Project1- Group 39 Linux Kernel Best Practices

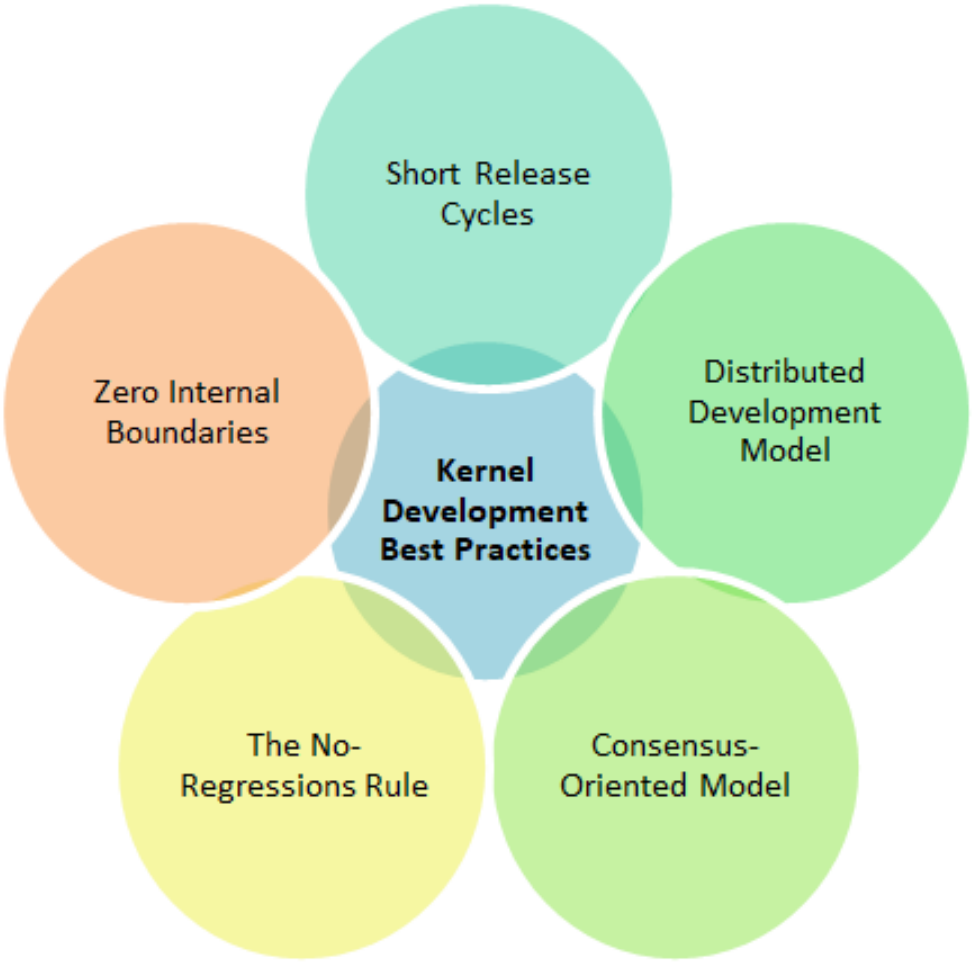
Gokul Krishna Koganti*
gkogant@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Rachana Kondabala
rkondab@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Smayana Pidugu
spidugu@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Harikrishna Selvaraj
hselvar2@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Kiran Bharadwaj
kganesh4@ncsu.edu
North Carolina State University
Raleigh, NC, USA



Unpublished working draft. Not for distribution.

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

ABSTRACT

The document explains the Linux kernel best practices and provides evidence that the practices are followed in the project. This document also gives an insight on how we have adapted the best practices to our project.

CCS CONCEPTS

• **Software and its engineering → Software development process management**.

KEYWORDS

Software engineering, Linux Kernel Best Practices

ACM Reference Format:

Gokul Krishna Koganti, Rachana Kondabala, Smayana Pidugu, Harikrishna Selvaraj, and Kiran Bharadwaj. 2021. Project1- Group 39 Linux Kernel Best Practices. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The five Linux best practices are 1. Short Release Cycles 2. Distributed Development Model 3. Consensus Orientation Model 4. The no regression rule 5. Zero internal Boundaries

2 LINUX KERNEL BEST PRACTICES

2.1 Short Release Cycles

Short release cycles are essential for time-constrained projects because team members should further review what can be improved in the project. Short release cycles are suitable for rapid development environments. Also, long release cycles are not recommended because after each long release, it becomes relatively difficult for the other team members to comprehend the code during code review sessions. Having long release cycles in short time constrained project would decrease the scope for considerable improvements, because the time after the release cycles wouldn't be enough to implement it.

In the case of our project, since the duration is one month, there has been only a single release. But code improvements have been done frequent enough after reviews which have been held weekly.

2.2 Distributed Development Model

This model is required for introducing parallelism into our project. In this model, major works are divided to tasks so that the work can be carried out autonomously without any dependencies.

We have divided our project work into several small tasks based on front end, back end and database categories. We have assigned each category to different team members, so that the work can be carried out in parallel. The input values were hard coded initially so that front end development can be developed independent of the back end module and vice versa. At the end of the development of each module we have integrated these components. Thus, allowing for distributed development.

2.3 Consensus Orientation Model

Even though a level of autonomy is given to each team member on the model that they are working on, each and everyone in the team must have a say, even in minute of details in the project. Each team member can be viewed as initial user of the project and therefore, their ideas and criticism must be taken into consideration to develop an efficient project. We must be under the assumption that a part a teammate doesn't like will not be liked by a sector of user groups, that the teammate falls into. And hence, any such changes which are not agreed upon by everyone must be discarded.

We have discussed the issues or changes to the code prior to committing them to the master branch. We have employed the "Six Thinking Hats"[1] [1] technique by Dr. Edward D Bono to discuss on the issues/changes. At the end of each meeting, which have decided on which issues/changes must be implemented/discarded.

2.4 The no Regressions Rule

No regression rule of Linux kernel states that an update to the system shouldn't break the existing functionality.

For implementing no regression rule, we downloaded the code onto the local system and checked the functionalities and behavior of all the preexisting modules. Then we made changes on the local system and before pushing the updated code on to the master branch, we verified if the previously checked modules are working just as they were prior to the changes.

2.5 Zero Internal Boundaries

Zero Internal Boundaries is vital for successful implementation of small team projects. Assigning rigid roles and strict boundaries in small teams will hinder the progress of the project. This could often lead to scenarios where one might need help of fellow team members, but the other member would be oblivious to the part in which help is required. Therefore, by removing the boundaries within the team members, the team could output an optimal performance, where everyone in the team whilst working on their part, can also assist their peers when needed.

In our project, everyone has access to the tools required to built the entire project. Each team member had similar configurations and same permissions. At any point in the development cycle, one always had access to the other people's work. This way, we always had a clear picture of what everyone is doing and how they are doing it. We had an open communication channel, weekly meetups, code review sessions to discuss any issues/queries encountered during the development.

REFERENCES

[1] DE BONO, E. (1985). Six thinking hats. Toronto, Ont, Key Porter Books.