Structural Quality & Software Evolution: Paper Summary

Alison Major
Department of Computer and Mathematical Sciences
Lewis University
Romeoville, Illinois, USA
AlisonMMajor@lewisu.edu

I. Introduction

This paper will investigate how well the Pylint score (more specifically, the Refactor score) measures a system's structural quality. To do this, we will analyze the correlation between the Refactor score and the ease of adding features to the system.

Good architecture takes into account maintainability. The effort to make a feature work should be easy and localized in maintainable projects. We will measure locality by the number of files edited in a commit. We will focus on commits that represent new features rather than bug fixes.

This paper will review how structural quality can impact software evolution.

II. Working with Data

The work done by Dr. Omari and Dr. Martinez involves collecting a sub-set 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 [1]. We will use this work to select appropriate Python systems for review by collecting meta-data on these code systems. We hope to understand the impact of structure quality on the software evolution process with this information.

To understand the scores we'll be working with, we must understand what Pylint itself is doing. Then, through the documentation of Pylint, we can understand how to use it and the scores it will provide [2]. We can also review the documentation to know how the Pylint Score is calculated, as well as the various features that the Refactor Score takes into account [3]. Finally, in respect to Python, it is also important to understand PEP 8, as this is the default set of standards that Pylint uses to judge Python code [4].

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 a study of the impact different programming languages may have on the code quality [5]. The group's methods and the ideas they have formed may help create direction and assumptions in our research.

III. Insights into Maintainability

Reviewing the "Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models" maintained by ISO may also provide interesting insights [6]. Does the Pylint checker follow the quality models outlined here? Are there benefits or drawbacks to the models that the ISO/IEC:25010:2011 suggests? These could influence more thoughts on interpreting final data results.

Within the course's textbook, "Software Architecture in Practice," chapter 18 provides some insight in documentation around architecture [7]. When reviewing code quality scores, it would be interesting to check some of the documentation around the best scoring software systems and some of the worst scoring software systems. Do these projects have adequate documentation? Does the level of documentation correlate to the ability to maintain a decent score? I would be curious if contributing developers are positively influenced by good documentation or other factors to maintainable code structures.

IV. Related Works

The authors of "Measurement and refactoring for package structure based on complex network" recently reviewed a similar idea with the focus on cohesion and coupling over time for a project [8]. It will be interesting to read and understand their findings and see how it compares to the data that we collect and understand.

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 [9]. This variable in the maintainability of a software system can influence how easy or difficult it is to make a change. Referencing the findings from these authors and the guidelines they provide for maintaining readability could be helpful when building conclusions from the data we collect ourselves.

Another 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 [10]. 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. The research that this group provides could provide valuable insights into our project.

V. Conclusion

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 in regards to maintainability. Good projects will inevitably continue to grow and evolve. Understanding methods to keep code refactors on a level that makes code easy to change is important. We may also find that projects with worsening scores slow down with updates and have reduced engagement.

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.

Understanding the impact of structural quality on the evolution of a project can provide compelling perspectives.

References

[1] Safwan Omari and Gina Martinez. Enabling empirical research: A corpus of large-scale python systems, 2018. [Provided by Dr. Omari].

[2] Logilab and contributors. Pylint. Logilab, 2020. https://pylint.org/ [Online; accessed 7-December-2021].

[3] PyCQA Logilab and contributors. Pylint features. Logilab and PyCQA, 2021. https://pylint.pycqa.org/en/latest/technical_reference/features.html#reports-options [Online; accessed 7-December-2021].

[4] Python Software Foundation and contributors. Pep 8 – style guide for python code. Heroku Application, 2021. https://www.python.org/dev/peps/pep-0008/ [Online; accessed 7-December-2021].

[5] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. *COMMUNICATIONS OF THE ACM*, 60(10):91–100, 2017.

[6] Technical Committees: ISO/IEC JTC 1/SC 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/standard/35733.html [Online; accessed 7-December-2021].

[7] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice: Third Edition*. Addison-Wesley Professional, 2012.

[8] Yangxi Zhou, Yanran Mi, Yan Zhu, and Liangyu Chen. Measurement and refactoring for package structure based on complex network. *Applied Network Science*, 5(50), 2020.

[9] Valentina Piantadosi, Fabiana Fierro, Simone Scalabrino, Alexander Serebrenik, and Rocco Oliveto. How does code readability change during software evolution? *Software Qual J*, 25:5374—-5412, 2020.

[10] Robert Baggen, José, Pedro Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. *Software Qual J*, 20:287—307, 2012.