CmyPlot Development: How Group 1 Follows the 5 Linux Kernel Best Practices

Brad Erickson (bericks*)
Aakriti Aakriti (aaakrit*)
Ashwin Kumar Muniswamy (akumarm*)
Jainam Ketan Shah (jkshah*)
Sharath Kumar Vijaya Kumar (svkumar2*)
*@ncsu.edu
North Carolina State University
USA

ABSTRACT

The following is a report based on how the Linux best practices are followed for Group 1 of the first project. Our project is titled CmyPlot, an online graphing tool that allows users to upload data and look at it in both tabular and graphical formats.

KEYWORDS

best practices, software engineering

ACM Reference Format:
Brad Erickson (bericks*), Aakriti Aakriti (aaakrit*), Ashwin Kumar Muniswamy (akumarm*), Jainam Ketan Shah (jkshah*), and Sharath Kumar Vijaya Kumar (svkumar2*). 2021. CmyPlot Development: How Group 1 Follows the 5 Linux Kernel Best Practices. In *Software Engineering F'21, August – December, 2021, Raleigh, NC*. ACM, New York, NY, USA, 2 pages. https://doi.org/None

1 SHORT RELEASE CYCLES

The first Linux Best Practice is short release cycles. This allows for new features and bug fixes to be implemented in a timely manner instead of waiting for a larger release. This process is great for consumers as the product they are using is constantly being updated. Additionally, this eases the developers to have the flexibility of testing their premature code and being able to fix any changes or bugs in the code with the next version release. This is a great way to test any new features which can be extended if they get good reviews from the user base.

Our group follows this model and CmyPlot currently has two alpha releases. The first release, *v0.1-alpha*, focuses on the general structure of the codebase, without much functionality. This release was mainly for getting the Zenodo citation set-up. The second release, *v0.2-alpha* is our final release for Project 1. This encompasses the overall goal of the project: providing an online data visualization tool to our users. For the rubric items, see survey items 4.8, 10.2, and 10.3.

Evidence of this can be found in our Github release tags.

2 DISTRIBUTED DEVELOPMENT MODEL

In the distributed development model, no one develop is responsible for writing code and approving what gets added into the codebase. Instead, these tasks are split among all team members. A benefit of this model is that all team members have a working idea of what's in the codebase and new features do not get hung up on a single person managing everything.

Our team handles this a few ways: 1) the issues are open for any member to work on and 2) when someone opens a pull request, it cannot be merged until it has at least 1 reviewer that is not the author of the pull request. For some issues, it might be best to choose a specific group member as a reviewer. When working on implementation of a item with a framework someone hasn't used before, they should include someone who has experience working with that framework. For instance, when someone is implementing something related to the Dash framework, they should include Brad since he has the most experience with it.

Evidence for this can be found in the project issues and pull requests. Also in the repository, under insights, we can see how much each person has added to the codebase over time.

3 CONSENSUS ORIENTED MODEL

The Consensus Oriented Model tells us that no group of users should block another group of users from contributing. This also means that their is a general shared mindset between developers. All proposed changes should be okayed with the core developers before being fully implemented into the system.

To achieve this best practice, our team started with determining our goal: providing an online data visualization tool to our users. From here, we determined the tickets required to reach this goal and began working. In our project Discord server, we held discussions to hash out which best fit the goal and which did not. Additionally, we used our server to vote general logistics of the project, such as name and theme.

For the development side, requiring reviewers on the pull requests helped us achieve this model. If the reviewer is not satisfied with the change, we would be asked to make additional changes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Software Engineering F'21, August – December, 2021, Raleigh, NC
© 2021 Association for Computing Machinery.
ACM ISBN 0…$0.00
https://doi.org/None

The primary evidence of this is focused on the project issues, pull requests, and discussion in Discord.

4 THE NO-REGRESSION RULE

The No-Regression Rule is all about moving forward while still maintaining backwards compatibility. If the system upgrades to a newer version, the users should still be able to do all of the things they did on the prior version. In smaller systems, developers should not implement pieces of features that the users might start using before the main feature is implemented.

We follow this rule by not implementing any pieces of features that are not yet finished. For instance, we wanted to add dynamic filtering to CmyPlot; however, this functionality was not added because of time restraints and thus, dynamic filtering got moved to the road map.

Had we attempted to implemented portions of this in the code base, the users may have began using those features and expect them to be there in the future. If this feature then got scrapped and we took out those sub-features, the users may have been upset that they are no longer able to use them. Since we did not do, we have avoided violated the no-regression rule.

The main evidence of this rule comes from the design of the system, the code itself, and pull requests (nothing was removed).

5 ZERO INTERNAL BOUNDARIES

The Zero Internal Boundaries best practice enables anyone to be able to make changes to any part of the project. This allows for problems and features to be implemented by any of the developers (similar to that of the distributed model). To achieve this, all the developers must have access to the same development tools.

Our team implemented this in a few ways: 1) all developers on the team had access to the full code base and 2) all developers used a virtual environment to manage packages. The third-party tools such as CodeCov and Zendodo were not accessible to all team members as the owner of the repository, Brad, had to configure access for third-party applications. With the virtual environment, we are able to specify specific versions of the packages we used in our *requirements.txt*.

Evidence for this can be found in the instructions for project set-up in our repository's README and specific packages can be found in the *requirements.txt*.