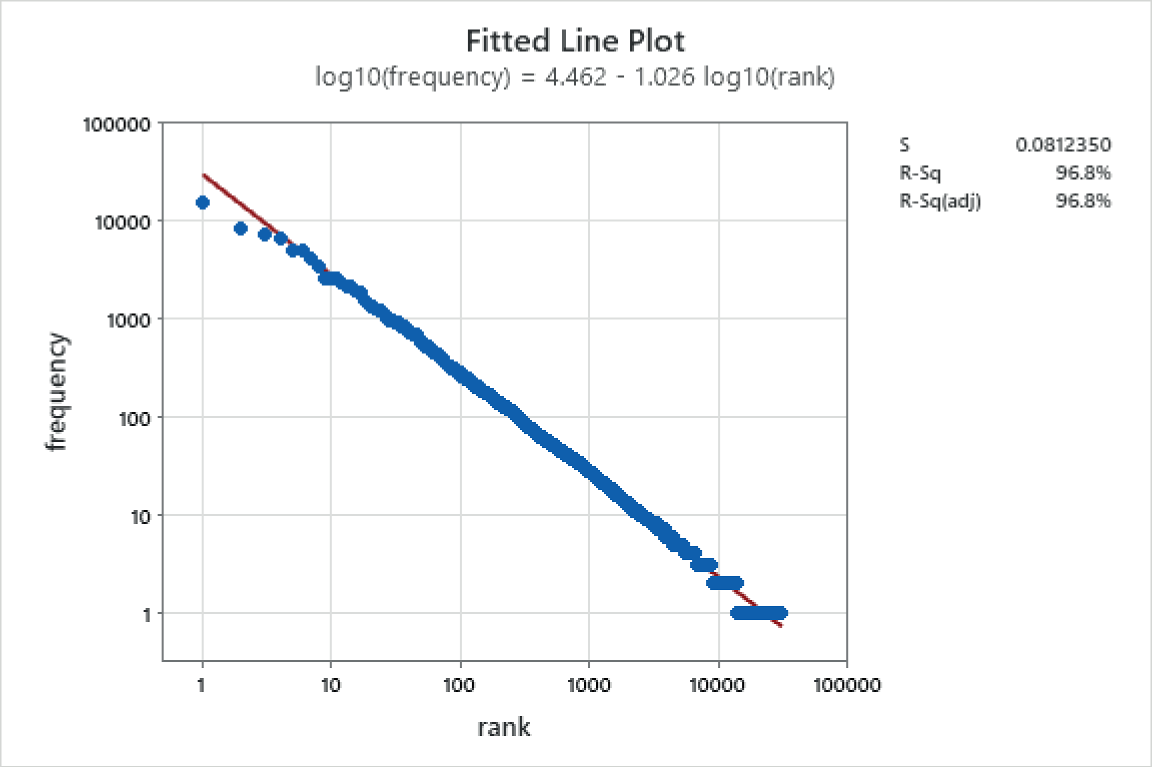


Figure 5.1: Rank Frequency Distribution of Words

$$f \approx \frac{C}{r} \tag{5.2}$$

When Zipf plotted the rank of words from *Ulysses* against the frequency of use on a double logarithmic scale, the result was a line with a slope of negative one. Zipf's graph is shown in Figure 5.1. This characteristic of the line is easily demonstrated mathematically. Since Zipf's plots are logarithmic, logarithms of both axes are taken, and the equation is adjusted to show the following:

$$log(f) \approx log(C) - log(r) \tag{5.3}$$

Equation 5.3 mathematically presents Zipf's equation when plotted on a double logarithmic scale, with a slope of negative one. For the purpose of consistent terminology in this paper, going forward, ***log(f)*** will be referred to as **Zipf Frequency**. That is to say; in this dissertation, the “Zipf Frequency” is the logarithm of how frequently the word is used.

Zipf continued his work by describing how humans, in all actions, select tasks to perform based on efficiency and energy required to complete the task [206]. Zipf believed the Principle of Least Effort is fundamental to human behavior. Zipf illustrated this principle by describing artisans, who must survive by performing work and using the most efficient tools. As a result, we can expect artisans will arrange their tools such that the tool furthest from the artisan would be the tool requiring the most effort to use. Since efficient tools are used most often, they must also be the best understood.

5.4 Research Objective

I believe although Zipf’s work has been applied in the area of software, Linguistic Economy has not been sufficiently leveraged to measure program readability and reliability. Therefore I investigate how **Zipf Frequencies** can be applied to measure the complicacy of statements and expressions found in program source code.

Applied linguistics is described as a field that investigates language-related, real-life problems. According to Peniro and Cyntax, fields within applied linguistics include education, psychology, communication research, anthropology, and sociology [130]. This dissertation explores software implementation using computer languages as a new area of applied linguistics.

5.5 Linguistics Study of Statements and Expressions

In this study, I demonstrate that the most commonly used programming statements and expressions are the most efficient for making understandable source code for programmers, and this source code is both easier to read and more reliable in operation. Statements and expressions are used to control assignments and also express logical decisions within software. As previously mentioned, this evaluation of statements and expressions in software is based on work by George Zipf, including Zipf's law of vocabulary balance and the principle of linguistic economy.

5.6 Methods and Data

5.6.1 Step 1 - Creation of the Statement and Expressions Corpus

In creating the statements and expression corpus database, the same steps performed by George Zipf with respect to Linguistic Economy were followed [206]. As previously mentioned, Zipf analyzed the vocabulary of James Joyce's novel "Ulysses" and Zipf plotted the rank of words from "Ulysses" against the frequency of use on a double logarithmic scale, showing a line with a slope of minus one.

The software corpus was created using a project base made from the large collection of projects gathered in Chapter 2. These projects collectively contain millions of lines of C and C++ source code. Next, the tool Tokenizer, written by Diomidis Spinellis, was used to extract the statements and expressions from each project file [162]. In this process, the statements and expressions were sanitized such that more generic names replaced unique names of identifiers. For example, the expression **'i = i + 1'** would be changed to **'ID = ID + NUM'**. Statements and expressions from all the files in the downloaded projects were counted.

5.6.2 Step 2 - Organize Projects by Bugginess

A second descriptive attribute usable with source code or projects refers to how buggy it might be. As mentioned in Chapter 2 of this thesis, accurate data for project bugginess is difficult to acquire and perhaps even subjective. This chapter uses the same methods as described in Chapter 2 for classifying bugs.

It has been suggested that all software projects have between 15 and 50 BKLOC, regardless of the programming language utilized, and this chapter considers projects that fall into this range to be normal [111]. Going forward, I will refer to this range as **McConnell-Normal**.

Two methods were used for organizing the project base for bugs. In the first method, projects were grouped based on different ranges of BKOLC. In the second method, projects were divided into groups based on their size and bugginess. Projects were considered to be non-buggy if they have less than 15 BKLOC, and projects were considered buggy if they had more than 15 BKLOC.

5.6.3 Step 3 - Gather Pleasant and Unpleasant to Review Files

As discussed earlier in this thesis, for the 19th International Conference on Agile Software Development, a survey was performed to identify parts of source code that programmers find unpleasant [49]. For this survey, various C++ source files were put online for evaluation. Participants were asked if they felt a particular source file would be pleasant or unpleasant to review.

Description	Count
Leaning Pleasant to Review Files	20
Leaning Unpleasant to Review Files	34
Unused Files (outliers)	36
Total Available Files	90

Table 5.1: Composition of Data From Pleasant to Review Survey

For this linguistics study, no attempt was made to balance lines of code or the number of included files. Files considered outliers were discarded, such as the truly loved source files and the truly hated source files. For example, one truly loved file only had about 20 lines of actual source code. The selected files were based on a ratio of pleasant to review versus unpleasant to review falling into the range of 0.5 to 1.5, with neutral being 1.0. For example, if five reviewers found a file pleasant to review and four reviewers found a file unpleasant to review, the ratio would be 1.25. Files with ratios from 1.0 to 1.5 were considered to have the descriptive attribute of leaning positive or pleasant to review. Files with a ratio of 0.5 to 1.0 were considered to have the descriptive attribute of leaning negative or unpleasant to review. Table 5.1 breaks down the final tally of files leaning positive and leaning negative to review. After selecting the files, they were processed in the same manner as the source code used to build the corpus.

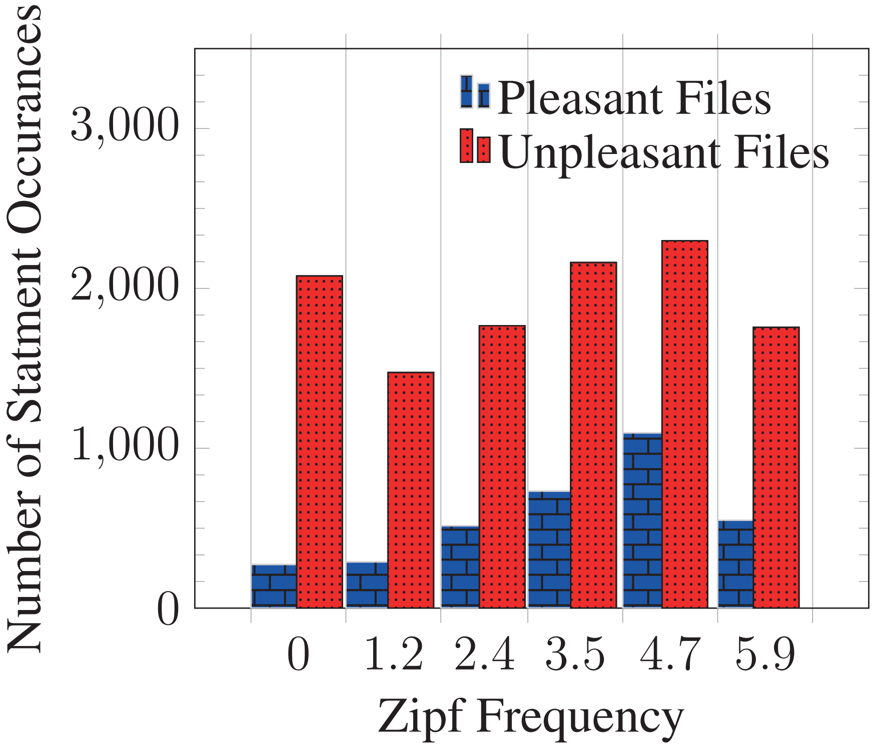


Figure 5.2: Zipf Frequency Histogram of Pleasant and Unpleasant Files

5.7 Results

5.7.1 Corpus Creation Results

Figure 5.3 shows the plotted results and demonstrates results very similar to those shown by the novel "Ulysses" [206]. Regression analysis of the data in Figure 5.3 produces Equation 5.4, which is an excellent match to Zipf's equation, previously shown in Equation 5.1.

When reading Figure 5.3, a higher value for frequency indicates the item (statement

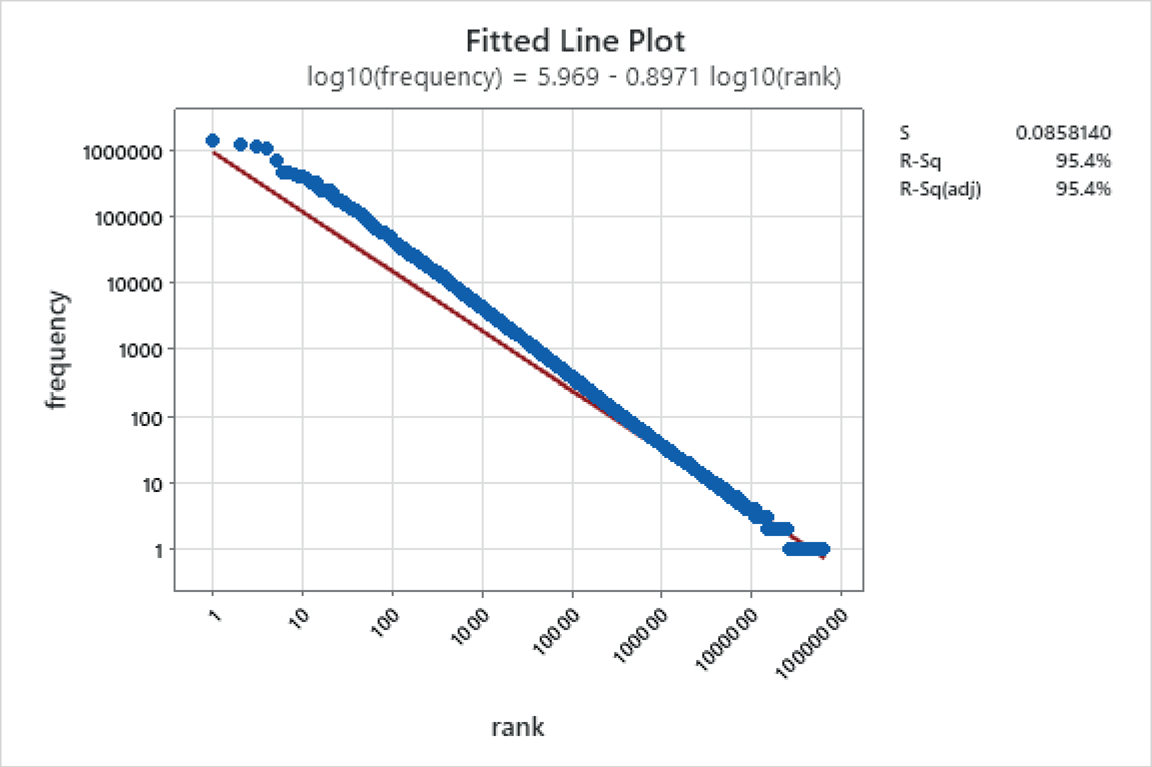


Figure 5.3: The Rank Frequency Distribution of Statements and Expressions

or expression) was used more often and thus is easier for a person to understand. As a reminder, going forward, the log base 10 of this value will be referred to as the **Zipf Frequency**. As is typical with ranking systems, a lower numerical value indicates a higher ranking. For example, a team ranked in first place is generally considered better than a team ranked in seventh place. Thus, a higher numerical value for rank consequently means a lower frequency of use, which is considered more complicated to understand.

$$log(f) \approx 5.969 - 0.8971 \times log(r) \tag{5.4}$$

Table 5.2 demonstrates the linguistic economy principle connected to software by showing the most popular items from the statements and expressions corpus. The “else” statement happens to be the most popular. Listing 5.1 shows one unpopular statement. Out of more than five million statements and expressions, this statement was only used once. Large statements and large expressions and statements and expressions with many opera-

Statement	Occurrences in Corpus
else	2,140,212
return_ID	1,544,827
ID_ID	1,518,153
ID_(_ID_)	1,489,444

Table 5.2: Most Popular Programming Statements From Examined Projects

tors are used less frequently. This is also analogous to the results found with Zipf's work on Ulysses. For example, In "Ulysses," words like 'chryselephantine' and 'systematization' were only used once; however words such as 'the' and 'of' were used thousands of times. This is analogous to how an artisan will use easier, more efficient tools first if possible.

Listing 5.1: A Single Rarely Used Programming Statement

```
auto_ID_=_ID_:_:_ID_(_)_._ID_(_ID_)_.
_ID_(_NUM_comma_STRING_LITERAL_)
_._ID_(_NUM_comma_NUM_comma
_NUM_comma_NUM_comma_false_)_._ID_
(_STRING_LITERAL_comma_STRING_LITERAL_)_.
_ID_(_)_._ID_(_)_._ID_(_ID_)
```

5.7.2 Bugginess Results

Bugs Per KLOC	Average Zipf Frequency	Number of Projects
$1 - 15$	4.0	457
$15 - 50$	3.99	181
$50 - 70$	3.84	15
$70 - 90$	3.59	3
$90 - 240$	3.35	4

Table 5.3: Zipf Frequency Relation to Bugs per KLOC

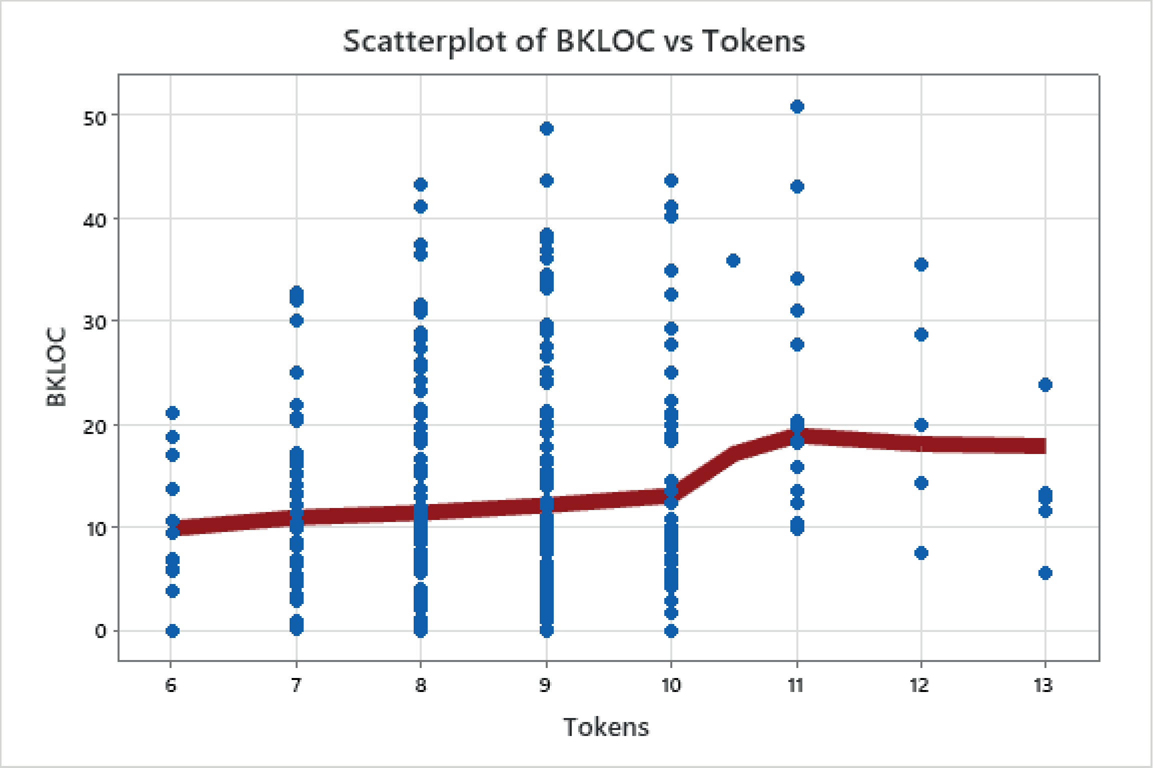


Figure 5.4: Median Token Count vs. Bugs Per KLOC (BKLOC)

As shown in Table 5.3, as the BKLOC increases, the average Zipf frequency decreases. Also shown in Table 5.3, projects below and within the McConnell-Normal range have an average Zipf Frequency of about 4.0. However, as the projects begin to exceed the McConnell-Normal range, the average Zipf Frequency declines, indicating that more challenging to understand statements and expressions are included.

Zipf's law of vocabulary balance and the principle of linguistic economy suggest that humans prefer to speak using less complicated words, and less complicated words are shorter in character count. Applying this concept to software, humans will prefer to use shorter and less complicated statements and expressions. A compiler considers a token to be fundamental and can not be broken down further[115]. By counting the tokens of statements and expressions, it is possible to measure their length and estimate complicacy. Tokenizer was again used and the number of tokens in each statement and expression was counted [161]. Next, the median lengths of statements and expressions were calculated for each project and analyzed with BKLOC. Locally weighted regression scatterplot smoothing (LOWESS) shows the trend that as the median length of statements/expression grows, so does the BKLOC. This is shown in Figure 5.4.

5.7.3 Pleasantness of Review Results

When considering the pleasantness of any topic, negativity bias should be kept in mind. Paul Rozin et al., in their paper on negativity bias, describe how quickly a meal is ruined by a short visit from a cockroach [150]. Negative influences seem to have greater importance than positive influences in many situations. With this in mind, it should be expected that projects that are unpleasant to review contain more statements and expressions of lower rank, and this seems to be the case, as shown in Figure 5.2. Otherwise, both the pleasant to review and unpleasant to review files, follow a very similar pattern. It can thus be said that the number of unique, and thus low-ranked, expressions influence the desirability of review. It is also worth noting that the median Zipf Frequency of projects leaning pleasant to review was 4.58, and the median Zipf Frequency of projects leaning unpleasant to review was 3.79. Remember, a higher Zipf Frequency reflects more understandable statements and expressions.

5.8 Threats to Validity

Because of the Agile conference paper's goals, the pleasant versus unpleasant to review survey was structured in such a way that it did not have the same reviewers for each reviewed file. Also, as the reviewers were allowed to select the number of files they wished to review, not all the files had the same number of reviews. Though this does introduce a degree of randomness, it is possible if all the reviewers reviewed the same files, the ratio of unpleasant to pleasant may be different. Nevertheless, more than 400 people participated in the survey, which makes it reasonable to believe sufficient insight is provided. Concerning “bugginess,” projects with enormous BKLOC values are rarer, and as such, the number of projects in our collection badly above McConnell-Normal [111] was relatively small. Regardless of the quantity, the included enormously buggy projects do provide interesting data.

5.9 Conclusion

The results shown in this chapter support the measurement of statement and expression complicacy using concepts from Zipf’s Law of vocabulary balance and the principle of linguistic economy. Unpleasant to review files were shown to have a greater number of low-ranking statements, indicating that Zipf’s methods can apply to code complicacy. The results also show that projects with statements and expressions that are identified as less understandable are indeed less desirable to review and likely are buggier. Since statements and expressions are a core part of a program source, it is crucial to the quality of a project that they are readable and understandable. Although the study in this chapter used data gathered from C and C++ projects, the concepts shown here can also apply to new programming languages. Human artisans will always arrange their tools most efficiently. The most efficient constructs of new programming languages will undoubtedly be used most often. This study demonstrates that these linguistic concepts can be applied to program quality. This knowledge can be employed to make more readable, higher-quality programs.

Part IV

Machine Learning and Code “Tumor” Identification

Chapter 6

Image Recognition for Code Reviews*

6.1 Introduction and Background

Code reviews are policy in many software development organizations, and it is commonly believed code reviews are an economical way to discover faults before a software product is deployed. Indeed, it is even suggested that inadequately reviewed code has twice the faults of reviewed code [17]. However, many software engineers are overwhelmed with work, so proper code reviews are often not done. The reviewability of software is affected by many factors such as documentation, logic, semantics, and syntax. Source code includes aspects that might even be considered aesthetic, and aesthetic aspects might turn tedious and possibly overwhelm the review process [95].

6.1.1 Existing Literature and Related Work

Code Reviews

As code reviews are an essential topic, many papers are written each year to address review process problems. In the paper, “Confusion Detection in Code Reviews,” Felipe Ebert et al.

*The text in this section is substantially taken from Conference Publication #10, “Using Machine Learning Image Recognition for Code Reviews.”

recognize code reviews do not always go smoothly and identify items causing confusion in the review process [54]. Fatima et al. discuss the good and bad consequences of feedback in the review process [59]. A common theme of these papers is problems of the code review process itself and how automation of at least some of the process can improve the overall quality of the review.

Machine Learning and Image Analysis

Machine learning and image recognition have been used with success in many areas. For example, Lin et al. describe the successful use of deep learning for laser positioning [104]. An even more applicable subject is image processing and sentiment analysis. Qian et al. analyzed Twitter messages, attempting to capture human expressiveness with image recognition [136]. In 2015, Zhang et al. described performing sentiment analysis on "microblogs" by integrating text and image features [202]. Although these papers are not software related, they demonstrate the success of machine learning in the context of image analysis and show the possibility of detecting text sentiment. In a paper by Mehrdad Yazdani et al., non-photographic images, such as screenshots and images of text messages, were analyzed and found useful in predicting social trends [196].

Machine Learning and Source Code

Concerning research related directly to software source code, Ron Coleman et al. in their paper "Aesthetics Versus Entropy in Source Code," found evaluating code beauty could be used for style checking [36]. Other studies have used machine learning and deep learning in code review systems to analyze code errors automatically. Bielki et al. introduced a machine learning-based system where the analyzer learned to produce static analysis tools using a decision tree algorithm [22]. The system showed a coverage improvement but mentioned scalability and generalizability could be improved. Anshul Gupta et al. created a system using a "long short-term memory" network called DeepCodeReviewer, which learned to review from human reviewers. They explain they plan to improve the DeepCodeReviewer tool to learn continuously and personalize itself to a team or a repository [78]. These papers demonstrate the applicability of machine learning to the code review process but do not address reviews using image processing.

6.1.2 Research Objective

This study aims to evaluate the possibility of using screenshots of source code with machine learning image recognition as part of the software code review process. Tools to reduce monotonous tasks related to reviews could be very valuable. This study begins by discussing the readability aspects of code and estimates the impact style has on reviews. Images were created of poorly styled code and properly styled code and machine learning was used to train an image recognizer to identify poorly formatted code and present positive results. Creating source code screenshots for analysis could be part of automating code reviews. Using automation as part of the review process could make software engineers more efficient.

6.2 Chapter Background

This chapter uses data initially gathered in preparation for the 2019 IEEE Aerospace Conference in Big Sky, Montana (Aeroconf). Although Chapter 4 covers this study in detail, a brief review is warranted as data from the Aeroconf study is used here. As a reminder, for Aeroconf, "code snippets" were created and shown to programmers [46]. Programmers were then asked to determine the proper outcome should the code be executed. The results demonstrate problematic code takes longer to review and is more often reviewed incorrectly. Most significantly applicable for this chapter, improperly formatted code had a review success of less than 90 percent and required nearly 60 seconds to review on average. Nicely formatted code only required about 37 seconds to review. Feedback received from

Aeroconf participants suggested many issues could be avoided simply by following coding standard rules. Most issues identified were stylistic, not logic-based. This suggests these issues may be spotted visually, analogous to a tumor in a medical CT scan. Going forward in this thesis, the term *tumor* refers to these problematic coding style issues. See snippets in Sections 6.2.1 and 6.2.2 for examples of code without a tumor and code with a tumor.

6.2.1 Example of Code Without a Tumor

```
function3()
{
  int a=1,b=1,c=3,d=0;
  if (a < b)
  {
    if (b > c)
    {
      printf("1\n");
    }
    else if (a > d)
    {
      printf("2\n");
    }
    else if (d > a)
    {
      printf("3\n");
    }
  }
  else
  {
      printf("4\n");
  }
      printf("\n");
}
```

`Note: Braces line up and braces even for one statement.`

6.2.2 Example of Code Containing a Tumor

```
int a=1,b=1,c=3,d=0;
if (a < b)
  if (b > c)
    printf("1\n");
  else if (a > d)
    printf("2\n");
  else if (d > a)
    printf("3\n");
else
  printf("4\n");
```

```
Note: No braces and if/else blocks do not even align
```

6.2.3 Problem Description and Impact

Since modern code editors can enforce properly formatted code, it was suprising to see how much existing code violates style rules. It seems even though modern tool kits are helpful, some issues of poorly formatted code linger. To demonstrate the ramifications of this problem, the projects first used in Chapter 2 were reviewed and scanned for common issues. As shown in Table 6.1, most projects had at least some software issues, and two projects had more than 15 percent of their lines associated with issues. Static analysis tools nsiqcppstyle [198] and lizard [197] were used to identify issues.

A hypothetical project illustrates the possible consequences of tumors in source code. The size of the hypothetical project was determined using projects in the collection with very few tumors, specifically the projects where 0 percent to 2 percent of their lines were associated with tumors, as shown in Table 6.1. Static analysis of this group showed the median number of lines of code was 61,649, and the median number of tumors was 499, with the probability of a line being part of a tumor being 0.8 percent. The hypothetical project was given characteristics based on these numbers and is shown in Table 6.2 with the results. Results from the Aeroconf study [46], as discussed in Chapter 4, were used to estimate how long code with and without tumors takes to review. In this hypothetical

Table 6.1: Software Tumor Density of Sample Projects

Percentage of Lines **Associated With Tumors**	**Number of Projects**
0% to 1%	180[a]
1% to 2%	181
2% to 3%	121
3% to 4%	51
4% to 5%	36
5% to 6%	23
6% to 9%	22
9% to 12%	15
12% to 15%	3
15% to 100%	2[b]

[a] Read as 180 projects have 0% to 1% of their source impacted by tumors.
[b] Read as two projects have more than 15% of their source impacted by tumors.

project, the presence of tumors increased the review time by 21 percent. It is also important to remember that not only does the presence of tumors increase the review time, but it also reduces the accuracy of the review [46].

Table 6.2: Specifications for a Hypothetical Software Project

Description	**Value**
Project Lines	61,649
Project Tumors	499
Lines in Each Code Segment	56
Number of Segments	1,100
No tumor Segment Review Time	37.5 Seconds
Segment Review Time with tumor	59.5 Seconds
No Tumors Review Time	**11.5 Hours**
Tumor Review Time	**14.5 Hours**
Increased Time	**21%**

6.2.4 Deep Learning Background

The Convolutional Neural Network (CNN) is a mathematical construct inspired by the organization of the animal visual cortex. A CNN is built in layers and generally has three layer types: convolution, pooling, and fully connected. Feature extract happens in the convolution and pooling layers. The fully connected layer performs classification. [195].

The training process provides information to the machine learning model from which

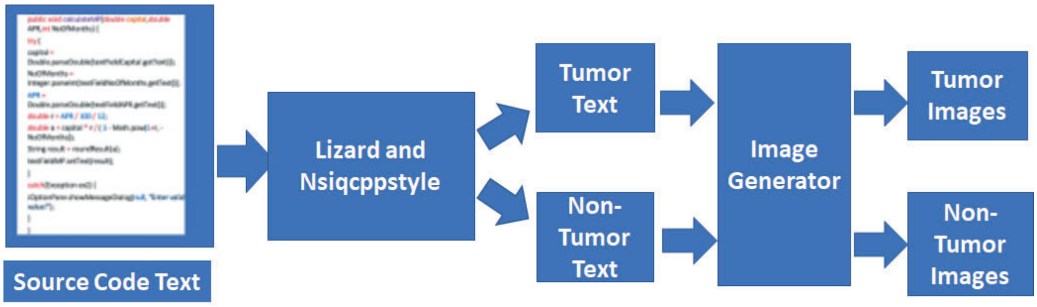
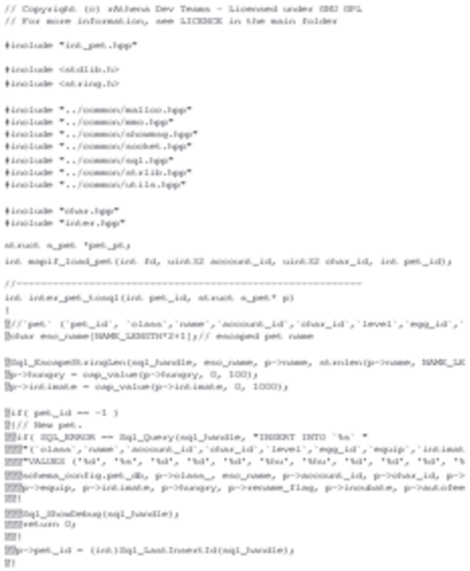


Figure 6.1: Preparation of Code Images for Machine Learning

it can learn [40]. It is also possible to take knowledge acquired from an already trained model and apply it to a new application through a concept called transfer learning[153]. For example, VGG-19 (Visual Geometry Group) and ResNet50 (Residual Networks) are pre-trained CNNs using the ImageNet dataset [42]. VGG-19 and ResNet50 alow developers to leverage transfer learning to solve image classification tasks [154] [3].



```
// Copyright (c) rAthena Dev Teams - Licensed under GNU GPL
// For more information, see LICENCE in the main folder

#include "int_pet.hpp"

#include <stdlib.h>
#include <string.h>

#include "../common/malloc.hpp"
#include "../common/mmo.hpp"
#include "../common/showmsg.hpp"
#include "../common/socket.hpp"
#include "../common/sql.hpp"
#include "../common/strlib.hpp"
#include "../common/utils.hpp"

#include "char.hpp"
#include "inter.hpp"

struct s_pet *pet_pt;
int mapif_load_pet(int fd, uint32 account_id, uint32 char_id, int pet_id);

//---------------------------------------------------------
int inter_pet_tosql(int pet_id, struct s_pet* p)
{
	//`pet` (`pet_id`, `class`,`name`,`account_id`,`char_id`,`level`,`egg_id`,`
	char esc_name[NAME_LENGTH*2+1];// escaped pet name

	Sql_EscapeStringLen(sql_handle, esc_name, p->name, strnlen(p->name, NAME_LE
	p->hungry = cap_value(p->hungry, 0, 100);
	p->intimate = cap_value(p->intimate, 0, 1000);

	if( pet_id == -1 )
	{// New pet.
		if( SQL_ERROR == Sql_Query(sql_handle, "INSERT INTO `%s` "
			"(`class`,`name`,`account_id`,`char_id`,`level`,`egg_id`,`equip`,`intimat
			"VALUES ('%d', '%s', '%d', '%d', '%d', '%hu', '%hu', '%d', '%d', '%d', '%
			schema_config.pet_db, p->class_, esc_name, p->account_id, p->char_id, p->
			p->equip, p->intimate, p->hungry, p->rename_flag, p->incubate, p->autofee
		{
			Sql_ShowDebug(sql_handle);
			return 0;
		}
		p->pet_id = (int)Sql_LastInsertId(sql_handle);
	}
```

Figure 6.2: Sample Image for Machine Learning

Note: This is an image meant for machine learning training. It is not meant for reading by human eyes.

6.3 Methods

6.3.1 Step 1 - Image Acquisition

As mentioned previously, the same open source projects used in Chapter 2 are used in this chapter. A complete list can be found in Appendix C. Static analysis was used to identify tumors and then image files were created from those results. Good code snippets (non-tumors) were likewise identified and separated, and image files for non-tumors were also produced. The process for image creation is shown in Figure 6.1.

In total, a set of about 44,000 images was prepared. This set was used for model training and testing, with about 38,000 images labeled as non-tumor and the remaining labeled as tumor (about 6,000). However, due to the imbalanced distribution of data, about 6,000 non-tumor images were randomly selected, along with the 6,000 tumor images. By doing this, the model potentially avoids the problem of over-fitting, which is a major factor leading to poor performance. The created images are monochromatic and 228 pixels by 280 pixels in size. An example is shown in Figure 6.2.

6.3.2 Step 2 - Model Definition

VGG-19 and ResNet50 are predefined and pre-trained using the ImageNet data set. No further definition is required, however, for informational purposes, VGG-19 is a classic CNN with 19 layers with trainable weights, 16 convolutional layers, and three fully connected layers [108]. ResNet50 has 50 layers [3]. The customized CNN in this project consists of three convolutional blocks, each constructed with multiple layers. The convolutional blocks are followed by two fully connected blocks, also built with multiple layers. Stochastic gradient descent (SGD) was selected as the optimizer, with a learning rate of 0.0001. An example of how to create a customized CNN is provided by Eijaz Allibhai, in his work “Building a Convolutional Neural Network (CNN) in Keras” [6].

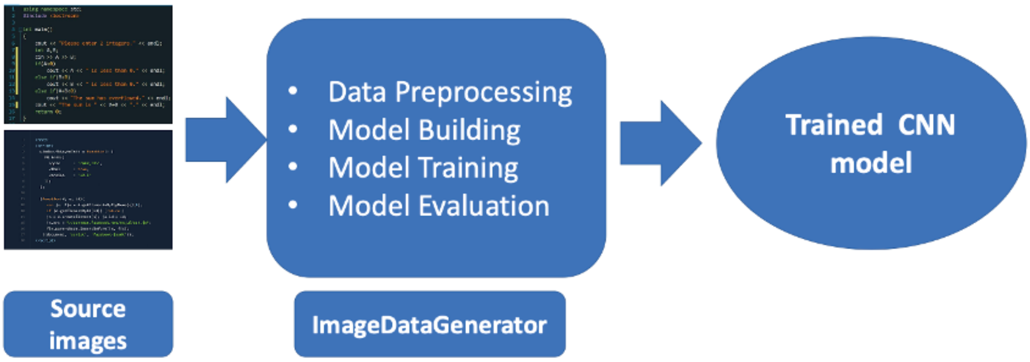


Figure 6.3: Pre-processing Image Data and Model Training

6.3.3 Step 3 - Training the Model

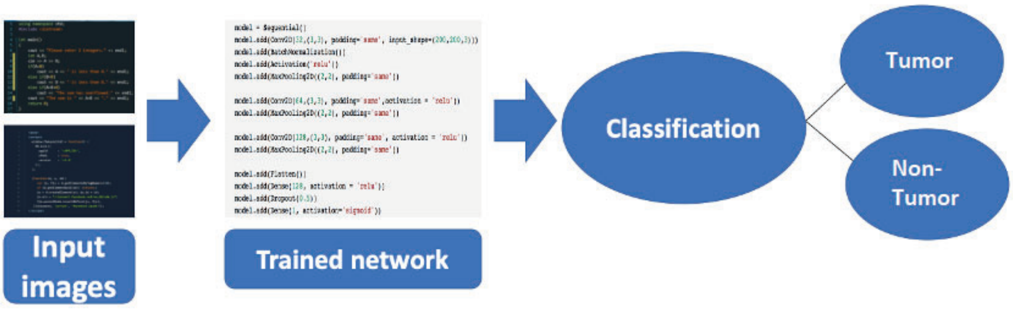
To train the models, a Keras framework [89][106] is utilized with a Tesla T4 GPU [122]. All the models are trained over 100 epochs, and data is broken up into batches of 64 images. The ImageDataGenerator utility [88] was employed to progressively manipulate and load data in batches and help monitor the training process. Figure 6.3 shows that ImageDataGenerator implements multiple stages in a single utility, from preprocessing code snippets to training and evaluating resulting models.

6.3.4 Step 4 - Classification

VGG-19, ResNet50, and the customized CNN were trained on the same data to have comparable accuracy rates. The operational flow of the process is shown in Figure 6.4.

6.4 Results and Discussion

Two metrics were selected to evaluate the performance of the different architectures: accuracy and F1-score. (F1-score is a measure of precision and recall.) The customized CNN architecture performed best, with 80 percent accuracy and a 0.79 F1-score. VGG-19 and ResNet50 did not perform as well on this task, having accuracy results of only 59 percent and 55 percent. Overall comparisons are shown in Table 6.3. The problem with VGG-



```
model = Sequential()
model.add(Conv2D(32,(3,3), padding='same', input_shape=(200,200,3)))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D((2,2), padding='same'))

model.add(Conv2D(64,(3,3), padding='same',activation = 'relu'))
model.add(MaxPooling2D((2,2), padding='same'))

model.add(Conv2D(128,(3,3), padding='same', activation = 'relu'))
model.add(MaxPooling2D((2,2), padding='same'))

model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
```

Figure 6.4: Model Evaluation Flow Diagram

19 and ResNet50 may be caused by the amount and nature of the data presented. These CNNs use transfer learning and the application of transfer learning attempts to use previous knowledge to solve a problem. In this case, it seems the tumor data does not align well with transfer learned data, leading to poor performance. In addition, the tumor training data may be insufficient to properly exercise the number of layers built into ResNet50 and VGG-19. The customized CNN has fewer layers and a less complicated architecture than both ResNet50 and VGG-19. Also, since this is a binary classification problem, it may be inappropriate for pre-trained models such as ResNet50 and VGG-19 since they were created for more sophisticated applications.

Table 6.3: Comparison of Machine Learning Model Accuracy Rates

Description	**Accuracy**	**F1-Score**
VGG19	59%	0.62
ResNet50	55%	0.61
Custom CNN	80%	0.79

Since there was a deliberate reduction in training data performed earlier, class distribution over the data is balanced, and the problem of overfitting was avoided. This is demonstrated by the confusion matrix shown in Table 6.4. The number of correctly predicted tumors is appropriate for the number of tumors in the dataset. The precision and recall rates of the target classes are relevant and consistent, demonstrating the model's efficacy.

Table 6.4: Classification Report of the Final Model

Description	Precision	Recall	F1-Score
No Tumor	0.77	0.85	0.81
Tumor	0.83	0.74	0.78
Macro Average	0.81	0.8	0.8
Weighted Average	0.81	0.8	0.8

6.5 Conclusion

Though code reviews are commonly accepted as a necessary software development activity, various impediments often hinder proper assessment. Software reviewability is also influenced by many code characteristics, including visually recognizable attributes. Since some code attributes are visually recognizable, this study demonstrates a means of evaluating source code using screenshots of code snippets. Screenshots of tumor and non-tumor source code were created. Different CNNs were used to identify tumors and the customized CNN could identify code tumors with satisfactory accuracy.

Though the patterns used in this work can be spotted simply by using standard static analysis tools, this might not be the case for other tumor styles. If it is possible to identify any bad-looking code using machine learning and image recognition, it is possible to refine the process for identifying more issues. Since this work shows promising results, more research should be done. Even for humans, visually recognizing a tumor in software can be a difficult task. Applying deep learning image classification brings an exciting advancement to this activity.

Part V

Preparation and Planning for Avoiding Complicacy

Chapter 7

Applying Hazardous Operational Planning to Software Development

7.1 Introduction and Related Work

In Chapter 8, a lifecycle model demonstrating playwriting and code walk-throughs is presented as a method to facilitate software design. As this approach is new, introducing these ideas with some background information is beneficial. This chapter discusses the motivation for the playwriting and walkthrough ideas and demonstrates their connection to software.

A valuable activity in planning for software dependability is understanding how those involved in analogous endeavors prepare for success in their critical activities. Inspiration is available from various places, but for mission-critical software, activities that protect people and property in dangerous situations are very insightful. In this chapter, consideration of the preparation required for convoy protection in dangerous regions is presented. The preparation steps are summarized below [10]:

1. Gather intelligence on route, people, and surrounding areas.

2. Issue a "Warning Order."

3. Publish timeline for and perform critical tasks, such as maintenance, pre-mission checks and inspections, and rehearsals.

4. Issue a convoy mission brief that details actions against known dangers, routes, speed, employment of support vehicles, order of movement by vehicle, casualty plan, strip map, and convoy manifest.

5. Confirm the communication plan.

6. Conduct a leader back-brief and perform final rehearsals.

7.1.1 Existing Literature

Military organizations around the world have used rehearsals for centuries. It is said that the Romans rehearsed battles using sand tables with icons to visualize the battlefield. [155] Modern armies believe the rehearsal is a tool for commanders to make sure parties involved understand the intent and scope of the operations. Rehearsals provide opportunities to identify previously unrecognizable inadequacies in plans. Rehearsals contribute to external and internal coordination [10]. In other words, rehearsals of all shapes and sizes are used to ensure efficient battlefield operations. Dress rehearsals can be entire battlefield simulations with whole army units participating, or they can be small, where individuals take on the role of entire units. Events are simulated in real-time and participants act out their responsibility at different points of the exercise [10].

7.1.2 Research Objective

Obviously the military is not alone in using rehearsals, and this is a powerful tool that can be well used in software engineering. In software engineering, using rehearsals leads to a very unencumbered design work flow that nicely partitions the modules of a system. Responsibilities are identified by highlighting the nouns and the verbs in the requirements, such as user stories. Verbs are candidates for actions a class can perform, and nouns are candidates for information that the class we should maintain [193].

7.1.3 Application of Operational Planning to Software

The first step is very applicable to software operations. Gathering intelligence can easily be mapped to what is known as the requirements workflow in software engineering. Gathering intelligence is an important activity for a successful mission, and it is unlikely any military would start a convoy without proper intelligence. In software engineering, proper requirements gathering is the basis for a successful software project.

In the second step of mission preparation, the military issues a “warning order.” In software engineering, is analoous to approval for a mission-critical project.

The third step in convoy planning includes much detail relevant to the successful creation of a mission-critical software project. The mission timeline is undoubtedly analogous to a schedule. The remainder of the listed tasks map to the analysis, design, implementation, and test software development workflows. These tasks are basically the work required to develop a software product. As with the requirements workflow, abundant literature exists describing these everyday activities [152].

In Step four of convoy protection planning, final details of the mission are established. In a waterfall software engineering model, this is analogous to the completion of the design, and the team being ready to begin coding.

In the final step the convoy protectors give a final briefing and perform final rehearsals. The importance of the rehearsal is apparent in the steps followed by military planners. It is instructive to learn why such importance is placed on this activity. As described by convoy protection planners, the following points describe the importance of rehearsals [184]:

- Reinforce training and increase proficiency in critical tasks.

- Reveal weaknesses or problems in the plan.

- Synchronize the actions of the signal teams.

- Improve each member's understanding of the operation.

This approach to ensuring understanding of the activity presents an opportunity to create a novel new approach for engineering dependable software systems. Understanding the architecture of the software is undeniably an essential aspect of dependability. An opportunity exists during analysis and design to walk through and rehearse the software's internals to validate understanding of the requirements and architecture. Though what could be called "operational rehearsals" are performed regularly in the form of execution-based testing, rehearsals of the software design itself are seldom, if ever, done. Section 8 explores this concept of rehearsal in software development in greater detail. A rehearsal can reveal weaknesses in the requirements or the proposed design, benefiting the project.

7.2 Conclusion

Understanding how humans prepare for hazardous work provides insight into how software engineers can prepare software for critical missions. The concepts presented by the military demonstrate how important it is to understand a task that must be performed. Software engineers can mimic these techniques in their efforts to create dependable programs. Performing rehearsals during analysis and design can help create more dependable systems.

Chapter 8

Software Design Using Operational Planning Inspired Rehearsals[*]

8.1 Introduction

For whatever reason, designing software has not been perceived as exciting as writing it. In the days when design relied heavily on flowcharts and data flow diagrams, programmers would complain about management requiring those steps. Some organizations even believe architecture design is too expensive and time consuming. Another contributing factor to this, at least at the beginning of a project, is that the software problem to be solved is not well understood [64]. Immediately writing code is seen as a way for engineers to begin understanding the domain with the thought of writing the "real code" later, which more often than not never happens. In the eyes of the customer and management, the code is working and the team is demonstrating progress. At this point piecemeal growth of the software begins and development starts to grow in an uncontrolled fashion [64]. Put another way, rather than design and architecture structuring the code, the code defines the design and architecture. This results in an overly complicated code base which is hard to expand

[*] The text in this section is substantially taken from conference publications #6 and #9, "A life cycle model for creating uncomplicated software" and "Teaching software engineering to career-changers."

and maintain [64].

Before going too far, the importance of software design cannot be stated, and designing an architecture should not be an afterthought. Stephen Schach describes the code-and-fix life cycle model as a software product implemented without requirements, specifications, or any attempt at design [137]. Schach points out that maintenance costs of the code-and-fix model are higher than for formally designed software products [137]. The traditional view of maintenance is that it begins after product delivery to correct faults. Many organizations have a large, existing code base and many projects are products based on code reuse. Maintenance may begin as soon as the source code needs to be looked at again. A software engineering mentor once said reading great code is like reading a book [51]. The intent of the author is so clear that the code "reads" like a story [52]. Successful literary works are rarely written without some plan or outline. Even the best coding standard will not make up for a lack of design.

8.1.1 Background and Existing Literature

Alternatives to the ad-hoc design approach have been proposed, such as Responsibility-Driven Design. As stated by Rebecca Wirfs-Brock, "Responsibility-Driven Design is a way to design that emphasizes behavioral modeling using objects, responsibilities, and collaborations. In a responsibility-based model, objects play specific roles and occupy well-known positions in the application architecture." [193]. This concept of Responsibility-Driven Design is beneficial for the analysis and design work flows of software engineering.

In Responsibility-Driven Design, objects have a very specific part of the application. Each object is responsible for doing one portion of the work. Objects do only one job, and they must do that one job well. Objects then communicate with each other to fulfill the larger goals of the application [192]. As an example, consider a system that requires the ability to navigate. In this system there is GPS receiving equipment, a compass, and inertial sensors. It is possible that the entire application could be monitoring these different sensors, distributing the responsibility for navigation throughout the system. Following Responsibility Driven Design, there will be one module that is completely responsible for navigation, and other parts of the system will query the services of this module for their navigation needs. It is likely this module will have several classes supporting the different types of navigation, but all of the navigation responsibility will be compartmentalized in one place.

8.1.2 Research Objective

Although using Responsibility-Driven design for identification of required classes is productive, it can be inadequate for deriving the flow of a system. Suppose Responsibility-Driven Design is explored from a different direction that described by Wirfs-Brock. Rehearsals were introduced in Chapter 7 as a means to validate activity planning. I demonstrate Responsibility-Driven Design based rehearsals as a new method of analysis.

This chapter describes how to use rehearsals to perform analysis and design workflows by having a person or persons rehearse (simulate) candidate object behavior. Responsibility-Driven Design is proposed as the starting point for engineers to relate to software modules. Software engineers, should try to describe how their software will work from the perspective of humans, rather than algorithms, doing the work, with the caveat that each person may do only one thing and must do it well. Developers must decide what roles are needed to resolve this conflict and the characters are given a significant problem to solve immediately [45].

8.2 Suggested Work Flow

8.2.1 Play Writing

It may be considered odd that information on writing a play would be included in a discussion of software engineering life cycle models. However, when considering using rehearsals as tool, using a play as a structure should not be overlooked. In his book, "Writing Your First Play", Roger Hall outlines elements of a play [79]. Chapter 1 covers action and how dynamic action employs verbs. In software engineering, verbs can be used to represent methods or functions in your code. Chapter 2 discusses obstacles and conflict, such as the conflict faced by stakeholders who do not have the required software.

Though this technique can work with any architecture design, the Model-View-Controller design pattern (MVC) works very well for this approach. MVC defines a plan for organizing components. The model portion handles data storing and the algorithms for processing data. The view portion is responsible for displaying information and results to the user. The controller is in charge and sends commands to the model and the view [149].

There are many resources describing how to write a successful play. However applying artistic information to software design is not always obvious. In playwriting, it is important to come up with a main character, then decide on a conflict or problem [188]. Next identify a beginning point and show the story in actions and "speech." Don't over do it, as one group of students wrote their play based on Star Wars characters and upon rereading it at a later date they could not remember the roll of each character.

There is one more suggestion that can benefit the success of a play, especially for new authors. Characters with special skills should be provided or generated before playwriting begins. In the sample play, characters with different skill sets participate in completing the required task. For example, the "Artist" character is responsible for communication. A "Boss" (Controller) character is responsible for the overall operation. Other characters for security, data management, and direct communications are included. See Table E.1 for a complete list. The "Professor" and the "Student" and the only two human characters in the play and they represent the users of the software.

The main character of the play is the "Professor," though the "Controller" has an active supporting role. As play writers should give characters a significant problem to solve immediately, in the sample play, the problem to solve is how the professor can best communicate with students during class. An example of the play created, as well as the UML used, can be found in Appendix E.

8.3 Final Analysis and Design Work flow

Human personalities are given to the software modules. At this point, a beginning point is determined, and the engineering analysis comes from the story via actions and speech. Responsibility-Driven Design coupled with plays produces an analysis that easily is communicated to all stakeholders.

When the play is finished it is rehearsed. When the participants agree that the flow of the application is proper and the requirements are met, the play is converted to a UML sequence diagram. The Unified Modeling Language (UML) defines a standard set of diagrams used in designing software [99]. Since UML sequence diagrams visually describe the actions of objects in a time sequence, they are idea to represent the lines of a play.

When the sequence diagram is complete, a UML class diagram can be made. UML Class diagrams show the relationships and dependencies among classes and are used to show the overall system architecture. It is very easy to make a class diagram from the sequence diagram, as the messages in the sequence diagram become the methods in the class diagram. Strict UML rules should not be enforced as the goal is to arrive at a candidate architecture for the software system. When the class diagram is complete, a candidate architecture is ready including identified classes, associations, and method names. An example of a play, as well as related sequence and class diagrams, are included in the Appendix E.

8.4 Results

To determine if this approach has merit, software design projects that were assigned to graduate students at the University of St. Thomas in St. Paul Minnesota, were analyzed. The beginning software engineering class has been consistently organized for the past three years. Students were required to form teams of two or three persons and design a major software project. Students were allowed to select the theme of their own projects, but in general, students are guided towards projects where the user interface is a prominent part of the application. Some example-projects include a WhatsApp like application that translates text to the receivers native language, classroom management applications, games, and medical patient management systems.

In the first year studied (2015), pre-play, students were asked to generate two-column use cases from user stories and then derive a design. Students generally had no trouble with the initial use-case, which showed user and system interactions. These describe "the user does this" and "the system does that" interactions. However, at this time many students were unable to identify the classes required to build a system. Deconstructing the system into smaller objects was a frustrating task for many students.

When assessing team progress, it was apparent that generally only one student per group understood how to undertake this task adequately. This problem was reflected through a summative assessment where nearly 50 percent of the students were unable to correctly create multiple two-column use cases, then perform analysis to derive the required UML diagrams. Also, nearly 25 percent of the students who had correctly created two-column use cases and adequately identified classes were unable to properly suggest functions or methods within those classes. Informally, students also indicated frustration with this approach.

During winter semester of 2016 performance style plays where introduced as a method of analysis, and became evident that the level of participation in the group activity rose dramatically. Performance style plays solved a significant problem facing the students, the partitioning of the system object. Students had less trouble identifying classes. Resolution of this difficulty was helped through the suggestion of characters with specialized skills for the play. Students could now envision a collection of specialists performing the tasks required for the system to operate. Pre-play, it was difficult to envision how to divide up the work of the system. Post-play, with the provided suggested characters, assigning tasks became very practical and was no longer perceived as impossible. Students also no longer had trouble identifying the methods required of each class, as methods were built upon the dialog between the characters in the play.

All team members took part in the creation of the play, and the post-play summative assessment rose to nearly 80 percent success. Thirty-nine final exams from two sections of pre-play classes and 119 final-exams from four sections of post-play classes where reviewed. Though the numbers of reviewed exams pre-play and post-play differ, the success percentages were consistent among classes. Additionally, post-play students who were not wholly successful were also not completely lost. In general, their issues were not severe. For example, “methods” might show up in the wrong class or “methods” may be missing. With a little bit more practice, these students can master this topic.

Table 8.1: Percent of Student Projects Successfully Completed

Curriculum	**Success Rate**
Pre-play	50%
Post-play	80%

8.5 Conclusion

In this chapter, a new approach to analysis and design workflows was presented with the goal of avoiding complicated software. The concept of responsibility-driven design is strengthened by employing the novel design concept of playwriting for separating object responsibilities while establishing a software architecture. Terminology and characteristics of complicated software are provided. How to creatively perform software engineering analyses and design workflows by writing plays inspired by Responsibility-Driven Design is shown. Information on creating UML sequence and UML class diagrams is given and a summative assessment was used as a measure of the overall success or failure of the approach. Though this approach was only formally studied once, it has been used successfully and consistently since 2016 in Software Engineering classes at the University of St. Thomas. The ongoing success suggests further research is warranted to analyze the more specific issues students had pre-play and how the performance style play could resolve those issues. In addition, a formal evaluation of large programming projects to verify good design quality and good programming practice is also necessary.

Chapter 9

Application of Coding Standards*

9.1 Introduction to Coding Standards

Coding standards are thought to be an excellent way to encourage consistent software quality. As stated by Jean-Pierre Rosen, the goal of a coding standard is to improve the quality of code, not just having one for the sake of having one [148]. An unnecessarily complicated program is not generally thought of as quality code. Because not all software engineers have mastery of, or even respect for, established coding rules, positive influence from guidelines is not always found. A significant and essential part of coding standards is a defined coding style, and while modern programming editors can enforce many stylistic rules, some specific tasks are still up to the programmer. If a programmer is required to remember too many things it is possible that some coding standard requirements will be missed.

This chapter includes a discussion of literature investigating other reasons why essential directions are not always followed. Additional research was also done in an attempt to clarify what is considered complicated software. The resulting recommendation urges organizations to create two-layer coding standards. Layer-1 should be made from rules

*The text in this section is substantially taken from Journal Publication #1, "Coding Standards and Human Nature."

that are easy to remember. Layer-2 should enforce more formal rules. Detailed coding standards should not be eliminated; instead, they should be supplemented.

9.2 Existing Literature

In 1994, Lawrence Zeitlin did an experiment involving activities with chainsaws. Safety orientation was provided, and upon completion of training, understanding of the rules was verified. When the investigation finished, it was determined that only 55.6 percent of the group followed the rules. Perhaps even more surprising, more experienced users were less in compliance [201]. A paper by Reason et al. describes the steps organizations with safety-critical applications take to assure compliance with production rules [141]. Consider how aircraft maintenance workers are required to follow workplace standards for their safety and the safety of those using the aircraft they maintain. If we couple lack of rule compliance with the immediate danger presented by chainsaw operation, it is not too surprising that software engineers disregard some of the rules of a coding standard as there is no physical danger present. However, it is possible that much like the aircraft maintenance workers, the work of the software engineer may put someone's life in peril. Unlike the aircraft maintenance worker, it is possible that faulty software is not just on one device, but many devices.

Individual differences in working memory also affect how people follow directions. George Miller observed that the number of objects an average person can hold in his or her working memory is about seven [116]. If you consider a mind maintaining a collection of coding standard rules while an engineer is creatively writing software, it seems probable that some of the rules are likely to be dropped. Some participants in an online conversation felt that coding standards are a creativity and productivity destroyer [53]. It has also even been suggested that disregarding the rules may be a rebellious act directed towards company policy or management [142].

The governmental Occupational Safety and Health Administration (OSHA) provides one more reason for a failures to follow directions. OSHA feels a lack of compliance is because many do not understand why compliance is essential [128]. For example, it may be concluded that after safety training and more personal practice with chainsaws, the users did not see the connection of the rules to safety so disregarded them. The chainsaw experiment showed that even in the presence of immediate physical consequences of failure to comply, guidelines are still not followed. It seems that if a coding standard is made, the most critical rules must be explained concisely so they are easy to remember. The software engineer will need a clear understanding that a mistake in the software they write may have dire consequences on the lives they touch. Dire consequences can exist for all types of software, not merely software written to control apparatus. For example, Lewis Morgan of IT Governance Blog states that his January 2018 report of data breaches is one of the most extensive lists he ever put together [118]. Undoubtedly though software may fail with no physical consequences, it still can have grave consequences on peoples lives. The Jet Propulsion Laboratory provides a thorough coding standard for applications, including requirements for flight-related software [120]. Linus Torvalds has produced a very real-world coding standard as well, and many of his ideas are incorporated as part of the recommendations of this chapter [182].

9.2.1 Research Objective

I propose a small layer or shim be placed on top of current and accepted standards for an organization. This top layer is to provide a fixed number of “must-haves” that are not impossible for a software engineer to remember. The must-haves need to be completely compatible with more detailed standards, such as the JPL standard, as the written code will ultimately have to comply with those more detailed standards.

In the 1960 movie “The Magnificent Seven,” a group of seven was hired to protect a small village in Mexico from a group of plundering bandits [146]. With the obvious consideration of Miller's law, it seems appropriate for the first layer coding standard to have seven rules protecting software from marauding bugs [116].

9.3 Coding Standard Layer One Recommendations

These rules are supported by the results of the conducted survey as well as pertinent suggestions from the Torvald's standard [182].

Consistent Indentation - The results of the conducted survey indicated files with proper indentation were more pleasant to review. Proper indentation gives a programmer a picture of the control flow of a function or method with a glance. Programmers should not have to struggle or rely on comments to find the beginning and end of a block [52]. Indentation should be consistent across all constructs such as structures and switch statements. Torvald's standard suggests indentations should match at eight characters and he argues that if you need to indent more than three times, you should rewrite your code [182]. Intuitively, it seems multiple indentations lead to more complicated code.

Limit Preprocessor Directives - Depending on the programming language used, preprocessor directives can get in the way of understanding. In C language, many conditional preprocessor directives, #if, #ifdef, #else, #ifndef, etc., can make code entirely undesirable for review, and ultimately unmaintainable. When nesting #ifdefs and #ifs, it can be very difficult to determine which statements are are actually used without painful examination of preceding lines or simulating the execution of the preprocessor [52].

No "Dead Code" - There should never be "dead code" in comments or anywhere else in a source file. It is extremely frustrating to spend time looking at and trying to understand a function only to later find out this function is no longer called. Do not use "dead code" as a source control system [182].

Limit Line Length - An unsurprising result from the source file survey, Chapter 4, was that programs with lines longer than 120 characters were not pleasant to review. Limiting line length to what can be seen without effort is an essential consideration for any programming language [137]. Torvald's coding standard suggests that lines should not be longer than 80 characters [182].

Be Mindful of Working Towards Efficiency - "More computing sins have been committed in the name of efficiency (without necessarily achieving it) than any other reason [194]." The software survey indicated that programs undesirable to review also generally had higher Cyclomatic Complexities [110]. Code with high cyclomatic or decision complexities should be considered to be a technical debt and should be rewritten [137].

Show Intent - When reviewing code, it is crucial that code reviewers understand the intent of your work. Comments should tell what a function does, not how a function works. It is not okay to use comments as a substitute for weak program structure or poorly named functions. The code should be self-documenting, and its operation should be apparent to a reviewer [137]. A program written in self-documenting code has carefully chosen variable and function names, and the code is crafted exquisitely, almost wholly removing the need for comments [52].

Braces for even one statement also make the programmer's intention clear. Indicating a beginning and and end for even one statement makes the intent abundantly clear. Use of parentheses is also a good idea to clarify intent, even for those who have learned the many precedence rules. Not everybody has learned or remembers all the precedence rules. The placement of the parentheses telegraphs the intent of the programmer, and there is no doubt as to how the logical expression will be evaluated [52].

Have a Consistent and Straightforward Naming Convention - Names of variables and functions should be descriptive and consistent. In the 1970's variable name length was a concern. This is no longer the case, and auto-complete in coding editors eliminates the need to memorize complicated identifiers, so there is no excuse for having nondescript names. Abbreviations should not be used as people tend to lose consistency in making abbreviations over time [137]. Code reviewers sometimes have to guess what a variable name is during a debugging exercise. When named consistently, it can be easy to deduce. If abbreviations are used, and especially if they are used inconsistently, it can be challenging to guess a name to find the code that needs evaluation. For example, consider how the following four variables are declared: averageFreq, frequencyMaximum, minFr, and frqncyTotl. A programmer reviewing the code must know if freq, frequency, fr, and frqncy all refer to the same thing [137]. Finally, choose a style such as camelCase or under_bars, but do not mix both in the same project.

9.4 Coding Standards Conclusion

Software development organizations constantly strive to create higher quality products. As in many aspects of life, established rules and instructions are not always followed when writing software. Implementing a two-layer coding standard where one layer is composed of easy to follow rules and the second layer is built from more detailed regulations will help software engineers write code that is compliant or, if nothing else, write code that is easier to bring into compliance. By understanding seven basic rules, engineers can produce less complicated programs and write code with fewer faults. For many organizations, maintenance costs are as much as three times the cost of the original software project [137]. Writing less complicated, less faulty code will make maintenance easier and bring down maintenance costs. A coding standard will not magically make bad programmers into good programmers, and a coding standard cannot make up for a bad design,but it can substantially benefit a development organization [52]. Organizations should create a Layer-1 coding standard from easy to remember rules to foster creation of higher quality, less complicated software.

Part VI

Working With Existing Projects

Chapter 10

Software Streamlining*

It is not difficult to find examples of popular software becoming more complicated over time. Users generally appreciate the arrival of new features, but the fact remains that new issues also arrive. In some environments, the cost of complications may not be worth the value of the new feature. For example, if a new feature requires a faster processor or more memory, it may render the software useless on some existing supported hardware. It is also possible that build size eliminates portability to new targets. Consider Linux. When new in 1997, many versions of Linux require about four megabytes (MB) of memory to operate. Ten years after Linux's initial release, many versions required more than 256 MB of memory to operate [76]. Presently it is recommended that desktop free versions of Debian Linux have 512MB available. [41]. Intuitively, it seems possible that new features can be added to a project without raising complicacy. One obvious consequence of the new complicacy is an opportunity for new bugs. However, another often overlooked result is the increased difficulty of use and the greater demand for operational resources. This chapter presents taking an existing software product and streamlining it into a smaller and less complicated version.

*The text in this section is substantially taken from Conference Publication #1, "Software Streamlining: Reducing Software to Essentials."

10.1 Existing Literature

This issue of expansion of features resulting in the expansion of complicacy and complexity has not been ignored in software engineering circles. One example is shown in the paper, “A survey of Software Refactoring.” Tom Mens et al. describe various techniques for handling software expansion to meet new requirements while maintaining quality [114]. Konstantinos Stroggylos et al. investigate if refactoring improves quality at all [166]. At the time of this writing, a search of Google Scholar [73] shows more than 62,000 works related to software refactoring. Most works describe the management of expansion of features rather than slimming systems down. Arie Van Deursen et al. address refining and slimming to some extent in their paper, “Aspect mining and refactoring.” They suggest mining source code in a manner almost analogous to mining gold. Through isolating and then refactoring code, reusable modules can be made from existing software products [186]. This project demonstrates module reusability in new projects and quality improvements in existing projects, making their work supportive of works concerning reduction in software complicacy.

Also, the importance of small and uncomplicated is widely addressed by the concept of the “minimal viable product” (MVP). There are many definitions of what the term MVP means, as described by Valentina Lenarduzzi et al. in their paper “MVP Explained: A Systematic Mapping Study on the Definitions of Minimal Viable Product [101].” From the perspective of this work, an MVP is software system with the least number of features required to fulfill most users’ needs. MVPs are generally streamlined from inception, rather than as a result of a streamlining project.

With the thought of avoiding complicated code in mind, one might think about irreducible complexity. Irreducible complexity is not a universally accepted concept in the biological sciences. Michael Behe defines irreducible complexity as a single system composed of several well-matched, interacting parts that contribute to the basic function, wherein the removal of any one of the parts causes the system to effectively cease functioning [18].

Put simply, if a piece is taken away, the system no longer performs as it was intended to. Supporters of intelligent design believe this shows that evolution cannot be completely responsible for life on this planet, and that there had to be an intelligent creator involved.

The merits of biological intelligent design will not be debated here, but one cannot avoid noticing the parallels between computer/software evolution and biological evolution. Code is said to evolve, but code cannot evolve without the hand of the creator. Some have suggested code "rots" if left alone long enough. In practice though, it can be recognized that the code that is not rotting, but the environment that it was designed to run on is changing. In computer software, it is impossible not to recognize the hand of an arguably intelligent creator. An accounts receivable program may one day evolve into a full accounting system program, but it will not do so by mutation.

As a software engineer, this concept should be kept in mind during analysis and design. If a design is overly complicated the software engineer should work to eliminate extra complexity, with the final target being reduced until the program can not further be reduced without destroying product functionality. Extraneous parts that do not contribute to the program's functionality should be removed.

It should be kept in mind that creating more complicated systems is not always beneficial and sometimes ends with catastrophic results. Consider the Boeing 737-MAX as a dramatic example to illustrate the point. The 737-MAX was created as an upgrade to the successful 737 line of aircraft, but the new physical characteristics of the Boeing 737-MAX made it prone to stall in certain circumstances. Additional software was required to address this increased stall risk. Sadly, the new system did not work correctly in all circumstances, resulting in the loss of aircraft and human life [85]. Obviously, the 737-MAX issues were not merely due to software changes, and perhaps even more apparent, not all software expansion leads to catastrophic results. However, this story can serve as a reminder that added complicacy is not always the answer.

Table 10.1: Classic Rodos Features

Object oriented C++ interfaces
Ultra fast booting
Real time priority controlled primitives multi-threading
Thread safe communication
Synchronization
Event propagation
Publish/Subscribe middleware

10.1.1 Research Objective

Although there is an abundance of research on refactoring as part of ongoing software maintenance and much research exists on MVPs, the concept of creating of entirely less complicated systems from larger systems (streamlining) has not been sufficiently explored or advocated. In contrast to previous research, this goal is to streamline by "reducing to essentials" an entire existing application in both size and scope.

Rodos (Classic Rodos) is a preemptive, priority-based operating system having features summarized in Table 10.1 [117]. Classic Rodos is has been successfully integrated into many projects, including the TET-1 micro-satellite operated by the German Aerospace Center [58].

As part of the exploration of software streamlining, this work begins with the Classic Rodos operating system, mines the essential aspects, and delivers a less complicated, less resource-intense, Instant-Up version (Instant-Rodos). Instant-Up operating systems are tiny and require very few resources to be operational. Using an instant-up operating system requires only a standard C/C++ compiler and minimal hardware startup code [47]. Generally, access to only a few hardware specific operations is required to launch the systems, making porting to new targets very easy. On some processors, Arduino for example, no special access to processor registers is required, all due to the rich tools provided for software construction. As Instant-Up operating systems do not have access to registers, the thread scheduling must be cooperative [47]. Even so, all types of schedulers are possible, including round-robin, priority, and fair, though fair scheduling does require a tick or other timing mechanism. Instant-Rodos implements a priority scheduler with fair scheduling for threads of equal priority.

The resulting product, Instant-Rodos, shows it is possible to streamline and simplify software making it less complicated.

10.2 Methods / Implementation Sequence

Table 10.2: Supported Classic Rodos Header Files

Header
CommBuffer.h
Fifo.h
LocalDefines.h
Semaphore.h
Thread.h
TimeEvent.h
TimeModel.h
Timer.h

As a project goal was to maintain existing functionality, I gathered **important header files,** from Classic Rodos. Table 10.2 lists the files that define the application programming interface (API) of the supported features.

After defining the required features, the **queue implementation** was then performed. A queue is a first-in, first-out data-structure which is very important inside operating systems. As mentioned, Classic Rodos is preemptive and priority-based. To make the new design less complicated, the often-used data structure, the 'heap,' was chosen to implement required priority queues. A heap is a standard data structure that allows for efficient adding and removal of items which is very popular for priority queue applications. Use of heaps is applicable in many areas of the new operating system; thus, it is used for the ready queue, semaphore waiting queues, and timer queues.

Table 10.3: Classic Rodos Stack Layout

Thread	Stack Allocation
thread 0	allocation unused
thread 1	thread 0 stack
thread 2	thread 1 stack
thread 3	thread 2 stack
no thread	thread 3 stack
no thread	remaining allocation

The next challenge was implementing a simplified **context switching** method. Instant-Rodos uses the functions "setjmp" and "longjmp" for context switching. Setjmp and longjmp are defined in the C and C++ standard library for providing non-local jumps. One way to understand non-local jumps is to think of the often villainized 'goto' statement, though setjmp and longjmp are a little more sophisticated. Context switching is achieved by using setjmp to save current context information. When the scheduler determines which thread to execute, the longjmp function is used to restore a previously saved context. Basically, when a call to longjmp is made, the execution continues using context information saved by setjmp. It is possible to build Instant-Rodos without a standard C/C++ library, but the programmer would need to implement setjmp and longjmp. The good news is that implementing setjmp and longjmp requires only a few lines of assembly, and ample example code exists.

At this point, **the startup** was implemented. During startup, stack space for each thread is allocated in the initialization routine. Because Instant-Up operating systems do not have the ability to directly manipulate the processor stack pointer, each thread must allocate space for the previously initialized thread. Space is allocated at the top of the initialization function as a local array variable. This allows the previous thread to safely "push" data onto the stack without crossing into its neighbors' stack-space. Table 10.3 has a small

Table 10.4: Rodos Size Comparison Report

Version	Size
Classic Rodos Compiled STM32	40k
Instant-Rodos Compiled STM32	13k
Instant-Rodos Compile 68k	10k
Classic Rodos Lines of Code	6,400
Instant-Rodos Lines of Code	1,650

illustration of the stack layout. This does mean some memory might be wasted, as the very first thread will not have a predecessor. However, this implementation is clean and elegant; addressing the unused space allocated by the first thread could add new complications. As soon as all the threads are initialized, the last thread is “run,” launching the operating system.

Performance measurements were performed upon completion of the source code for the new operating system. For these measurements the system was built using different targets to measure both size requirements and verify utility. A very simple test application was created with threads sending messages to each other. Long running tests were performed on two different STM discovery board classes. The long running tests were performed using 2700 mAh batteries and measured memory requirements and duration.

10.3 Operational Comparison

Table 10.4 shows the different memory requirements of each version. Instant-Rodos requires less memory than Rodos for the test application (13KB for Instant Rodos compared to 40KB for Rodos). This is supported by the lines of code for each operating system (1,650 for Instant vs. 6,400 for Rodos). Table 10.5 shows the results of the first long running test.

The first long running test ran on the STM32F4-Discovery board and included both Rodos and Instant-Rodos. In the second long running test, only Instant-Rodos was run because it was not practical to get Rodos on the smaller board. Table 10.6 shows the results of the second long running test. Summarizing the results, at 14 hours versus 14.5 hours, there was little to no performance gain when running Instant-Rodos on the same hardware platform as Rodos. However, Instant-Rodos saw a large gain in longevity by moving to a board with less resources running for 27 hours.

Table 10.5: Rodos “Hello Word” Application on STM32F4-Discovery

Version	Size	Idle Ticks	Duration
Classic Rodos	64K	1,770,512[1]	(approx) 14 hours
Instant-Rodos	35k	1,961,241[1]	(approx) 14 hours
[1]Context switch time is longer on Classic-Rodos			

Table 10.6: Rodos Application on STM32F0-Discovery

Version	Size	Duration
Classic Rodos	NA	NA
Instant-Rodos	22K	27 hours

10.4 Level of Complicacy

In addition to performance measurements, **complicacy measurements** were also made against the newly created operating system source code. Traditional metrics, such as lines of code (LOC), cyclomatic complexity, Static measurements of coding style were perforemd as shown in Table 10.7.

Using traditional software metrics, it is apparent that Instant-Rodos is comparable to Rodos in some respects, while being less complicated in others. The cyclomatic complexity per KLOC of Instant-Rodos and Classic Rodos are comparable to the average for the thesis projects (236 per KLOC). Decision complexity is significantly different with Rodos at 193.33 per KLOC, and Instant-Rodos at 62.41.

However, the total cyclomatic complexity of Instant-Rodos is only 214 compared to more than 12,000 for RODOS. LOC for Instant-Rodos is also much smaller, at about 25

Table 10.7: Style Metrics Applied to Rodos

Metric Name
Indent blocks
Avoid using more than four parameters
Avoid deep blocks

percent of the quantity of Classic Rodos. Finally, when it comes to programming stylistic rules, Instant-Rodos conforms better to style recommendations. Measurements on the three basic styles mentioned are shown in Table 10.8.

Table 10.8: Static Analysis Report of Rodos versus Instant-Rodos

Instant-Rodos	Counts
Total Applied Rules	3
Total Errors	0
Total Analyzed Files	29
Violated Files	0
Conformance	100.00%

Rodos	Counts
Total Applied Rules	3
Total Errors	29
Total Analyzed Files	37
Violated Files	13
Conformance	64.86%

10.5 Discussion

Instant-Rodos is designed to be less complicated than Rodos, and this is reflected by the values from the traditional metrics. This does not indicate that Rodos is bad or problematic, it simply reflects that Instant-Rodos, by supporting fewer features, was designed to be less complicated than Rodos.

As shown in Table 10.5, the performance of Instant-Rodos versus Rodos is very comparable when they are run on the same hardware. However, as shown in Table 10.6, Instant-Rodos providers longer operational duration on small systems where it is impractical to run full Rodos. In environments of lower resource availability, Instant-Rodos demonstrates an improved battery life is possible by moving to more simple software and hardware.

Further energy management enhancements to Instant-Rodos could be achieved by using a time-triggered scheduling approach, as advocated by Michael J, Pont [134]. Following Pont's suggestions, tasks can be created and executed at different priorities at different times as required. Implementing this design is a practical way to enable the system to sleep during idle times for even more battery life.

Instant-Rodos' ease of portability is demonstrated by the port to the 68K processor. The 68K memory footprint is shown in Table 10.4. This port of Instant-Rodos required less than 75 lines of hardware dependant code to configure the processor and launch the operating system.

As a final point of discussion, it should be pointed out that Pareto Principle illustrates a crucial point of this project. Put simply the Pareto principle indicates that 20 percent of inputs result in 80 percent of outputs [8]. The software manifestation of the Pareto Principle suggests 80 percent of users only use 20 percent of program features. This exercise creating Instant-Rodos shows that even when streamlining, desirable API maintenance is still possible, and common required application features can still be supported. If possible to select the proper 20% of features that satisfy 80% of users, it is possible to reduce system complicacy, perform desired work, and be less resource-intensive.

10.6 Conclusion

People generally expect new software releases to include expanded features and provide enhancements. The ongoing growth and expansion of software can result in new complicacies, which sometimes produce negative consequences. This paper explores the concept and benefits of transforming existing software systems into smaller, less complicated versions through streamlining. Streamlining creates a new, smaller system from existing work rather than simply engaging in refactoring or other popular techniques. Demonstrating the process of software streamlining, essential features from the Rodos operating system were extracted to produce Instant-Rodos, which is less complicated and requires fewer operational resources. As shown in Table 10.2, Instant-Rodos was built using many existing Rodos C++ headers, resulting in the preservation of essential Rodos features. Though this project focused on streamlining to create a new software product, there is no reason not to use streamlining for refactoring an existing product to adapt functionality without increasing complicacy. Overall, the Instant-Rodos exercise results show that the concept of creating less complicated systems from existing software is worth further exploration and promotion.

Chapter 11

Programming Style - Java For Satellites Case Study[*]

11.1 Introduction

Throughout this thesis, different ways of measuring software complicacy have been presented. This chapter presents a case study as an example path that might be taken for selecting existing source code for a project and determining if it is sufficiently uncomplicated. Though this chapter focuses on searching for a Java Virtual Machine (JVM), the concepts used can be applied to other types of software.

The 'C' programming language has been long accepted as the programming language of choice for aerospace and mission-critical applications. However, Java was released in 1995, and it is now more than 23 years old. It should not simply be dismissed as mission-critical work.

[*] The text in this section is substantially taken from Conference Publication #8, "The Suitability of Java for Satellite Applications"

11.1.1 Existing Literature

There are many examples of Java use in CubeSat applications. For example, the ham radio community has telemetry software written in Java [67], and there are programs and libraries written in Java to create simulations of orbit, and orbital decay [133]. Though there are many examples of Java used in ground software, there are few examples of Java used in orbital applications. As of August 2018, GitHub hosts more than sixty Java Virtual Machine (JVM) projects [71]. Wikipedia shows a similar count of active and inactive open source JVM projects.

In addition to hosting many JVM projects, GitHub also hosts more than four hundred math and science libraries written in Java [71]. Many universities teach Java as the introductory programming language, so large numbers of Java programmers are available in an academic setting [77]. Professionally, Java is just as popular as ‘C’ by programmers [135].

11.1.2 Research Objective

Based on the popularity of Java in aerospace applications and the number of JVMs available, it makes sense to consider using Java in embedded, mission-critical applications. This chapter endeavors to answer the question of whether any JVM is suitable for use in space flight applications. The first step is defining criteria for the evaluation of source code suitable for use in mission-critical applications. Then, various open-source Java virtual machines are evaluated for suitability. Finally, a report of findings is presented.

11.2 Justification for Java in Space

11.3 Criteria for Java Virtual Machine Selection

Many Java Virtual Machines exist so criteria must be defined to select which JVMs would be evaluated in this project. First and foremost, the source code must be readily available. For this study, the source code must be available on GitHub [71] or SourceForge [159]. If the source code is too challenging to acquire, it could be of no use to those interested in the project. The second criteria are that the JVM source code must be written in ‘C’ or ’C++’ and must be simple to build. This is because complicated builds are harder to port to embedded applications as such complicated builds and builds that required special tools were dropped from consideration. There are also example Java Virtual Machines written in Java, Python, and other interpreted languages. At this point, having a layered JVM does not seem practical for satellites, so none of these were elected. Ideally, the selected JVMs have some sort of open source license, but many projects have not established any license. However, any project that specifically limited type of use was dropped from consideration.

11.4 Criteria for Satellite Software

It is essential to be careful when selecting software for inclusion in a satellite mission as a software failure can cause the failure of the mission. This section defines what I believe are essential characteristics for software used in satellite and other mission-critical applications. The criteria presented here are based on two foundations. The first and most important foundation in determining if a piece of software is space-worthy is whether the source code is uncomplicated enough to understand.

If project code is too complicated for a proper review, it likely contains faults that go unnoticed. Recommendations included here are based on a previously mentioned study I demonstrated that software engineers can quickly visually spot code that is too complicated for review. In the study, programmers were asked to visually scan "C" and "C++" source code and immediately indicate if they felt the code would be pleasant or unpleasant to review [48]. Upon completion of the survey, the most important stylistic failures were determined and listed in Table 4.1 [48].

The complexity study also discovered that programs with a high cyclomatic complexity were considered unpleasant to review [48]. Cyclomatic complexity measures the number of independent paths through a portion of code [109]. In this thesis, it was determined that a cyclomatic complexity of 232.27 per KLOC the average value. Refer to Appendix 15.3 for more information on this.

The second foundation is based on coding recommendations from NASA's Jet Propulsion Laboratory (JPL) [96]. These standards were summarized through recommendations compiled by Gerald Holzmann from his paper, "The Power of Ten-Rules for Developing Safety Critical Code." These are listed in Table 11.1 [82].

Table 11.1: Gerald Holzmann's Ten Rules for Safety Critical Software

Rule Name
Use simple control structures including avoiding 'goto.'
Know how long control will remain in a loop.
Do not use dynamic memory.
Keep function length short.
Use assertions to check for conditions that should never happen
Use the smallest scope possible for variables and methods.
Check return codes from function calls.
Do not use preprocessor directives.
Limit pointers to only one level of dereferencing.
Do not ignore compilation warnings.

Since uncomplicated code is an essential aspect of this thesis, in this section I map how these recommendations are connected to uncomplicated code. For example, using simple control structures when programming is recommended. Though it is superficially easy to search for goto statements, to be utterly confident that software has simple enough control structures, the code must be easy to understand and review. Likewise with the upper bounds of loops; it is possible to do running time analysis on loops to determine upper bounds, but understanding the code is indeed the best way to be sure. Limiting pointers to only one level of dereferencing makes it easier for a human reviewer to understand.

Assertions are essential for checking conditions that should never happen. Uncomplicated source code is also very important in this matter. If a human is unable to understand code, they likely cannot wholly understand the conditions that should never happen to place assertions.

An often overlooked recommendation is that when calling a function, the return code should be checked rather than the caller assuming success. Intuition tells us it is easier to determine proper error handling when working with less complicated code. Straightforward things to verify are the absence of pre-processor directives such as #ifdef, the absence of compilation warnings, the inclusion of asserts, and that the the number of lines in a function or method should be small.

From knowing the characteristics of complicated code and the ideal characteristics of software for space applications, a methodology can be created to identify JVMs that are far from compliance.

The use of dynamic memory is not recommended, and in 'C' programming this means that malloc system calls are not allowed. An important feature of Java is garbage collection, which implies dynamic memory; however, it is theoretically possible to eliminate garbage collection and require developers to understand how much memory they need.

11.5 Methodology

More than sixty virtual machines are available, and manually reviewing each machine is impractical. Built on the understanding of what characteristics are important in satellite software, a mostly automatic process was created with the goal of eliminating JVMs which were far from meeting the recommendations. The process of evaluation was carried out over several steps.

11.5.1 Step 1

As mentioned, only JVMs with readily available source code were desired for this study. JVMs were downloaded from GitHub.com [71] and SourceForge [159]. The JVM's License files were reviewed, and those that explicitly forbade certain usage were eliminated. JVMs without a license were not eliminated, but the lack of license is an important consideration from a legal perspective.

11.5.2 Step 2

In this step, an attempt was made to build the virtual machine in a Unix environment. A complicated build process is likely to be risky, so if too much effort was required to make the build, or if the build could not be completed, the JVM was dropped from the study.

11.5.3 Step 3

In this step, the remaining machines were evaluated for software quality. A report based on the complexity study was generated for each machine. No machines were eliminated from the study based on these reports.

11.5.4 Step 4

The remaining JVMs were then checked for 'C' constructs such as #if, #ifdef, and malloc. They were also checked for usage of assert and the absence of goto statements.

11.5.5 Step 5

A small test program was compiled and run using the OpenJDK [126] environment, then run in the test enviornments. The rationale for this step is that if it is difficult to get a small application running, a more complicated application is likely impossible. The sample program is shown in Listing 1. Note that the test here is not suggested to be an exhaustive test of all of Java's capabilities, but only to exercise very basic Java capabilities.

11.5.6 Listing 1

```
// Hello JVMs
// Michael Dorin 2018
//
public class HelloWorld
{
  public static float math (int a, float b)
  {
    float returnVal;
    returnVal = a * b;
    return returnVal;
  }
  public static void main (String args[])
  {
    float a = (float) 1.1;
    int b = 6;
    float answer;
    answer = mathTest (b, a);
    System.out.println ("Hello world");
    System.out.println ("Result " + answer);
  }
}
```

11.6 Results

None of the virtual machines were able to meet all of the requirements perfectly. This does not necessarily mean that Java should not be considered for satellite applications. However, it does mean selecting Java an aerospace application is a more difficult decision because of the extra work that may be required to get a JVM ready. The results presented are broken up into three sections. First, the standard GNU JVM for Linux and the standard Oracle JVM are discussed. The second sections present results from analyzing the open source, “embeddable” machines. Finally, the last section shows two machines which did not make the final selection but were regarded as interesting.

11.7 Standard Linux JVMs

11.7.1 Oracle

This chapter discusses open source JVMs due to their reviewability. During the investigation, it became apparent that many open-source JVMs are not ready for mission-critical applications and it seemed wiser to consider the Oracle JVM. Except for the absence of source code to review, it cannot automatically be considered a bad choice. It is well maintained and is available for different target platforms. Oracle provides good support for their product, and it is easy to install in Linux environments. Oracle presently provides an embedded version of the software that can run in even Arm environments [127]. If a CubeSat has the resources to support the Oracle JVM and the need exists for Java, it may be the path to select.

11.7.2 OpenJDK and HotSpot

Open JDK is generally thought of as the standard set of Java tools for Linux. Open JDK uses HotSpot JVM, which is the self-proclaimed best JVM on the planet [75]. This is likely the most complete and most stable of the open source JVMs evaluated in this project. Installing Open JDK on various Linux distributions is easy. HotSpot makes regular use of assert, as recommended by the JPL summary. However, it is not an easy JVM to independently review based on its size and having a review quality report of only 30.53 percent. HotSpot source also has a cyclomatic complexity of 277.69 per KLOC, which is highter than the cyclomatic complexity of other projects studied in this thesis. Hotspot does use the "C" goto statement various places and there are thousands of preprocessor directives. HotSpot may not easily port to small target platforms or specialized real-time operating systems. However, this could be a good choice for systems running Linux with sufficient resources.

11.8 Embeddable Java Virtual Machines

None of the finalist JVM candidates for embedded systems met the requirements for mission-critical applications. However, the JVMs included in this section built and ran Java class files, meaning they may be good candidates for future updating.

11.8.1 sJVM by Jakub Veverka

This seems to be a student project, and no license is provided on GitHub [187]. The project has an excellent review quality report of 71.79 percent and it has a cyclomatic complexity of 232.40 per KLOC, which is nearly the average of other projects studied in this thesis. At present it only has twenty preprocessor directives and it uses no goto statements. sJVM is also somewhat functional and does operate. sJVM is also very easy to build on Linux and even Macintosh environments and the build generates no warnings. However, it is not a complete Java implementation as it will not run class files created with current Java build tools.

11.8.2 JVM by Arthur Emidio

This also seems to be a student project with no License provided, but the complete source code is available on GitHub [57]. The virtual machine put forth by Arthur Emidio has a good review quality report of 55.56 percent and it has a cyclomatic complexity of 168.67 per KLOC, which is decently lower than the average cyclomatic of projects found in this thesis. It does have a considerable number of preprocessor directives but it does not use "C" goto statements. It also calls malloc twenty-seven times in the code. Though an incomplete JVM, it is somewhat functional. This JVM was very easy to build on Linux but required cmake. sJVM could run compiled programs but unfortunately it was unable able to run the test program (Listing 1).

11.8.3 Simple Java Virtual Machine by ntu-android

The virtual machine put forth by ntuAndroid [121] violates many of the rules in Table 4.1 resulting in a low quality report of only twenty percent. However it does have some features that make it attractive. When reviewing the code, one can see many obvious stylistic violations that would be quick and easy to correct. This JVM is also somewhat functional and operates better than most when tested with compiled Java code. It has a cyclomatic complexity of 201.18 per KLOC, which is also lower than other projects measured in this thesis. It uses more than fifty preprocessor directives but uses no goto statements. It also calls malloc thrity-six times. The JVM was very easy to build on both Linux and on Macintosh environments, but does generate 53 warnings at this time. It is packaged with the GNU Public License which makes its use unrestricted.

11.9 Honorable Mentions

11.9.1 Tiny JVM by Julian Offenhäuser

During the JVM evaluation period, I was unable to get this virtual machine running correctly, but it is mentioned here because of its size and simplicity [124]. It will build on Linux, but a license has yet to be defined. Executing the included automatic test caused the application to crash on our Ubuntu system. That said, it would be very straightforward to port this JVM to different processors and operating systems. Based on a single file, the build process cannot be simpler. It has a pretty good cyclomatic complexity of 190.47 per KLOC. It uses very few preprocessor directives and it has no goto statements. At this time, it uses "malloc" in nine locations. The size and scope of this one file JVM is very interesting and compelling as a candidate for updating and porting.

11.9.2 Nano VM by Tharbaum

This virtual machine was not tested on Linux as it was been made for the Atmel AVR family [81]. It is a good implementation and was actively maintained for years. It has a reasonable review quality report of forty-five percent it has a cyclomatic complexity of 315.48/KLOC. There are 141 preprocessor directives and it has no "C" goto statements. It does use "malloc," but only in one place, making it ideal for handling dynamic memory concerns. Many of the complaints in the quality report are very easy to fix, such as adding braces even for one statement and adding spaces around operators. It was released under the GNU public license, so, has flexibility for use in different application environments.

11.10 Conclusion

As part of the preparation for this study, more than sixty Java Virtual Machines were reviewed for suitability for satellite and mission-critical applications. First, the results show it is possible to use Java for a satellite application, but presently, it is not a perfect choice. There is no single Java Virtual Machine that can meet all the requirements for an ideal application. The recommendation from this review is, for now, to use a standard JVM from Oracle or one provided by the OpenJDK project. Unless a CubeSat software team has the background and the time to do extra work, an embeddable JVM should not be considered. Many of the embedded JVMs do not build properly or are simply too complicated to adapt to an embedded target smoothly. It is unfortunate that a JVM suitable for satellite embedded systems was not identified during this study. Future work should be done taking an existing embeddable JVM such as Nano VM and bringing it up to a mission-critical software coding standard. During this process, an abstraction layer should be created allowing the adapted JVM to be ported to different real-time operating systems and processors. Second, and very important, the results show evaluating existing works for a specific task based on complicacy is reasonable and practical.

Chapter 12

Static Analysis - Loop Artificial Pancreas Case Study [*]

This chapter continues with the theme of evaluating existing software for complicacy. The Loop project's goal is to provide a substitute for the functionality of the pancreas. As such, it qualifies as a mission-critical application. Since Loop is a mission-critical application with a large and diverse user base, the information discovered here is very important for this thesis.

This chapter expands the evaluation of static analysis and also includes information on Loop operations and a report of how well the system has worked from the perspective of a user. As Loop is written in Swift, different tools for static analysis are required. Topics such as cyclomatic complexity, programming style, LOC, and even division by zero potential are addressed in this study. As such, the topics examined in this study are similar but not precisely the same as those in previous chapters. Analysis of Loop source code is performed, and considerations are presented regarding project reliability using the bug counting concept.

[*] The text in this section is taken substantially from Conference Publication #3, "Open Source Medical Device Safety: Loop Artificial Pancreas Case Report"

12.1 Introduction to Loop

The Loop project was selected because of the important service it provides users. Loop is an iPhone application that blends a continuous glucose monitor (CGM) with an insulin infusion pump, creating a class III medical device [61]. As a class III device, Loop is extraordinary in the world of open-source medical applications.

The Loop software application sits atop several commercially available medical devices to extend their functionality and interoperability (Figure 12.1a). Loop is a 'hybrid closed-loop system' since it requires user input. Loop attempts to improve control of insulin delivery using automatic adjustment of baseline (basal) insulin delivery and an advanced model for insulin delivery at mealtimes (bolus). Use of closed-loop insulin delivery systems has been shown to improve glycemic control, when compared to patient administered insulin delivery [15].

An iPhone replaces both the handheld insulin infusion pump controller and a continuous glucose monitor receiver. The iPhone receives Bluetooth Low Energy (BTLE) transmissions from the CGM. It sends radiofrequency (RF) commands to an insulin infusion pump (OmniPod or Medtronic) via a communications intermediary called a RileyLink. A RileyLink is a custom-built piece of hardware that bridges communication between the iPhone and the infusion pump [143]. The CGM transmits the user's blood glucose levels at regular intervals. If communications fail, the Apple Health application allows users to manually enter data collected from a traditional blood glucose measurement device. If all channels fail to deliver CGM data to Loop for more than 15 minutes, automated controls for insulin administration cease and rates of infusion default to pump settings preprogrammed by the user.

The Loop project is now more than two years old [176] and the software architecture is well established. Loop has a large and active user base, as evidenced by a dedicated Facebook group with more than 19,000 members. Loop is composed of a primary iPhone

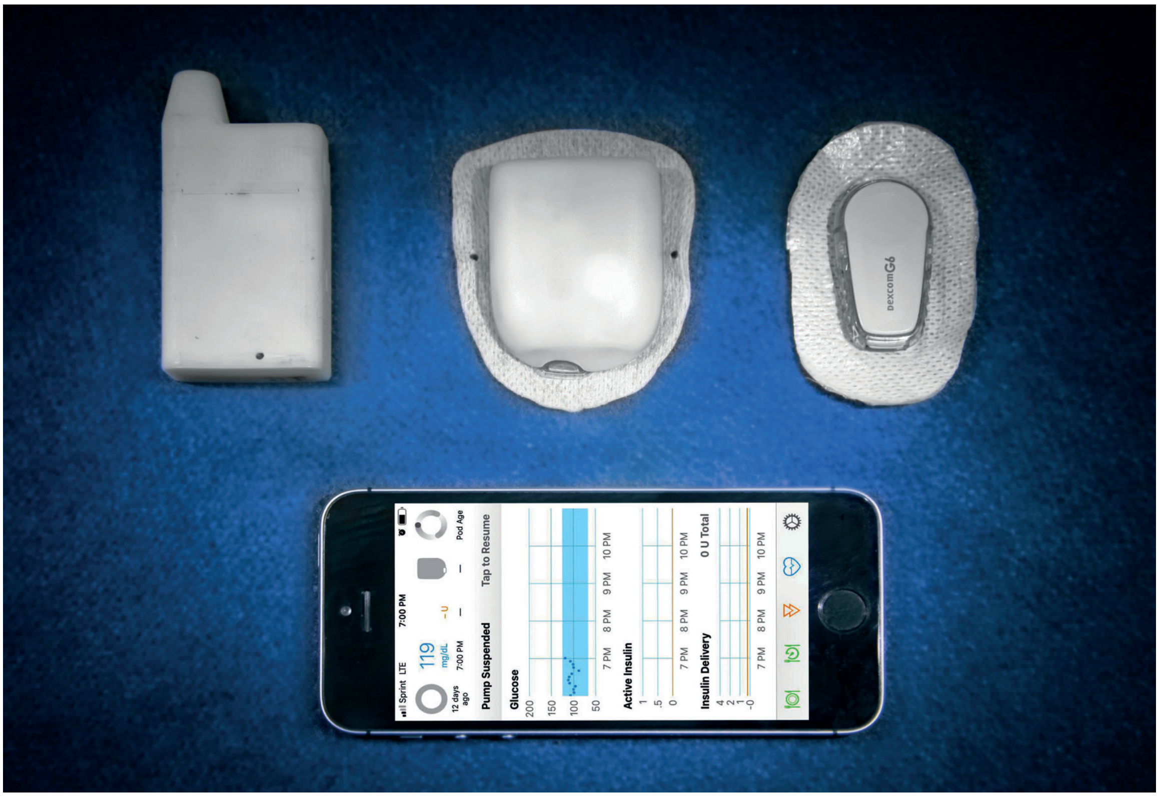


Figure 12.1: Loop System Components

Top Left to Top Right: RileyLink, Infusion Pump, Blood Glucose Sensor, iPhone application and seven supporting libraries. At this time, Loop runs exclusively on iPhones, but Android alternatives exist [189]. Loop v2.0 contains more than 90,000 lines of code, including libraries [105]. Loop is a complex system of hardware and software and, by its nature, allows points of failure to exist. For example, without a properly operating Riley Link, the Loop program cannot send instructions to the insulin infusion pump. Versions of Loop presently under development eliminate the need for a Riley Link device [179].

Since Loop is mission critical, I looked for instances where faults might not be readily identifiable inside common software execution paths. Static analysis is a technique used to identify software faults independent of code execution [203]. Static analysis was performed on Loop source code in two different ways. First, I used a program called SwiftLint, which is designed for static analysis on Swift-based source code [140]. Second, I wrote a specialized software tool to search for potential 'division by zero' errors. Division by zero (Div-0) errors can be catastrophic for a program if they are not correctly handled. The static analysis with SwiftLint produced a considerable number of recommendations, both warnings and errors regarding the source code. Analysis for Div-0 potential also showed these cases exist, which is not surprising as sophisticated math is required when calculating insulin dosage. These reports neither specifically indicate dangerous faults exist nor suggest that Div-0 problems are not handled properly, but a code inspection should be performed to verify application safety.

Estimates for the number of faults based on the size and scope of the project showed the software to be mature in quality. Depending on how faults were reported and counted, actual faults counted were between 1208 – 1499 (Table 12.1). Bug estimation techniques developed by Steve McConnell and Fumio Akiyama [112] [4] relied on lines of code. McConnell suggested 15 bugs per KLOC which gives an estimate of 1,080 bugs (given that Loop and its supporting libraries are about 72 KLOC). Akiyama suggested 18 bugs per KLOC, giving an estimate of 1,296 bugs. Using the average bugs, 14.16 per KLOC for mature sample projects (see Table A.2), my own estimate is 1012 bugs. At the time of writing for this project, Akiyama came closest to the number of bugs. I do not suggest these results are typical. However, they do suggest that using lines of code as a bug estimation tool is reasonable. Each organization must maintain its own BKLOC statistics to obtain the best results possible.

Based on the bug estimates and the large active user base, it is possible to surmise that the majority of software system faults on the loop project have likely been identified and remedied or are being remedied, resulting in a mature software product. This is in line with the Bug-Counting Concept of reliability estimation.

12.1.1 Loop Efficacy from the User Perspective

“Rosa” is a 38-year-old, technology literate female living in the United States, with a 26-year history of type 1 diabetes. Rosa’s Omnipod Insulin Infusion Pump with Eros pods [125] and Dexcom G6 model CGM required the Loop version released during or after April 2019. Rosa selected the stable Loop software branch to avoid the volatility of updates in an active project. At the time of this writing, Rosa had been running Loop for more than five months. Before and during her transition to Loop, Rosa reported Blood Glucose (BG) information to Tidepool. Tidepool is an organization that provides data collection and reporting of BG information [103].

People with diabetes make up a diverse community. Physiological and behavioral differences between individuals make generalizations difficult or impossible. Rosa was only comfortable conducting care via an automated software system because of her technical background and familiarity with devices used in the treatment of diabetes.

Overall, Loop managed Rosa’s glucose levels as indicated by Time in Range reporting on Tidepool. However, inconsistent readings from the Dexcom G6 were problematic, leading Rosa to believe the reporting was inaccurate. In December 2019, Rosa spoke with a Dexcom Care Line nurse who advised her on proper sensor calibration. Rosa felt that Dexcom documentation could have been clearer on this issue[44] [185]. When documentation is inadequate, users must contact manufacturers or seek other support when encountering problems.

Some difficulties experienced may be attributable to the upgrade from the G4 model CGM to the G6 when Rosa began using Loop. As the G6 was a new device, neither Rosa nor her doctor had much time working with it. This inexperience lead Rosa to experiment with sensor calibration techniques that, in all probability, ultimately increased sensor error. In retrospect, continuing to use the devices she had experience with while adding the new software system would have allowed for more rapid identification of subsystem failures.

Issues with server connectivity were also problematic, but some may have been avoided through following developer configuration recommendations. For example, Rose was using Dexcom Follow alongside Dexcom Share. Loop developers recommended against using this configuration [165].

Daily use of the Loop system also identified several environmental concerns that may need addressing. Operation in cold weather is a possible concern as Loop requires battery-powered hardware. Also, the impact of background RF and of sport or work clothes on BTLE communications should be quantified. Finally, further research is warranted to understand how variables such as illness or intense anaerobic exercise can impact operations.

12.1.2 Analysis Results - Software Perspective

There are several factors both users of Loop and for future open source developers to consider, particularly when applications have the potential to impact human health and safety. Points of failure need to be identified, and default execution paths that allowing code to pass failure points should be provided. The server outage from Dexcom is a good example. Many Loop users, unaware of how to properly configure their device, were unable to use the Loop application during this time.

As discussed, the number of faults found in Loop is typical among similarly sized software projects, leading me to believe that most of the faults have been found in the system.

Table 12.1: Loop and Component Bug Counts as of January 2020

Loop Component	**Bugs Reported on GitHub**	**Bugs Suggested by Git Logs**
Amplitude-iOS	81	124
CGMBLEKit	32	54
G4ShareSpy	0	2
MKRingProgessView	46	16
LoopKit	38	256
dexcom-share-client-swift	3	6
SwiftCharts	325	156
rileylink_ios	71	463
Loop	612	422
Total	1208	1499

However, addressing the issues identified by static analysis early might have prevented faults from being released to users in the field and reduced the risk of using the Loop product. Abundant research exists on this topic and an excellent paper by Al Bessey et al. discusses finding faults without code execution [21]. Avoiding releasing faults in the first place is a priority for mission-critical systems. The performance of static analysis should be part of the software development process, and abundant static analysis tools exist. Upon finding errors and warnings through static analysis, they should immediately be addressed.

12.1.3 Conclusion

Providing coaching advice in hindsight is never problematic and it has not been my intention to disparage the Loop project. This chapter shows the successful use of the Loop open-source medical project and has reported both the positive and negative aspects of Loop from a an operational as well as a software development perspective.

The Loop project is a volunteer effort and the success of their work must be commended. They have accomplished a lot and have positively impacted many lives. The

Loop software system has benefited many people but it is not necessarily a good idea for everybody. At this time, those interested should do their own thorough evaluation of the system, discuss operations with other Loop users, talk to their doctor, and then finally make their own educated decision as to whether or not they should attempt using Loop. As with the JVM study, this investigation can be a model for evaluating existing systems for complicacy and reliability.

Part VII

Conclusion

Chapter 13

Exploring Code of “Rock Star” Developers

13.1 Introduction and Background

Throughout computer science history, many programmers became famous because of the work they produced. As with famous artists and musicians, the study and understanding of the work of rock star programmers can help others become better at software development. Several famous software engineers were selected, and their works were gathered and analyzed. In some cases, their work may still be in active use. The authors selected for this analysis are Brian Kernighan, Dennis Ritchie, Donald Knuth, Linus Torvalds, Douglas Comer, and Richard Stallman. These “rock stars” were selected based on their notability in the industry and the availability of source code attributed to them.

13.1.1 Existing Literature

Brian Kernighan was part of the team developing the Unix operating system, but he is probably most famous for co-authoring, with Dennis Ritchie, “The C Programming Language [178].” The book “The C Programming Language” made both Kernighan and Ritchie legends in C programming practice. More information about Kernighan and his work can be found in the 2017 interview made by the Computer History Museum [43]. Kernighan also wrote a book, “Elements of Programming Style” where he described important aspects of programming style [90]. Dennis Ritchie also worked on the Unix operating system [144], and according to his obituary, he was the inventor of the ‘C’ programming language [32]. Donald Knuth was famous for his insights on computer programming and designed the text formatting language “TeX” [5]. In his essay, “Literate Programming,” Donald Knuth also wrote about programming style [93]. Linus Torvalds is famous for developing the Linux operating system [181]. Douglas Comer famous for his book “Operating System Design: The XINU approach,” which was first published in 1984 [50]. Since 1984, the book has been updated several times and the XINU operating system has been incorporated into more than a dozen products. Richard Stallman is famous for pioneering free software and founding the Free Software Foundation [191]. Stallman was also inducted into the Internet Hall of Fame in 2013 [156].

13.1.2 Research Objective

Although there is literature on the lives of rock star coders and their opinions on programming style, analyzing their code gives more complete perspective on their work. Since their work is and has been widely used, exploration is essential to understanding the characteristics of their source code. This project studies classic works using traditional metrics such as coupling and decision complexity, and examines programming styles and linguistic aspects. The results of this analysis are compared to results gathered from various open-source projects.

13.2 Methods

13.2.1 Acquire Appropriate Source code

It was not difficult to find code from all the rock stars, though some had more code available than others. No attempt was made to normalize the rock star authored code based on size for this study.

Brian Kernighan and Dennis Ritchie. The source code from Kernighan and Ritchie came from examples found in "The C Programming Language, Second Edition [145]." Source code examples from their text can be found on GitHub [157]. As it was not always possible to separate the work of Kernighan and Ritchie, their work is analyzed as an aggregate. In total, 5,384 lines of code attributed to Kernighan and Ritchie were analyzed.

Donald Knuth. Knuth's "Buddy Allocation" was selected and the code came from his book, "The Art of Programming, Vol 1, Fundamental Algorithms, Section 2.5C [92]." This source code is also available on GitHub [91]. Several of Knuth's programs written in "CWEB" were analyzed. "CWEB" which is a programming system designed by Donald Knuth and Silvio Levy [94]. The syntax of CWEB contains C constructs, so it was possible to use this as an additional source of work written by Knuth. In total, 1,154 lines of code authored by Knuth were used in this project.

Douglas Comer. This research used the XINU operating source code to represent Douglas Comer's work. Various versions of this code can be found on the XINU website [38], but for this project, the x86 variant was selected. Approximately 20,000 lines of code attributed to Comer were analyzed.

Linus Torvalds. Although Linux source code is easy to find, for this project, the original release of Linux was used. This code was found at soft.lafibre.info by Appliwave [76]. The original Linux was selected to maximize the amount of code completely written by Linus Torvalds. The original Linux contained approximately 9,000 lines of source code.

Richard Stallman. The source code used for Richard Stallman is from an early version of Emacs. This source code is presently hosted by Apple in their open-source repository [164]. The version of Emacs studied comprises about 33,000 lines of code.

Mainstream Programmers. “Mainstream” code, from various sources, was used for comparison to the work of the rock stars. Some mainstream code came from the open-source projects used in Chapter 2. Also included are Beginning ‘C’ code examples compiled by Gourav Thakur [175]. More sophisticated mainstream code comes from the GNU Scientific Library published by the Free Software Foundation [66] and from the Operating System textbook, “Operating Systems: Three Easy Pieces [11].” Finally, to include code deliberately designed to be difficult to read, code from the International Obfuscated C Code Contest was included [29].

It is presumed that textbook and beginning ‘C’ source code is purposefully written to be understandable. It is thought that scientific source code will naturally be more challenging to read. Finally, it is supposed that the code written for the International Obfuscated C Code Contest is intentionally written to be challenging to understand.

13.2.2 Further Organize Mainstream Code

The selected open source projects, first used in Chapter 2, were further classified to get an insight into reliability. As young projects may not yet have had a chance to demonstrate their reliability, only mature projects were analyzed. Since the average age of the projects was about seven years, only files older than seven years were selected. These files were further divided by the number of times the word “crash” appeared in their git log. This research presupposed that seven-year-old files with zero crashes were more straightforward to read than seven-year-old files with multiple crashes.

Table 13.1: Coding Issues Impacting Source Code Desirability

Problematic Coding Issues
Do not write over 120 columns per line
Indent blocks inside of a function
Use less than five parameters for a function
Avoid deeply nested blocks
Use braces even for one statement

13.2.3 Perform Measurements

Estimate Coupling. Coupling measures the number of classes and data types known by a module, and for this study, an approximate measurement coupling was made using the Sheficom metric was made. More information on this tool can be found in Chapter 3.

Measure Decision Complexity. The “Succinct Code Counter” (Scc) tool is used [25] is used to count the number of decisions in a file. Chapter 2 has more information on decision complexity as well as Scc.

Perform Static Analysis of Style. Conventional wisdom suggests coding style plays an essential role in the comprehensibility of source code. Poorly styled code is more formidable to review and sometimes is even reviewed incorrectly. Chapter 4 identifies problematic style characteristics and measurement methods. These methods from Chapter 4 were used on the projects contained in this chapter. As a reminder, Table 13.1 lists the important problematic coding issues.

Perform Linguistic Analysis. The linguist George Zipf suggests humans prefer to speak in the most efficient manner possible [206]. This is covered in detail in Chapter 5, but generally speaking, people try to use the most straightforward words possible to communicate successfully. The linguistic style source code measurements mimic the spoken word analysis based on the presumption that programmers will generally want to write code using the most uncomplicated statements and expressions necessary to have a successfully working program. The linguistics analysis steps follow the process described in Chapter 5.

Map Trends of Random Open-Source Mainstream Projects. To visually portray characteristics of mainstream work, trends were analyzed from the measured metrics of the crash reporting open-source. Locally estimated scatter plot smoothing (LOESS) was used to illustrate these trends [35]. Note that LOESS, is not used for predictions but instead to show tendencies based on the measured metric vs. reported crashes, similar to the work of John Lachin et al. in their paper "Impact of C-peptide preservation on metabolic and clinical outcomes in the Diabetes Control and Complications Trial" [97]. The $\sqrt{N}$ was used for the LOESS K value, as described by Saravanan Thirumuruganathan [177].

13.3 Results

13.3.1 Mainstream Projects

The results from the mainstream projects are reported first to facilitate comparison with the rock stars. As previously mentioned, source files that reported crashes were separated by the number of crashes reported. Decision complexity, coupling, style, and linguistic measurements were made for all mainstream projects. For the open-source files organized by crashes, trends showing metric measurements vs. crash reporting are shown here.

Style compliance is shown in Figure 13.1. Each point represents the percentage of files reporting crashes compared with their conformance to style rules. Measurement represents the percentage of the number of files in compliance.

Decision complexity vs. crash reporting file results are shown in Figure 13.2. For this graph, the number of data points (N) was 1,100 and the k value (window size) for LOESS was 33. The maximum number of decisions per file was 307, the minimum was 37.

Coupling results are shown in Figure 13.3. For this graph, the size of the sample (N) was 2,000 and the K value was 44.7. On the low end, the coupling estimate is about 13 "includes" per source file. On the higher end, the coupling estimate is 33 "includes" per source file.

Linguistic token count values are shown in Figure 13.4. The smallest number of tokens found in statements/expressions was 10. The largest, for this study, was 11. A sample size of 2,200 (N) was used with a k value of 47.

Most of the style issues in the mainstream projects were related to having braces for even one statement, though, for crash reporting files, as the number of reported crashes increased other issues crept in. Regarding coupling, none of the projects not classified as crash reporting, those being Beginning "C", Obfuscated, OSTEP, GNU Scientific, had a coupling measure higher than six. However, as shown in Figure 13.3, coupling does tend to creep up as the number of reported crashes increase with crash reporting files. As with coupling, decision complexity creeps up most in files reporting five crashes. The obfuscated code projects reported a decision complexity on par with files reporting five crashes. Statement/Expression length was between five and eleven. The beginning 'C' files and the OSTEP textbook code had the smallest average statement/expression length. Open source files with zero and five crashes, the GNU Scientific Library, and the Obfuscated code had the largest statement/expression lengths.

13.3.2 Donald Knuth

Concerning conformance to the style rules listed in Table 13.1, Knuth violated the rule banning long lines, and the rule mandating the use of braces for even one statement. However, he only had thirteen infractions within his 1,154 lines of analyzed code. With minimal effort, conformance could reach 100 percent, suggesting style conformance is not a large problem in the work of Knuth.

Compared to the files in which crashes were reported, Knuth wrote code with low coupling and with a low decision complexity. From a linguistics perspective, his work was uncomplicated, with an average statement token count of 8.4, less than files reporting crashes. Regarding decision complexity, Knuth's work was also measured less decisions than files reporting any crashes, but more than other projects (except for Obfuscated work).

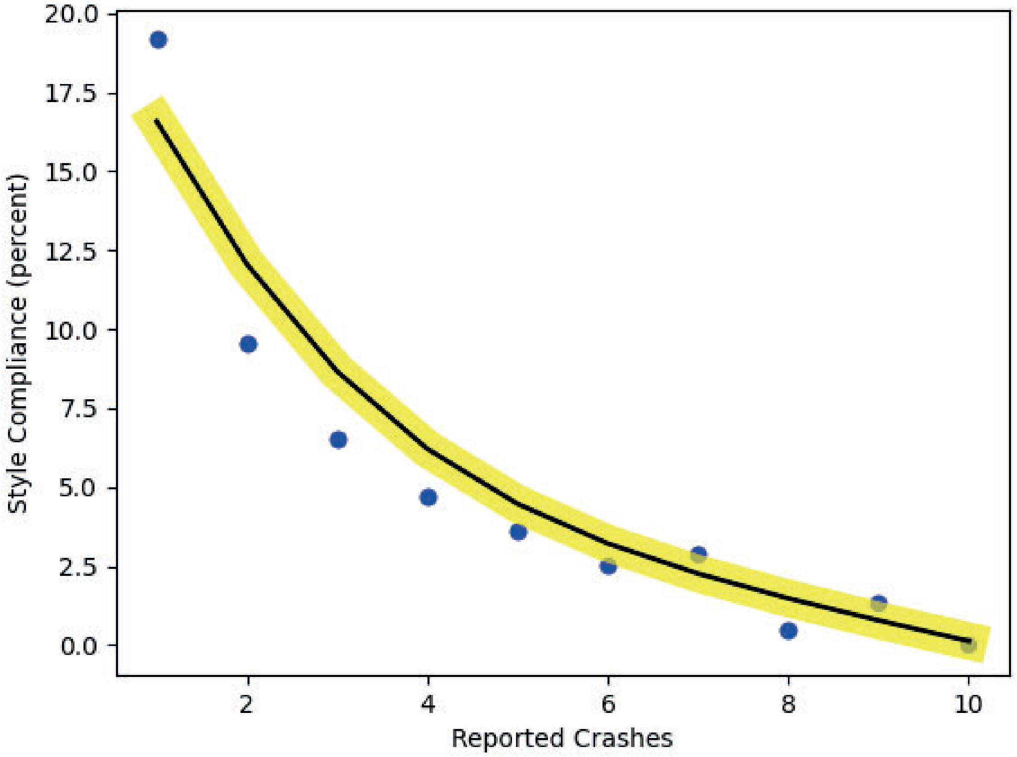


Figure 13.1: Scatter Plot of Files Reporting Crashes vs. Compliance

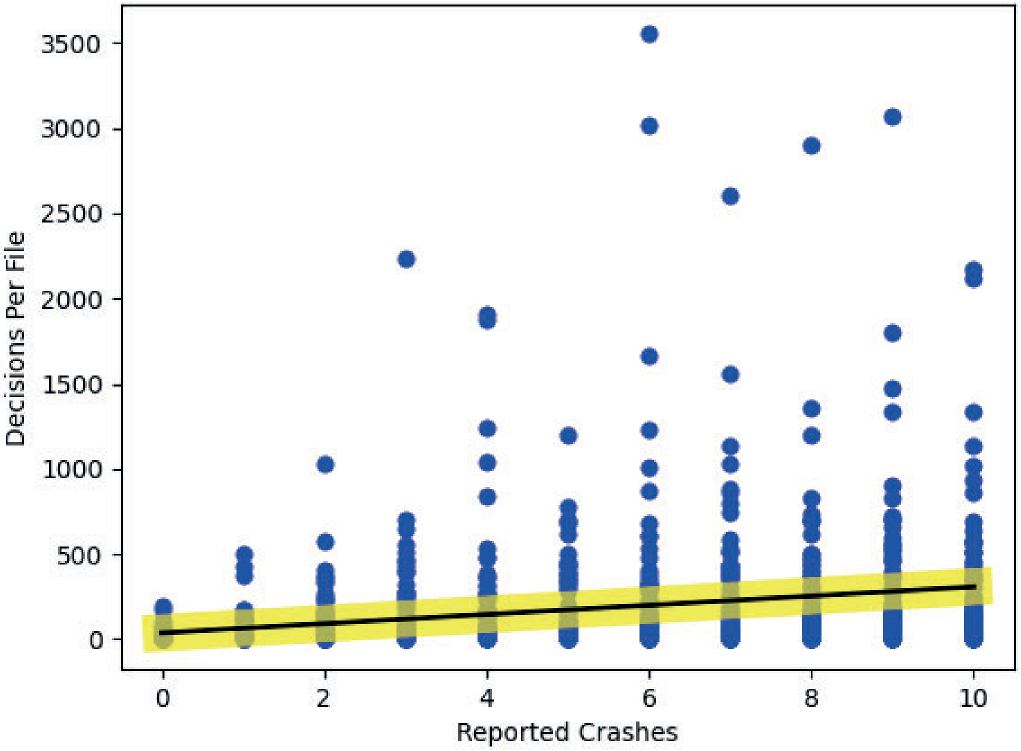


Figure 13.2: Scatter Plot of Files Reporting Crashes vs. Decisions

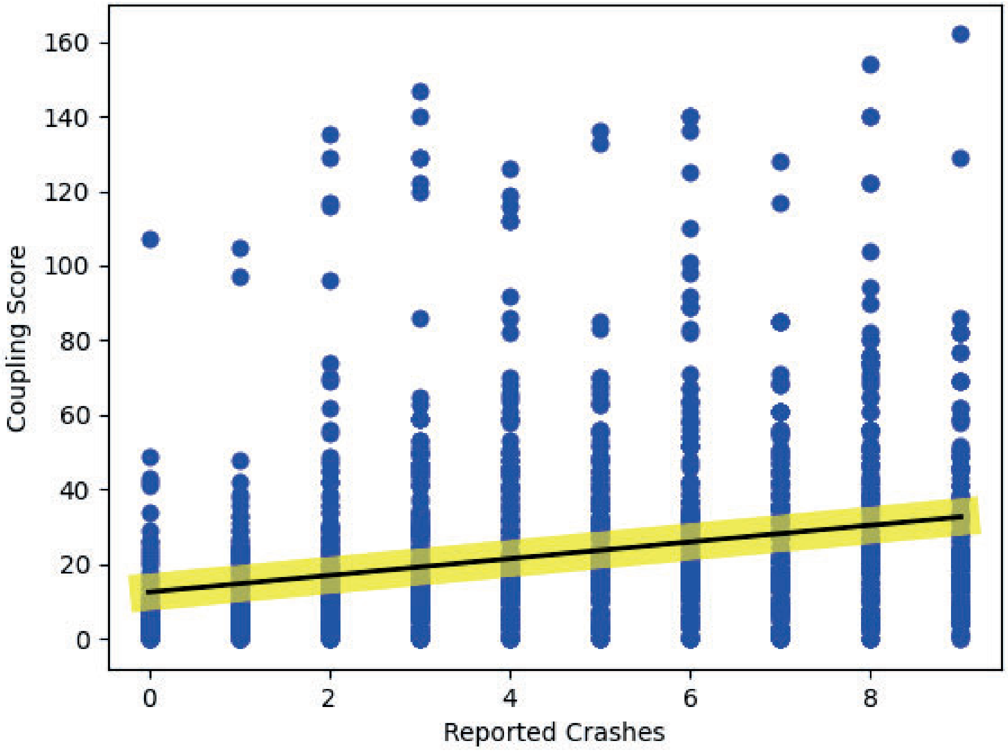


Figure 13.3: Scatter Plot of Files Reporting Crashes vs. Mean Coupling Score

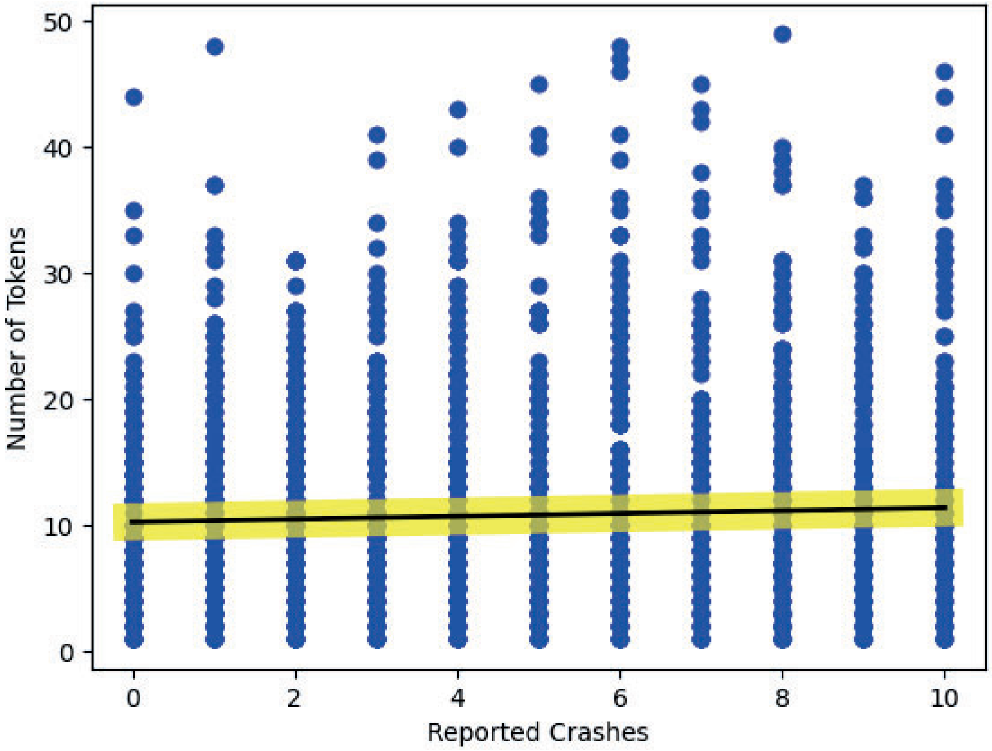


Figure 13.4: Scatter Plot of Files Reporting Crashes vs. Linguistics

Coupling measurements were smaller than all projects except for Beginning ‘C.’

13.3.3 Kernighan and Ritchie

Regarding programming style, Kernighan and Ritchie (K and R) had 434 violations for the 5,364 lines of analyzed source code. However, 96 percent of those violations concerned the mandate to use braces for even one statement, so K and R’s could quickly be brought to a superior level of style conformance once again indicating that style is not a major problem. Another interesting observation is that K and R’s code has the least amount of coupling. This is likely because their code comes from textbook examples which normally only require standard header files. Linguistic analysis of their code indicates an average token count of 7.3, which is less than most of the included mainstream software. Except for OSTEP and Beginning ‘C’, and K and R’s work has less coupling, fewer decisions per file, and fewer tokens per statement than all the mainstream projects included.

13.3.4 Linus Torvalds

Regarding stylistic quality, the Linux source code from Torvalds has a build quality of nearly 55 percent. As with Knuth and K and R, most violations arise from not using braces for one line of code. Considering that this accounted for 721 of the 764 violations, Torvalds’ code is of good quality. Torvalds’ measured coupling averages about five header files per file, less than files reporting zero crashes. Decision complexity is lower than files reporting zero crashes, with 20 vs. nearly 36. Compared to the mainstream Beginning ‘C,’ Linux has higher values for coupling, decisions per file, and tokens per statement/expression, which is expected as Linux is a much more sophisticated product. Linux generally has better numbers than files reporting zero crashes. The algorithms in the Linux source code make far fewer decisions per file than the Obfuscated code.

13.3.5 Richard Stallman

Of all the rock stars, Richard Stallman ranks the best at programming style, with a quality report of 72.38 percent. As with the other authors, most of Stallman's infractions were related to not using braces for single lines (87 percent). Coupling measurements were lower than most mainstream projects. Only Obfuscated and Beginning 'C' had lower coupling measurements (both groups represent small programs). Files written by Stallman tend to make more decisions than those by the other authors, averaging nearly 31 per file, but this is still less than files reporting any crashes. Stallman also tends to use shorter statements/-expressions, with a linguistic token count of 7.9.

13.3.6 Douglas Comer

As with K and R and Knuth, Douglas Comer's work originates within a textbook. Comer's programming style is very consistent. He has 74.37 percent of files in compliance. so he has very few issues with programming style. A deeper analysis shows that from nearly 19,000 lines of code, Comer has only 327 style violations, so it would be very easy to adapt the code to be in full compliance. With regards to coupling, Comer's work is a bit higher than the other authors and higher than the code reporting five crashes. This is due to Comer using a 'cheater file.' Linguistically, the average length of statements/expressions in Comer's is 7.1 tokens, which is a lower value than most mainstream projects. Decision complexity is about ten decisions per file, which is much lower than code reporting any crashes.

13.4 Discussion

The trends shown in Figures 13.1, 13.2, 13.3, and 13.4, demonstrate behaviour of the different metrics as the number of crashes a file reports increases. These characteristics of style, coupling, decision complexity, and statement/expression size are important to measure because their values can indicate a propensity for crashing, and more crashes can indicate poor reliability. From the perspective of linguistics, although there is not a significant trend demonstrated as the number of reported crashes increases, it is interesting to note that none of the rock star work approaches the token count of any crash reporting file. The rock stars had no systemic issues concerning programming style, and all of their work could easily be improved to whatever standard desired. To some extent, this could also be said for the mainstream programmers. However, as the number of reported bugs increases by file, the style improvements would be more challenging. Considering coupling, only the XINU code had an issue flagged. Coupling from the other rock stars was well below the coupling of any files reporting crashes. Finally, in examining decision complexity, none of the rock star work is even close to the code reporting five crashes or the obfuscated code.

Code deliberately written to be easier to understand tends to have lower values for coupling, complexity, and statement/expression length. Such code has better compliance with style rules.

13.5 Conclusion

Software rock stars are famous for implementing new ideas with exceptional programming skills. Rock star work continues to impact software engineering today, and understanding the styles and characteristics of their code can be an essential tool for creating reliable software. This research compares the work of rock stars to mainstream programmers' work. Source code from both mainstream authors and rock stars was gathered and analyzed for style, coupling, decision complexity, and the number of tokens in statements/expressions. Where possible, mainstream code was further organized based on crashes reported by a particular file.

A comparison of the the rock stars work to that of mainstream authors shows that rock star work is generally less complicated. Rock star code has little or no style issues, has few issues with coupling, and no issues with statement/expression length or decision complexity. Understanding the characteristics of rock star work can help software engineers improve their work. These concepts should be collected and put into educational materials so students can learn the techniques of the rock star programmers.

Chapter 14

Application of Discovered Concepts

This dissertation investigates many topics related to the characteristics of complicated software and reliability issues caused by software complicacy. Although this information is vital, it is also essential that these discoveries be put into software development practice. This chapter illustrates the concepts by summarizing actions that lead to less buggy and more reliable software.

14.1 Begin the Software Project With a Plan

It has been said that failure to plan is planning to fail [180]. Software planning begins with requirements gathering and architecture definition. Within this dissertation, Chapter 7 and Chapter 8 describe a straightforward and uncomplicated approach to requirements gathering and analysis, as well as a methodology for creating a software architecture.

14.2 Be Disciplined, Use a Coding Standard

As described in the preface, it is unethical to write undependable software knowingly, so software engineers must have discipline. In Chapter 9, a practical coding standard is defined that can be used to support and reinforce discipline on a project. The benefits of a coding standard are numerous, including easier maintenance, which leads to improved dependability.

14.3 Automate Code Reviews

Code reviews are an essential aspect of creating quality software. Research included in this dissertation provides multiple avenues for automating code reviews. I present three different concepts applicable to review automation.

14.3.1 Automation Using Metrics and Static Analysis

Metrics and static analysis are covered in great detail in Chapter 2 and all of the analysis was performed with free tools. Automation can give developers an edge to keep their projects from becoming buggy. Static analysis and metric calculation can show the level of complicacy in a source project and the propensity of bugs.

14.3.2 Automation Using Machine Learning

Chapter 6 demonstrates that repeated coding problems may be identified with tools built using machine learning. In the described application of machine learning, images of problematic coding blunders are made, analogous to images of tumors. Machine learning image recognition can identify these coding blunders in the same manner as identifying tumors in humans.

14.3.3 Automation Using Applied Linguistics

Chapter 5 shows that complicated statements and expressions can contribute to buggy software. Using tools based on applied linguistics makes it possible to identify code with excessive amounts of these complicated statements.

14.4 Evaluating Existing Code, Libraries, and Tools

Simply creating good source code in new projects is insufficient, it is essential that all the code included in a project is of good quality. Chapter 10 demonstrates how to take an already successful project and make it less complicated. Chapter 11 demonstrates how to evaluate tools and libraries which may be incorporated. Using these techniques, evaluate libraries and tools for the project to ensure they are not too complicated and too buggy.

Chapter 15

Conclusion

15.1 Outcomes to Research Questions

The motivation for this dissertation was to examine how the complicated nature of source code impact bugginess and ultimately reliability. At the beginning of this dissertation, six research questions were posed. An extensive amount of research was done, and source code analysis was completed to discover answers to these research inquiries. In this section, I summarize the answer to each question and provide references to different parts of the dissertation where specific information can be found.

15.1.1 How can we use traditional metrics and static analysis to foretell project bugginess and dependability?

As described extensively in Chapter 2, traditional metrics and static analysis can be used to foretell project bugginess. Chapter 2 describes how the various metrics are calculated, and their relationship to project bugginess. Also described is how static analysis works and its use in determining project complicacy.

15.1.2 How can software engineers write software with a lower probability of being buggy?

There are several actions that can be taken to proactively write software with a smaller probability of being buggy. These steps are summarized in Chapter 14 in some detail.

15.1.3 How can we apply principles from linguistics to measure complicacy?

A method for software analysis using techniques from linguists is described in Chapter 5. Zipf's Law of Vocabulary Balance and Zipf's Law of Least Effort can be applied to software to locate overly complicated parts of source code.

15.1.4 How can we apply machine learning to detect complicated parts of source code?

Machine learning presents ample opportunities for improving the code review process. In Chapter 6 the use of image processing is explored in detail demonstrating how to identify problematic code.

15.1.5 What steps can developers take to avoid complicacy and improve dependability when performing analysis and design workflows?

As described in this dissertation, the creation of less buggy and less complicated software begins with a good plan and good architecture. Actions that may be helpful to this end are described in Chapter 8.

15.1.6 How can we use knowledge from this dissertation to analyze existing works?

Chapter 10 shows in detail how to take a successful existing product and streamline it, making it less complicated.

15.2 Future Research

Three important topics illustrated in this dissertation show promise for valuable future research. Chapter 6, *Image Recognition for Code Reviews*, discusses how frequently made problematic coding errors can be automatically identified using machine learning. Further research is possible and recommended to better understand how to exploit the technology most suitably.

The second area that warrants further research is software applied linguistics, as described in Chapter 5. This chapter shows that programming languages have sufficient characteristics of spoken languages that more linguistics study of programming languages could be valuable. As with machine learning, there is ample opportunity in this area for further research.

As this thesis mainly focused on “C” based projects, future efforts should be made to expand these efforts to demonstrate applicability to programming languages. As there are many different programming languages for many different system architectures, many different avenues could be explored.

15.3 Closing Summary

This thesis expands software engineering knowledge regarding complicated source code and software bugginess. It begins by discussing ethical considerations of software development and the many ways software touches our lives, businesses, and scientific endeavors.

Exploration of methods for writing better software is essential. The thesis continues by exploring traditional code evaluation metrics and static analysis of code. Chapter 2 explains traditional code evaluation metrics and provides examples of bug trends applied to those metrics. An interesting observation from Chapter 2 is how more authors per line of code leads to more bugs. Chapter 3 presents a new tool for estimating coupling. This tool is straightforward to understand, and since it is written in Python, it is easy to expand to other languages or port to different environments. This makes it very practical. Coupling directly impacts a developer's ability to understand the code they are working on, and having a practical tool for estimating coupling will benefit software development efforts. Chapter 4 describes using static analysis tools. In this chapter, a survey was performed to see which programming constructs were most confusing to programmers. The results in Chapter 4 demonstrate how poorly formatted code takes more time to read and is misread more often. This information is used throughout the thesis.

Chapter 5 and Chapter 6 present new and novel ways of looking for code complicacy. Chapter 5 shows how techniques from linguists examining spoken languages can be applied to software source code. Using linguistics, we can determine the location of complicated source code in a project. This code can then be reviewed and then refactored if necessary. Alternatively, complicated code may be assigned to an engineer with appropriate experience. Chapter 6 discusses software complicacy identification using machine learning and image recognition. Using convolutional neural networks, a machine learning system may spot problematic aspects of code a human reviewer might miss.

One commonality between creating dependable code and dangerous public safety or military missions is proper preparation and planning. An intriguing concept in this type of planning and preparation is the importance of rehearsal for missing understanding. Chapter 7 introduces the concepts of planning and rehearsals to software development. Chapter 8 describes an approach to software architecture creation using rehearsals. Teaching developers the power of system walkthroughs via rehearsals will make it possible for them to create a more robust architectural design when creating new systems.

The thesis continues by offering strategies for how developers can avoid and cope with software complicacy. Within Chapter 9, an innovative new coding standard style is proposed to make compliance with organizational coding practices easier. This coding standard strives to separate easy-to-follow rules from necessary but overbearing aspects of coding standards. The chapter emphasizes the significance of the concept that hard-to-follow rules may not be followed. The thesis continues with Chapter 10 presenting the process of taking an existing product and making it less complicated. In Chapter 11 a case study demonstrates a path to find the least complicated and presumably most reliable software for a system. This chapter analyzed Java virtual machines, but the techniques described are broadly applicable. Continuing with the theme of existing products, Chapter 12 demonstrates how static analysis can be used to determine if existing code is satisfactory for a mission-critical application. This thesis then moves to apply concepts learned throughout to the work of software engineers considered masters of the art. In Chapter 13, the work of "Rock Star" programmers is measured and compared to mainstream work. Overall, Chapter 13 shows the code produced by the "Rock Star" programmer is generally less complicated than code produced by mainstream programmers.

Software complicacy does not simply cause software-related problems but also impacts human endeavors at many levels. This research ends by summarizing strategies for developers to avoid and cope with software complicacy. As a closing reflection, software authorship should be considered analogous to the authorship of books and articles. Natural language tools can help us write by identifying syntax and suggesting potential semantic errors. However, computer-based tools lack a human perspective and cannot actually identify whether a written paragraph is correct. Since software is dynamic and execution can take many paths, tools for automatically determining program correctness will be as challenging to create as tools for checking the correctness of natural language writing. A generic tool to perfectly determine software correctness is a difficult challenge. However, using techniques described in this paper, it is possible to suggest that a piece of software has a propensity for bugs and put effort into the potentially buggy code to improve it.

Appendix A

Common Values of Metrics

A.1 Important Metrics

Table A.1 presents values of important metrics, per project, in the study regardless of age. Projects smaller than one KLOC were excluded and outliers from other metrics were removed before calculations were made. The average age was approximately seven years.

Table A.1: Metrics of All Project

Metric	Minimum	Maximum	Mean	Median
Thousands of Lines of Code (KLOC)[1]	1.0	8359.1	212.5	48.2
Decision Complexity Per KLOC[1]	5.83	257.51	127.59	126.7
Efferent Coupling (Sheficom)[2]	0.000	67.17	22.01	18.76
Number of Authors Per KLOC[3]	0.0003	6.05	1.34	0.84
Number of Bugs Per KLOC[3]	0.00	36.94	8.86	5.9
Number of Bad Bugs Per KLOC[3]	0.00	0.88	0.15	0.047

[1] KLOC measurement made with Source Code Counter tool

[2] These values were based on the Sheficom tool. More information can be found in Appendix F

[3] These values were based on counts retrieved from Git log files [161]

A.2 Mature Projects

Table A.2 presents values of important metrics of projects older than seven years. As before, projects smaller than one KLOC were excluded, and outliers of other metrics were removed before calculations were made.

Table A.2: Metrics of Mature Projects

Metric	Minimum	Maximum	Mean	Median
Thousands of Lines of Code (KLOC)[1]	1.1	5745.19	291.1	89.9
Decision Complexity Per KLOC[1]	13.28	261.78	136.18	135.15
Efferent Coupling (Sheficom)[2]	0.031	60.33	20.73	18.53
Number of Authors Per KLOC[3]	0.017	6.64	1.48	0.99
Number of Bugs Per KLOC[3]	0.000	51.45	14.17	10.65
Number of Bad Bugs Per KLOC[3]	0.00	1.64	0.35	0.22

[1] Measurement made with Source Code Counter tool

[2] These values were based on the Sheficom tool. More information can be found in Appendix F

[3] These values were based on counts retrieved from Git log files [161]

Appendix B

Source Code from Survey

In this survey, developers were asked to review a some source code and immediately decide if it is pleasant or unpleasant to review.

The source code files as well as a CSV files of the overall results can be found here: `https://github.com/mikedorin/thesis_survey_1.git`

Appendix C

Open Source Projects Used

All projects were found on GitHub and downloaded on September 22, 2019. Each of the projects is listed below.

- 1store
- 2048.cpp
- 360Controller
- 3dgame-shaders-for-beginners
- acmchallenge-workbook
- actorframework
- Adafruit_NeoPixel
- AI4Animation
- AirSim
- albert
- aleth
- Algo_Ds_Notes
- Algojammer
- algorithms
- algorithms_and_data_structures
- AliceVision
- AliSQL
- alkhaser
- amazondsstne
- ammo.js
- anbox
- androidgpuimage-plus
- AndroidJniBitmapOperations
- AnimeEffects
- antimony
- AnyQ
- anyRTCRTMP-OpenSource
- aoapcbac2nd
- apitrace
- apkstudio
- appjs
- AppleALC
- appleseed
- arangodb
- ArcadeLearning-Environment
- arcoreandroid-sdk
- ArduinoIRremote
- ArduinoJson
- ardupilot
- arrayfire
- arrow
- aseprite
- asio
- asmdom
- asmjit
- assimp
- asyncprofiler
- AtomicGameEngine
- audiorouter
- AutoHotkey_L
- avian
- AwesomeBump
- awtk
- backwardcpp
- baiduCDP
- basis_universal
- beast
- benchmark
- benchmarks
- beringei
- bettersqlite3
- bfs
- bgslibrary
- bish
- blackbird
- Blackbone
- blazingsql
- BlingFire
- bloaty
- blynklibrary
- boden
- BOLT
- bond
- BootNTR
- botan
- botnets
- box2d
- bpftrace
- braft
- brigand
- bsf
- btfs
- bugs_text

- busy
- butteraugli
- bwapi
- caffewindows
- carla
- cartographer
- CataclysmDDA
- catboost
- ccls
- cefpython
- cereal
- ceressolver
- cgal
- cgdb
- ChaiScript
- Checkpoint
- cherrytree
- chineseocr_lite
- Cinder
- citra
- ckb
- ckbnext
- clasp
- Clementine
- client
- cling
- clip
- clivisualizer
- CNTK
- co
- cocos2djs
- code
- codelite
- CodingInterviewChinese2
- CodingInterviews
- compute
- ComputeLibrary
- concurrentqueue
- confluo
- conky
- coolqhttp-api
- CopyQ
- CPlus-Plus
- Cplusplus-Concurrency-In-Practice
- cplusplus-_Implementation_Of_Introduction_to_Algorithms
- CPlusPlusThings
- cppcheat-sheet
- cppcheck
- CppCon2014
- CppCon2015
- CppCon2016
- CppCon2018
- cpphttplib
- cppinsights
- cppjieba
- CppPatternsPatterns
- CppPrimer
- CppPrimer
- cpprestsdk
- cpptaskflow
- CppTemplateTutorial
- cpr
- cquery
- crackingthe-coding-interview
- crawl
- crow
- cryptopp
- csmith
- CS_Offer
- cuberite
- cuml
- CuteMarkEd
- cvxpy
- CxbxReloaded
- cxxopts
- DALI
- date
- dbgmacro
- deepdetect
- DeepImage-Analogy
- deepinwine-ubuntu
- DeepMimic
- deletes
- design_patterns
- DeskGap
- Detours
- devilutionX
- dexed
- dexui
- DHTsensor-library
- differentialprivacy
- DirectXGraphics-Samples
- DirectXShaderCompiler
- DirectXTK
- dnscat2
- Dobby
- doctest
- dogecoin
- domoticz
- DOOM3-BFG
- doom3.gpl
- doxygen
- draco
- drake
- drogon
- duilib
- dumpDex
- dxvk
- DynamicAPK
- dynet
- earthenterprise
- EAST
- EASTL
- easyloggingpp
- EasyPR
- easy_profiler
- edbdebugger
- edge
- electron
- ELF
- ELL
- elliptics
- embree
- EmulationStation
- encfs
- endlesssky
- engine
- entityx
- entt
- eosio.cdt
- EpicSurvivalGameSeries
- EPIJudge
- ESPEasy
- esphome
- Espradio
- espurna
- ethminer
- euler
- eurorack
- evpp
- falco
- Familia
- fann
- FastLED
- fastnetmon
- faust
- fbthrift
- feather
- fetlang
- fibjs
- Fido
- firesheep
- Firmware
- fivem
- flameshot
- flamingo
- flann
- flat_hash_map

- FLIF
- freeablo
- FreeCAD
- FRIEND
- fritzingapp
- fr_public
- fswatch
- g2o
- GAAS
- GacUI
- GameNetworking-Resources
- GameNetworkingSockets
- GamePlay
- gdal
- GDevelop
- gemmlowp
- gflags
- ggpo
- ghostwriter
- GildedRose-Refactoring-Kata
- gitcrypt
- gitmosts
- glog
- glow
- glsloptimizer
- gnuradio
- goldendict
- gperftools
- gqrx
- griddb_nosql
- grive
- grpcweb
- GSL
- GuiLite
- halflife
- Halide
- halley
- HandlerSocketPlugin-for-MySQL
- handy
- Hardcoder
- hardwareeffects
- harfbuzz
- Hazel
- hed
- heif
- HElib
- helioworkstation
- hermes
- hex
- HexRaysCodeXplorer
- HIP
- Hippy
- HoRNDIS
- hotspot
- hpx
- HyperDex
- hyperion
- HyperLandmark
- HyperLPR
- hyperscan
- i2cdevlib
- i2pd
- ice
- icinga2
- icomet
- identifier_textfiles
- iisnode
- ikos
- immer
- Impala
- IncludeOS
- includewhat-you-use
- incubatordoris
- incubatormxnet
- incubatorpagespeed-ngx
- InfinityHook
- instantmeshes
- interpret
- ion
- IRremoteESP8266
- ispc
- iverilog
- jack2
- jetsoninference
- jittor
- json
- jsoncpp
- JUCE
- kaggle2014-criteo
- Kaggle_CrowdFlower
- kakoune
- kalibr
- kaolin
- Karabiner
- katran
- KB2E
- kbdaudio
- kcws
- keepassx
- kenlm
- KeyDB
- keystone
- kids
- KlayGE
- klee
- krita
- Kryptotrading-bot
- kudu
- kungfu
- kurentomedia-server
- LaiNES
- LAVFilters
- LearningOpenCV-3_examples
- LearnOpenGL
- ledger
- leelazero
- LeetCode
- LeetCodeSolutions
- lepton
- libcds
- libchaos
- libco
- libffm
- libfm
- libfreenect2
- libgo
- libigl
- libquic
- librealsense
- LibreCAD
- librime
- libsass
- libtins
- libtorrent
- libuinode
- libzmq
- licode
- LIEF
- lilliput
- Lilu
- liquidfun
- liteide
- LiveVideoCoreSDK
- llilc
- llvmproject
- lmctfy
- lmms
- lnav
- lodepng
- logcabin
- LogDevice
- Logstalgia
- lsd_slam
- ltp
- LumixEngine
- lux
- LxRunOffline
- mace
- MachOView

- mactype
- magnum
- mailcore2
- maim
- mame
- mapboxgl-native
- maplab
- mapnik
- MarbleMarcher
- matplotlibcpp
- mcrouter
- mediasoup
- MegEngine
- meshlab
- meshoptimizer
- mesos
- microsoftpdb
- milesdeep
- milvus
- mindforger
- minetest
- miniblink49
- minicap
- minigo
- MITIE
- mixxx
- mlpack
- MNN
- moderncpp-tutorial
- moderngpu
- ModSecurity
- monero
- msgpackc
- MTuner
- MulticoreTSNE
- MultiMC5
- multipass
- multiwiifirmware
- mumble
- MuseScore
- musikcube
- Mutate
- MVision
- mysql
- mysql5.6
- mysqlserver
- MyTinySTL
- namecoinlegacy
- nan
- nana
- nanogui
- nanonode
- napajs
- Natron
- nbind
- nccl
- ncmdump
- nebula
- nethogs
- neutralinojs
- newstuff
- nghttp2
- ngraph
- ninja
- nix
- nmslib
- NoahGameFrame
- nodeaddon-examples
- node.bcrypt.js
- nodefibers
- nodegui
- nodejava
- nodememwatch
- node.native
- nodeopencv
- nodepacker
- nodeqt
- nodewebrtc
- nodewebworker-threads
- NonEuclidean
- notbusy
- note
- notepadqq
- nppPluginManager
- nsjail
- NymphCast
- oatpp
- oboe
- OceanProject
- oclint
- ogl
- ogre
- olive
- omim
- omniscidb
- oneDNN
- oneTBB
- onnxruntime
- oomd
- Open3D
- openalpr
- openauto
- openbr
- OpenCC
- opencv
- OpenCV3Intro-Book-Src
- opencv4nodejs
- opencv_contrib
- openh264
- OpenJK
- OpenMQTTGateway
- openMVG
- openmw
- opennsynth-super
- OpenPano
- openrasp
- openscad
- OpenSceneGraph
- opensource-search-engine
- open_spiel
- OpenSubdiv
- openthread
- opentoonz
- opentrack
- OpenTTD
- opentx
- openvino
- openvr
- openvslam
- ORB_SLAM2
- original_busy_notbusy
- ortools
- oryol
- osrmbackend
- otterbrowser
- OTTO
- PacketSender
- PacVim
- PaddleLite
- panda3d
- passenger
- PAT
- pbrtv3
- pcl
- pcsx2
- pdns
- pdqsort
- pegasus
- pesieve
- pgmodeler
- PhoenixGo
- PHPCPP
- phpdesktop
- phxpaxos
- phxqueue
- phxrpc
- phxsql
- PhysX
- PhysX3.4
- picotorrent
- pika

- pistache
- PJON
- plaidml
- PlayLeetcode
- Polycode
- ppsspp
- ProcessHider
- protobufc
- proxysql
- PrusaSlicer
- pubsubclient
- pugixml
- pulseeffects
- pushpin
- pydensecrf
- Pyjion
- pytorch
- pywin32
- qBittorrent
- QConf
- QGIS
- qgroundcontrol
- QOwnNotes
- QtAV
- qtcreator
- qTox
- QtScrcpy
- QuantLib
- Qv2ray
- Rack
- rainmeter
- rangev3
- rathena
- RawTherapee
- rcswitch
- re2
- readerwriterqueue
- recastnavigation
- recipes
- recompiler
- redex
- RedisStudio
- RefineDet
- Relativ
- renderdoc
- reshade
- retdec
- rethinkdb
- RetroShare
- Revive
- RF24
- rfid
- rhino
- ricochet
- rippled
- robotstxt
- root
- RosettaStone
- rpirgb-led-matrix
- rr
- rtags
- rtorrent
- rttr
- runtime
- RuntimeCompiledCPlusPlus
- rustbindgen
- RxCpp
- s2clientapi
- s3fsfuse
- screencapture-recorder-to-video-windows-free
- scribe
- scummvm
- scylla
- ScyllaHide
- sdsllite
- SEAL
- seastar
- SeetaFace2
- SeetaFaceEngine
- selfdriving-car
- sentencepiece
- serenity
- SeriousEngine
- server
- servicefabric
- serving
- SFML
- SHADERed
- shogun
- shotcut
- Sigil
- silicon
- simbody
- singa
- slambook
- Slic3r
- sling
- SmartDeblur
- SmartOpenCV
- Sming
- snap
- snapcast
- snappy
- snowboy
- socket.ioclient-cpp
- sofapbrpc
- sol2
- solvespace
- sourcesdk-2013
- spectrum
- sphinx
- spring
- SPTAG
- sqlcheck
- sqlyogcommunity
- Squirrel.Windows
- srsLTE
- ssr
- Stacer
- StarSpace
- statements
- steem
- stellarium
- stepmania
- stkcode
- STL
- Stockfish
- StreetMap
- subconverter
- subsurface
- supercollider
- SuperWeChatPC
- Surround360
- SwarmUI
- swift
- swig
- sysdig
- taichi_mpm
- taiga
- tair
- Tars
- tclip
- tcpflow
- td
- teeworlds
- TelegramGallery
- Tengine
- TensorComprehensions
- tensorflowopencl
- TensorRT
- tera
- terminal
- termite
- terra
- tesseract
- TheForge
- TheOpen-Book
- ThePowder-Toy
- therubyracer
- ThreadPool
- thrust
- thundersvm

- tigervnc
- tink
- tinydnn
- tinyfecVPN
- tinykaboom
- tinyobjloader
- tinyraytracer
- tinyrenderer
- tinyxml2
- tippecanoe
- toggldesktop
- ToGL
- tomahawk
- tools
- TorchCraft
- Torque2D
- Torque3D
- toy
- trafficserver
- TranslucentTB
- TrinityCore
- Triton
- tritoninference-server
- tungsten
- TurboDex
- twistercore
- two
- typesense
- udp2rawtunnel
- UDPspeeder
- UEFITool
- uncrustify
- unfork
- unrealcv
- Urho3D
- USD
- uTensor
- uWebSockets.js
- v8js
- valhalla
- vcmi
- verge
- verona
- VINSFusion
- VINSMono
- visualboyadvancem
- vogl
- VTK
- Vulkan
- VulkanTutorial
- wabt
- waifu2xcaffe
- Waifu2xExtension-GUI
- wangle
- wav2letter
- WAVM
- WaykiChain
- wdt
- webdsp
- WebServer
- websocketpp
- WeChatRobot
- wesnoth
- WhateverGreen
- WickedEngine
- WiFiManager
- WikiSort
- wil
- Win2D
- WLED
- wxWidgets
- xcbuild
- xenia
- xeuscling
- xlearn
- xmrig
- xmrstak
- xournalpp
- Xposed
- xray16
- xtensor
- yamlcpp
- ycmd
- Yolo_mark
- yue
- yuzu
- Z0FCourse_ReverseEngineering
- z3
- zcash
- zeek
- Zero0ad
- ZeroTierOne
- zetasql
- zindex
- ZLMediaKit
- znc
- zopfli
- zynaddsubfx

Appendix D

Instant Rodos

Instant Rodos may be found here.

```
https://dori3417@bitbucket.org/dori3417/instant-rodos.git
```

Appendix E

Educational Play

E.1 Example Play, The Happy Class

Table E.1: Happy Class Play Characters

Name	Character Title	Responsibility
Claudia	File Clerk	Storing and retrieving data
Diego	Student	A student using the system (Human)
Gonzalo	Security	Provides user validation
Patrick	Professor	A professor using the system (Human)
Rebecca	Controller (Boss)	Manages software operations
Sergio	Operator	Provides internal communications
Valeria	Artist	Generates output to users

Setting

Three Software People (Valeria, Rebecca, Patrick) sit around piles of paper showing user stories and use cases. A low-resolution prototype is taped to the wall. Cups are full of coffee. The three are very pleased with the quantity and quality of their requirements gathering and analysis but harbor some doubts with respect to moving to the next phase.

Narration

Valeria: This is sure good coffee; do you think we have enough?

Rebecca: I hope not, I want to wrap up and go home … but now what?

Patrick: Now we have to come up with a candidate architecture, but where do we begin? We have gathered so much information and talked to so many people. We even have fantastic low-resolution user interface prototypes.

Valeria: Perhaps we start small. Rebecca, please grab me a minor use case.

Rebecca: (Rebecca finds a minor use case and gives it to Valeria.)

Valeria: Ok, I have the "Professor Logs In" use case, but it is still not obvious how to continue.

Rebecca: I know, let us pretend to be the software. Perhaps we can get an idea of how to construct this thing!!

Patrick: Don't be silly.

Rebecca: Wait, wait, let us just give it a try and see where it takes us.

Patrick: Ok, I will pretend to be the Professor in the use case. Rebecca, you be the Software.

Patrick: I'll start.

Patrick: Hola 'Rebecca,' I want to set up a class.

Rebecca: I am not sure what you want or how to help. Valeria, can you show him what he can do?

Valeria: Ok, I will pretend to be in charge of showing stuff.

Valeria: Ok, here are your options. (Valeria shows Patrick a sheet of paper. Patrick pretends his options are written on it.)

Patrick: I think this is getting closer, but rather than calling you by name, I am going to call you by a title to help this stay organized. I will be the Professor. Rebecca, you seem to be the boss so that I will call you "Controller." Valeria, you seem to be a communicator, so I will call you Artist.

Professor: Hola, Controller, I want to set up a class.

Controller: Artist, please show this Professor their options.

Artist: Professor, welcome, here is our main screen. Professors need to sign in.

Controller: STOP! I don't know how people authenticate. I just know how to boss people around. We need somebody like a 'Security' to handle this. (Just then they notice Gonzalo sitting in the corner.)

Patrick: Hey Gonzalo, come over here for a second. We need you to be a Security in our software world.

Gonzalo: Hey, a Security, wow that sounds like fun.

Patrick: Ok, let us continue again, remember Rebecca is the Controller.

Professor: Hey, Controller, I want to set up a class.

Controller: Artist, please show this Professor their options.

Artist: Professor, welcome, here is our main screen. Professors need to sign in.

Professor: Security, here is my info.

Security: Controller, the Professor, has signed in

Controller: Artist, please show the Professor how to create a class.

Artist: Professor, please provide a class name and let us know when you are ready to start.

Professor: Controller, I want to start a class named, SEIS610. It is my favorite class, and a fantastic Professor teaches it.

Rebecca: STOP. I don't know how to create a class. We need somebody to keep track of all of this. Wait, Claudia, come in here. Can you pretend to be an File Clerk for us? Please. Just for a bit. Ok, Claudia, when I call out File Clerk, you answer.

Claudia: Do I have to call you Controller??

Rebecca: Yes, you do.

Rebecca: Ok, now let us continue

Controller: File Clerk, please create a class named SEIS610.

File Clerk: Ok, Controller, here is that class you wanted.

Rebecca: Stop again. Ok, Claudia has created a class for us but how do I talk to it. Claudia does not know anything about talking to classes. It's like, we need, a telephone operator. Sergio, come in here a second. We need you to pretend to be a telephone operator.

Sergio: Do I have to?

Rebecca: Yes, you do, and yes you need to call me Controller.

Rebecca: Alright, let us continue again.

Controller: Operator, can you please give me a new channel.

Operator: Why yes, Controller, here you go.

Controller: File Clerk, hey, sorry to bug you again, but can you store this channel information.

File Clerk: Controller, consider it done!

Controller: Artist, can you show the Professor a classroom based on their new class with this id and channel information?

Artist: Professor, here is your class, make sure you tell your students the id 1234!

(Diego walks by. . . decides to be funny and pretends to be a student.)

Diego: Hey, look at me, I am a student!

Student: Hola, Controller, I want to participate in class with the ID of 1234

Controller: File Clerk, do we have a class with the ID of 1234? If so can you give it to me?

File Clerk: Controller, yes, we do have that class. Here is the info.

Controller: Artist, can you display a class to this student with this info?

Artist: Student, here is your class

Student: Controller, can you tell the Professor that I don't understand the problem just described?

Controller: File Clerk, what is the channel that class 1234 uses?

File Clerk: Class 1234 uses this channel.

Controller: Operator, can you relay this question via this channel to the Professor.

Operator: Yes, I will, and I have.

Patrick: Well gang, I think this gives us a fascinating picture!

The end!

E.2 Play UML

E.2.1 Sequence Diagram

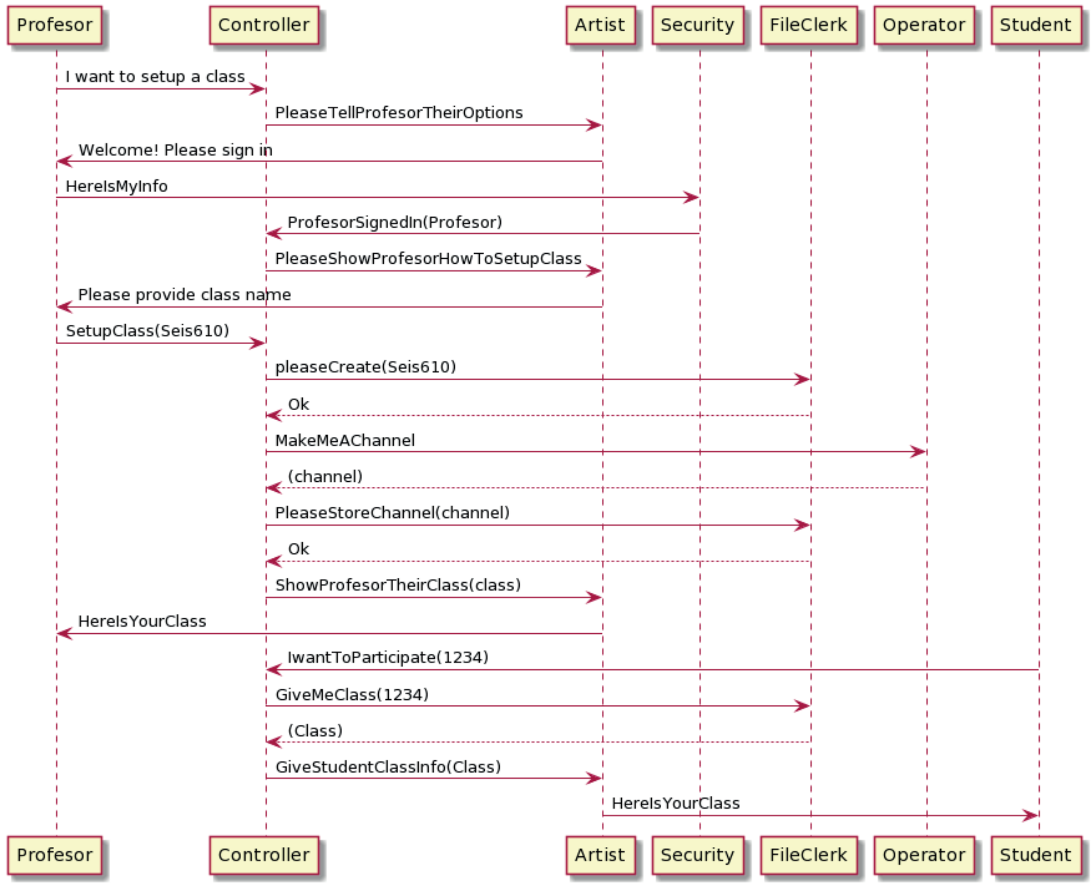


Figure E.1: Happy Class Example Sequence Diagram

E.2.2 Class Diagram

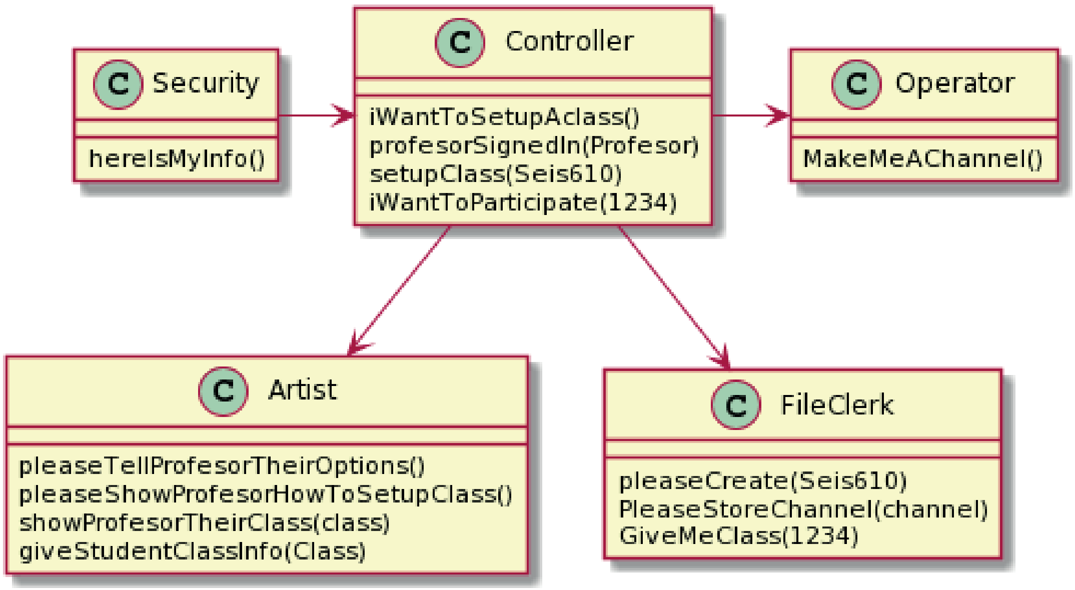


Figure E.2: Happy Class Example Class Diagram

E.3 Play Python Source

Source code for The Happy Class can be found here:

```
gitclonegit@bitbucket.org:dori3417/happy-class.git
```

Appendix F

Sheficom (Efferent Coupling)

F.1 Sheficom Efferent Coupling Tool

Download from here:

```
https://github.com/mikedorin/sheficom.git
```

Bibliography

[1] Nuclear Energy Agency. *Collection and Analysis of Common-Cause Failures due to Nuclear Power Plant Modifications*. Mar. 2020. URL: `http://www.oecd.org/officialdocument`.

[2] Syed Nadeem Ahsan and Franz Wotawa. "Fault prediction capability of program file's logical-coupling metrics". In: *2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*. IEEE. 2011, pp. 257–262.

[3] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. "Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes". In: *arXiv preprint arXiv:1711.04325* (2017).

[4] F Akiyama. "An example of software system debugging. IFIP Congress 71, Vortragsauszüge". In: *Computer Software* (1971), pp. 37–42.

[5] Donald J Albers and Lynn Arthur Steen. "Biographies [A Conversation with Don Knuth]". In: *Annals of the History of Computing* 4.3 (1982), pp. 257–273.

[6] Eijaz Allibhai. *Building a Convolutional Neural Network (CNN) in Keras*. Accessed: 2022-07-30. Oct. 2018. URL: `https://towardsdatascience.com/building-a-convolutional-neural-network-cnn-in-keras-329fbbadc5f5`.

[7] Javier Alonso et al. "The nature of the times to flight software failure during space missions". In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE. 2012, pp. 331–340.

[8] Yoram Amiel and Frank Cowell. "Monotonicity, dominance and the Pareto principle". In: *Economics Letters* 45.4 (1994), pp. 447–450.

[9] Idan Amit and Dror G. Feitelson. *The Corrective Commit Probability Code Quality Metric*. 2020. arXiv: `2007.10912 [cs.SE]`.

[10] US Army. *FM 6-0 Commander and staff organization and operations*. 2015.

[11] Andrea C. ArpaciDusseau and Remzi H. ArpaciDusseau. *Operating Systems: Three Easy Pieces*. 2022. URL: `https://pages.cs.wisc.edu/~remzi/OSTEP/`.

[12] A. Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. DOI: `10.1109/TDSC.2004.2`.

[13] Nathaniel Ayewah et al. “Using Static Analysis to Find Bugs”. In: *IEEE Software* 25.5 (2008), pp. 22–29. DOI: 10.1109/MS.2008.130.

[14] Bence Babati et al. “Static analysis toolset with Clang”. In: *Proceedings of the 10th International Conference on Applied Informatics (30 January–1 February, 2017, Eger, Hungary)*. 2017, pp. 23–29.

[15] Lia Bally et al. “Closed-Loop Insulin Delivery for Glycemic Control in Noncritical Care”. In: *New England Journal of Medicine* 379.6 (Aug. 2018), pp. 547–556. ISSN: 0028-4793. DOI: 10.1056/NEJMoa1805233. URL: http://www.nejm.org/doi/10.1056/NEJMoa1805233.

[16] Rajiv D. Banker et al. “Software Complexity and Maintenance Costs”. In: *Commun. ACM* 36.11 (Nov. 1993), pp. 81–94. ISSN: 0001-0782. DOI: 10.1145/163359.163375. URL: http://doi.acm.org.ezproxy.stthomas.edu/10.1145/163359.163375.

[17] Gabriele Bavota and Barbara Russo. “Four eyes are better than two: On the impact of code reviews on software quality”. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2015, pp. 81–90.

[18] Michael J Behe. “Irreducible complexity: Obstacle to Darwinian evolution”. In: *Philosophy of Biology: An Anthology* 32 (2009), p. 427.

[19] Mordechai Ben-Ari. “The bug that destroyed a rocket”. In: *ACM SIGCSE Bulletin* 33.2 (2001), pp. 58–59.

[20] Al Bessey et al. “A few billion lines of code later: using static analysis to find bugs in the real world”. In: *Communications of the ACM* 53.2 (2010), pp. 66–75.

[21] Al Bessey et al. “A few billion lines of code later: using static analysis to find bugs in the real world”. In: *Communications of the ACM* 53.2 (2010), pp. 66–75.

[22] Pavol Bielik, Veselin Raychev, and Martin Vechev. “Learning a static analyzer from data”. In: *International Conference on Computer Aided Verification*. Springer. 2017, pp. 233–253.

[23] Malaya Kumar Biswal M and Ramesh Naidu Annavarapu. “A Study on Mars Probe Failures”. In: *AIAA Scitech 2021 Forum*. 2021, p. 1158.

[24] Jean-Louis Boulanger. *CENELEC 50128 and IEC 62279 standards*. John Wiley & Sons, 2015.

[25] Ben Boyer. *A tool for counting lines of code, blank lines, comment lines, and physical lines of source code in many programming languages*. https://github.com/boyter/scc.git. Accessed 9-2021. 2018.

[26] Annika Brändström. *Coping with a Credibility Crisis: The Stockholm JAS Fighter Crash of 1993*. Försvarshögskolan, 2001.

[27] Robert W Brill. “Instrumentation And Control System Failures In Nuclear Power Plants”. In: *International Symposium on Software Reliability Engineering, San Jose, CA*. Citeseer. 2000.

[28] Benjamin Brosgol. “Do-178c: The next Avionics Safety Standard”. In: *Proceedings of the 2011 ACM Annual International Conference on Special Interest Group on the Ada Programming Language*. SIGAda ’11. Denver, Colorado, USA: Association for Computing Machinery, 2011, pp. 5–

6. ISBN: 9781450310284. DOI: 10.1145/2070337.2070341. URL: https://doi.org/10.1145/2070337.2070341.

[29] Leo Broukhis, Simon Cooper, and Landon Curt Noll. “The International Obfuscated C Code Contest”. In: *IOCCC,[Online], Available: http://www. ioccc. org/years. html [Accessed 27 05 2022]* (2022).

[30] Bugzilla. *Bugzilla, Installation List*. 2018. URL: https://www.bugzilla.org/installation-list/ (visited on 03/28/2018).

[31] Zubaidah Bukhari, Jamaiah Yahaya, and Aziz Deraman. “Software metric selection methods: A review”. In: *2015 International Conference on Electrical Engineering and Informatics (ICEEI)*. IEEE. 2015, pp. 433–438.

[32] Martin Campbell-Kelly. “Dennis Ritchie obituary”. In: *The Guardian* (2011).

[33] Davide G Cavezza et al. “Reproducibility of environment-dependent software failures: An experience report”. In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE. 2014, pp. 267–276.

[34] Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007.

[35] William S Cleveland and Susan J Devlin. “Locally weighted regression: an approach to regression analysis by local fitting”. In: *Journal of the American statistical association* 83.403 (1988), pp. 596–610.

[36] Ron Coleman and Brendon Boldt. “Aesthetics versus entropy in source code”. In: *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*. The Steering Committee of The World Congress in Computer Science, Computer … 2017, pp. 113–119.

[37] Douglas Comer. *Operating system design: the Xinu approach, Linksys version*. Chapman and Hall/CRC, 2011.

[38] Douglas Comer. *The XINU Page*. Accessed: 2022-6-20. URL: https://xinu.cs.purdue.edu/.

[39] Manuel Cruz and Clyde Chadwick. “A Mars Polar Lander Failure Assessment”. In: *Atmospheric Flight Mechanics Conference*. 2000, p. 4118.

[40] Kajaree Das and Rabi Narayan Behera. “A survey on machine learning: concept, algorithms and applications”. In: *International Journal of Innovative Research in Computer and Communication Engineering* 5.2 (2017), pp. 1301–1309.

[41] *Debian Linux*. Accessed: 2020-10-20. URL: http://www.debian.org/.

[42] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[43] Peter J Denning. *Oral history interview with Peter J. Denning*. 2013. URL: https://conservancy.umn.edu/bitstream.

[44] *Dexcom Product Guides - User Guides, Quick Start, Tutorials*. https://www.dexcom.com/guides. Accessed: 2020-01-20.

[45] Jonathan. Dorf. *Playwriting 101 How to Write a Play*. 2018. URL: http://www.playwriting101.com.

[46] M. Dorin and S. Montenegro. "Eliminating Software Caused Mission Failures". In: *IEEE Aerospace Conference Proceedings*. Vol. 2019-March. 2019. ISBN: 9781538668542. DOI: 10.1109/AERO.2019.8741837.

[47] Michael Dorin. "Building" instant-up" real-time operating systems-Here are three ways to build an instant" up and running" RTOS for use on any target system requiring only some compilation and minimal hardware resources." In: *Embedded Systems Design* 21.5 (2008), p. 18.

[48] Michael Dorin. "Coding for Inspections and Reviews". In: New York, NY, USA: ACM, 2018. DOI: 10.1145/3234152.3234159.

[49] Michael Dorin and Sergio Montenegro. "Coding Standards and Human Nature." In: *International Journal of Performability Engineering* 14.6 (2018).

[50] Comer Douglas. *Operating System Design: The XINU approach*. 1984.

[51] Michael Drew. *Personal Communication*. 2018.

[52] Michael L. Drew. *A Coding Standard for the C Programming Language*. Oct. 22, 1999.

[53] David Eaton. *Answer To: Do all programmers follow coding standards?* 2014. URL: https://www.quora.com/Do-all-programmers-follow-coding-standards.

[54] Felipe Ebert et al. "Confusion detection in code reviews". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2017, pp. 549–553.

[55] Mustafa Efendioglu, Alper Sen, and Yavuz Koroglu. "Bug prediction of systemc models using machine learning". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.3 (2018), pp. 419–429.

[56] Pär Emanuelsson and Ulf Nilsson. "A Comparative Study of Industrial Static Analysis Tools". In: *Electronic Notes in Theoretical Computer Science* 217 (2008). Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008), pp. 5–21. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2008.06.039. URL: https://www.sciencedirect.com/science/article/pii/S1571066108003824.

[57] Arthur Emidio. *Arthur Emidio JVM*. Mar. 2016. URL: https://github.com/ArthurEmidio/jvm.

[58] Muhammad Faisal, Atheel Redah, and Sergio Montenegro. "DESIGN OF A COMPACT ACADEMIC COURSE FOR SATELLITE NAVIGATION AND CONTROL WITH A REAL-TIME OPERATING SYSTEM". In: *INTED2016 Proceedings*. IATED. 2016, pp. 4886–4895.

[59] Nargis Fatima, Sumaira Nazir, and Suriayati Chuprat. “Understanding the Impact of Feedback on Knowledge Sharing in Modern Code Review”. In: *2019 IEEE 6th International Conference on Engineering Technologies and Applied Sciences (ICETAS)*. IEEE. 2019, pp. 1–5.

[60] FDA. *Medical Device Recalls*. `https://www.accessdata.fda.gov`. Accessed: 2021-1-18. 2021.

[61] *FDA - Product Classification*. URL: `https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfPCD/classification.cfm?ID=OYC`.

[62] Glenn Fleishman. *In space, no one can hear you kernel panic – increment: Software architecture*. Feb. 2020. URL: `https://increment.com/software-architecture/in-space-no-one-can-hear-you-kernel-panic/`.

[63] U.S. Food and Drug Administration. *Becton Dickinson (BD) CareFusion 303 Inc. Recalls Alaris System Infusion Pumps Due to Software and System Errors*. `https://www.fda.gov/medical-devices/medical-device-recalls/becton-dickinson-bd-carefusion-303-inc-recalls-alaris-system-infusion-pumps-due-software-and-system`. Accessed: 2021-1-18.

[64] Brian Foote and Joseph Yoder. “Big ball of mud”. In: *Pattern languages of program design* 4 (1997), pp. 654–692.

[65] Matthieu Foucault et al. “Impact of developer turnover on quality in open-source software”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 829–841.

[66] Free Software Foundation. *GSL-GNU Scientific Library*. ”Accessed 2022-6-25”. URL: `https://www.gnu.org/software/gsl/`.

[67] *FoxTelem Software for Windows, Mac, and Linux*. Aug. 2018. URL: `https://www.amsat.org/foxtelem-software-for-windows-mac-linux/`.

[68] Yasao M Funami H Halstead. *A Software Physics Analysis of Akiyama’s Debugging Data*. Tech. rep. 1975. URL: `https://docs.lib.purdue.edu/cstech/93`.

[69] Mike Gaines. “Software fault caused Gripen crash”. In: *Flight International* (1989).

[70] Simson Garfinkel. “History’s worst software bugs”. In: *Wired News, Nov* (2005).

[71] *GitHub*. Aug. 2018. URL: `https://www.github.com`.

[72] GitHub. *GitHub, Inc.* https://github.com/. Accessed: 2020-01-20. 2020.

[73] Google. *Google Scholar*. Accessed: 2020-10-20. URL: `http://scholar.google.com/`.

[74] Michael Grottke, Allen P Nikora, and Kishor S Trivedi. “An empirical investigation of fault types in space mission system software”. In: *2010 IEEE/IFIP international conference on dependable systems & networks (DSN)*. IEEE. 2010, pp. 447–456.

[75] The HotSpot Group. *HotSpot*. 2018. URL: `http://openjdk.java.net/groups/hotspot/`.

[76] Vivien GUEANT. *Old Versions of Linux*. Accessed: 2020-10-20. URL: `https://soft.lafibre.info/`.

[77] Philip Guo. *10 Most Popular Programming Languages In 2018: Learn To Code*. July 2014. URL: `https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext`.

[78] Anshul Gupta and Neel Sundaresan. "Intelligent code reviews using deep learning". In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*. 2018.

[79] Roger Hall. *Writing your first play*. Focal Press, 2012.

[80] Maurice Howard Halstead et al. *Elements of software science*. Vol. 7. Elsevier New York, 1977.

[81] Till Harbaum. *NanoVM*. June 2006. URL: `http://www.harbaum.org/till/nanovm/index.shtml`.

[82] Gerard J Holzmann. "The Power of Ten–Rules for Developing Safety Critical Code1". In: *Software Technology: 10 Years of Innovation in IEEE Computer* (2018).

[83] ISO. *Road vehicles – Functional safety*. Norm. 2011.

[84] Chris W Johnson. "The natural history of bugs: Using formal methods to analyse software related failures in space missions". In: *International Symposium on Formal Methods*. Springer. 2005, pp. 9–25.

[85] Phillip Johnston and Rozi Harris. "The Boeing 737 MAX saga: lessons for software organizations". In: *Software Quality Professional* 21.3 (2019), pp. 4–12.

[86] Natsuda Kasisopha and Panita Meananeatra. "Applying ISO/IEC 29110 to ISO/IEC 62304 for Medical Device Software SME". In: *Proceedings of the 2nd International Conference on Computing and Big Data*. ICCBD 2019. Taichung, Taiwan: Association for Computing Machinery, 2019, pp. 121–125. ISBN: 9781450372909. DOI: `10.1145/3366650.3366670`. URL: `https://doi.org/10.1145/3366650.3366670`.

[87] Prabhjot Kaur. "A Review of Software Metric and Measurement". In: *International Journal of Computer Applications & Information Technology* 9.2 (2016), p. 187.

[88] *Keras: Image Data Processing*. `https://keras.io/preprocessing/image/`. Accessed: 21-7-2020. 2020.

[89] *Keras: the Python deep learning API*. `https://keras.io/`. Accessed: 21-7-2020. 2020.

[90] Brian W Kernighan and Phillip James Plauger. *Elements of programming style*. McGraw-Hill, Inc., 1974.

[91] Donald Knuth. *The Art of Programming*. Accessed: 2022-5-30. URL: `https://github.com/thornto4/knuth`.

[92] Donald E Knuth. *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX–A RISC Computer for the New Millennium*. Addison-Wesley Professional, 2005.

[93] Donald Ervin Knuth. "Literate programming". In: *The computer journal* 27.2 (1984), pp. 97–111.

[94] Donald Ervin Knuth and Silvio Levy. *The CWEB system of structured documentation*. Addison-Wesley Professional, 1993.

[95] Aaron Kozbelt et al. "The aesthetics of software code: A quantitative exploration." In: *Psychology of Aesthetics, Creativity, and the Arts* 6.1 (2012), p. 57.

[96] Jet Propulsion Laboratory. *JPL Institutional Coding Standard for the C Programming Language*. 800 Oak Grove Dr, Pasadena, CA 91109: Jet Propulsion Laboratory, 2009.

[97] John M Lachin et al. "Impact of C-peptide preservation on metabolic and clinical outcomes in the Diabetes Control and Complications Trial". In: *Diabetes* 63.2 (2014), pp. 739–748.

[98] Martin Langer and Jasper Bouwmeester. *Reliability of cubesats-statistical data, developers' beliefs and the way forward*. 2016.

[99] Craig Larman, Philippe Kruchten, and Kurt Bittner. "How to fail with the rational unified process: Seven steps to pain and suffering". In: *Valtech Technologies & Rational Software* (2001).

[100] Inhwan Lee and Ravishankar K Iyer. "Software dependability in the Tandem GUARDIAN system". In: *IEEE Transactions on Software Engineering* 21.5 (1995), pp. 455–467.

[101] Valentina Lenarduzzi and Davide Taibi. "Mvp explained: A systematic mapping study on the definitions of minimal viable product". In: *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2016, pp. 112–119.

[102] Nancy G Leveson. "Role of software in spacecraft accidents". In: *Journal of spacecraft and Rockets* 41.4 (2004), pp. 564–575.

[103] *Liberate your diabetes data — Tidepool*. `https://www.tidepool.org/`. Accessed: 2020-01-20.

[104] Chern-Sheng Lin et al. "The application of deep learning and image processing technology in laser positioning". In: *Applied Sciences* 8.9 (2018), p. 1542.

[105] *LoopKit/Loop: An automated insulin delivery app template for iOS, built on LoopKit*. `https://github.com/LoopKit/Loop`. Accessed: 2020-01-20.

[106] Navin Kumar Manaswi. "Understanding and working with Keras". In: *Deep Learning with Applications Using Python*. Springer, 2018, pp. 31–43.

[107] André Martinet. "Économie des changements phonétiques, Berne, A". In: *Francke* 396 (1955), pp. 2–15.

[108] Muhammad Mateen et al. “Fundus image classification using VGG-19 architecture with PCA and SVD”. In: *Symmetry* 11.1 (2019), p. 1.

[109] T. J. McCabe. “A Complexity Measure”. In: *IEEE Trans. Softw. Eng.* 2.4 (July 1976), pp. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837. URL: http://dx.doi.org/10.1109/TSE.1976.233837.

[110] Thomas J McCabe. “A complexity measure”. In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[111] Steve McConnell. *Code complete*. Pearson Education, 2004.

[112] Steve McConnell. *Code complete*. Pearson Education, 2004.

[113] Andrew Meneely and Laurie Williams. “Secure open source collaboration: an empirical study of linus’ law”. In: *Proceedings of the 16th ACM conference on Computer and communications security*. 2009, pp. 453–462.

[114] Tom Mens and Tom Tourwé. “A survey of software refactoring”. In: *IEEE Transactions on software engineering* 30.2 (2004), pp. 126–139.

[115] Microsoft. *Elements of C, C Tokens*. ”Accessed 2022-6-25”. URL: https://docs.microsoft.com/en-us/cpp/c-language/c-tokens?view=msvc-170.

[116] George A Miller. “The magical number seven, plus or minus two: Some limits on our capacity for processing information.” In: *Psychological review* 63.2 (1956), p. 81.

[117] Sergio Montenegro and Frank Dannemann. “Rodos-real time kernel design for dependability”. In: *ESASP* 669 (2009), p. 66.

[118] Lewis Morgan. *List of data breaches and cyber attacks in January 2018*. 2018. URL: https://www.itgovernance.co.uk/blog/list-of-data-breaches-and-cyber-attacks-in-january-201-2.

[119] Nachiappan Nagappan and Thomas Ball. “Use of relative code churn measures to predict system defect density”. In: *Proceedings of the 27th international conference on Software engineering*. 2005, pp. 284–292.

[120] JPL NASA. “JPL Institutional Coding Standard for the C Programming Language”. In: *Jet Propulsion Laboratory, California Institute of Technology* (2009).

[121] ntuAndroid. *Simple VM*. Mar. 2014. URL: https://github.com/ntu-android/simple%5C_vm.

[122] NVIDIA. *NVIDIA T4 Tensor Core GPU for AI Inference NVIDIA Data Center*. https://www.nvidia.com/en-us/data-center/tesla-t4/. accessed 21-7-2020.

[123] J. Oberg. “Why the Mars probe went off course [accident investigation]”. In: *IEEE Spectrum* 36.12 (1999), pp. 34–39. DOI: 10.1109/6.809121.

[124] Julian Offenhäuser. *Tiny JVM*. July 2018. URL: https://github.com/metalvoidzz/TinyJVM.

[125] *Omnipod FAQs - LoopDocs*. URL: https://loopkit.github.io/loopdocs/faqs/omnipod-faqs/.

[126] *OpenJDK*. Aug. 2018. URL: http://openjdk.java.net/.

[127] Oracle. *Oracle Java Embedded*. July 2018. URL: http://www.oracle.com/technetwork/java/embedded/overview/index.html.

[128] OSHA Oregon. *"Safety and the Supervisor"*. 2018. URL: http://osha.oregon.gov/OSHAEdu/safety-and-the-supervisor/1-160i.pdf.

[129] María Jesús Paredes Duarte et al. “”El principio de economía lingüística.””. In: (2008).

[130] Rocio Peniro and Jorde Cyntas. “Applied linguistics theory and application”. In: *Linguistics and Culture Review* 3.1 (May 2019), pp. 1–13. DOI: 10.37028/lingcure.v3n1.7. URL: https://lingcure.org/index.php/journal/article/view/7.

[131] Ivars Peterson. *Fatal defect: Chasing killer computer bugs*. Mckay, David, 1996.

[132] Kristine Pinedo. *4 of the Worst Computer Bugs in History*. Sept. 2017. URL: https://www.bugsnag.com/blog/4-worst-computer-bugs-in-history.

[133] André Platzer. *Orbital Library*. Feb. 2017. URL: http://symbolaris.com/orbital/.

[134] Michael J Pont. *Patterns for time-triggered embedded systems*. TTE System, Ltd, 2008.

[135] Manisha Priyadarshini. *10 Most Popular Programming Languages In 2018: Learn To Code*. June 2018. URL: https://fossbytes.com/most-popular-programming-languages.

[136] Chen Qian et al. “Text-Image Sentiment Analysis”. In: ().

[137] Stephen R Schach. *Object-oriented and classical software engineering*. McGraw-Hill Companies, 2011.

[138] Vincent van Ravesteijn. *Is LyX Bugzilla closed?* 2010. URL: https://www.mail-archive.com/lyx-users@lists.lyx.org/msg80875.html.

[139] Eric Raymond and B Young. *Thecathedral and the bazaar-musings on Linux and open source by an accidental revoltionary,(rev. ed.)* 2001.

[140] *realm/SwiftLint: A tool to enforce Swift style and conventions*. URL: https://github.com/realm/SwiftLint.

[141] James Reason, Dianne Parker, and Rebecca Lawton. “Organizational controls and safety: The varieties of rule-related behaviour”. In: *Journal of occupational and organizational psychology* 71.4 (1998), pp. 289–304.

[142] Rodger Richard. *Why I Have Given Up on Coding Standards*. 2012. URL: http://www.richardrodger.com/2012/11/03/why-i-have-given-up-on-coding-standards.

[143] *RileyLink FAQs - LoopDocs*. URL: https://loopkit.github.io/loopdocs/faqs/rileylink-faqs/.

[144] Dennis M Ritchie. "The UNIX System: The Evolution of the UNIX Time-sharing System". In: *AT&T Bell Laboratories Technical Journal* 63.8 (1984), pp. 1577–1593.

[145] Dennis M Ritchie, Brian W Kernighan, and Michael E Lesk. *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[146] William Roberts. *The Maginificent Seven*. Screenplay, 1960.

[147] Simon Rogerson. "The chinook helicopter disaster". In: *IMIS Journal* 12.2 (2002).

[148] Jean-Pierre Rosen. "Designing and checking coding standards for ada". In: *SIGAda Ada Letters* 31.3 (2011), p. 13.

[149] Arnold Rosenbloom. "A Simple MVC Framework for Web Development Courses". In: *Proceedings of the 23rd Western Canadian Conference on Computing Education*. WCCCE '18. Victoria, BC, Canada: ACM, 2018, 13:1–13:3. ISBN: 978-1-4503-5805-7. DOI: 10.1145/3209635.3209637. URL: http://doi.acm.org.ezproxy.stthomas.edu/10.1145/3209635.3209637.

[150] Paul Rozin and Edward B Royzman. "Negativity bias, negativity dominance, and contagion". In: *Personality and social psychology review* 5.4 (2001), pp. 296–320.

[151] Stephen R Schach. *Object-oriented and classical software engineering*. Vol. 6. McGraw-Hill New York, 2007.

[152] Google Scholar. *Google Scholar*. https://scholar.google.com//. Accessed: 2021-02-20.

[153] Seldon. *Transfer Learning for Machine Learning*. Accessed: 2022-07-30. July 2021. URL: https://www.seldon.io/transfer-learning#.

[154] Hoo-Chang Shin et al. "Deep convolutional neural networks for computer-aided detection: CNN architectures, dataset characteristics and transfer learning". In: *IEEE transactions on medical imaging* 35.5 (2016), pp. 1285–1298.

[155] Roger Smith. "The long history of gaming in military training". In: *Simulation & Gaming* 41.1 (2010), pp. 6–19.

[156] Internet Society. *INTERNET HALL OF FAME INDUCTEES*. Accessed: 2022-5-30. URL: https://www.internethalloffame.org/inductees/2013.

[157] Internet Society. *Source Code from The C Programming Language, 2nd Edition*. Accessed: 2022-5-30. URL: https://www.internethalloffame.org/inductees/2013.

[158] McCabe Software. *More Complex = Less Secure, Miss a Test Path and You Could Get Hacked.* 2013. URL: `http://www.mccabe.com/pdf/More%20Complex%20Equals%20Less%20Secure-McCabe.pdf` (visited on 04/22/2018).

[159] *SourceForge*. Aug. 2018. URL: `https://www.sourceforge.com`.

[160] Diomidis Spinellis. "Git". In: *IEEE software* 29.3 (2012), pp. 100–101.

[161] Diomidis Spinellis. "Git". In: *IEEE software* 29.3 (2012), pp. 100–101.

[162] Diomidis Spinellis. *Tokenizer*. Accessed: July 22,2020. URL: `https://github.com/dspinellis/tokenizer`.

[163] Merriam-Webster Staff. *Merriam-Webster's collegiate dictionary*. Vol. 2. Merriam-Webster, 2004.

[164] Richard Stallman. *Emacs*. Access:2022-5-9. 1986.

[165] *Step 3: Add CGM - LoopDocs*. URL: `https://loopkit.github.io/loopdocs/operation/loop-settings/cgm/`.

[166] Konstantinos Stroggylos and Diomidis Spinellis. "Refactoring–does it improve software quality?" In: *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*. IEEE. 2007, pp. 10–10.

[167] Dan Sturtevant et al. "Technical Debt in Large Systems: Understanding the cost of software complexity". In: *Unpublished thesis, MIT* (2013).

[168] Daniel Joseph Sturtevant. "System design and the cost of architectural complexity". PhD thesis. Massachusetts Institute of Technology, 2013.

[169] Ann T Tai, Leon Alkalai, and Savio N Chau. "On-board preventive maintenance for long-life deep-space missions: a model-based analysis". In: *Proceedings. IEEE International Computer Performance and Dependability Symposium. IPDS'98 (Cat. No. 98TB100248)*. IEEE. 1998, pp. 196–205.

[170] Andrew S Tanenbaum. *Operating system design and implementation*. PHI, 2009.

[171] Antony Tang. "Software Designers, Are You Biased?" In: (2011).

[172] GCC Team. *GCC Home Page*. 2018. URL: `https://gcc.gnu.org/`.

[173] LyX Team. *LyX Home Page*. 2018. URL: `https://www.lyx.org/`.

[174] Sudo Team. *Sudo Main Page*. 2018. URL: `https://www.sudo.ws/`.

[175] Gourav Thakur. *Beginners C Program Examples*. "Accessed 2022-6-25". URL: `https://github.com/gouravthakur39/beginners-C-program-examples`.

[176] *The History of Loop and LoopKit - Nate Racklyeft - Medium*. URL: `https://medium.com/@loudnate/the-history-of-loop-and-loopkit-59b3caf13805`.

[177] Saravanan Thirumuruganathan. *A Detailed Introduction to K-Nearest Neighbor (KNN) Algorithmhm*. May 2010. URL: https://saravananthirumuruganathan.wordpress.com/2010/05/17.

[178] Ken Thompson et al. “History of Unix”. In: ().

[179] *Tidepool Loop — Tidepool*. URL: https://www.tidepool.org/loop.

[180] Andy Tomkinson. “Risk assessment: vulnerability”. In: (2000).

[181] Linus Torvalds. “Linus Torvalds”. In: *Free and Open Source Software* (1969), p. 82.

[182] Linus Torvalds et al. “Linux kernel coding style”. In: *Also available as https://www. kernel. org/-doc/Documentation/CodingStyle* (2001).

[183] Gihan M Ubayawardana and Damith D Karunaratna. “Bug prediction model using code smells”. In: *2018 18th international conference on advances in ICT for emerging regions (ICTer)*. IEEE. 2018, pp. 70–77.

[184] Headquarters United States Army. *ATP 6-02.2 Signal Platoon*. Dec. 2020. URL: https://armypubs.army.mil/epubs/DR_pubs/DR_a/ARN31376-ATP_6-02.2-000-WEB-1.pdf.

[185] *User Manual For use with Dexcom G4 ® PLATINUM with Share Now including Apple ® Watch Information*. Tech. rep. 2015. URL: www.dexcom.com.

[186] Arie Van Deursen, Marius Marin, and Leon Moonen. “Aspect mining and refactoring”. In: *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE), with WCRE*. 2003, pp. 11–21.

[187] Jakub Veverka. *Jakubveverka sJVM*. Dec. 2015. URL: https://github.com/jakubveverka/sJVM.

[188] William Victor. *Creative Writing Now, How to Write a Play*. Accessed: 2018-08-11. 2009. URL: https://www.creative-writing-now.com/how-to-write-a-play.html.

[189] *Welcome to the AndroidAPS documentation — AndroidAPS 2.5.1 documentation*. URL: https://androidaps.readthedocs.io/en/latest/EN/.

[190] Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. “Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models”. In: *Empirical Software Engineering* 13.5 (2008), pp. 539–559.

[191] Sam Williams. *Free as in Freedom [Paperback]: Richard Stallman’s Crusade for Free Software*. O’Reilly Media, Inc., 2011.

[192] B. Wilkerson Wirfs-Brock and L. Wiener. *Designing object-oriented software*. Prentice Hall, 1990.

[193] Rebecca Wirfs-Brock and Brian Wilkerson. “Object-oriented design: a responsibility-driven approach”. In: *ACM SIGPLAN Notices*. Vol. 24. 10. ACM. 1989, pp. 71–75.

[194] William A Wulf. “A case against the GOTO”. In: *Proceedings of the ACM annual conference-Volume 2*. 1972, pp. 791–797.

[195] Rikiya Yamashita et al. “Convolutional neural networks: an overview and application in radiology”. In: *Insights into imaging* 9.4 (2018), pp. 611–629.

[196] Mehrdad Yazdani and Lev Manovich. “Predicting social trends from non-photographic images on Twitter”. In: *2015 IEEE international conference on big data (big data)*. IEEE. 2015, pp. 1653–1660.

[197] T Yin. *A simple code complexity analyser without caring about the C/C++ header files or Java imports, supports most of the popular languages*. https://github.com/terryyin/lizard. Aaccessed 21-7-2020. 2012.

[198] JunHo Yoon and Kunal Tyagi. *nsiqcppstyle*. 2014. URL: `https://github.com/kunaltyagi/nsiqcppstyle` (visited on 08/19/2014).

[199] Shuiyuan Yu, Chunshan Xu, and Haitao Liu. “Zipf’s law in 50 languages: its structural pattern, linguistic interpretation, and cognitive motivation”. In: *arXiv preprint arXiv:1807.01855* (2018).

[200] Nico Zazworka et al. “Comparing four approaches for technical debt identification”. In: *Software Quality Journal* 22.3 (2014), pp. 403–426.

[201] Lawrence R Zeitlin. “Failure to follow safety instructions: Faulty communication or risky decisions?” In: *Human Factors* 36.1 (1994), pp. 172–181.

[202] Yaowen Zhang, Lin Shang, and Xiuyi Jia. “Sentiment analysis on microblogging by integrating text and image features”. In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2015, pp. 52–63.

[203] Jiang Zheng et al. *On the Value of Static Analysis for Fault Detection in Software*. Tech. rep. URL: `http://www.gimpel.com/html/products.htm.`.

[204] Sarah Zielinski. “Computer failure caused loss of Mars spacecraft”. In: *Eos, Transactions American Geophysical Union* 88.17 (2007), pp. 192–192.

[205] George K Zipf. *1965. The Psychobiology of Language*. 1935.

[206] George Kingsley Zipf. *Human behavior and the principle of least effort.* addison-wesley press, 1949.