LEWIS UNIVERSITY

Structural Quality & Software Evolution

A Thesis

By

Alison Major

Department of Computer and Mathematical Sciences

Submitted in partial fulfillment of the requirements

for the degree of
Master of Science in Computer Science,

Concentration in Software Engineering

April 29, 2022

The undersigned have examined the thesis entitled '**Structural Quality & Software Evolution**' presented by **Alison Major**, a candidate for the degree of **Master of Science in Computer Science (Concentration in Software Engineering)** and hereby certify that it is worthy of acceptance.

Advisor	Date
Program Director	Date
Department Chair	Date

Abstract

Some software engineering projects fail to evolve, which makes them obsolete. This topic is interesting and important to developers because the software that fails to evolve will fail to generate user engagement, leading to revenue loss. We review a number of projects and resources to understand the correlation of software structure quality and its impacts on a system's ability to evolve. With this understanding, we explore ways to improve the evolution of a software system through tools and suggestions.

TODO: Update the abstract with each iteration of this paper.

Acknowledgements

This research and working towards my degree has been an endeavor and one that I would not dream of tackling on my own.

Firstly, my thanks go to God, the ultimate creator, who formed us all to be creators fitting our skills.

Second, much gratitude goes to my very creative and patient family. I am thankful for my husband, Chris, who was foundational in finding the time and focus on working on my studies. My amazing kids, Ewan and Gwynnie, have provided patience, encouragement, and numerous trips to the library. I could not have gotten through this without all of your support.

Finally, I am grateful for my coworkers and my professors. The Robotic Process Automation (RPA) team at Sysco has been cheering me on from the sidelines. The professors at Lewis University have provided mentorship and guidance. They have all encouraged me as I've worked to apply my knowledge from the last decade of my experience as a software developer and gather new information that has helped us build better software solutions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

When building software systems, we have several areas of concern: cost, delivery timeline, quality, etc. The cost and time-to-market are often the two problems given the highest priority in a project. However, engineers must consider the software quality to preserve the system's longevity. Despite its importance, the code and architecture quality can be challenging to understand and measure.

The art of programming began as something very tedious and error prone, with programmers writing every step of a process in a machine language. As this was not a great way to grow the industry, languages were developed so that programmers could reduce the manual effort of their programming and also improve the quality of their work by allowing the machine itself to transform the high-level code into something executable and efficient. [2]

With the creation of programming languages, the ability to build systems and applications has become a skill that most anyone can learn. Open source project create communities of learning and advancement. However, all systems need structure and an investment of time.

When we think about projects, we can assume that as time goes on and changes and additions occur within a system's source code, the complexity of that system will grow. However, when we manage the code structure, we can keep the complexity in check, allowing systems to evolve. Developers can maintain this structure through simple steps like having readable code and more complex considerations, like how coupled and cohesive a system is.

One way to understand the quality around a system is to discuss its "maintainability," the ease of receiving new features or resolving bugs. For example, developers may find that adjusting one area to add a new feature requires touching several other code areas in tightly coupled systems. Some code measuring systems provide a Maintainability Index (MI), a well-known quality measure. While the effectiveness in quantifying software quality is debated, many quality models include MI measurements [3], [1].

Adewumi et. al found that maintainability is measured by 55% of the existing Open Source Software (OSS) quality models, making it the most common quality characteristic measured [1]. From "Fig. 1.1", we can infer that maintainability is more important than functional stability. If the code is maintainable and accessible, missing features can be incorporated, but if it is difficult to read and understand and is not well documented, those missing features would be hard to implement [1].

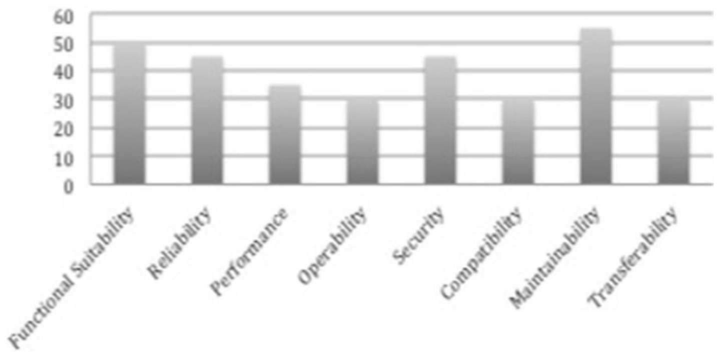

Figure 1.1: Frequency distribution of ISO 25010 product quality characteristics in OSS quality models, as found by Adewumi et. al in 2016 [1].

Code smells are used extensively by practitioners to identify low-quality spots in the software system. These areas would need the teams' attention and are good candidates for refactoring. Rather than putting focus on building comprehensive models with all possible software characteristics, developers should instead focus on the essential characteristics: maintainability, usability, and maintenance capacity of a software community [1].

1.1 Maintainability Index and Pylint Refactor Scores

"Software maintainability is one of the fundamental external quality attributes and is recognised as a research area of primary concern in software engineering [4]." As such, it is important for us to find ways to measure the maintainability of a software system. As mentioned earlier, many OSS quality models already include maintainability as a characteristic measurement.

Pylint is a static analysis tool that identifies several classes of code quality concerns. Particularly relevant to our study are refactor violations, which report on various code smells. We can assume that there must be some correlation between Maintainability Index and the type and number of code smells in a software system, quantified by the Pylint refactor score.

This study explores such assumptions and systematically investigates any correlation between the Maintainability Index metric and the Pylint Refactor score. Furthermore, we perform analysis on specific refactor violations to reveal and shed light on the relative effectiveness of the different refactor violations and their relationship to Maintainability Index.

The structural quality of a software system will impact the software evolution. If the project has poor structural quality, the architecture will minimize its ability to evolve, and the software system will eventually "die-off" so to speak.

We will look at many open-source Python systems using Pylint and attempt to correlate the data from the Pylint scores to the level of ease in adding new features to the system. This will determine if a system is more maintainable with better Pylint scores.

TODO: Add info about Radon and MI and scores

1.2 Paper Structure

In Chapter 2, we will dig into a deeper background of the topic, exploring ideas of why software systems need to keep users engaged long term (Section 2.1). We will explore automated measurements that provide evaluation scores of software systems. By using some of these quality and maintainability scores, we can see how structure impacts evolution (Section 2.2). Additionally, we'll explain how maintainability is measured, as well as the different attributes that can factor into maintainability. We will also review related works (Section 2.3).

With more background on the problem, we can then review the methodology for our research in Chapter 3. Here we will review where we found our initial data set (Section 3.1) and what criteria we used to filter it to a manageable size for our tests (Section 3.2).

Chapter 4 will review the results of our research, using the methodology previously explained. TODO: Expand on this a little.

We will then provide final conclusions and recommendations in Chapter 5. TODO: Expand on this a little.

Chapter 2

Background and Literature Review

In this chapter we will gain a better understanding of the topic at hand, first by understanding long term user engagement. From there we will explore the impact of structural quality within a software system. Once we understand the problem thoroughly, we will determine a useful dataset that we can apply our theories to, as well as explore related works.

2.1 Keeping Users Engaged Long Term

When developing a new system or a new software idea, getting the project off the ground and in front of users is one thing. However, keeping that project alive with a thriving community of engaged users is another.

The systems we create could be customer-facing web applications, games, or internal applications used to carry out tasks. Regardless of the type of system, the product will no longer provide usefulness without evolving with the user's needs. As users become familiar with a system that is designed to depend, at least in part, on their attitudes and their practices, the users will modify their behavior to minimize their own effort or maximize the effectiveness of the tool [2]. Even in a corporate setting with internal business systems, over time, users will need change; how a system can adapt to those needs requires a level of flexibility.

2.1.1 Why does software evolution matter?

> "Software evolution is the continual development of software after its initial release to address changing stakeholder and/or market requirements." [5]

When a system cannot evolve, the impact is primarily felt by the users. However, this impact will eventually get back to those who created and continue to support the system. With users that are either unsatisfied or unable to use the system any longer, the engagement levels will drop. The decline in users will ultimately result in a loss of income, as the system can no longer deliver to the needs of its audience.

Because organizations invest large amounts of money in the software systems that they create, they depend on the software's continued success. Software evolution will allow the system to adapt to new or changing business requirements, fix bugs and defects, and integrate with other systems that have changed and evolved that may share the same software environment.

As a system is used, inevitably, users will stumble into situations that even the best quality assurance testers will miss. When defects are found, they will require fixing. THOUGHT: This is a maintenance thing; maybe inappropriate to include here

To keep a system up-to-date, we must add new features. For example, there may be a need to improve a system's performance or reliability, especially if the user base expands. For a business or group to maintain user satisfaction, the software must continue to evolve; this will result in a system's increased size and complexity, as well as quality decline (unless very closely monitored over time) [6].

Security can also impact the need for a system to be maintained. New ways to infiltrate a system can be uncovered, so it is important to stay on top of newest versions of dependencies and technologies in order to avoid potential breaches of data and experience.

2.1.2 How do we ensure software evolution?

Because the maintainability of a system can ultimately influence the ability to generate revenue, we must find ways to ensure that a project will evolve. One of these ways could be to ensure that a project continues to be considered "maintainable" throughout its lifetime. This system characteristic will ensure that bugs can be fixed quickly, but new features should be easy to add as the users' needs evolve.

Not only is the user engagement necessary for the activation of the evolution process, but the engagement of the development community itself, especially when considering open source projects, is critical. As Adewumi et al. points out, a key feature that distinguishes open source software is that it is built and maintained by a community [7]. Additionally, the quality of the community will determine the quality of the open source software [8].

2.2 The Impact of Structural Quality

2.2.1 Software Maintenance

The structural quality of a software system will impact the software evolution. Similarly, as a project evolves, there is a likelihood that it will degrade, as changes are often made quickly and in ways that the original design did not anticipate [9]. If the project has poor structural quality, its ability to evolve will be minimized, and the software system will eventually "die-off" so to speak.

There is much planning involved in all software creation projects in what the product will be, will do, who it is for, etc. One of the things that should also be on the planning list is long-term maintenance and growth. That is, how do we build a thing that will be easier to add features to down the road?

Let us define maintainability in the context of software. For example, a system would be considered easy to maintain if it is easy to debug and easy to add new features. These new features are generally considered minor features, and may often be reported as bugs by users, when in reality, they are looking for functionality enhancements [10].

> "Software maintenance in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes." [10]

It may be easier to understand what characteristics define a system with poor maintainability. These types of systems will have poor code quality, leading to defects. For example, there could be undetected vulnerabilities or vulnerabilities that have been ignored. It may be that the system is overly complex. In addition to the complexity, it could be hard to read due to poor naming or dead (unused) code throughout the source code.

A project is known to have good maintainability when there is an enforced set of clean and consistent standards for the code. This often involves having human-readable names for functions, methods, and variables. Any complex code is minimized, and methods are small and focus on a single thing. Parts of the system are decoupled and organized, making it easy to work on different parts with low impact on unrelated parts. For example, the code is DRY (there is limited redundancy in the code), unused code has been removed, and there is a level of documentation that supports an easy understanding of the system.

Why should we care about whether the code is maintainable? It is assumed that a large amount of the cost over the lifetime of a project is attributed to maintainability. Fred Brooks, in his book "The Mythical Man-Month" (1975) even claimed that over 90% of the costs for a typical software system come up in the maintenance phase [11]. In 1977, Meir M. Lehman noted that 70% of a program budget was spent on maintenance, with the remaining 30% spent on development. In 1993, it was again observed that only 20-40% of the resources (money, time, effort) were used for development of a project, with 60-70% used for maintenance activities [12].

Once the bulk of the system is off the ground and live worldwide, how well the team can improve the system with new features and fix bugs, even working on different parts in parallel, can be impacted by its maintainability. Any successful piece of software will inevitably need to be maintained.

TODO: Anything about SOLID and when too many files are being touched? [9]

2.2.2 Software Evolution

There is a distinction to be made between **software maintenance** and **software evolution**. We will refer to software maintenance as bug resolution and for minor functional improvements. For example, we can consider this routine maintenance when we must fix a broken route in the application or provide a subtle enhancement on the user experience. However, when we look at upgrades to the system, adaptations to the changing and growing needs of the user, or migrating the system to a new technology, we can refer to this as evolution of the software.

The evolution of software can result from new laws that have come into being. As technology itself changes, governing bodies must continually revisit data collection and information sharing policies. Changes in technology and laws may lead to adaptations in the software systems.

It is also fair to say that systems will change because we can never fully determine a user's needs at the start of a project. It would be safe to say that the user's needs will change over time themselves. This leads to a neverending project that will always need some form of enhancement.

Meir “Manny” Lehman and László “Les” Bélády contributed to a list of laws involving software evolution known as Lehman’s Laws that describe a balance between forces that drive new developments while also slowing progress. These laws apply to programs that were written to perform some real-world activity, where its behavior is linked to the environment in which it runs; additionally, this program category assumes that the program needs to adapt to varying requirements and circumstances in that environment. Eight laws were created and are listed below. [13]

TODO: Read and reference “An Empirical Study of Lehman’s Law on Software Quality Evolution” [6].

1. **Continuing Change** *(1974)*
2. **Increasing Complexity** *(1974)*
3. **Self Regulation** *(1974)*
4. **Conservation of Organisational Stability** *(1978)*
5. **Conservation of Familiarity** *(1978)*
6. **Continuing Growth** *(1991)*
7. **Declining Quality** *(1996)*
8. **Feedback System** *(1996)*

The first law, “Continuing Change,” tells us that if a system does not adapt, it will become progressively less satisfactory. The second, “Increasing Complexity,” explains that as a system evolves, unless work is done to maintain or reduce complexity, the complexity will increase. This can be due to the added volume of the code from new features or even an increasing number of developers that have edited the code. Unless this phenomenon of increased complexity is actively addressed during changes, it can impact the maintainability (and the ability of a project to continue evolving) in the future.

Lehman’s fifth law, “Conservation of Familiarity,” explains how the average incremental growth does not change over time as a system evolves. The people interacting with the system, such as the developers, business persons, or users, must still continue using and working within the system at the same “level of mastery.” If the system grows and changes excessively, the mastery will drop, slowing down the next set of changes. This could be because the source code or architecture has become more complex (impacting the developers’ ability to adapt and enhance the system) or because the user features have changed so that the system audience needs time to master the new interfaces or new tools. Because of this natural “slow-down” for excessive change, the average incremental growth will remain steady. We can see a simplified visual in “Fig. 2.1” showing that when the number of changes spikes (that is to say, when there is excessive growth in a system), it will be followed by an iteration of fewer changes, leading to a nearly consistent average of incremental growth (the thick, horizontal line) over time.

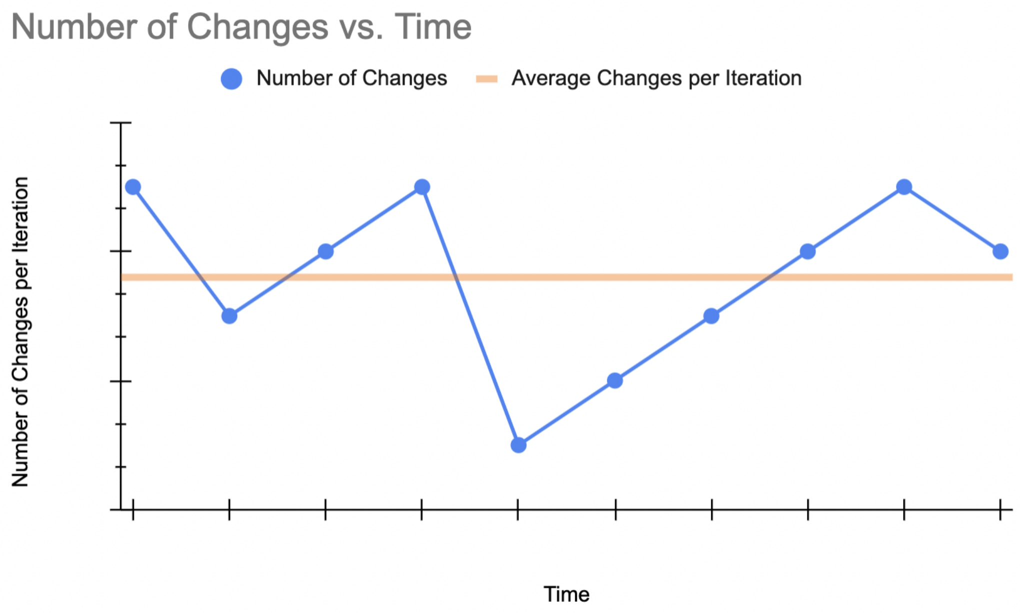

Figure 2.1: A simplified visual of Lehman's fifth law, "Conservation of Familiarity."

In Lehman's sixth law, "Continuing Growth," we see that the system user's satisfaction will not be maintained without continually increasing the functional content. Along a similar idea, the law pertaining to "Declining Quality" states that if the operational environment for the system does not change, the system's quality will appear to decline. Yu and Mishra performed focused research on supporting Lehman's Laws, especially in the case of the seventh law pertaining to declining quality, by defining a metric for software quality and reviewing the bug history, growth of size, complexity, and quality of two large open-source systems [6]. Therefore, we must continue adapting for even the appearance of the maintained quality of a system.

With all of these characteristics surrounding the evolution of software, we benefit from the Internet that has positively improved the experience. Two common resources currently available to developers have impacted software evolution [5]:

1. The rapid growth of the World Wide Web and Internet Resources make it easier for users and engineers to find related information.
2. Open source development where anybody could download the source codes and modify it has enabled fast and parallel evolution (through forks).

These two suggestions are very evident in modern development. For example, a developer may regularly use resources like StackOverflow to find solutions to problems and use open-source tools that the developer and their team can contribute to or adjust to their specific needs.

2.2.3 Measuring Maintainability

Despite the nuanced differences between *maintainability* and *evolution*, the two characteristics run parallel to each other. If a system is easy to maintain, it will also be easier to evolve. If we can measure our system's maintainability, we can also determine if our system is in a good position to continue evolving to meet our future needs.

> "The unit cost of change must initially be made as low as possible and its growth, as the system ages, minimized. Programs must be made more alterable, and the alterability maintained throughout their lifetime. The change process itself must be planned and controlled." [2]

Several tools attempt to provide some value around these ideas. In this paper, we will focus on the metrics that Pylint provides, specifically looking into the Refactor score of Pylint.

We will look at many open-source Python systems using Pylint and attempt to correlate the data from the Pylint scores to the level of ease in adding new features to the system. This will determine if a system is more maintainable with better Pylint scores.

FUTURE EDITION: To do this, we will measure the locality of the changes by the number of files that are edited in a commit. We will also focus on commits that represent new features, not on commits that are bug fixes.

2.2.4 Maintainability Scores

First, let us consider our original understanding of software maintainability. While this definition focuses primarily on bug fixes and minor enhancements, maintainable projects should also have ease in their ability to evolve. Therefore, we can study the impact maintainability (structural quality) has on software evolution by reviewing the scores provided by automated code review tools.

In this study, we will be using Pylint and will be focused on the values of the Refactor score regarding a set of open-source Python systems. To understand the scores we will be working with, we must understand what Pylint itself is doing.

Through the documentation of Pylint, we can understand how to use it and the scores it will provide [14]. The Pylint score itself is calculated by the following equation [15]:

```
10.0 - (( float( 5 * e + w + r + c) / s ) * 10 )
```

Numbers closer to `10` reflect systems that have fewer errors, fewer warnings, and have overall better structure and consistency. In the above equation, we are using the following values [16]:

- **statement** (`s`): the total number of statements analyzed
- **error** (`e`): the total number of errors, which are likely bugs in the code

- **warning** (`w`): the total number of warnings, which are python specific problems
- **refactor** (`r`): the total number of refactor warnings for bad code smells
- **convention** (`c`): the total number of convention warnings for programming standard violations

The Refactor score is of special interest to us and considers many features that are meticulously outlined on the Pylint site [17]. These types of warnings include a number of checks, such as when a boolean condition could be simplified, or a useless **`return`**, and so on. This score, in particular, will be part of our focus.

To calculate the Refactor score, Pylint will check the code for code smells based on the definitions for checks that have been documented. For every infraction, the score increases by one count.

> "In computer programming, a **code smell** is any characteristic in the source code of a program that possibly indicates a deeper problem." [18]

We can use these Refactor scores to help us spot architecture smells. After all, code smells can point the way to deeper problems in our system. There are fundamental design principles that have been established that we should consider when creating software; code smells alert us to areas that have deviated from these principles. These smells are drivers for refactoring and, when addressed, can help us maintain the integrity of our architecture rather than creating a patchwork construction. Because of the relation of refactor scores to the code structure itself, we will be spending much of our focus on this particular value.

Finally, in respect to Python, it is also helpful to be familiar with PEP 8, as this is the default set of standards that Pylint uses to judge Python code [19]. This standard can be used to make code more readable and more consistent, which may contribute to the code being more maintainable. These standards cover things like indentation spacing, maximum line length, where to break lines, how to handle imports, and more. By defining a set of standards, teams can ensure they have a defined set of rules so that any contributors to the code understand the expectations (and so that automated systems like Pylint can enforce those standards to maintain readability and consistency).

2.2.5 Other Maintainability Characteristics

The authors of "Measurement and refactoring for package structure based on complex network" recently reviewed a similar idea focusing on cohesion and coupling over time for a project [20]. In a software system, we desire low coupling (allowing for changes to one area to remain independent of changes to another area) and high cohesion (indicating reduced complexity in modules, which improves maintainability). Through a few experiments on open-source software systems, the authors determined that their algorithm that calculated metrics was capable of improving package structures to have high cohesion and low coupling. Their study gives us confidence that metrics around the software's structure can provide value in keeping systems in a maintainable state, which allows for software evolution.

Another variable that may impact the maintainability of code is readability. For example, in the article "How does code readability change during software evolution?" the authors have addressed this concern and found that most source codes were readable within the sample they reviewed. Additionally, a minority of commits changed the readability; if a file was created as less readable, it was likely that it remained that way and did not improve [21]. This variable (readability) in the maintainability of a software system can influence how easy or difficult it is to make a change. The authors also found that big commits, usually associated to adaptive changes (a form of software evolution), were the most prone to reduce code readability [21]. This assumes that smaller commits are almost always better and can lead to more readable code.

Piantadosi et al. found that changes in readability, whether improvements or disintegrations, often occurred unintentionally [21]. By enforcing the PEP 8 standard, we know that Pylint is encouraging systems to remain readable. Therefore, projects that use some form of automated system in their pipeline benefit from keeping their project on track in this regard, limiting the effects of readability on a software's potential for evolution.

The paper "Standardized code quality benchmarking for improving software maintainability" provides additional insights into how the code's maintainability is impacted by the technical quality of source code [22]. Within their paper, the authors seek to show four key points: (1) how easy it is to determine where and how the change is made, (2) how easy it is to implement the change, (3) how easy it is to avoid unexpected effects, and (4) how easy it is to validate the changes. Their approach has shown that some tools and methods can be used to improve and maintain technical quality within their projects, allowing systems to continue to evolve at a reasonable pace.

2.2.6 Documentation and Maintainability

Our assumption is that the Refactor score in projects should correlate to the evolution of the system. The first pass through the data is not conclusive in this particular detail, as the projects reviewed have many other factors contributing to the evolution of the project (number of contributors, size of the code system, etc.). Our assumption is that the correlation between software quality and software evolution would indicate that the better-scoring code systems are readable in themselves. In addition, it would be helpful to understand whether there are any similarities in how a system is documented that could contribute to improved software evolution of a system.

The textbook, "Software Architecture in Practice," chapter 18 provides some insight in documentation around architecture [23]:

> "If you go to the trouble of creating a strong architecture, one that you expect to stand the test of time, then you *must* go to the trouble of describing it in enough detail, without ambiguity, and organizing it so that others can quickly find and update the needed information."

The book describes how documentation holds the results of significant design decisions, providing valuable insights into decisions down the road. While not directly related to the Pylint Refactor score and not within the source code itself, it is still helpful to remind ourselves that documentation can also influence the ability of a software system to evolve.

> “Our study has shown that the primary studies provide empirical evidence on the positive effect of documentation of designs pattern instances on programme comprehension, and therefore, maintainability.”
>
> “...developers should pay more effort to add such documentation, even if in the form of simple comments in the source code.”

In research done by Wedyan and Abufakher (quoted above), it was found that documenting design patterns was useful in enhancing code understanding [24]. In turn, the comprehensibility impacts the maintainability of the code in a positive way, which continues to reinforce the impact that documentation can have and how it ties well into considerations for software structure.

Borrego et al. discussed that software development teams must have access to architecture knowledge, as that is the base in which they will build and understand the technical part of any software development cycle [25]. It has also been recognized that an absence of knowledge management is a factor in failure of a development project [25]. Bjørnson and Dingsøyr noted that knowledge, in software engineering, is managed mostly through repositories that will support knowledge flows [26].

2.3 Related Work

In our research, we are running with the assumption that Maintainability Index (MI) is our primary indicator and we will look for the correlation between the MI and other Pylint scores. We hope to find which correlations align and which are the most important.

2.3.1 Considering Data Sets

When exploring the correlations of maintainability and refactoring, there are many sources available for research. Some researchers have looked at proprietary systems as they evolve over time, while others have chosen open-source code available to the general public.

A study conducted by Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov begins by programmatically collecting a sample set of projects in GitHub that vary in languages. Then the group of projects is appropriately culled, resulting in a final set used for the review. The results are then studied for the impact different programming languages may have on the code quality [27]. Through their research, they were able to determine which languages were more prone to defects, and that individual languages are more related to individual bugs rather than bugs overall.

The authors of “Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison” performed a very similar study to what we are doing here. Their study focuses on object-oriented software (specifically C/C++ and Java) and a correlation analaysis between object-oriented metrics and software maintainability, looking for the best metrics to predict maintainability [28]. This particular study focuses on a few hand-picked software systems with an analysis of the change logs. Our study, however, will be of a larger scale (about 50 software systems) and focused solely on Python-heavy projects.

2.3.2 Design Patterns and Software Quality

In the paper “Impact of design patterns on software quality: a systematic literature review” the authors compared the use of design patterns to software evolution and maintainability. They found that design patterns provided clear flexibility when they reviewed changes that extended (evolved) software [24].

> “Changes performed in a class can be corrective, adaptive, perfective, or preventive. These changes can occur due to new requirements, debugging, changes that propagate from changes in other classes and refactoring.”

Wedyan and Abufakher found that there were two reasons that a class had more frequent changes [24]:

1. The class was easy to extend.

2. The class correlated to other classes (raising alarms about class modularity).

With these findings in mind, we intentionally aim to focus our research on changes for system extensions and adaptations rather than bug fixes that appeared to be larger change due to high coupling. Within this paper, we were able to focus on Refactor scores (code smells) rather than Error scores (bugs) within the system.

2.3.3 Software Architecture and Maintainability

The research done in "Software Architecture Metrics: A Literature Review", the authors discuss how early detection of issues within the software's architecture is key to mitigating the risk of poor performance and can lower the cost of repairing faults [29]. While most developers have had access to these types of metrics for several decades, the industry and open-source community have not really latched onto their use for keeping code in easy-to-work-with condition.

The review done by Coulin et al. called out five important qualities of software architecture [29]:

1. Maintainability
2. Extensibility
3. Simplicity, Understandability

4. Re-usability

5. Performance

Focusing on these qualities can narrow down the choice between different design options to end up with the most ideal solution. Keeping these five qualities in top-of-mind for new (and changed) code allows for easier future development and evolution of the software system.

ISO/IEC 25010:2011 is a detailed standard for software quality that contains eight product quality characteristics [30]. Each characteristic is further comprised of various sub-characteristics. "Fig. 2.2" lists these characteristics, with maintainability being one of the most significant characteristics in some studies [31], [1].

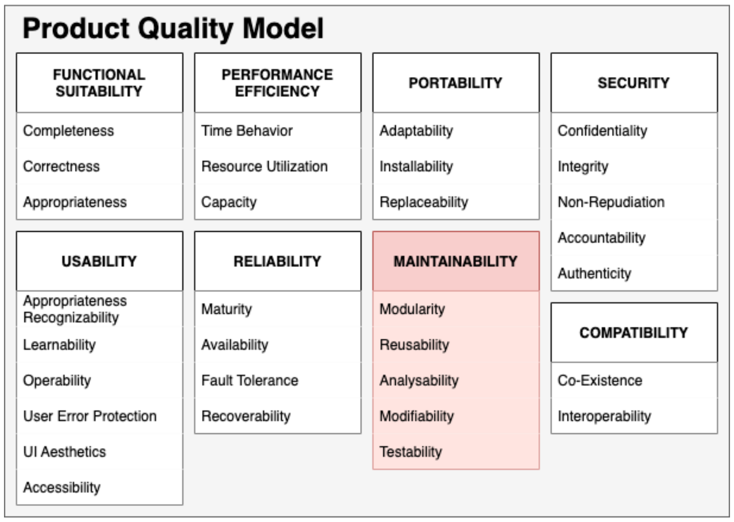

Figure 2.2: The eight product quality characteristics and sub-characteristics.

Chapter 3

Methodology

There are a number of factors to consider when reviewing data and considering which software systems to consider. An obvious starting place is with open source software (OSS), as it is freely available to study.

To find projects of a caliber worth studying, we may also consider the maintenance capacity of a project.

> "Maintenance capacity refers to the number of contributors to an OSS project and the amount of time they are willing and able to contribute to the development effort as observed from versioning logs, mailing lists, discussion forums and bug report systems. Furthermore, sustainability refers to the ability of the community to grow in terms of new contributors and to regenerate by attracting and engaging new members to take the place of those leaving the community." [1]

Open source projects, as stated above, have many factors and variables that contribute to the successful ability to not only maintain, but also evolve a project. Not only must the architecture be built in a fashion that is able to be enhanced, but the community of developers must be adequately engaged and interested in the continuing improvement.

3.1 Initial Repository Set

We have established that we have a problem with projects that fail to evolve, resulting in a loss of revenue. We also understand that evolving software is essential in order to keep users engaged; without it, there is an appearance in the decline of quality and the program becomes less satisfactory to the user, as well as potential for competitors to outpace us with features available. We must now understand how we can ensure that our systems evolve. For this, we will look to understand how the system's structural quality impacts software evolution.

The work done by Dr. Omari and Dr. Martinez involves collecting a subset of Python projects that we can use for further research. The bulk of the effort they have provided is determining which classifiers to use to pare down the public set of Python systems into a good collection for further analysis [32]. The work that they have provided was used to select appropriate Python systems for review by collecting meta-data on these code systems.

Selection criteria employed by Omari and Martinez included "popular projects with long development history and multiple release cycles [32]." All projects are open source, enabling other researchers to have access to the defined set. Additionally, they captured meta-data that was then used to define our own subset of repositories for our particular focus.

From their subset of repositories, we reviewed current Pylint scores from each of the 132 systems. This gives us a sampling of data that we can now dig deeper into, comparing similar systems (similar size, similar number of contributors, etc.) and their evolution process by reviewing past commits rather than merely the current state of the system, as we have done here.

3.2 Filtered Respository Set

Once we had a narrowed set of projects from GitHub that were primarily written in Python, we culled the set more using several criteria:

1. Projects that are at least 80% Python
2. Projects with a long history of commits
3. Projects with large development teams (community of contributors)
4. Projects with many releases
5. Projects of a substantial age

Armed with this list, we were able to use the metadata from GitHub for each of our repositories already collected and determine a cross-section of these criteria that would result in about 50 repositories for futher study.

Beginning with the languages field from GitHub, we could easily narrow down projects that had at least 80% of the code in Python. In our set, 103 repositories contained 80% or more Python code.

With this narrowed set, we then looked to see at which percentile all the remaining criteria would yield the desired number of repositories. We determined that using the value at the 20th percentile in each of the above categories would yield the size set we'd need.

Criteria	20th Percentile Value
Number of Commits	2,968
Number of Contributors	90
Number of Releases	44
Age (in months)	66.4

Table 3.1: Criteria used to filter down the initial set of repositories.

Table 3.1 shows the values found for each of our criteria. Using these values as our minimum requirements, we can narrow our repository set to 46 repositories (see Table 3.2).

Respository Names
ansible astropy autobahn-python aws-cli beets biopython boto buildbot celery cobbler conda cython django django-rest-framework electrum fail2ban gensim luigi matplotlib mongoengine mitmproxy mongo-python-driver mopidy networkx paramiko nova numba pandas peewee pelican pip pyramid ranger raven-python salt scikit-image scikit-learn scrapy sentry sqlalchemy swift sympy tornado web2py werkzeug youtube-dl

Table 3.2: List of the 46 repositories for research focus.

Chapter 4

Results

With our narrowed focus of repositories, we can now spend some more efforts looking through the data that can be collected. Specifically, we gathered data on the Refactor Warnings provided by Pylint. In Table 4.1 we have a shortened listing of the repositories from our set with their total count of refactor warning messages, in addition to the most commonly appearing refactor warning message. For the entire set, see Table 7.1 in the Appendix.

We can see that the "worst" project in our set, in regards to the number of refactor warning messages provided, was the repository Sympy [33], with 14,206 refactor messages. However, project size may also weigh into how many warnings and errors are present, as larger projects are presumed to have more errors.

If we instead look at the ratio of refactor warnings to the total lines of source code, we get a very different picture, provided in Table 4.2. In this

Repository Name	Msg Count	Top Msg	Top Msg Count
sympy	14,206	too-many-arguments	6,601 (46%)
ansible	10,431	no-else-return	1,711 (16%)
salt	7,814	too-many-arguments	1,640 (21%)
–	–	–	–
ranger	109	no-else-return	29 (27%)
sentry	66	no-self-use	32 (48%)
raven-python	20	too-few-public-methods	7 (35%)

Table 4.1: Total refactor messages per repository for the 3 best and 3 worst offendors. Also provided with the refactor message that had the most warnings and its total count (and the percentage of that message from the total refactor warnings for that repository).

case, the repository Raven-Python [34] (which was the "best" in regards to refactor message count) is the "worst" with the highest ratio of refactoring warnings compared to the lines of source code in the project. Raven-Python also has the smallest count of SLOC in the entire repository set!

Given that we now have an idea of which projects have the most refactor messages per lines of source code ("worst" offenders) and the projects with the least refactor messages ("best" projects for maintainability), we can look a little closer at the messages that are most frequent. Common to all six of these repositories at a high frequency are the message "no-self-use" and "no-else-return". The message "no-self-use" means that 'self' is used as an argument but is not used in the method and should be handled differently.

Repository	Total Refactor Msgs	Total Project SLOC	Ratio
raven-python	20	1,474	1.35
scrapy	381	58,768	0.64
numba	126	20,192	0.62
–	–	–	–
electrum	722	425,576	0.16
youtube-dl	1,003	667,075	0.15
cython	1,718	1,183,863	0.14

Table 4.2: Total refactor messages per repository, total project source line of code (SLOC), and the ratio of refactor messages to SLOC.

The message “no-else-return” highlights when an unnecessary block of code follows an if-conditional.

Some of the other common messages call out the use of “too many” of different types of objects. For example, “too-many-branches”, “too-many-arguments”, “too-many-locals”, “too-many-statements”...

> “Software maintainability these days has become one of the essential external attributes of software, which further forms a basis of research for many researchers working in the fields related to software engineering. Software maintainability can be described as the extent to which a particular software system can be changed concerning the number of Lines of Code (LOC).” [31]

Now, it would be helpful to understand where we stand on the Maintainability Index (MI) for these projects, as this is our form of measurement that will help us understand how the code might be able to evolve over time. The

MI is calculated on a per-module basis, but for the sake of conversation (and acknowledging that we will lose some nuances by reducing it this way), let's find the average MI at the project level and add this to our table (Table 4.3).

Repo	Refactor Msgs	Project SLOC	Ratio	Avg Project MI
raven-python	20	1474	1.35	87.02
scrapy	381	58768	0.64	64.47
numba	126	20192	0.62	62.55
–	–	–	–	–
electrum	722	425576	0.16	39.41
youtube-dl	1003	667075	0.15	54.16
cython	1718	1183863	0.14	31.02

Table 4.3: Total refactor messages per repository, total project source line of code (SLOC), the ratio of refactor messages to SLOC, average Maintainability Index (MI).

When reviewing the MI for our repositories at either end of the spectrum and in context with the entire set (see Appendex 7.2), we'll find that while Raven-Python has the highest number of refactor warnings when compared to the SLOC count, it also has the second highest average MI across all of its project files.

Of the full set, our highest average MI is 87.38 (belonging to Sentry) and our lowest average MI is 28.85 (belonging to MatPlotLib). Regardless, all of these average values are above a score of 20, which is Radon's lowest score provided for an "A" rank, that is, what would be considered a project "very high" maintainability.

Chapter 5

Conclusions and Recommendations

By collecting data and drawing our conclusions from it, with help from the insights from the studies done before ours, we may better understand metrics that can be useful regarding maintainability. Good projects will inevitably continue to grow and evolve. Understanding methods to keep code refactor on a certain level makes code easy to change. We may also find that projects with worsening scores slow down with updates and have reduced engagement.

When reviewing our data with current project Refactor message ratios, our three worst offenders were *Raven-Python*, *Scrapy*, and *Numba* [34], [35], [36]. *Raven-Python* has been deprecated in favor of a newer system, *Sentry-Python*. The other two projects are still quite active with development despite their poor scores. The two active projects are used in thousands of projects each, which may be the reason for continued development despite potential difficulty in maintenance.

Projects that may be open source or have many contributors are especially vulnerable to maintainability degrading over the evolution of a project. Having a reliable metric can be very useful in programmatically avoiding code smells and keeping code in a state that is easy to manage through simple metric checks in deployment pipelines.

While not one of our worst refactor-to-SLOC ratios, *SymPy* is a repository with a large number of refactor warnings. We can see an example of code smell issues in reviewing some current symptoms that *SymPy* is experiencing, with only 72% code coverage and a failing build (see “Fig. 5.1”). Despite the engagement and continued development, we suspect that real adaptations and evolution of the software may be difficult with this code.

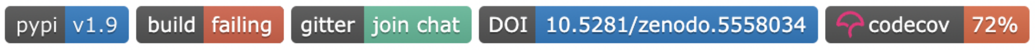

Figure 5.1: A snapshot of the badges from *SymPy*’s repository.

We have further work to do in this study to gain better understanding. With a set of several “best” and “worst” Python software systems, we will look into the history of the projects’ commits. It would be useful to see how the Refactor scores have changed over time, and if the rate at which changes were pushed correlated to the increase or decrease in that Refactor score.

Additional data can be gathered from this set that may provide more insights than this first brush of the data provides us. Understanding the impact of structural quality on the evolution of a project can provide compelling perspectives.

References

[1] A. Adewumi, S. Misra, N. Omoregbe, B. Crawford, and R. Soto, “A systematic literature review of open source software quality assessment models,” *SpringerPlus*, vol. 5, no. 1936, 2016.

[2] M. M. Lehman, “Programs, life cycles, and laws of software evolution,” *Proceedings of the IEEEs*, vol. 68, no. 9, pp. 1060–1076, September 1980.

[3] A. V. Deursen, “Think twice before using the “maintainability index”,” 2014, https://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/ [Online; accessed 23-Jan-2022]. [Online]. Available: https://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/

[4] H. Alsolai and M. Roper, “A systematic literature review of machine learning techniques for software maintainability prediction,” *Information and Software Technology*, 2019.

[5] W. contributors, “Software evolution — Wikipedia, the free encyclopedia,” 2021, https://en.wikipedia.org/wiki/Software_evolution [Online; accessed 12-December-2021]. [Online]. Available: https://en.wikipedia.org/wiki/Software_evolution

[6] L. Yu and A. Mishra, “An empirical study of lehman’s law on software quality evolution,” *Int J Software Informatics*, vol. 7, no. 3, pp. 469–481, 2013.

[7] K. Haaland and A.-K. Groven, “Free/libre open source quality models-a comparison between two approaches,” 2010.

[8] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos, “The sqo-oss quality model: Measurement based open source software evaluation,”

IFIP International Federation for Information Processing, vol. 275, pp. 237–248, 2008.

[9] R. C. Martin, "Design principles and design patterns," *Object Mentor*, 2000.

[10] W. contributors, "Software maintenance — Wikipedia, the free encyclopedia," 2021, https://en.wikipedia.org/wiki/Software_maintenance [Online; accessed 17-December-2021]. [Online]. Available: https://en.wikipedia.org/wiki/Software_maintenance

[11] F. Brooks, *The Mythical Man-Month.* Addison-Wesley, 1975.

[12] "Ieee standard for software maintenance," *IEEE Std 1219-1993*, pp. 1–45, 1993.

[13] W. contributors, "Lehman's laws of software evolution," 2021, https://en.wikipedia.org/wiki/Lehman%27s_laws_of_software_evolution [Online; accessed 12-December-2021]. [Online]. Available: https://en.wikipedia.org/wiki/Lehman%27s_laws_of_software_evolution

[14] Logilab and contributors, "Pylint," Logilab, 2020, https://pylint.org/ [Online; accessed 14-December-2021]. [Online]. Available: https://pylint.org/

[15] P. Logilab and contributors, "Pylint features," Logilab and PyCQA, 2021, https://pylint.pycqa.org/en/latest/technical_reference/features.html#reports-options [Online; accessed 14-December-2021]. [Online]. Available: https://pylint.pycqa.org/en/latest/technical_reference/features.html#reports-options

[16] R. Kirkpatrick, "A beginner's guide to code standards in python - pylint tutorial," 2016, https://docs.pylint.org/en/1.6.0/tutorial.html [Online; accessed 18-December-2021]. [Online]. Available: https://docs.pylint.org/en/1.6.0/tutorial.html

[17] P. Logilab and contributors, "Pylint features," Logilab and PyCQA, 2021, https://pylint.pycqa.org/en/latest/technical_reference/features.html#refactoring-checker [Online; accessed 14-December-2021]. [Online]. Available: https://pylint.pycqa.org/en/latest/technical_reference/features.html#refactoring-checker

[18] W. contributors, "Code smell — Wikipedia, the free encyclopedia," 2021, https://en.wikipedia.org/wiki/Code_smell [Online; accessed 18-December-2021]. [Online]. Available: https://en.wikipedia.org/wiki/Code_smell

[19] P. S. Foundation and contributors, "Pep 8 – style guide for python code," Heroku Application, 2021, https://www.python.org/dev/peps/pep-0008/ [Online; accessed 14-December-2021]. [Online]. Available: https://www.python.org/dev/peps/pep-0008/

[20] Y. Zhou, Y. Mi, Y. Zhu, and L. Chen, "Measurement and refactoring for package structure based on complex network," *Applied Network Science*, vol. 5, no. 50, 2020.

[21] V. Piantadosi, F. Fierro, S. Scalabrino, A. Serebrenik, and R. Oliveto, "How does code readability change during software evolution?" *Software Qual J*, vol. 25, pp. 5374–5412, 2020.

[22] R. Baggen, José, P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Qual J*, vol. 20, pp. 287–307, 2012.

[23] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice: Third Edition.* Addison-Wesley Professional, 2012.

[24] F. Wedyan and S. Abufakher, "Impact of design patterns on software quality: a systematic literature review," *IET Software*, vol. 14, no. 1, 2020.

[25] G. Borrego, A. L. Morán, R. R. P. Cinco, O. M. Rodríguez-Elias, and E. García-Canseco, "Review of approaches to manage architectural knowledge in agile global software development," *IET Software*, vol. 11, no. 3, pp. 77–88, 2017.

[26] F. O. Bjørnson and T. Dingsøyr, "Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used," *Information and Software Technology*, vol. 50, no. 11, pp. 1055–1068, 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584908000487

[27] B. Ray, D. Posnett, P. Devanbu, and V. Filkov, "A large-scale study of programming languages and code quality in github," *COMMUNICATIONS OF THE ACM*, vol. 60, no. 10, pp. 91–100, 2017.

[28] M. Dagpinar and J. H. Janke, "Predicting maintainability with object-oriented metrics - an empirical comparison," *Reverse Engineering - Working Conference Proceedings*, pp. 155–164, 12 2003.

[29] T. Coulin, M. Detante, W. Mouchère, and F. Petrillo, "Software architecture metrics: A literature review," 2019.

[30] T. C. I. J. S. 7, Software, and systems engineering, "Iso/iec 25010:2011: Systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models," ISO, 2011, https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en [Online; accessed 14-December-2021]. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en

[31] S. Gupta and A. Chug, "An extensive analysis of machine learning based boosting algorithms for software maintainability prediction," *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 7, no. 2, pp. 89–109, October 2021.

[32] S. Omari and G. Martinez, "Enabling empirical research: A corpus of large-scale python systems," 2018.

[33] S. contributors, "Sympy," 2021, https://github.com/sympy/sympy [Online; accessed 18-December-2021]. [Online]. Available: https://github.com/sympy/sympy

[34] ——, "Sentry," 2021, https://github.com/getsentry/raven-python [Online; accessed 18-December-2021]. [Online]. Available: https://github.com/getsentry/raven-python

[35] ——, "Scrapy," 2021, https://github.com/scrapy/scrapy [Online; accessed 23-April-2022]. [Online]. Available: https://github.com/scrapy/scrapy

[36] N. contributors, "Numba," 2022, https://github.com/numba/numba [Online; accessed 23-April-2022]. [Online]. Available: https://github.com/numba/numba

[37] M. Lacchia, "Radon 4.1.0 documentation," 2020, https://radon.readthedocs.io/en/latest/commandline.html#the-raw-command [Online; accessed 19-April-2022]. [Online]. Available: https://radon.readthedocs.io/en/latest/commandline.html#the-raw-command

[38] C. contributors, "Cython," 2022, https://github.com/cython/cython [Online; accessed 23-April-2022]. [Online]. Available: https://github.com/cython/cython

[39] Y.-D. contributors, "Youtube-dl," 2022, https://github.com/rg3/youtube-dl [Online; accessed 23-April-2022]. [Online]. Available: https://github.com/rg3/youtube-dl

[40] E. contributors, "Electrum - lightweight bitcoin client," 2022, https://github.com/spesmilo/electrum [Online; accessed 23-April-2022]. [Online]. Available: https://github.com/spesmilo/electrum

[41] A. contributors, "Ansible," 2021, https://github.com/ansible/ansible [Online; accessed 17-December-2021]. [Online]. Available: https://github.com/ansible/ansible

[42] S. contributors, "Salt," 2021, https://github.com/saltstack/salt [Online; accessed 18-December-2021]. [Online]. Available: https://github.com/saltstack/salt

Appendix

7.1 Pylint: Refactor Scores

The score in Pylint is a value out of 10, with 10 being the best.

Pylint has a number of types of messages:

1. Convention
2. Error
3. Fatal
4. Information
5. Refactor
6. Warning

There is a very large list of all messages at Overview of All Pylint Messages.

We spend most of our time reviewing the Refactor messages, found at Pylint Refactor Messages.

Additionally, the Convention messages are also helpful for review, found at Pylint Convention Messages.

7.2 Radon: Maintainability Index

In this paper, we use Radon as one method for finding the Maintainability Index (MI) for projects, at the module (file) level.

MI Score	Rank	Maintainability
100 - 20	A	Very high
19 - 10	B	Medium
9 - 0	C	Extremely Low

7.3 Data: Repository Refactor Messages

TODO: fill this table with ALL the respositories

```
-- Find total number of refactor messages
SELECT
  t1.name AS "Repository Name",
  SUM(t1.n::numeric) AS "Total Refactor"
FROM new_pylint_metrics_by_msg AS t1
  WHERE t1.code = 'R'
GROUP BY t1.name
ORDER BY SUM(t1.n::numeric) DESC;

-- Find "worst offender" messages for each repository
SELECT
  t2.name AS "Repository Name",
  t2.msg AS "Most Common Refactor Message",
  SUM(t2.n::numeric) AS "Most Common Refactor Message Count"
FROM new_pylint_metrics_by_msg AS t2
  WHERE name = 'sympy'  -- set repository name here
  AND code = 'R'
GROUP BY t2.name, t2.msg
ORDER BY SUM(t2.n::numeric) DESC
LIMIT 1;
```

Repository Name	Msg Count	Top Msg	Top Msg Count
sympy	14,206	too-many-arguments	6,601 (46%)
ansible	10,431	no-else-return	1,711 (16%)
salt	7,814	too-many-arguments	1,640 (21%)
ranger	109	no-else-return	29 (27%)
sentry	66	no-self-use	32 (48%)
raven-python	20	too-few-public-methods	7 (35%)

Table 7.1: Total refactor messages per repository. Also provided with the refactor message that had the most warnings and its total count (and the percentage of that message from the total refactor warnings for that repository).

7.4 Data: Repository Refactor-to-SLOC Ratio

It is good to note that the value for the Source Line of Code (SLOC), when added to multi-line comments, single-line comments, and blank lines, should add up to the total lines of code in the project [37]. For this table, we are looking only at the SLOC, which are lines that are only actual code (no comments or blank lines).

```
-- https://radon.readthedocs.io/en/latest/commandline.html#the-raw-command
-- The equation sloc+multi+singlecomments+blank=loc should always hold.

SELECT
  t1_radon_raw.name AS "Repository",
  SUM(
    CASE
      WHEN t3_python_metrics.code = 'R' THEN n::numeric
      ELSE 0
    END
  ) AS "Total Refactor Msgs",
  SUM(t1_radon_raw.sloc::numeric) AS "Total Project SLOC",
  CONCAT(TRUNC(
      SUM(
        CASE
          WHEN t3_python_metrics.code = 'R' THEN n::numeric
          ELSE 0
        END
      ) / SUM(t1_radon_raw.sloc::numeric) * 100,
      2 -- Number of decimal places
    ), ''
  ) AS "Ratio",
  CONCAT(
    TRUNC(AVG(t2_radon_mi.mi::numeric), 2),
    ' ±',
    TRUNC(STDDEV_POP(t2_radon_mi.mi::numeric), 2)
  ) AS "Avg Project MI",
  CASE
    WHEN AVG(t2_radon_mi.mi::numeric) >= 20 THEN 'A'
      WHEN AVG(t2_radon_mi.mi::numeric) >= 10 THEN 'B'
      ELSE 'C'
  END AS "MI Rank",
  CASE
    WHEN AVG(t2_radon_mi.mi::numeric) >= 20 THEN 'Very high'
      WHEN AVG(t2_radon_mi.mi::numeric) >= 10 THEN 'Medium'
      ELSE 'Extremely low'
  END AS "Maintainability",
  CONCAT(TRUNC(
    SUM(t1_radon_raw.comments::numeric)
      / SUM(t1_radon_raw.sloc::numeric)
      * 100,
    2  -- Number of decimal places
  ), '%') AS "Total Project Comment-to-SLOC Ratio"
FROM new_radon_raw_metrics AS t1_radon_raw
INNER JOIN new_radon_mi_metric AS t2_radon_mi
  ON t1_radon_raw.module_name = t2_radon_mi.module_name
    INNER JOIN new_pylint_metrics_by_msg AS t3_python_metrics
      ON t1_radon_raw.module_name = t3_python_metrics.module_name
        INNER JOIN new_repo_details AS t4_repo_details
          ON t1_radon_raw.name = t4_repo_details.repo_name_2
GROUP BY t1_radon_raw.name
ORDER BY (
  SUM(
      CASE
      WHEN t3_python_metrics.code = 'R' THEN n::numeric
      ELSE 0
      END
    ) / SUM(t1_radon_raw.sloc::numeric)
) DESC;
```

Repository	Refactor Msgs	Project SLOC	Ratio	Avg Project MI	MI Rank	Maintainability	Comment-to-SLOC Ratio
raven-python	20	1474	1.35	87.02 ±13.75	A	Very high	8.41%
scrapy	381	58768	0.64	64.47 ±17.72	A	Very high	5.59%
numba	126	20192	0.62	62.55 ±21.08	A	Very high	13.11%
sentry	66	10770	0.61	87.38 ±19.47	A	Very high	8.89%
boto	1398	237761	0.58	58.00 ±22.88	A	Very high	14.91%
celery	1626	302080	0.53	41.73 ±24.26	A	Very high	6.00%
pyramid	452	85302	0.52	70.25 ±24.18	A	Very high	10.29%
mopidy	211	40381	0.52	59.15 ±20.46	A	Very high	7.19%
sympy	14206	2873930	0.49	31.16 ±26.81	A	Very high	8.08%
aws-cli	585	126303	0.46	61.73 ±20.44	A	Very high	15.89%
luigi	299	68973	0.43	53.56 ±22.51	A	Very high	14.81%
networkx	568	147166	0.38	44.66 ±20.48	A	Very high	24.00%
scikit-learn	1485	396109	0.37	41.74 ±19.67	A	Very high	13.88%
peewee	163	44595	0.36	31.26 ±20.32	A	Very high	4.76%
mongo-python-driver	418	117448	0.35	43.62 ±23.74	A	Very high	12.44%
buildbot	1129	318794	0.35	56.60 ±23.29	A	Very high	17.88%
django	1720	485681	0.35	57.28 ±28.51	A	Very high	12.97%
pandas	1197	357537	0.33	42.07 ±25.86	A	Very high	9.24%
mitmproxy	488	148319	0.32	56.21 ±21.53	A	Very high	5.60%
biopython	1930	615445	0.31	43.54 ±23.97	A	Very high	17.19%
django-rest-framework	241	77488	0.31	50.68 ±28.56	A	Very high	9.35%
werkzeug	348	116937	0.29	34.06 ±20.89	A	Very high	6.20%
autobahn-python	541	182842	0.29	46.86 ±27.75	A	Very high	19.61%
sqlalchemy	2040	703340	0.29	32.98 ±29.94	A	Very high	7.17%
cobbler	221	77348	0.28	58.76 ±22.87	A	Very high	12.01%
scikit-image	499	187819	0.26	53.87 ±23.64	A	Very high	9.64%
nova	2593	981563	0.26	64.72 ±31.14	A	Very high	16.51%
paramiko	257	97387	0.26	43.79 ±22.00	A	Very high	13.78%
tornado	326	123535	0.26	37.14 ±22.48	A	Very high	17.28%
conda	660	260363	0.25	44.82 ±27.94	A	Very high	16.42%
beets	471	186165	0.25	43.66 ±26.32	A	Very high	15.64%
astropy	1965	780617	0.25	40.20 ±28.61	A	Very high	19.38%
salt	7814	3131332	0.24	45.69 ±24.78	A	Very high	8.43%
ranger	109	44487	0.24	45.74 ±25.48	A	Very high	9.49%
pelican	117	48096	0.24	35.95 ±19.42	A	Very high	7.82%
swift	1159	523880	0.22	37.13 ±24.09	A	Very high	14.30%
pip	1215	590140	0.20	47.52 ±28.78	A	Very high	12.61%
matplotlib	1926	943406	0.20	28.85 ±27.37	A	Very high	11.10%
fail2ban	157	84340	0.18	48.41 ±20.03	A	Very high	30.77%
mongoengine	145	80473	0.18	30.75 ±29.57	A	Very high	9.24%
ansible	10429	5818380	0.17	46.82 ±22.87	A	Very high	5.98%
web2py	1555	903496	0.17	37.78 ±29.88	A	Very high	10.16%
electrum	722	425576	0.16	39.41 ±28.06	A	Very high	9.14%
youtube-dl	1003	667075	0.15	54.16 ±19.95	A	Very high	5.04%
cython	1718	1183863	0.14	31.02 ±29.19	A	Very high	12.74%

Table 7.2: Total refactor messages per repository, total project source line of code (SLOC), the ratio of refactor messages to SLOC, average Maintainability Index (MI) with standard deviation, and code comment-to-SLOC ratio.

7.5 Data: Repository Refactor By Message

This section contains information from the repositories that were on the edges of refactor message counts.

The sections for the projects with the highest ratio of refactor messages compared to the source lines of code are contained in Subsections 7.5.1 (Raven-Python), 7.5.2 (Scrapy), and 7.5.3 (Numba).

The sections for the projects with the lowest ratio of refactor messages compared to the source lines of code are contained in Subsections 7.5.4 (Cython), 7.5.5 (YouTube-DL), and 7.5.6 (Electrum).

The sections for the projects with the most total refactor messages are contained in Subsections 7.5.7 (Sympy), 7.5.8 (Ansible), and 7.5.9 (Salt).

```
SELECT
  t2.name AS "Repository Name",
  t2.msg AS "Most Common Refactor Message",
  SUM(t2.n::numeric) AS "Most Common Refactor Message Count"
FROM new_pylint_metrics_by_msg AS t2
  WHERE name = 'sympy'  -- Change repository name for each one
  AND code = 'R'
GROUP BY t2.name, t2.msg
ORDER BY SUM(t2.n::numeric) DESC
LIMIT 10;
```

7.5.1 Raven-Python: Most Common Refactor Messages

This repository, Raven-Python [34], had the highest ratio of refactor messages when compared to the number of source lines of code.

Only 6 were returned because there is such a low number of refactor warning messages for this project.

Repository Name	Most Common Refactor Message	Most Common Refactor Message Count
raven-python	too-few-public-methods	7
raven-python	useless-object-inheritance	5
raven-python	no-self-use	4
raven-python	no-else-return	2
raven-python	inconsistent-return-statements	1
raven-python	too-many-branches	1

Table 7.3: The 6 most common refactor messages for Raven-Python.

7.5.2 Scrapy: Most Common Refactor Messages

This repository, Scrapy [35], had the second highest ratio of refactor messages when compared to the number of source lines of code.

Repository Name	Most Common Refactor Message	Most Common Refactor Message Count
scrapy	useless-object-inheritance	102
scrapy	no-self-use	79
scrapy	inconsistent-return-statements	49
scrapy	no-else-return	44
scrapy	too-few-public-methods	42
scrapy	too-many-arguments	29
scrapy	too-many-instance-attributes	17
scrapy	too-many-locals	5
scrapy	no-else-raise	4
scrapy	too-many-return-statements	4

Table 7.4: The 10 most common refactor messages for Scrapy.

7.5.3 Numba: Most Common Refactor Messages

This repository, Numba [36], had the third highest ratio of refactor messages when compared to the number of source lines of code.

Repository Name	Most Common Refactor Message	Most Common Refactor Message Count
numba	too-few-public-methods	54
numba	no-else-return	25
numba	useless-object-inheritance	13
numba	no-self-use	8
numba	too-many-branches	5
numba	too-many-public-methods	4
numba	too-many-arguments	4
numba	too-many-locals	3
numba	too-many-return-statements	2
numba	too-many-instance-attributes	2

Table 7.5: The 10 most common refactor messages for Numba.

7.5.4 Cython: Most Common Refactor Messages

This repository, Cython [38], had the lowest ratio of refactor messages when compared to the number of source lines of code.

Repository Name	Most Common Refactor Message	Most Common Refactor Message Count
cython	no-else-return	417
cython	no-self-use	293
cython	too-many-branches	182
cython	too-many-arguments	162
cython	useless-object-inheritance	132
cython	too-many-locals	107
cython	too-few-public-methods	79
cython	too-many-statements	79
cython	inconsistent-return-statements	61
cython	too-many-instance-attributes	53

Table 7.6: The 10 most common refactor messages for Cython.

7.5.5 YouTube-DL: Most Common Refactor Messages

This repository, YouTube-DL [39], had the second lowest ratio of refactor messages when compared to the number of source lines of code.

Repository Name	Most Common Refactor Message	Most Common Refactor Message Count
youtube-dl	too-many-locals	413
youtube-dl	too-many-branches	116
youtube-dl	inconsistent-return-statements	85
youtube-dl	too-many-statements	84
youtube-dl	no-else-return	83
youtube-dl	too-many-arguments	49
youtube-dl	no-self-use	49
youtube-dl	no-else-raise	31
youtube-dl	useless-object-inheritance	26
youtube-dl	too-many-nested-blocks	19

Table 7.7: The 10 most common refactor messages for YouTube-DL.

7.5.6 Electrum: Most Common Refactor Messages

This repository, Electrum [40], had the third lowest ratio of refactor messages when compared to the numbe

Repository Name	Most Common Refactor Message	Most Common Refactor Message Count
electrum	no-self-use	163
electrum	too-many-locals	87
electrum	inconsistent-return-statements	77
electrum	no-else-return	75
electrum	too-few-public-methods	69
electrum	too-many-arguments	51
electrum	too-many-instance-attributes	38
electrum	too-many-statements	38
electrum	too-many-branches	32
electrum	too-many-public-methods	26

Table 7.8: The 10 most common refactor messages for Electrum.

7.5.7 Sympy: Most Common Refactor Messages

This repository is SymPy [33], one of the repositories with the highest count of refactor warning messages.

Repository Name	Most Common Refactor Message	Most Common Refactor Message Count
sympy	too-many-arguments	6,601
sympy	no-else-return	2,813
sympy	no-self-use	831
sympy	inconsistent-return-statements	809
sympy	too-many-locals	741
sympy	too-many-branches	608
sympy	too-many-return-statements	376
sympy	too-many-statements	294
sympy	too-many-nested-blocks	159
sympy	too-many-ancestors	148

Table 7.9: The 10 most common refactor messages for SymPy.

7.5.8 Ansible: Most Common Refactor Messages

This repository is Ansible [41], one of the repositories with the highest count of refactor warning messages.

Repository Name	Most Common Refactor Message	Most Common Refactor Message Count
ansible	no-else-return	1,711
ansible	too-many-branches	1,265
ansible	inconsistent-return-statements	1,138
ansible	useless-object-inheritance	1,060
ansible	no-self-use	869
ansible	too-many-locals	816
ansible	too-many-statements	633
ansible	too-many-arguments	582
ansible	too-few-public-methods	512
ansible	too-many-nested-blocks	446

Table 7.10: The 10 most common refactor messages for Ansible.

7.5.9 Salt: Most Common Refactor Messages

This repository is Salt [42], one of the repositories with the highest count of refactor warning messages.

Repository Name	Most Common Refactor Message	Most Common Refactor Message Count
salt	too-many-arguments	1,640
salt	no-else-return	1,565
salt	too-many-branches	1,024
salt	too-many-locals	891
salt	too-many-statements	495
salt	too-many-nested-blocks	340
salt	inconsistent-return-statements	278
salt	too-many-return-statements	242
salt	too-few-public-methods	206
salt	useless-object-inheritance	206

Table 7.11: The 10 most common refactor messages for Salt.

7.6 Data: Pylint Scores for Best & Worst

This data was generated by cloning the existing repository and then deciding which folder contained most python files. Then check out a specific commit hash (beginning at the stage where our data was generated and moving towards current) and run pylint. The Pylint rating is a score out of 10.

```
########## Cython ##########
# Clone the repository
git clone https://github.com/cython/cython.git

# Navigate into the repository folder
cd cython

########## Raven-Python ##########
# Clone the repository
git clone https://github.com/getsentry/raven-python

# Navigate into the repository folder
cd raven-python

########## Generic Steps ##########
# Checkout the commit hash
git checkout <hash>

# Find the date of the last commit at this stage
git log

# Run pylint
pylint ./*
```

7.6.1 Cython Pylint Scores

Release	Commit Hash	Commit Date	Pylint
(start)	433e6992ca89e0c7059b87cbae0e9536f11aa58f	14-Dec-2018	8.11
0.29.25	488e21a34259258210f0be92c58618e1ea8a928f	6-Dec-2021	8.14
0.29.26	3028e8c7ac296bc848d996e397c3354b3dbbd431	16-Dec-2021	8.13
3.0.0a10	ddaaa7b8bfe9885b7bed432cd0a5ab8191d112cd	6-Jan-2022	7.95
0.29.27	229a4531780863c8a5c311d6b3c70a545988f85f	28-Jan-2022	8.13
0.29.28	27b6709241461f620fb25756ef9f1192cc4f589a	17-Feb-2022	8.13
(latest)	d48d0a038e2838d3bd2981e2687557a86936076b	21-Apr-2022	7.91

Table 7.12: Pylint score from various releases of Cython, the "best" ranked repository for maintainability.

7.6.2 Raven-Python Pylint Scores

Release	Commit Hash	Commit Date	Pylint
(start)	2d55add71f3a4d4cbb86ba56770b0e5038cb57cf	24-Oct-2011	2.68
5.3.0	0c9b082a5f6dc70e7526e9107688a19d75a9da45	30-Apr-2015	4.93
5.4.0	3b0f6a0374b122d0b26cbdd641a70c4e01316598	6-Jul-2015	5.01
6.2.1	12b388a76dba4d1de82314f0184efcfcd1e2070a	22-Sep-2017	5.92
6.3.0	1a0d697dd11459bfa8ce933b3b53264098dad60d	29-Oct-2017	5.96
6.4.0	15ccf495f070a9dd8b3a0501cd1e7225429ca29c	11-Dec-2017	5.93
6.5.0	a2923ea42aeb0bfc8180dbec64dd037dfe381fc1	17-Jan-2018	5.95
6.6.0	f579e6809b01d27da5fe515d8572b497c98b4b43	14-Feb-2018	5.95
6.7.0	2c4ed64beecf9af4b45d4028eae7d540fa8df767	18-Apr-2018	5.85
6.8.0	595d69558d8a093b72423b07404c25863b1d0f31	12-May-2018	5.86
6.10.0	d7d14f61b7fb425bcb15512f659626648c494f98	19-Dec-2018	5.89
(latest)	5b3f48c66269993a0202cfc988750e5fe66e0c00	18-Jan-2021	5.89

Table 7.13: Pylint score from various releases of Raven-Python, the “worst” ranked repository for maintainability.

Release	Commit Hash	Commit Date	Pylint
(latest)	4cce4b5d9f5b34379879a332b320e870ce0ce1ad	20-Apr-2022	7.03

Table 7.14: The latest Pylint score from Sentry-Python, which replaced Raven-Python.