Feature Hunt : Software Engineering Project1

Mithil Dave
North Carolina State University
Raleigh, USA
mdave2@ncsu.edu

Raj Shah
North Carolina State University
Raleigh, USA
rshah28@ncsu.edu

Nirav Patel
North Carolina State University
Raleigh, USA
nkpatel8@ncsu.edu

Parth Kanakiya
North Carolina State University
Raleigh, USA
pkanaki@ncsu.edu

Bhargav Jethwa
North Carolina State University
Raleigh, USA
bjethwa@ncsu.edu

ABSTRACT

Linux Kernel Best Practices are widely accepted and followed to facilitate smooth development cycles. In this paper, the five Linux Kernel Best Practices for Software Development are explained, as well as the relationships between Linux Kernel Development Best Practices and Software Development.

KEYWORDS

software development, Linux kernel, Best practices

ACM Reference Format:
Mithil Dave, Raj Shah, Nirav Patel, Parth Kanakiya, and Bhargav Jethwa. 2021. Feature Hunt : Software Engineering Project1. In *Proceedings of Project 1, SE21 (Project 1).* Raleigh,NC, 2 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The Linux kernel is a free and open-source, monolithic, modular, multitasking, Unix-like operating system kernel. It is also an prime example of the power of open source project and constructive collaboration. The linux kernel is a very huge project and they have had their own challenges. A part the reason behind the success of linux kernel is their development process. This process has enabled the linux kernel team to deliver multiple releases without issues. This process has a set of well defined rules, which guide the overall development cycle. These set of rules are called linux-kernel development practices. Some of them are mentioned below.

(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 the connection between the items in our project rubric and the Linux Kernel best practices. We will also give the evidence of the use of these practices within our project and assess the outcomes at the end.

2 APPLICATION OVERVIEW

A lot of products are built with pure assumptions which does not always makes up to a great product. We have built a platform called 'Feature Hunt' to collect, analyze, and organize feature feedback/requests to make better product decisions. User will be able to see different products and their feature details. Users can share, vote and discuss feature requests on the platform. 'Feature Hunt' will automatically show all the submissions from your users on a product dashboard to get a good sense of where you are headed. In future, it can include the status tracking for feature development of various products.

3 LINUX KERNEL DEVELOPMENT BEST PRACTICES

3.1 Zero Internal Boundaries

The idea of Zero Internal Boundaries states that there should be no restrictions on who may access specific parts of a project among its contributors. Anyone working on a project should be able to see the tools and code that are utilized in various areas of the project.

We implement the Zero Internal Boundaries approach in our project since each team member contributes to multiple source files across the project, and we exclusively use cross-platform open-source tools so that any team member may contribute to all of the project's work. We kept our requirements.txt up to date with the libraries we used in Project 1 to make sure we followed this criterion. We also ensured that all members of our team were capable of creating and launching the website. That is to say, there were no tools or libraries available to only a few of us.

3.2 No Regressions Rule

The No Regressions Rule for Linux Kernel Best Development Practices states that as new upgrades are released, no existing functionality or code quality will be lost. This entails extensive documentation of prior and current revisions of the code base, allowing any software user to examine if any changes have removed functionality or reduced the product's quality. This way, it's apparent whether we've broken the No Regressions Rule, and if so, whether we've acknowledged the problem and taken prompt steps to correct it.

All features, updates, and bugs/bug fixes were documented in our problems and on our project board to ensure we followed this guideline. We created a RoadMap to keep track of all the issues and their progress. This way, any modifications can be easily evaluated to determine whether there have been any substantial reductions in features or code quality. We were able to ensure that when each significant update was merged back into the main branch, it did not influence the functionality of the other features in our project since we maintained each significant update on its own branch.

3.3 Consensus Oriented Model

Every update to the codebase must be accepted by all (respected) developers on the team, according to the consensus-oriented development approach. This ensures that no changes are made at the expense of other features or groups, and that the code stays scalable and flexible. This criteria is addressed in one key aspect by the rubric: we ensure that each problem is discussed before being closed. We can ensure that not only is the issue closed with the completion of the task(s), but that any component of the code fix or upgrade is not in conflict with the remainder of the code by discussing it. 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 that at least one reviewer checks the code before it is merged to ensure that it does not clash with other features. We double-check for syntactical incompatibilities using Github, as well as looking at the logic itself. We attempted to keep each issue short to ensure that the resultant code is not too difficult or confusing to compare to the rest of the codebase, which helps to ensure that conflicts are minimized for readability when merging.

3.4 Distributed Development Model

The Distributed Development Model divides a project into components and assigns each component to a separate developer. As a result, no individual developer is responsible for the entire project's development and assessment. This best practice is included in the rubric, along with points for ensuring that workload is distributed evenly across the entire team. Despite the fact that everyone is given areas with which they are more familiar, the rubric ensures that all members have a thorough understanding of the rest of the project and can complete reviews and integration more efficiently. Communication is a crucial aspect of the Distributed Development Model since it provides for seamless achievement of project objectives and consistency. As a result, the team must be able to communicate and stay connected via a chat channel.

We kept a Google chat room for communication to ensure that we followed these principles for our project. We also had a weekly face-to-face meeting. Given everyone's schedules, the overall work was evenly distributed among all team members. Two of us worked on the front-end and the remaining three on the back-end and documentation once the project idea was developed and agreed upon. Across all divisions, however, everyone was informed of the status and development.

3.5 Short Release Cycles

Short Release Cycles are crucial so that there are clear goals to strive for and the program as a whole is examined on a regular basis to see that what's being added. This provides for continuous evaluation of what is important, as well as flexibility in direction and the capacity to include new technology that might not be available when a project was first conceived. Short release cycles are beneficial because they make it easier to forecast how long something will take by specifying a unit of work and a time frame rather than trying to plan out large features over a long period of time.

We have only been working on this project for one month, therefore this is a difficult approach to follow for a project of this complexity and duration. As a result, rather than full releases, our short release cycles were done as frequent merged pull requests.

4 CONCLUSION

Following the linux-kernel standard development practices for our project made the execution of our project easier and quite efficient. The work was very well divided between the team members, which enabled us to do the tasks with increased focus. Also, the communication had played a major role in clarifying doubts among the team members. Creating a road-map helped us to visualize our common goal and trouble shoot the flaws in the design. We made all our decisions by incorporating every team member's consent and opinions. Overall, our project benefited a lot from the standard development practices and we were able to build a truly useful product at the end.

5 FUTURE SCOPE

In extension to what we have built, there are several other use cases that can be added to our platform. Some of them are pointed out below:

- There can be a separate dashboard to manage product feature feedback. Owners will be allowed to prioritize any feature for development.
- Owners can download the data about the product and feature feedbacks in a CSV format. It can be used for analysis of the user responses.
- It can have a separate page to show "Trending" products. Several filters i.e "Newest", "Most Popular", "Trending" etc can be applied. Product owners can have a separate page to track the progress of development of any feature. They can decide timelines, select developers or assign reviewers.
- Users will be allowed to see if their feature suggestions are "Under consideration" or have been "Rejected".
- The product feature review dashboard can be modified to allow only selected group of users to post a review.
- The website can allow anonymous posting and up voting to gather honest reviews.
- Receiving feedback from a specific customers groups i.e students, commuters, doctors etc should be available. One can send out personalized invite to these groups automatically, asking for a product review.