Structural Quality & Software Evolution: First Draft

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

***Abstract*—Some software engineering projects fail to evolve, which leads to them becoming obsolete. This topic is interesting and important to developers because software that fails to evolve will fail to generate user engagement, which leads to loss of revenue. Say what my solution achieves. Say what follows from my solution.**

I. Introduction

When building systems as software engineers, we have several areas of concern. How much will it cost? How long will it take to deliver? What will the quality be? Often, the cost and time-to-market are the two concerns given the highest priority in a project. However, the quality must be considered to preserve the system's longevity.

One way to understand the quality around a system is to discuss its "maintainability," that is, the ease of receiving new features or resolving bugs. For example, if a system is tightly coupled, it may be that adjusting one area to add a new feature requires touching several other parts of the system.

To give some simple examples for tightly and loosely coupled systems, let us consider your eyes. If you are a near-sighted individual, a loosely coupled solution to improve your vision is to get glasses or contacts. You can change out the glasses for different types and styles while providing a solution to your problem (poor vision). However, you cannot simply change your eyes; these are tightly coupled to your body and would be difficult to remove and swap for another set of eyes without considerable work to separate and reattach them to the correct systems. By having a loosely coupled system (wearing glasses), we allow for easy feature changes (turn those glasses into sunglasses! pick a new style!) and easy bug fixes (adjust the prescription).

Like how it may be easier to change how we look and how well we can see by using prescriptive eyewear, creating a structured software system will impact how the software can evolve with new features. If a system is difficult to evolve, users will lose favor as it cannot offer the newest features that competitors may offer.

State your contributions. Do not leave the reader to guess what my contributions are. Should be provable (refutable). Contributions will be things we have proven or worked out in the following sections. Can link to those sections as we describe those contributions.

CLAIM: We can use maintainability scores to see how structure impacts evolution (Section IV-A).

CLAIM: Following shared standards improves maintainability (Section IV-B).

CLAIM: Documentation can improve maintainability (Section IV-C).

remove page break - the problem - 1 page

II. The Problem

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

The systems we create could be customer-facing web applications, games, or internal applications used within a business to carry out tasks. Regardless of the type of system, without evolving with the user's needs, the product will no longer provide usefulness. However, even in the business world with internal systems, needs change, and how a system must be used needs to have enough flexibility for tasks to continue to be completed.

Let's consider what software evolution is, from this simple definition on Wikipedia [1]:

> Software evolution is the continual development of a piece of software after its initial release to address changing stakeholder and/or market requirements.

When a system cannot evolve, the users mainly feel the impact. 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, and so will the users. This 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 are dependent on the continued success of the software. Software evolution will allow the system to adapt to new or changing business requirements, fix bugs and defects, and also 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.

To keep a system up-to-date, we must add new features. There may be needs to improve performance or reliability of a system, especially if the user-base expands.

Security can also impact the needs 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.

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 characteristic of a system will ensure that bugs can be fixed quickly, but new features should be easy to add as the users' needs evolve.

If a system is large, it may involve a number of steps in order to get changes through. For example, a process like that in "Fig. 1" shows 7 indivudal steps just to get to a new feature release. Any amount of complexity can make this processes even more cumbersome.

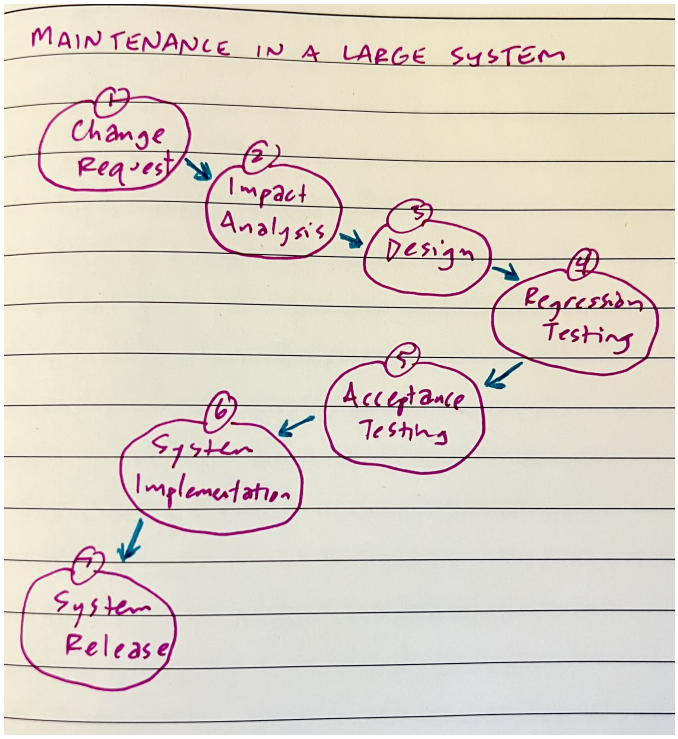

Fig. 1. Large systems take a number of steps to add new features.

examples? anything about other evolution processes? better diagrams? See example diagrams by searching for "software evolution" on google

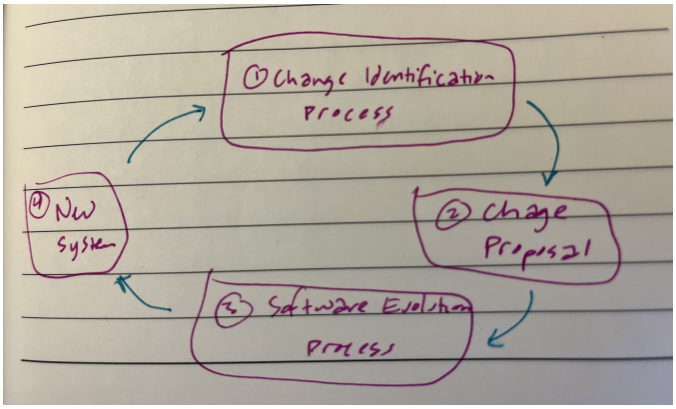

Fig. 2. New features are part of the evolution process.

remove page break - my idea - 2 pages

III. My Idea RENAME THIS SECTION

The structural quality of a software system will impact the software evolution. 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, and so on. One of the things that should also be on the planning list is for 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 adds new features.

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 have been ignored. However, 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), 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" even claimed that over 90% of the costs for a typical software system come up in the maintenance phase [2]. 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 by fixing bugs can be impacted by its maintainability. Any successful piece of software will inevitably need to be maintained.

There is a distinction to be made between **software maintenance** and **software evolution**. Software maintenance is what we'll refer to as bug resolution and for minor enhancements. For example, when we must fix a broken route in the application, or provide subtle improvement on the user experience, we can consider this normal maintenance. 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.

Evolution of software can be a result of new laws that have come into being. As technology itself changes, governing bodies must continually revisit policies on data collection and information sharing. This may lead to adaptations that result from the new laws, or even adaptations to take advantage of new technlogies.

It is also fair to say that systems will change because we can never fully determine the needs of a user at the start of a project. In fact, it would be safe to say that the needs of the user will change over time themselves. This leads to a never-ending project that will always need some form of enhancements.

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 devel-

opments while also slowing progress. These laws specifically 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 summarized below. [3]

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 continue to 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 the added volume of the code from new features, or even as a result of 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 personel, or users, must still be able to continue to use and work within the system at the same "level of mastery." If the system grows and changes excessively, the mastery of the system 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 in such a way that the system audience needs time to master the new interfaces or new tools. Because of this natural "slow-down" for excessive change occurs, the average incremental growth will remain steady. We can see a simplified visual in "Fig. 3" 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.

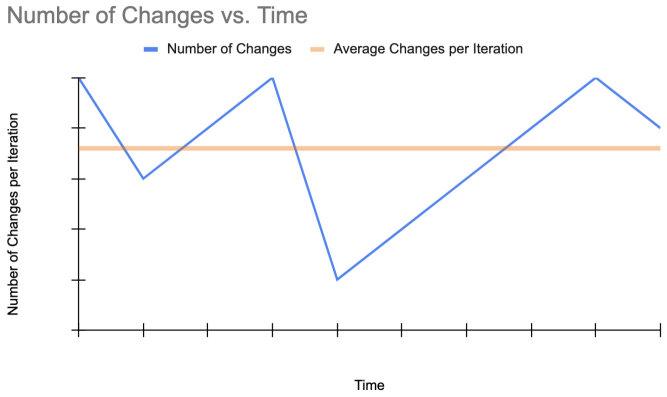

Fig. 3. A simplified visual of Lehman's fifth law, "Conservation of Familiarity."

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 quality of the system will appear to decline. We must continue adapting for even the appearance of maintained quality of a system.

With all of these characteristics surrounding the evolution of software, we have the benefit of the Internet that has positively improved the experience. Two common resources currently available to developers have impacted software evolution [1]:

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

These two suggestions are evident if you are a developer. It is quite possible that you regularly use resources like StackOverflow to find solutions to problems, as well as using open source tools that

you and your team can contribute to or adjust to your specific needs.

Despite the nuanced differences between *maintainability* and *evolution*, the two characteristics run parallel to each other. If a system is easy to maintain, it follows that 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.

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 when it has better Pylint scores. 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.

any examples for this section?

remove page break - the details - 5 pages

IV. The Details

A. Maintainability Scores

claim 1

To understand the scores we will 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 [4]. 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 [5]. Finally, in respect to Python, it is also essential to understand PEP 8, as this is the default set of standards that Pylint uses to judge Python code [6].

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 [7]. 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 [8]. 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 [9]. 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.

B. Shared Standards

claim 2

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 [10]. 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.

C. Documentation

claim 3

Within the course's textbook, "Software Architecture in Practice," chapter 18 provides some insight in documentation around architecture [11]. 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.

remove page break - related work - 1 to 2 pages

V. Related Work

give credit to those who have come before, 1-2 pages

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 [12]. 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.

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 [13]. The group's methods and the ideas they have formed may help create direction and assumptions in our research.

remove page break - conclusion - half page

VI. Conclusions & Further Work

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 refactor on a level that makes code easy to change. 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] Wikipedia contributors. Software evolution — Wikipedia, the free encyclopedia, 2021. [Online; accessed 12-December-2021].

[2] Frederick Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.

[3] Wikipedia contributors. Lehman's laws of software evolution, 2021. [Online; accessed 12-December-2021].

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

[5] 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].

[6] 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].

[7] 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.

[8] 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.

[9] 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.

[10] 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].

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

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

[13] 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.