Group 30 Project 1 Rubric Comments

Qiuyu Chen
qnchen@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Yasitha Rajapaksha
yrajapa@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Jiacheng Yang
jyang31@ncsu.edu
North Carolina State University
Raleigh, NC, USA

J. Hugh Wright
jhwrigh2@ncsu.edu
North Carolina State University
Raleigh, NC, USA

1 INTRODUCTION

This essay compares the project 1 rubric with the Linux kernel best practices. It also includes notes on how we followed those practices in our own project, Item Stock Tracker.

The sections below each represent one of the rules of development listed in the 2017 Linux Kernel Report. We provide a short description of the rule or lesson, how it is reflected in the project 1 rubric, what is missing in the rubric, and how we have followed that tenant while working on our project. Through this reflection, we learned some practices suitable for group projects and how to develop a comprehensive rubric.

2 SHORT RELEASE CYCLES

Short release cycles are important. A short release cycle means that changes reach users quickly, so that bugs or security concerns can be resolved in a reasonable time frame after they are discovered.

In the rubric, this is best represented by "short release cycles" section. The reasons for this are self evident.

Although we only have one complete "release" of our project, we fulfill the spirit of this request by rapidly and consistently pushing our changes to GitHub, so that they can be reviewed by other team members and integrated with the main branch. People do not keep their work to themselves, and merge often, which reduces conflicts when merging. At the same time, the rubric should also include relevant assessments. We want to ensure that team members can commit in time so that other team members can start integration and find problems early.

Following this rule also makes it easier to create a code base that does not have internal boundaries, and where decisions are made by consensus. Since everyone can access each others code, and know that that code is recent and represents the latest state of the project, we can understand what everyone is doing to inform our future design

3 A DISTRIBUTED, HIERARCHICAL DEVELOPMENT MODE

It is not efficient and practical to let one person in charge of all code review and integration in a large team project. Therefore, distributed hierarchical development mode spreads out those tasks to individual developers to ensure the efficiency and smoothness of project integration.

The rubric mentions spreading the workload over the team. However, it doesn't specify the review and the integration process. I think it would be better to include some detailed points about this in the rubric. For example, how did the team complete the reviews, and did everyone participate?

In the process of completing the project, our team did not formally follow the rules. With only five people involved in the project, a strict hierarchy is generally not necessary. Each team member took turns reviewing others pull requests and code. Regarding integration, each team member was responsible for integrating the part they wrote, while communicating with the rest of the team. Before the start of the project, we established some basic group rules but did not mention how to implement reviews and integration. After finalizing the project idea, we simply assigned each person to do different parts of the program. Based on the part that everyone is responsible for, we should formulate more detailed rules on how to integrate.

4 TOOLS

Tools are vital elements in the development process. Both the Linux Kernel Best Practices and our rubric mention how important the right tools are to a project.

In the Linux Kernel Best Practices, it describes the right tool as the support which prevents the collapse of a kernel.

In the rubric, there are many points to check whether we use a tool(such as the version control tool), or whether the whole team is using the same tool. Specifically the points "Use of version control tools," "Use of style checkers," "Use of code formatters," etc.

This rule was actually very helpful during our development. For example, we are all using git as the version control tool, and it makes us can easily integrate our codes and cooperate. Without this tool, we would make more mistakes in integrating and the whole development process would become slower. We also all use the same IDE, PyCharm, which includes built in style and syntax checkers. After each commit, Travis CI will automatically run our program to ensure that it builds successfully and passes the test. It helps us to review our updates, especially before the pull request to the main branch. If anything is going wrong, we don't want the main branch to be polluted.In addition to that, we also ran style checks using the PyCodeStyle tool, to ensure our style and syntax matched with PEP8 standards. We analyzed our code coverage using the free version of Coveralls.

5 CONSENSUS-ORIENTED MODEL

A consensus-oriented model ensures that code is not put into production until it is approved by a general consensus. This guarantees that features that might inconvenience other developers or users are not put through. No individual should have the power to radically change a code base.

The rubric focuses quite a bit on ensuring that communication happens, and that decisions are discussed by the group as a whole. The point "issues are discussed before they are closed" is the most obvious example of this, and the rubric also makes sure that a chat channel exists for the project.

In our group, we communicated frequently over Discord about issues, and no issue was closed without the awareness of the team. Communication either happened through discord or over Zoom. We had regular Zoom meetings to make design decisions and ensure that everyone was on the same page.

Summaries of what was decided for how to fix an issue, and what features where implemented to close that issue, can be found in the form of GitHub comments on an issue, although very little discussion actually took place through GitHub directly.

6 "NO REGRESSIONS" RULE

The "No Regressions" Rule ensures the updates of a program will not break the original functionalities.

In the Linux Kernel Best Practices, this rule allows people to follow the kernel as it updates, since they don't have to worry about their OS no longer being supported, or a feature that they rely on disappearing.

This rule is not directly represented in the main rubric, mainly because of the time scope of the project. However, adequate testing does guarantee that features will not be removed accidentally, and there are several points in the rubric that deal with testing. Most notably "test cases exist" and "test cases are routinely executed."

In this project, since we expect other developers to add more features to the program in the future, there are also various checkpoints from our rubric to make sure we followed this rule. For example, we need to maintain an "always" stable branch by multiple passing tests, and we also need to have many issues for the failing cases. At the same time, the use of different tools also helps us. Github as the version control tool lets us have different branches for development and keep the main branch stable. Travis CI makes sure our program can build successfully after each commit. All of these eventually help us to achieve there were no regressions during development.

7 CORPORATE PARTICIPATION

Corporate participation is important in a commercial project. Companies need to balance their contributions to the project to make sure they will not hurt others companies or dominates the development.

"Corporate" participation does not apply directly to our rubric and project. But there is a very similar point of view. The rubric points out the rules of contribution- if others want to work on this project, they should extend the system without screwing things up. This is a very crucial rule because we hope other will pick up our project to work on.

We included those rules in our CONTRIBUTING.md file. We state that if others want to add a new feature to the project, they need to follow those rules. Such as regulations for making new issues for others to review and creating a new branch.

The rubric also includes several points that make projects appealing to outsiders hoping to join them. The points begging with "Docs:" all fall into this category. Although we do not anticipate any actual corporations joining our project, if our project where larger in scope, those resources would encourage other organizations to use and contribute to our project. For our documentation, we have a detailed README.md file, documentation generated through Sphinx based on our code comments, and a short animated video selling our idea.

8 NO INTERNAL BOUNDARIES

With no internal boundaries, all developers have access to the entire code base. This is necessary because bugs that are discovered while working in one section of code may need to be fixed in another code section entirely. Furthermore, when one developer is not available, others can pick up the slack- even when they do not necessarily specialize in that area.

The rubric covers these points by requiring "evidence that the members of the team are working across multiple places in the code base," and "evidence that the whole team is using the same tools." Everyone working with the same tools and having access to the same files allows us to collaborate more easily, and people can access other's code to resolve issues or make improvements.

In our project, everyone was working from the same GitHub repository, so everyone had access to all relevant files. We also communicated often so that everyone understood each others code, and could easily ask questions if something was unclear.

This greatly sped up development of our project, particularly when integrating back-end functionalities with the GUI.

One of our group members also dropped the course early on in our development. Since everyone had access to each others code, we were able to adapt to having one less team member with relatively little hassle. That could have been a disaster if that group member had code they had not shared, or knowledge that wasn't shared with the rest of our group.

9 CONCLUSION

Overall, the rubric covers all the fundamental points we need to pay attention to. It is worth noting that the rubric is intended to adapt the Linux kernel best practices to projects that are much smaller in scope, therefore some changes will necessarily have to be made to make the rubric practical.

For example, the rubric lacks some of the detailed expectations that are in the Linux Kernel Best Practices. The rubric mentions the spread over of tasks but does not include the expected actions of review and integration specifically. Therefore, the rubric can add more detailed content based on the Linux Kernel Best Practices.Through this project, we have learned that it seems troublesome to formulate group rules. But in the subsequent practice, it has been shown to be very necessary. By following the rules, we can make the project process more efficient, reduce conflicts and communication problems.