Structural Quality & Software Evolution

Structural Quality & Software Evolution

Alison Major

Lewis University

2022

- My name is Alison Major.
- I have researched the impact that the structural quality of software in open source Python projects can have on the evolution of the software.

Introduction

Maintainability Index and Refactor Scores

- Areas of concern: cost, timeline, quality
- Quality is hard to understand
- Pylint & Radon are a static analysis tools
- Refactor violations point out code smells

- When we build new software solutions, there are several categories that we consider during planning.
- Namely, COST, TIME-TO-DELIVER, and also, often to a slightly smaller extent, QUALITY.
- Though quality can be a factor we consider when planning a project, it is a hard characteristic to understand and measure.
- There are a number of tools available for enforcing standards, some built into IDEs (like Visual Studio Code) and others as 3rd party linter tools.
- Two such tools are Pylint and Radon, which can be used to analyze Python projects.
- In particular, we will focus on the REFACTOR violations that are noted in the code reports from Pylint and Radon, as these warnings can lead us to code smells.

Keeping Users Engaged Long Term

Why does software evolution matter?

- Users find bugs
- Users want new features
- New security threats
- New laws from governing bodies

Need a thriving community of engaged users in order to keep apps and games successful.

In an open source system, need a thriving community of engaged developers in order to continue evolving.

- Sometimes "evolution" is just a matter of maintenance and fixing bugs that users find.
- However, if users are engaged and actively using software, they are going to push the software to its limits and they will want new features.
- We want this kind of engagement, because it means that we have users, which, depending on your business model, means that your software can be profitable.
- Over time, there can also be new security threats and new laws that require software to be updated.
- These different forces will all drive software evolution; otherwise a system will become unused and deprecated.

How do we ensure software evolution?

Keep the project maintainable.

- Bugs should be quick and easy to fix
- New features should be easy to add
- Consistent standards (naming, small methods, etc)

Software Maintenance
Large portion of project cost in a typical software system is in the maintenance phase.

- So how do we make sure that our projects that we plan to build will evolve?
- We need to make sure the project stays maintainable. Bugs and new features should be easy to handle
- Part of what allows for easy changes is having consistent standards.
  - Naming conventions
  - Keeping methods small
  - Having organized files
- The majority of a typical project's cost lies not in the development phase, but in the maintenance phase!
- Maintenance is an important part of a software solutions lifecycle; we cannot ignore it as we plan!

Measuring Maintainability

- Easy to maintain = Easy to evolve
- Pylint & Radon Maintainability Index (MI)
- PEP 8 is a set of Python standards
- Refactor Messages (Pylint)
  - Refactor warnings are generally "code smells"
  - Code smells point out problems in Architecture

- If we want to keep our projects easy to maintain so that they are easy to evolve, how do we MEASURE that maintainability?
- Pylint and Radon are tools I mentioned earlier that provide a measurement known as the Maintainability Index.
- You can get this value from several different tools, all using the same standard equation with slight modifications.
- In Pylint's reports, they also provide a number of different messages: Convention, Error, Fatal, Information, Refactor, Warning
- These types of messages are defined using PEP 8, which is a collection of Python standards than can be used as a list of "best practices"
- Today we focus on the refactor messages with the data we collected, as they will generally lead us to "code smells", which are problems in the software's architecture.

The Impact of Structural Quality

Other Maintainability Characteristics

- Low coupling, high cohesion
- Readability
  - Big commits reduce maintainability
  - PEP 8 enforces readability
- Confidence that metrics around software structure provide value in keeping systems maintainable (and therefore can evolve)

- Refactor warnings aren't the only only thing we should care about for good maintainability.
- We should also remember other good practices, like low coupling and high cohesion.
  - Coupling refers to the degree to which different modules depend on each other (we want modules to be independent from each other; separate; low coupling)
  - and cohesion refers to the degree in which elements of a module belong together (we want to bind related code together; high cohesion)
- We also want good readability (big commits can be hard to follow, as well as big files with a lot of code).
- PEP 8 standards enforce readability.
- And having confidence around the metrics that tools like Pylint and Radon can offer should be able to help us keep our systems maintainable.

The Impact of Structural Quality

Documentation and Maintainability

- Documentation holds the results of significant design decisions
- Can influence the ability to evolve because...
  - Enhances code understanding
  - Comprehensibility impacts maintainability in a positive way

- To add to the ideas of maintainability and readable code, we should also remember that documentation can feed into our success in the quality of our architecture.
- When we build a software system, good documentation can hold significant design decisions.
- This “historical record” can help developers in the open source community understand the code and why things are the way they are.
- Better comprehension makes it easier to maintain and evolve the code.

Related Work

Design Patterns and Software Quality

- Design patterns provide flexibility
- Classes with frequent changes are either...
  - Easy to extend (okay)

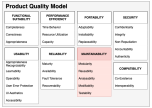
  ...or...
  - Correlate to other classes (high coupling... red flag!)
- We look at refactor score (code smell) not error score (bugs)

Keeping this in mind, we focus on *changes for system extensions and adaptiation*, not bug fixes.

- Through our studies, we also know that design patterns can lead us to better ways to build our code, which provides us with more flexibility.
- As we review open source systems, finding code classes with frequent changes generally mean one of two things. . .
  - the class is easy to extend (which is good!) (low coupling)
  - or the class is highly coupled (highly tied) to other classes which means updating one requires updates to another (code smell!)
- Keeping this in mind, we again are reminded to steer towards understanding the frequency of refactor messages, or code smells, that we get from Pylint.
- We are less concerned with error scores, which are just bugs and not an indicator of how maintainable the code is.

Related Work

Software Architecture and Maintainability (from ISO/IEC 25010:2011)



Keep these in mind for easier future development when adding or changing code.

- With all of these attributes and characteristics we've discussed, we can boil a lot of this down to the product quality model that the committee of the International Organization for Standardization and the International Electrotechnical Commission has provided us with:
  - Maintainability (our topic of interest)
  - Extensibility
  - Keeping things simple
  - Keeping things reusable
  - Focusing on performance
  - etc.
- Helping developers keep these in mind when adding or changing code will allow for easier FUTURE development.
- This all comes back to maintainability (our structural quality) and its impact on software evolution.

Methodology

Initial Repository Set

- Popular
- Long development history
- Multiple release cycles

Filtered Respository Set

At least 80% Python code and top 20th percentile in these categories:

- Long history of commits (2,968+ commits)
- Large number of contributors (90+ contributors)
- Many releases (44+ releases)
- Substantial Age (66.4+ months)

Results in 46 repositories for further research.

- Now we're ready for the data!
- We started with a larger set of open source python projects on GitHub that were popular (lots of stars), had a long commit history, and contained multiple release cycles.
- That set was collected from an earlier study conducted by Dr. Omari and other colleagues.
- From there, we filtered the set down to any projects that contained at least 80% Python code.
- And finally, we found the top 20th percentile in several key categories.
  - Long history of commits (2,968+)
  - Large number of contributors (90+)
  - Many releases (44+)
  - Substantial age (66.4+ months . . . over 5.5 years!)
- This left us with 46 repositories for deeper study.

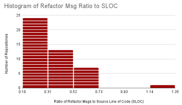
Results

- Radon MI for all repositories rank as grade "A" which is considered "very high maintainability"
- Open source systems with engaged community of developers tend to have higher scores
- For comparison, calculated ratio of refactor message count to SLOC as well as the average MI for a project.

Repo	Ratio	Avg MI	Status
cython	0.14	31.0	active
youtube-dl	0.15	54.16	active
electrum	0.16	39.41	active
numba	0.62	62.55	active
scrapy	0.64	64.47	active
raven-python	1.35	87.02	deprecated

- With our smaller data set on hand, we ran Radon against each repository to gain a Maintainability Index.
- For a very simplified view, we averaged the Maintainability Index of all files within each project.
- This gave us a single number for general comparison with each repository.
- This calculated value is shown in the "Avg MI" column.
- In Radon's grading system, any value over 20 is considered "very maintainable" code.
- We also used a count of the Pylint REFACTOR messages and calculated the ratio of messages to source-line-of-code for an idea of how many code smells each project contained, relative to its size.
- This calculated ratio is shown in the "Ratio" column.
- Listed in the table are the "best 3" (cython, youtube-dl, and electrum) and the "worst 3" (raven-python, scrapy, and numba) in regards to their refactor ratios.

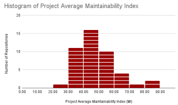
Results



Diligent development communities can keep refactor warnings low, regardless of system size (lines of code).

- With this information, we can get an idea of where these projects stand in regards to their refactor message ratios compared to each other.
- This histogram shows that the majority of our repository set have a very low ratio of refactor messages.
- This could indicate that our projects are all highly maintainable based on our assumptions that refactor messages are related to maintainability.
- Having few refactor messages (relative to the number of lines of code) means we have a smaller number of code smells. This is good for our architecture and quality!
- These low ratios may be a impacted by the data set we chose; perhaps open source projects with highly engaged communities tend to keep their code in a maintainable state, because this would be the only way for many people (over 90 contributors per project) to be involved.
- It has been interesting to see how low the ratio of refactor messages has been in our data set.

Results



Many repositories average in the mid-score to high-score. Radon considers 20 points and up to be very maintainable.

- To view the same repository set with their Maintainability Index averages, we have another histogram.
- This Radon score can range from 0 (awful) to 100 (perfect).
- Remember that Radon considers any value above 20 to be an "A" or "very maintainable".
- We can see that all of our projects receive an "A" grade, with the majority in the middle of the range (around 40 to 50).
- All projects in our data set were all mid- to high-range scores.

Conclusions

- Structural quality impacts software evolution.
- Good projects will grow and evolve.
- Poor structure leads to deprecation.
  - If the development community is engaged, deprecation of the project may lead to a fresh, improved code base.
- Open source and projects with many contributors are vulnerable to degrading maintainability.
  - Popular repositories with a long history of commits and releases (i.e. our repository data set) tend to have good maintainability.
  - The high maintainability is a testament to their longevity.

- From our set, the very best and very worst repositories are all still actively being improved and evolving, even today.
- An exception to this is our "worst" repository, raven-python, which was deprecated (this means the code is no longer supported), but in exchange for an improved system.
- The raven-python community didn't actually die, but the instead recognized that a better architecture was needed for the system to continue to evolve; they built a new code system (Sentry-Python) and shifted support there, where they have a better architecture and ability to evolve.
- Open source systems, and any project with many contributors or have many changes are vulnerable to degrading quality. It's just the nature of change over time.
- However, when we review a set of popular repositories that already have a long history of commits (these repositories are still active and over 5 years old!), we find that they tend to have good maintainability.

Recommendations

- Good architecture is important for evolution.
- Reliable quality metric can be a useful way to measure maintainability, which promotes ability to evolve a project.
- Pick a set of standards to maintain good architecture even with a large, open source community.
  - Limit complexity as project changes and grows.
  - SOLID principles.
  - Keep it DRY.
  - And other design patterns known to be best practice.
- Auto-enforce by using quality measurements for desired standards in the project's CI/CD pipeline.

- Where do we go from here?
- In regards to the data, we'd like to do more study to understand the correlation between the Pylint metrics and Maintainability Index in order to gain better insights into our assumptions.
- What we've found so far is a reinforcement of what many of us already know.
- Good architecture is important for software to be able to evolve.
- Reliable quality metrics can be helpful to keep an architecture ready for enhancements.
- Regardless of your code base, your team should understand what set of standards will work best for your software solution.
- Then auto-enforce these standards in your pipeline. This keeps everyone honest and will improve the ability to evolve your project.
- That is a high-level view of my research on how structural quality impacts software evolution.
- Thank you for your time and attention! Any questions?