CSC 510 Software Engineering, Group 14's essay on the use of the Linux Kernel's best practices on project 1

Connor Smith
cpsmith6@ncsu.edu

Ashwith Ramesh Shetty
ashetty2@ncsu.edu

Rahul Ramakrishnan
rramakr@ncsu.edu

Vasu Agrawal
vagrawa5@ncsu.edu

Leonardo Villalobos-Arias
lvillal@ncsu.edu

Muhammad Alahmadi

ACM Reference Format:
Connor Smith, Ashwith Ramesh Shetty, Rahul Ramakrishnan, Vasu Agrawal, Leonardo Villalobos-Arias, and Muhammad Alahmadi. 2021. CSC 510 Software Engineering, Group 14's essay on the use of the Linux Kernel's best practices on project 1. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Linux Kernel is a ongoing project with more than 28 million lines of code, 60,000 test files, and over 900,000 commits [3, 4]. This project has been extensively studied over the years in the fields of software evolution [2], software product lines [5], vulnerability analysis [6], and software testing [1] among many other applications. A key component to the success of this kernel is the development practices used by the Linux Foundation. To emulate the success of this development project, we tried to apply some of these practices into the process of developing our SE project. We approached this by guiding the way we work as a group, as well as reflecting this on our development artifacts. We believe we have successfully emulated the most important parts of each and everyone of these practices.

2 THE LINUX KERNEL BEST PRACTICES

According to the 2017 report on the Linux Kernel[1], there are 6 successful lessons than can be translated into development practices:

- Short release cycles are important.
- Process scalability requires a distributed, hierarchical development model.
- Tools matter.
- The kernel's strongly consensus-oriented model is important.
- A related factor is the kernel's strong "no regressions" rule.
- There should be no internal boundaries within the project

Note that in the report there are originally 7 principles. We excluded 1 of these (Corporate participation in the process is crucial), as they are difficult to apply in a project with a scope as small as hours (1 month, 6 people, project for a SE class).

[1]https://www.linuxfoundation.org/resources/publications/state-of-linux-kernel-development-2017/

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.1 Short release cycles

The short release cycles principle argues that shorter release cycles (around 2 months) are preferable over longer ones. This is justified as there is less code to be integrated, features get faster to the users (for which feedback can be collected faster). Developers are not as pressured to miss a cycle, as they can just wait until the next release to incorporate their code.

This principle is naturally built into the nature of this class' project, as there are 3 deadlines distanced by around 1 month each. Thus, we were strongly coerced to adapt this principle. To accomplish short release cycles, we had to plan accordingly: prioritize what objectives we have, divide them into small, achievable goals, and check that these goals are getting accomplished. We also have adapted to the mindset of peer-reviewed coding practices. For example, GitHub issues must be reviewed by one of the team members before they are closed; so that our code is tested quicker, before moving on to new tasks.

While one might think that the time spent in software development activities is mostly software development, shorter release cycles have also had us focus on other stuff: design, test cases, and documentation. Our software is thus a basic, functional product by the end of the first project; which will help us gather feedback to better improve this project (and our development practices) on project 2. The quality of our software out of the gate is also higher than if we rather left all testing and revisions at the end of a larger development cycle.

2.2 Distributed development

Distributed development involves spreading out the responsibilities for writing and reviewing code of a project over the team. This involves team members working on certain parts of the software and eventually merging the functionalities to build a comprehensive, complete software.

We used this principle to distribute our workload for the project. Each team member had certain tasks assigned to them. These included creating new functions, debugging, testing and writing up the documentation. The software ended up with basic functionality early on with complex commands being added as people finished their part of the code.

For example, one of us was responsible for creating the bot and another person handled the addition of the function that enables the user to get their schedule for the day. This principle helped us

when certain members could not attend the group meetings as we were still able to work on the project without any hindrance.

2.3 Tools matter

While not a principle itself, the tools matter lesson states that the Linux Kernel would have collapsed under its own weight were it not for the team having the tools to maintain its size. For example, distributed development would be impossible at that point without github or similar version control systems.

Similarly, we strongly rely on github for this project. Our development process centers a lot on many of the features github provides: issues, projects for keeping track of the state of these issues, automated actions (for test cases, linting, documentation), and discussion. We rely on multiple libraries, without which our project would not be able to have the release speed it currently has. And many of these processes are also automated by our IDEs (VS Code), which aids on our individual development tasks. Thus certainly, the right tools have pushed forward our project.

2.4 Consensus-oriented model

The Consensus-oriented model states that a proposed change cannot be integrated into the code base as long the majority of developers are opposed to it. In addition, no changes can be incorporated into the main branch unless they have been reviewed by a senior, knowledgeable developer. This is justified because no single group/individual should be able to make changes at the expense of other groups. Further it ensures the sanity of the repository.

We incorporated this principle within our project by ensuring that all the major functionalities of the project were discussed in detail before we started working on them. Also, any features implemented got rectified by some other developer before being added to the main branch. Once they are checked (and possibly rectified), we added them into our main branch. Doing this ensured that the developers were in sync with what everyone was working on and it provided them with a wholesome knowledge of the project.

2.5 No-regressions rule

The no-regressions rule states that if a software releases a specific interface (user, API), all future versions of that software should support that interface as well. For example, if a user of your software has a certain setup, this principle assures that any update to the software won't crash on that setup.

This principle does not directly apply to our project, as it deals with future releases, which start on phase two of development. However, right now what this means is that once the first version of our bot is released, we will not change the existing command list that the users are familiar with when upgrading the bot. We may change how the commands work behind the screen, and the way the bot stores information; but our promise is to not change the way the users interact with the bot.

This has helped us out by making us place more weight in our decisions from the beginning. We also have to focus on design and the important decisions before jumping on to code. This way, the no-regressions rule has made us place more emphasis in our design, and to stick more closely to that design.

2.6 Zero internal boundaries

By having no internal boundaries, developers of a software are allowed to change any part of that software, as long as that change is reasonable. This means that no part of the software is "owned" by someone, and instead can be modified by the community. This way, developers should have a more general viewpoint and knowledge about the entire project.

We endorsed the zero internal boundaries in our project by the way our work is distributed. While issues are handled by one developer (who does coding and testing), they must be checked by another developer before being considered done. This review process includes reading and commenting on their source code, reviewing that it works, and giving feedback on it. While this does not completely alleviate the issue (we still wanted to work on the stuff we worked on before), it has helped out the team gain a broader view of the source code. The zero internal boundaries principle has also made us better at knowledge transfer, as we all need to be able to work on all parts of the code. Thus, as we have documented and made all tools required to run the code more readily available, new developers should have an easier time getting to work on this project.

3 CLOSING THOUGHTS

While our project does not (and possibly will never) have the outreach, scope, size and importance of the Linux Kernel; we have as a development team learned from their lessons and improved our software engineering practices. To accomplish this, we have changed the way we developed and our development artifacts as well. The short release cycles has made us design and test our code with greater frequency, and focus on code quality more than quantity. The distributed development principle has made our work both more independent and less reliable on particular group members. The right tools for the job has made us focus less on getting things ready for work and more on getting the job done. The consensus oriented model has made the project driven by the group as a whole and not by particular individuals. The no-regressions rule has made us place more emphasis in our design and stick to our decisions. Lastly, the zero internal boundaries has made sure that our project can be picked up and maintained by other people. These development practices have improved our experience developing this project.

REFERENCES

[1] Imanol Allende, Nicholas Mc Guire, Jon Perez, Lisandro G Monsalve, Javier Fernandez, and Roman Obermaisser. 2021. Estimation of Linux kernel execution path uncertainty for safety software test coverage. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1446–1451.

[2] Ayelet Israeli and Dror G Feitelson. 2010. The Linux kernel as a case study in software evolution. *Journal of Systems and Software* 83, 3 (2010), 485–501.

[3] M Larabel. [n. d.]. The linux kernel enters 2020 at 27.8 million lines in git but with less developers for 2019. *Web page at linux. com, https://www. phoronix. com/scan. php* ([n. d.]).

[4] Johann Mortara and Philippe Collet. 2021. Capturing the diversity of analyses on the Linux kernel variability. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A*. 160–171.

[5] Alcemir Rodrigues Santos and Eduardo Santana de Almeida. 2015. Do# ifdef-based Variation Points Realize Feature Model Constraints? *ACM SIGSOFT Software Engineering Notes* 40, 6 (2015), 1–5.

[6] Alireza Shameli-Sendi. 2021. Understanding Linux kernel vulnerabilities. *Journal of Computer Virology and Hacking Techniques* (2021), 1–14.