Project 1 Analysis: Movie-Recommender

Adam Yen*
Dani Hove*
Kan No Lee*
fyen@ncsu.edu
dphove@ncsu.edu
klee32@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Figure 1: Seattle Mariners at Spring Training, 2010.

ABSTRACT

In the following paper, we describe the links between the rubric for Project 1 and and the Linux Kernel Development Best Practices. In particular, we emphasize on the ways the that the rubric items reinforce the principals laid out by the Best Practices, and in subsequently contribute to creating a productive development environment.

KEYWORDS

best practices, rubric, development, software engineering

ACM Reference Format:
Adam Yen, Dani Hove, and Kan No Lee. 2021. Project 1 Analysis: Movie-Recommender. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 SHORT RELEASE CYCLES

Within the context of the best practices document, short release cycles refer to the practice of emphasizing quick releases, in part for two reasons: keeping an active codebase, and updating (and more importantly, testing) frequently enough that if any problems crop up in the process of implementing a larger feature, issues are caught fast, and while changes are still small enough that the difficulty of debugging is curbed by smaller scope. Additionally, short release cycles, especially in consumer environments, can minimize consumer frustration, as it allows designers to more dynamically react to shifting consumer needs.

A number of rubric items apply to this particular category. The most apparent applicants are the group of criteria aimed at tracking the quantity of interaction with the source code. Particularly, things like the number of commits, the number of issues being reported and the use of version control tools. A little less obviously, many of the rubric items aimed at discussing testing procedures fall in line with the principles that inspired short release cycles. A key benefit of regular releases is that it prioritizes an efficient testing suite, and effective unit testing eventually leading into integration testing. Our team pushed to implement on this with combination of unit testing and brief cycles between pushing our updates. Operating in our mostly online context, we aimed to implement changes quickly around one another so as to best catch errors and problems as early as we could, and make the appropriate changes before anything else was built on top of it. This was especially key for the foundation building integral to Project 1.

2 ZERO INTERNAL BOUNDARIES

A lack of internal boundaries is absolutely esssential for a dynamic working environment. While specialization and delegation are important for good production management, forcing engineers to stay within those confines can be detrimental, especially when a problem in one section is being caused by another. In order for short

*All authors contributed equally to this research.

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

release cycles to be feasible, developers have to have equal access to the environment if need be. This minimizes on delays brought by authentication/second-hand editing, and as a side effect, encourages developers to work with an awareness of the teams around them, even if they don't interact directly with them at all times. The rubric largely covers this ground in its questions about evidence that all users have access to the same tools and environments, and most importantly, understand those tools and environments. Another key part of a system like this is presented by rubric items like the workload spread, and evidence that users are working across different parts of the code. Again, a key part of good production is the proper delegation of tasks, breaking down internal boundaries is meant to aid that, as opposed to completely eliminating all project roles. Our work was made far better without boundaries: the quirks of Javascript can often require changes in how other code in a module calls the same function, and our ability to equally access and contribute to the GitHub repository allowed us to dynamically make those changes in such a way that enhanced the rest of our production cycle.

3 NO REGRESSIONS

In the scope of Kernel development, it's absolutely essential that one never regresses they're version/code. Primarily from the angle of a consumer, no-regression rules are on of few assurances that updates won't mangle they're existing work. Additionally, operating on a principle that emphasises this backwards compatibility provides additional incentive for effective testing, and ensuring that releases meet a base functioning standard at all times, at risk of losing the trust of your consumers.

In the context of our rubric, many of the Software Sustainability Evaluation questions directly address a lot of the practices necessary for maintaining a code base with a no-regression rule. This is furthered by rubric points around basic tools like version control, stability features, and contributing etiquette for open source contributors that may not be familiar with what core features have to stay static to ensure compatibility. Our project in particular aimed to build modular enough work that updates could be slotted in without disturbing core functions, which are regularly tested. As a rule, base functionality is enforced by our testing suite.

4 CONSENSUS-ORIENTED MODEL

In Kernel environments, consensus is key to ensuring a smooth development process. Especially in an open source environment, development often depends on the continued satisfaction and interest of other developers. Consensus oriented models aim to allow contributors have equal say in the direction of the development process and prevent internal boundaries from creeping in. The rapid development, in conjunction with it's low margin for release error, demands that developers are in lock-step on their goals, even if they're working from different departments or even different companies.

The rubric ensures these practices are carried out by ensuring that issues are being extensively used to document next tasks, and that in particular, group members are regularly communicating. Communication is key to not only reaching, but maintaining consensus. Our group used a combination of issues (for tracking larger goals) and our main chatting platform, Discord, to coordinate our goals. Meetings were held in between larger milestones to ensure that everyone's voice was being heard, and that we could reach appropriate compromise when our visions for the project began to stray from one another. Regular communication was key to the eventual completion of our project.

5 DISTRIBUTED DEVELOPMENT MODEL

In Kernel development, especially the ideal open source environment where changes are being submitted frequently, it's overwhelming to expect a single developer to keep track of them. Distributed development models diffuse the load across trusted readers and testers who can ensure that the speed of development isn't bottlenecked by the speed or quality of verification. Again, with the short margins for error, and emphasis on efficient, speedy development cycles, a distributed model is key, especially once development grows larger.

The rubric encourages the seeds of this modeling, in part with the ways it encourages everyone understands the code base, and can perform the task of verifying a change. There are less opportunities for a distributed model to really shine in the context of our Project, given the fact that our team was composed of three people, who were already taking on more hats than usual for a full development cycle. However, the familiarity and distribution of labor encouraged by the rubric helped us both in refining the speed of our production process and the quality of it. Many rubric points of scalability and maintainability, particularly in how we laid out contributing etiquette, are essential to keeping contributors informed of how our model can be best interacted with.