Project 1 Rubric and Linux Kernel Best Practices

Pradhan Sudhanva Gowda Chetan Venkataramaiah
North Carolina State University
Raleigh, USA
pchetan@ncsu.edu

Wei Chen Chu
North Carolina State University
Raleigh, USA
wchu2@ncsu.edu

Steven Gregory Jones
North Carolina State University
Raleigh, USA
sjones9@ncsu.edu

Shikha Vasant Nair
North Carolina State University
Raleigh, USA
svnair@ncsu.edu

Alexander Viktor Snezhko
North Carolina State University
Raleigh, USA
avsnezhk@ncsu.edu

ABSTRACT

This paper explains the five the Linux Kernel Best Practices for Software Development, how they streamline development and the benefits of each, and discusses the connections between these and the items listed in the CSC 510 Software Engineering Project 1 Rubric.

CCS CONCEPTS

• **Software Development**; • **Software Development Practices** → *Linux Kernel*;

KEYWORDS

software development, Linux kernel best practices

ACM Reference Format:
Pradhan Sudhanva Gowda Chetan Venkataramaiah, Wei Chen Chu, Steven Gregory Jones, Shikha Vasant Nair, and Alexander Viktor Snezhko. 2021. Project 1 Rubric and Linux Kernel Best Practices. In *Project 1.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Linux Kernel is a vast open source project which has been in use for decades. Part of what makes it such an effective tool is the practices which have made it a prime example of collaborative development. These practices have made it possible for Linux to exist for as long as it has, without any compromise to its quality which can often be rampant in projects of its scope and longevity. So what gives Linux its ability to stick around for so long, with such frequent releases without issues which cause users to abandon it? Linux Kernel maintains a few simple but crucial rules for people who work on it, which have allowed it to stay in use for so long. These five practices are:

(1) Zero Internal Boundaries
(2) No Regressions Rule
(3) Consensus Oriented Model
(4) Distributed Development Model
(5) Short Release Cycles

In this paper, we will discuss our Project 1 rubric and how it relates to these 5 best development practices from Linux Kernel, while giving examples as to how we applied these practices in our own work to make our team as efficient and communicative as we could.

2 LINUX KERNEL DEVELOPMENT BEST PRACTICES

2.1 Zero Internal Boundaries

The Zero Internal Boundaries principle dictates that there should be no boundaries between contributors to a project as to who can access different parts of the project. Anyone working on a project should be able to have access to the tools and code used in different parts of the project. This principle corresponds to the "evidence that the whole team is using the same tools" and "evidence that the members of the team are working across multiple places in the code base" criteria in the project 1 rubric [1]. We implement the Zero Internal Boundaries principle in our project as it can be seen that each of the team members contribute throughout various source files in the project, and we only use cross-platform open-source tools to ensure that team members can all work across the whole project.
To make sure we followed this rule in Project 1, we kept our requirements.txt completely up to date with the libraries we used. We also made sure that all members of our group could run the bot and had the same administrative permissions in our test server so that if one person could run a test or command, all members could also run it. In other words, there were no tools or libraries that only some of us had access to. We also kept a chat channel with meeting notes and resources so that if anyone had questions, they could be answered as quickly as possible, and any tutorials for using the Discord API could be shared. We made sure all code was up to date in the repository for other members to view and approve.

2.2 No Regressions Rule

The No Regressions Rule for Linux Kernel Best Development Practices means that as new updates come to a software, there is no removal of features that exist or the quality of code. The project 1 rubric has many points related to documented all features and testing of the project. With these points, it is easy for someone looking at our repository to determine that as new releases of the software are released, no features are being removed and no code is being changed in a way which negatively affects the quality of our code. The rubric asks if we "store [our] documentation under revision control with [our] source code" and if we "publish [our] release history e.g. release data, version numbers, key features of each release etc. on [our] web site or in [our] documentation" [1]. Both questions involve thorough documentation of previous and

current iterations of the code base, which allows any user of the software to check if there have been any updates which remove functionality or quality of the product. This way it is clear if we have violated the No Regressions Rule and if so, have we addressed this issue and taken immediate action to fix the violation. To make sure we followed this rule, all features, upgrades, and bugs/bug fixes were documented in our issues and on our project board. This way, any changes can be quickly viewed to see if any significant reduction in features or quality of code have been made. Since we kept each major feature was worked on in its own branch, we were able to make sure that when they merged back into the main branch, they did not affect the functionality of the other features in our project.

2.3 Consensus Oriented Model

The consensus-oriented model of development means that for every change made to the codebase, there has to be agreement across all (respected) developers on the team. This makes sure that there are no changes made at the expense of any other feature or group, and the code remains flexible and scalable [1]. The rubric addresses this requirement in one major way: we make sure that each issue is discussed before being closed. By discussing, we can make sure that not only is the issue being closed with completion of the task(s) included, but any portion of the code fix or upgrade being done is no tin conflict with the rest of the code. This includes the merging of pull requests, as we link our pull requests to our issues and a merge closes the issue. We make sure at least 2 reviewers check the code being merged before it is done to make sure there are no conflicts with other features of the Bot. Github checks for syntactical conflicts, so we double check that as well as viewing the logic itself. To help make sure to minimize conflicts for readability when merging, we tried to keep each issue small so that the resulting code is not too difficult or confusing to compare to the rest of the codebase, and if we needed to roll back any changes it wasn't a significant step back.

2.4 Distributed Development Model

The Distributed Development Model breaks the project into parts and assigns those parts to different developers. That way, one person is not responsible for development and review of the entire project. The rubric includes this best practice with points on making sure "workload is spread over the whole team" [1] and recording team member's contribution. This is shown through the number of commits made by each member and reporting discussions of issues before they are closed. Although everyone is assigned parts that they will be more familiar with, the rubric makes sure the members still have a good understanding of other parts of the project and can better complete reviews and integration. The team must provide "evidence that the members of the team are working across multiple places in the code base." The tools and configuration files in the project will show that "the whole team is using the same tools" [1] and can run the system. An important part of the Distributed Development Model is communication because that allows for seamless integration of the project and consistency. For that reason, the team must be able to stay connected and up-to-date through a chat channel.

In order to make sure we followed these points for our project, we kept a Discord server for communication as well as testing. We also kept a weekly in-person meeting. All changes and design choices were decided on together and updates on each feature team members worked on were kept in channels designed for those purposes. Since we all had access to the same testing server, we could also see exactly what the Bot was giving as its output for any given feature, so in addition to tracking bugs and issues on our Github repository, the specifics of each output was very clear amongst our team. Although we did branch for each feature, we made sure each branch was relatively short-lived, so each was merged back into main within 2 or 3 days of it being created. That way we all could make sure that any necessary libraries and tools were constantly being shared amongst our team and working properly, and we could troubleshoot when needed.

2.5 Short Release Cycles

Short Release Cycles are important so that there are clear milestones to aim for and that the software as a whole is periodically being reviewed for what is getting added to it. This allows for constant evaluation of what is important, allows for flexibility in direction, and the ability to possibly incorporate new technologies that may have not existed when a project was originally thought out. Short release cycles are good because it is more accurate as well for predicting how long something is going to take; defining a unit of work and a time-frame becomes easier than trying plan out huge features over a long period.

This is a difficult practice to follow for a project of this scope and time period, as we have only been working on it for one month. Therefore, our short release cycles were done as frequent merged pull requests rather than full releases.

3 CONCLUSION AND FUTURE WORK

Overall, we found that we liked these five practices for development. We felt that they were both easy to implement without any major learning curve, but were effective in keeping the development of our Bot project streamlined and well-documented. We feel that there are a few specific things we did well, particularly regarding Zero Internal Boundaries and Consensus-Oriented Model. We kept communication open and made sure that in all of our meetings, all voices were heard. Our resulting product is a good mixture of all of our ideas.

Some things we could have done better are having shorter release cycles (as in actually putting out more than 1 release in the month we worked on our project), as well as doing better with the Distributed Development Model. While we each have a pretty thorough understanding of all parts of our code and features, maybe it would be better if next time, instead of each working on specific features, we switch features throughout development or each work on parts of the same features. For a project of this scope and time limit, however, we feel that this did not limit our progress in any significant way.

REFERENCES

[1] Tim Menzies. 2021. Project 1 Rubric. Retrieved September 30, 2021 from https://github.com/txt/se21/blob/master/docs/proj1rubric.md CSC 510 Software Engineering.