Enigma rubrics mapping to Linux Kernel Best Practices

Shubham Dua[1], Rahul Gautam[2], Shreya Buddaiahgari[3], Srilekha Gudipati[4], Chetana Chunduru[5], and Sarika Vishwanatham[6]

sdua2@ncsu.edu[1],rgautam3@*ncsu.edu*[2],sbuddai@*ncsu.edu*[3],sngudipa@*ncsu.edu*[4],cchetan2@*ncsu.edu*[5] and svishwa2@*ncsu.edu*[6]

[1,2,3,4,5,6]*Computer Science Department*

[1,2,3,4,5,6]*North Carolina State University, North Carolina, USA*

Abstract—Enigma has been developed as a one-stop destination for acquiring song recommendations and playing music depending on the user's inputs. It is an interactive Discord bot that plays songs on Discord voice channels and provides music recommendations. It makes it easier for different users to listen to the same song simultaneously and to play it on several speakers to create a surround sound experience. The foundation of Enigma is a data set that includes the top 50 Spotify tracks from 2010 to 2019. Users can also pause/resume the music and listen to their own custom tracks without having to go searching for them again.

ACM Reference Format:

Shubham Dua, Rahul Gautam, Shreya Buddaiahgari, Srilekha Gudipati, Chetana Chunduru, and Sarika Vishwanatham. 2022. Enigma rubrics mapping to Linux Kernel Best Practices. In Proceedings of ACM Conference (Conference'17). ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

INTRODUCTION

The Linux Kernel is a sizable open source project that has been in use for many years. Its development methods, which have made it a great example of collaborative development, contribute to its effectiveness as a tool. Due to these procedures, Linux has been able to last for as long as it has without experiencing any quality compromises, which are common in projects of its size and duration. What gives Linux its ability to endure for so long, with such frequent releases devoid of problems that prompt consumers to stop using it? For those who work on it, the Linux Kernel upholds a few straightforward but essential norms, which has allowed it to be in use for so long. These 5 techniques are:

(1) Short Release Cycles (2) Distributed Development Model (3) Zero Internal Boundaries(4) Tools Matter (5) Consensus Oriented Model (6) Low Regression Rule.

In this paper, we will discuss our Project 1 rubric and how it relates to these 6 best development practices from Linux Kernel, while giving examples as to how we applied these practices in our own development to make our team as efficient and communicative as we could.

1 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 to 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.

2 DISTRIBUTED DEVELOPMENT MODEL

Distributed development model ensures that the process of code review becomes smooth and without any compromise to kernel stability. In a centralized sys- tem, where one person reviews and approves any code changes, this person becomes the bottleneck for the faster deployment and implementation of the given code base. In a distributed system, each task is handled by an individual who is an expert in that particular field. This also ensures that bugs do not get past through the review process and thus the kernel stability is ensured. The elements that are a part of the distributed development model are: *a) Decisions made by unanimous vote:* Instead of a single person holding authority over a code base and making all the decisions, all the parties related to the project are involved and decisions are taken based on a general consensus. *b) Group meetings have a round robin speaking order:* This ensures that everyone's opinions are accounted for rather than one person making all the decisions. *c) Group meetings had a moderator that managed round robin* A group moderator ensures that everyone gets the chance to speak and the process is organized in an unbiased manner. *d) group meeting moderator ro- tated among the group. e)code conforms to some packaging standard* By keeping a standard format of writing code, integration and understanding of the code improves. This helps in making the overall process

easier. *e) code can be downloaded from some standard package manager f) workload is spread over the whole team (one team member is often X times more productive than the others* In distributed development model, everyone gets an equal opportunity to contribute to the code base. This also ensures that updates take place across the entire system rather than a single section. All the points mentioned further are different ways and systems that could be used to keep a track of all the work that is taking place on the distributed system. This makes sure that the process of code development in a distributed system remains seamless, fast and accounted for. *g) Number of commits h) Number of commits: by different people i)Issues reports: there are many j) issues are being closed k) License: exists l) DOI badge: exists m) Docs: doco generated , format not ugly n) Docs: what: point descriptions of each class/function (in isolation) o) Docs: how: for common use cases X,Y,Z mini- tutorials showing worked examples on how to do X,Y,Z p) Docs: why: docs tell a story, motivate the whole thing, deliver a punchline that makes you want to rush out and use the thing q) Docs: 3 minute video, posted to YouTube. That convinces people why they want to work on your code. r) code conforms to some known patterns*

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. We have used the MIT license and have a number of different badges like DOI badge, code coverage badge, python version badge, python application: passing badge. We have generated the different documents in a proper format, and with all the relevant information added so that anyone who uses our project can use it directly without any hassle. We have also added descriptions of all the commands in different files so that the users can have an idea on how to use these commands. These files also contain the video description of how one can use these commands.

3 ZERO INTERNAL BOUNDARIES:

A developer in most cases makes changes to only certain parts of the code base. But this does not prevent him from making changes to other parts of the code base. It's valid as long as the changes are justifiable. This practice ensures that the problems are fixed right when they originate. It is not necessary to provide multiple workarounds. It also indirectly helps the developer to gain a global perspective on the code base rather than sticking to a specific repository. The rubric has three tuples which could be mapped to this category: *evidence that the whole team is using the same tools: everyone can get to all tools and files* This indirectly translates the fact that every developer within the group knows each and every software or tool that is being used in the system. This ensures that everyone is aware of how the project actually works rather than just focusing on one aspect of the project. *evidence that the whole team is using the same tools (e.g. config files in the repo, updated by lots of different people)* This ensures consistency which makes the process of knowledge transfer, sharing and discussion easier. *evidence that the whole team is using the same tools (e.g. tutor can ask anyone to share screen, they demonstrate the system running on their computer) evidence that the members of the team are working across multiple places in the code base* This point provides a proof of the fact that the developers have indeed worked on different aspects of the code.
We have done most of our coding part in Python using and IDE like Pycharm, so all of us are using the same tools. We all are pushing to the main branch in or repository which gives us the idea that all the contributors are important and we trust them completely to directly push into the main branch. All of us can run the same code in our system and can perform all the tasks from their end. The main config files in our repository are being continuously updated by all the members along with the cog files.

4 TOOLS MATTER:

Every type of project requires a certain set of tools and resources which need to be used in unison for the system to work efficiently. These tools help in keeping a track of all the work that is taking place in the system. It also helps in a smooth on-boarding and knowledge transfer of a new person. A user doesn't have to manually enter every file and check for syntax and version control. Following are the tuples which map with this ideology.
a) Use of version control tools,

b) Repo has an up-to-date requirements.txt file

c) Use of style checkers

d) Use of code for-matters

e) Use of syntax checkers

f) Code coverage

g) test cases exist h) test cases routinely executed.
We all use Github as our version control tool and have an up to date requirements.txt file in our repository. Pycharm handles the part of style checkers, code for-matters and syntax checkers. For code coverage we have used codecov and we are getting 69.76% code coverage. Numerous test cases exists and they all get executed every time any push is made to the main branch

5 CONSENSUS ORIENTED MODEL:

The Linux community strictly adheres to the rule that a code cannot be merged into the main unless an experienced and respected developer approves it. This rule makes sure that the chances of the kernel going haywire is significantly reduced. One group cannot make unwarranted changes to the code base at the expense of other groups. While such restrictions are imposed the kernel code base remains flexible and scalable. Following tuples seems to match with the understanding of the practice. *a) the files CONTRIBUTING.md and CODE OF- CONDUCT.md have multiple edits by multiple people* This makes sure that everyone has contributed to the project and that all the decisions have been made by considering every team member's opinion. *b) the files CONTRIBUTING.md lists coding standards and lots of tips on how to extend the system without screwing things up c) multiple people contribute to discussions* All the decisions are taken unanimously by the entire team and stake- holders. *d) issues are discussed before they are closed.* A peer review process of the implemented feature makes sure that minimal amount of bugs slip through thus maintaining the stability of the product. Channel chat exists and is active and finally test cases: a large proportion of issues related to file handling cases.

6 LOW REGRESSION RULE:

Developers are always striving to upgrade the code base. However, this is not done at the cost of quality. This is one of the reasons to introduce the no-regressions rule. According to this rule, if a given kernel works in a specific setting, all the subsequent kernels must also be able to work in this setting too. However, in case the system becomes unstable because of the regression, the developers waste no time in addressing the issue and getting the system back in its original state. The tuple which could be mapped to this metric is: *features released are not subsequently removed*

REFERENCES

[1] *https : //medium.com/@Packt$_P$ ub/linux − kernel − development − best − practices −* 11c1474704*d*