Tai Phat Nguyen

Optimizing Developer Experience Through Clean Code

Metropolia University of Applied Sciences

Bachelor of Engineering

Mobile Solutions

Bachelor's Thesis

15 December 2024

Abstract

Author:	Tai Phat Nguyen
Title:	Optimizing Developer Experience Through Clean Code
Number of Pages:	47 pages + 2 appendices
Date:	15 December 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Specialisation option:	Mobile Solutions
Supervisor:	Amir Dirin, Senior Lecturer

The objective of this thesis was to explore the role of clean code principles in optimizing developer experience within modern software development. This study addresses the starting point that developer productivity and satisfaction are heavily influenced by code quality and communication within teams, emphasizing clean code as a core factor in efficiency and software quality.

The thesis is based on an extensive literature review of clean code principles and empirical research conducted by a case company's software development team. The research involved creating a commit message template adhering to standardized commit conventions. A comparison was also conducted between existing tools assisting developers in following clean code principles and a custom-made commit message template.

The findings demonstrate that adopting clean code principles significantly enhances developer experience by improving readability, maintainability and scalability of code. Structured commit messages streamlined team communication, reduced ambiguity and improved developers' working days. The outcome should help the case company foster collaboration, reduce developer frustration and minimize technical issues.

Keywords: Developer Experience, Clean Code principles

Table of Contents

List of Abbreviations

DX:	Developer Experience
A.I:	Artificial Intelligence
IDE:	Integrated Development Environment
VCS:	Version Control System
WIP:	Work in progress
CI/CD:	Continuous Integration/ Continuous Development
UI:	User Interface

1 Introduction

In the domain of software engineering, developer experience (DX) represents a crucial point in the pursuit of higher productivity, efficiency, and software quality (Cortex, 2024). Within this objective, the clean code concept emerges as a fundamental principle, serving as a guideline for the creation and maintenance of software solutions. Clean code, characterized by its simplicity, clarity and maintainability stands as one of the backbones of disciplined craftsmanship in software development (Martin and Feathers, 2009).

This thesis aims to exemplify the pivotal role of clean code in the optimization of developer experience within the context of contemporary software engineering practices. By examining empirical evidence, theoretical frameworks, and the industry best practices, this study aims to highlight the impact of clean code on software solution processes, especially on developer productivity, collaboration dynamics and system maintainability.

Through a comprehensive review of literature, the thesis will explore the theoretical definition of clean code, diving into its principles. It will analyze the practical implications of adhering to clean code practices, emphasizing the importance of clean code in software development. Moreover, the thesis aims to explain developer experience (DX) and how crucial DX is within the software domain.

By synthesizing theoretical insights with empirical findings, this thesis aims to provide guidelines on clean code principles by creating a commit message template for the case study company; moreover, the author aims to highlight how the custom commit message template positively affects the case study company team in optimizing developer experience. Through this, it is anticipated that relevant parties will gain valuable insights and will benefit from prioritizing clean code as a cornerstone of software engineering excellence.

2 Research questions and methodologies

In this section, the author focuses on determining the research questions and methods. Given the context of contemporary software development practices, the study seeks to determine the essence of developer experience and the significance of adhering to clean code principles. Through specific research queries, the study aims to introduce the definitions, importance, and usage of developer experience, clean code, and its principles. Moreover, by employing appropriate methodologies, the author can emphasize and provide more insights that contribute to a deeper comprehension of these aspects.

2.1 Research questions

Before deeper exploration, it is essential that a basic understanding of the study topic is acquired. Below is a set of questions that set a ground knowledge that can be helpful when moving further into the investigation of developer experience and clean code.

What is developer experience?

What are clean code principles?

What are the criteria for clean code?

How important is developer experience in software development?

How clean code is related to developer experience?

2.2 Methodologies

This study adopts a mixed-method approach to investigate the importance of clean code principles in software development. In addition, this study recommends how to apply the principles in practice.

Firstly, a literature review will synthesize existing knowledge, theories, and empirical findings about clean code, developer experience, and their roles. By analyzing and synthesizing diverse sources of information, this review aims to provide a comprehensive understanding of the theoretical foundations of clean code principles and their impacts on developer experience. Additionally, it seeks to identify gaps and areas for further investigation within the existing body of literature, thus laying the groundwork for subsequent empirical research.

In addition to the literature review, this study incorporates a case study approach to offer valuable insights from real-world software development practices. The study will focus on the creation of a git commit message template as a practical implementation of how clean code principles can be applied to enhance communication among developers with software development processes. By developing a standardized template for git commit messages, the study aims to showcase how clear and informative commit messages contribute to improved collaboration, code maintainability, and overall developer productivity. Furthermore, the case study provides an opportunity to explore the challenges and practical considerations involved in implementing clean code principles within real-world development environments.

Moreover, the study conducts a comparison between known tools assisting developers in adhering to clean code principles and the custom commit message template. This aspect of the research involves creating and analyzing how the custom commit template assists the case company developers during their typical working day. By comparing different tools and templates, the study aims to highlight the specific benefits and improvements gained through the application of clean code practices, such as enhanced readability, maintainability, and scalability. Through this practical demonstration, the study seeks to underscore the transformative impact of clean code principles on the quality and longevity of software systems, as well as their implications for developer experience and organizational success.

By harnessing findings from diverse resources and employing multiple methodologies including literature review, case study analysis, and tools comparison, this study aims to contribute to a nuanced understanding of significance of clean code in enhancing developer experience and advancing the practice of software engineering.

3 Theoretical research

3.1 Introduction

In this section, the author will focus on providing basic information related to clean code, including its definition, principles, differences between bad and good code. Moreover, the author dives deeper into developer experience by introducing its definition and exploring its role in modern software development. Furthermore, the relationship between clean code and developer experience will also be discussed, to give a fundamental understanding between the two concepts which are widely known by programmers and developers.

3.1.1 There will be code

In software engineering domain, it is certain that there will always be code written by programmers or developers. Code will never be gotten rid of, because code represents the details of the requirements (Martin and Feathers, 2009). It is understood by Martin and Feathers that the level of detail abstraction produced by code at some level is simply unreplaceable. As the world is moving forward rapidly, it is expected that the abstraction level of programming languages will continue to increase (Martin and Feathers, 2009). It can be understood that such development will not eliminate code.

A series of instructions given to a computer for execution is often thought of as code (Padolsey, 2020). However, this gives a false understanding of what developers are actually doing when writing code. When code is produced, humans are expressing their intents to computers, who cannot interpret

ambiguities with their initiative and experience. It is vital that a machine must be instructed with enough specificity to carry out every step (Padolsey, 2020). Therefore, the importance of programming languages – or normally called code – increases, which has narrowed down the distance between human capability and computing capability.

3.1.2 Good code and bad code

Good code

The presence of code within software development is not in question. Therefore, writing clean, understandble and maintainable code is one of the most important aspects of software development (Martin and Feathers, 2009). It is difficult to clearly determine whether a piece of code is good or bad; however, there are guidelines and recommendations from experienced peers, which programmers should follow in order to create a harmonized code pattern.

Just as in any form of communication, clarity is key. When a piece of code is clean, or so-called good, it communicates clearly one's intentions. It does not only enhance collaboration but also ensures that code is easy to maintain and modify in the future. Code is good when people can read and understand it without thinking (Beck and Andres, 2012).

By adhering to clean code principles, developers are able to produce good code, which brings more value than performance related aspects such as compile time, readability and scalability.

Bad code

On the contrary, bad code is what all programmers are trying to avoid. While there are many bad kind of code, the worst of them all is spaghetti code (Carullo, 2020). The name itself is quite self-explanatory, spaghetti code is limp, unstructured and can turn into a mess if not controlled. Spaghetti code violates many principles of structured code and is a potential threat.

Bad code can have negative impacts on a larger scale. If bad code is not maintained properly, or more and more bad code is combined, it will get to a point where fixing or maintaining will no longer be available. Bad code could bring a company down (Martin and Feathers, 2009). It is common that every programmer has encountered bad code. It is, as Martin and Feathers described, like a morass of tangled brambles and hidden pitfalls. The more time a developer spends on figuring out what is going on, the more senseless it can become.

As much as a programmer hates bad code, one must have written bad code before. Therefore, it is essential to question oneself about the decision to produce bad code. It may be due to lack of knowledge, where a developer has not been introduced to clean code principles. It can also be due to an important deadline that a developer has to meet, which results in an insufficient amount of time for neccessary steps to produce good code. Regardless of the reasons, it is common that all programmers have done it (Martin and Feathers, 2009).

3.1.3 Cost of bad code

Bad code is undeniably a threat and must be avoided. Owning bad code, or a so-called mess, is harmful both technically and financially. Bad code can slow teams down. The significance of slowdown can be huge. Every change made to the code can break other components, which results in more time invested in fixing a broken system. No change is trivial (Martin and Feathers, 2009). As the

pile of bad code builds up, the productivity of the team decreases, and with time, approaches zero.

Figure 1-1 Productivity vs. time

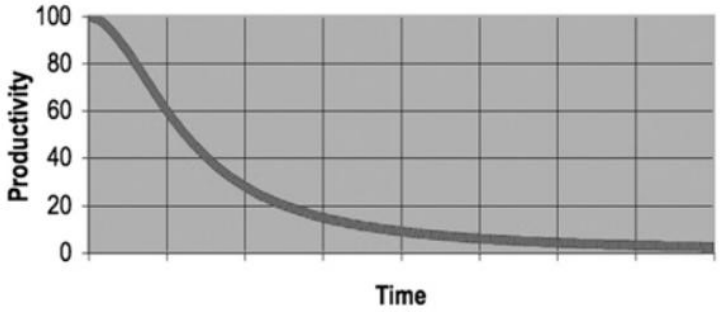

Figure 1: Productivity and bad code over time (Martin and Feathers, 2009)

As can be seen from figure 1, there is a strong connection between a team's productivity and bad code over time. As the curve in the figure illustrates, when bad code appears within a team's codebase and is not controlled, more chaos is added to a mess of bad code, which results in the team's demotivation. In this case, the company might solve the issue by employing more staff to boost productivity. However, the new staff is not versed in the design of the system, and they do not know whether a change will match the design intent. As a result, more and more mess are added, driving productivity further towards zero. The example shows that bad code can be harmful financially to a company if not properly addressed.

3.1.4 Art of clean code

Messy code is seen as a significant impediment by all programmers. It is important to determine the only way moving forward is to write clean code. However, there is no point trying to produce clean code if one does not understand what clean code means. Being able to write clean code is like painting a picture (Martin and Feathers, 2009). The majority of people recognize whether a painting is well-painted or not; however, being able to separate a good and bad painting does not mean people have the paint know-how.

Writing clean code requires a disciplined used of a myriad of techniques applied through a painstakingly acquired sense of “cleanliness” (Martin and Feathers, 2009). However, as mentioned earlier, it is not always easy to produce clean code ;moreover, in more common situations, it is not always simple to navigate through it when one is facing bad code. After all, it can be determined that a programmer who writes clean code or can see his/her way out of messy code is an artist who can take a blank screen through a series of transformations until it is an elegantly coded system (Martin and Feathers, 2009).

3.2 Clean code principles

Code is good when people read it without thinking (Martin and Feathers, 2009). Clean code principles have existed for more than a decade. They are meant to create a harmonized, clear pattern in writing code. The ultimate goal of clean code is to create software that is not only functional but also readable, maintainable and efficient throughout its lifecycle (Martin and Feathers, 2009).

In this section, the author will delve into the intricate components of clean code, providing detailed analysis of their implications within the software development domain. Throughout a systematic examination of each aspect, the author enlightens the reader about effects of clean code principles on efficiency, maintainability and scalability.

3.2.1 Naming

It is undeniable that naming has become one of the most important aspects of clean code. Naming is a universal principle. Names appear in almost, if not all, code systems (Martin and Feathers, 2009). There are multiple elements that need naming, namely variables, functions, arguments, classes, packages and source files, directories. In order to make code clean, being self-explanatory is one of the most important factors that should be taken into account. The code itself should tell a story or inform readers of what it does, and programmers can achieve such

a thing through naming. Choosing a good name might take time, but it saves more time than it takes (Martin and Feathers, 2009).

Purpose and concept

A good name indicates purpose (Padolsey, 2020). The name of an element (variable, function, class, etc) should answer a big question. It should tell why it exists and how it is used. Should a name require a comment, then the name does not reveal its intent, and it has not done its job as a name. The purpose of something is highly contextual; therefore, it should be informed by the surrounding code and the area of the codebase in which the name resides. A generic name is often used as long as it is surrounded by context that helps expressing its purpose.

A good name indicates also a contract with other parts of the surrounding abstraction (Padolsey, 2020). A name may indicate how it will be used for what type of value it contains or is expected. For instance, a variable name starting with the prefix “is” such as “isUser” ,can be expected to return a Boolean value. It is known that all names carry unavoidable expectations regarding their values and behaviours (Padolsey, 2020). It is important to be aware to misunderstandings or breaking contracts that others are relying on.

Avoidance of disinformation

Another good point to take into consideration is to avoid disinformation. Programmers must avoid leaving false clues that obscure the meaning of code (Martin and Feathers, 2009). Names with various meanings which can differ from their original meaning should not be a part of written code. For instance, instead of referring to a time variable, it is good to avoid using “d” for day; the “day” variable would be a better option in this case, as it expresses a clearer meaning of what the variable is for and its usage.

Distinction

This leads to another solid point, which is to make meaningful distinctions. Programmers create problems for themselves when they write code only for a compiler and interpreter to run (Martin and Feathers, 2009). It is not sufficient to add number series and noise words, even though the IDE does not throw errors. It is good to keep in mind that if the names must be different, they should also have different meanings.. Number-series naming (a1, a2, etc) is the opposite of naming with good intentions. Not only does it create confusion, but it also does not help writing to be readable, maintainable or scalable. Noise words make no difference. For example, if there was a “Product” class which was named “ProductInfo” or “ProductData”, then no differences would be made. Either of the variables expresses clear intent or provides extra information to readers.

One major difference between humans and IDEs is that a significant part of human brains is dedicated to the concept of word, and words are pronounceable. It is necessary to keep in mind such a difference when writing code. While programmers are writing code for an IDE to compile, there are, of course, other programmers who will need to read and understand. Therefore, it is good to make names pronounceable (Martin and Feathers, 2009). An unpronounceable name makes it difficult to discuss or talk about it. If there is a code-review where two or more people are participating, unpronounceable names create silly sounds and difficulties to address or imply functions or variables with such names.

Consistency

Consistency is another important aspect of clean code naming (Fowler, 2019). It is normally determined in big corporations or companies whether a certain guideline is given when starting to write code. However, it is not certain that all companies are providing guidelines for developers. Hence, it is important to have consistency when naming variables. Having consistency and everyone following

the same conventions reduce the cognitive load for developers, allowing them to have more time focusing on the task at hand. It is a bad practice if one part of the codebase uses camelCase for variables and functions, while another part harnesses snake_case. In such a case, confusion would be unavoidable.

Having mentioned all the above aspects related to naming when writing code, the most difficult thing about choosing names is that it requires good descriptive skills and a shared cultural background (Martin and Feathers, 2009). The author believes this is a teaching issue rather than a technical or business issue. Many people in software development are struggling to handle it. People may be afraid to rename variables or functions because different people might have different opinions. However, the author believes that most of the time programmers do not object naming change for the better; instead, programmers feel grateful. It is important to follow some of the mentioned rules and observe how one can improve the readability of his/her code. If a developer is mostly maintaining someone else's code, it can be beneficial to use refactoring tools to help solve problems. It will pay off in the short term and continue to pay off in the long run (Martin and Feathers, 2009).

3.2.2 Functions

Functions play the utmost vital role in modern programming languages. A function is a piece of code that performs a certain task. A function is a small independent unit of a program. A program contains one or more functions (Bhave and Patekar, 2012). By breaking down complex problems into smaller, more manageable components, functions enhance modularity and reusability within software development. Therefore, it is essential to write functions so that they comply to clean code principles. There are several principles which make good practice when it comes to clean code in functions.

Making a function small

A fundamental idea is to make a function small, both literally and metaphorically (Martin and Feathers, 2009). There are no exact measurements of how short or small a function should be; however, if a function is longer than one full page or more than 50 lines of code for instance, it is not considered a good function because as mentioned earlier, the use of a function is to de-structure complex problems into smaller and simpler components. Not only should a function be small, it should also contain proper blocks and indenting (Martin and Feathers, 2009). One of the most obvious examples can be if – else or while statements. Not only do blocks and indenting keep the enclosing function small, but they also add documentary values because the function called within the block can have a nicely descriptive name. Moreover, indenting makes functions easier to read and understand.

Doing one thing

According to clean code principles, a function should only do one thing (Bhave and Patekar, 2012). A function is an expression, a method of de-structuring complex problems; therefore, it is vital that functions remain their simplicity for readability and maintainability. A good way to tell whether a function is not doing “one thing” is to see if a programmer can extract another function from it with a meaningful name that is not just a description of its implementations. (Martin and Feathers, 2009).

Descriptive names

Naming principles have been mentioned earlier with their benefits. However, it is emphasized that descriptive name is one of clean code principles (Padolsey, 2020). It is difficult to overestimate the value of good names. Programmers and developers may spend a major part of their time following clean code principles by choosing good names for small functions that perform one task. The smaller

and more focused a function is, the easier it is to choose a descriptive name (Martin and Feathers, 2009).

Having mentioned that good naming can bring great benefits to software development and developer experience, it is essential that programmers are not afraid of using long names. In fact, a long descriptive name is better than a short enigmatic name (Padolsey, 2020). A long descriptive name is better than a long descriptive comment. Using a naming convention which allows long but easily readable functions create comfort and ease for developers whilst doing their daily tasks. Choosing a good name can take time; therefore, it is advised that developers and programmers should try several different names and read the code with each name in place. Time spent on having a good descriptive name will benefit users like developers greatly in the future. Moreover, modularizing and designing are easier with good names, which can help to improve the infrastructure restructuring of a user's codebase. Consistency in naming is also one good rule of thumb (Martin and Feathers, 2009). Phrases, nouns and verbs should be used in function names. It is worth considering names such as "includeSetupPage" or "editTarget" or "isNewValueANumber".

In conclusion, long descriptive names are valuable and provide developers with ease in software development. It is advised to spend time choosing a good name. It is not uncommon that companies spend a fortune on refactoring legacy code which consists of short enigmatic names, which can obviously be avoided

3.2.3 Comments

A comment is an interesting case within software development. Given the correct context, a well-placed comment is considered very helpful. To developers who are having a difficult time reading and understanding a piece of code (e.g. function, variable or if-else), a good comment is like good documentation. However, nothing can clutter up a module more than frivolous dogmatic comments. Nothing can be so damaging as an old crafty comment that expresses misinformation (Martin and Feathers, 2009).

It is difficult to determine whether comments are a bad or good thing within software development. The author believes that if programming languages were more expressive and communicative, it might be that there was no need for comments at all. The purpose of using comments can vary. It can be that developers need to add a piece of information to a certain part of code that they cannot explicitly express, or to describe what a function or a piece of code does within given context. However, whatever purpose comments have, it is thought that comments are used to compensate for failure to express oneself in code (Martin and Feathers, 2009).

Whenever developers or programmers find themselves in the position where a comment is needed, it is vital to take a step back and ponder within one's head. There are many questions a developer should be asking him/herself while being at this stage. The desire to have a comment can come from various reasons; however, most of the reasons begin with programmers having not followed strictly the clean code principles (Martin and Feathers, 2009). Is there another way to express without having to add comment to this code? – this is one of the first questions which should appear in the thought process.

Code evolves and changes; comments do not (Martin and Feathers, 2009). Codes are, for example, refactored, enhanced, omitted, added and subtracted

from time to time. Unfortunately, comments do not always follow. The older the comment is, and the further away it is from the code it describes, the more likely it is wrong. Programmers in a code refactoring or enhancing process have enough on their plates; adding maintenance of comments on top of the workload can have a negative impact on the outcome quality. Over time, good comments become bad comments, which eventually become false comments (Fowler, 2019). False comments are deluded and mislead.

3.2.4 Formatting/Indentation

When programmers or developers are reading code, it is critical that the layout of the codebase is readable, not to mention it has to be neat and consistent. The author believes that every developer wants to be impressed by the neatness and attention to detail of a particular piece of code. Code is good when people read it without thinking (Fowler, 2019). Code formatting not only helps programmers when writing code, it also makes reading code effortless. By being able to look through a codebase without pause or hesitation and by perceiving that clean code principles of formatting have been applied, it is easier for developers to focus on more important tasks at hand.

Purpose of formatting

Code formatting is nonetheless very important (Martin and Feathers, 2009). Programmers cannot ignore indentation and cannot take formatting for granted. Formatting code is about communication, which is one of the most important factors within software development.

Code formatting has a profound effect on the changes in the future. The code style and readability of code set precedents that continue to affect maintainability and scalability long after original code has been modified beyond recognition. Next the aspects of formatting/indentation that will assist developers become better at communication will be discussed.

Vertical formatting

How big should a source file be? – The author believes this is a good starting point.

An ideal format should be similar to a well-written newspaper article (Martin and Feathers, 2009). Users read a file vertically. At the top, a headline which expresses what the story is about is expected. The first paragraph gives a synopsis of the whole story, hiding all details whilst providing a broad-brush concept. As users continue, details will start to reveal themselves until the end of the file is reached.

Like in a source code file, there should be a simple yet explanatory title, which provides sufficient information in terms of whether users are in the correct module. Having a structure similar to a newspaper article, the source file is expected to contain high-level concepts and algorithms, whilst detail proximity increases the closer to the end of file.

Nearly all source code file is read left to right and top to bottom. Each line represents an expression or a clause, and each group of lines represents a complete thought or performs a certain action. Moreover, it is emphasized that programmers do not skip the meaning of blank lines. They have such a profound effect on the visual layout of the code. Each blank line is a visual clue that states a new and separate concept. This is called “vertical openness between concepts”. When users scan down the source code file, the eyes are drawn to the first line that follows a blank line (Martin and Feathers, 2009).

Concepts that are closely related should be positioned vertically close to each other. It is also vital to organize one’s data so that similar concepts can be found in the same source file, or source tree. Vertical separation should be a measure of how essential and related files are positioned and how important each file is to the understandability of others. It is advised to avoid forcing readers hopping around through source files to find needed information.

Declaration of variables should happen as close to their usage as possible (Martin and Feathers, 2009). Functions were mentioned earlier as a method of de-structuring complex problems into simpler and smaller ones; therefore, functions can be short. Position variables close to their usages reduces time spent searching or reading functions. Similarly, variables of loops should be declared within the loop statement.

Horizontal formatting

How wide should a line of code be? In the research carried out by Martin Roberts in 2009, line length distributions were studied in approximately seven projects.

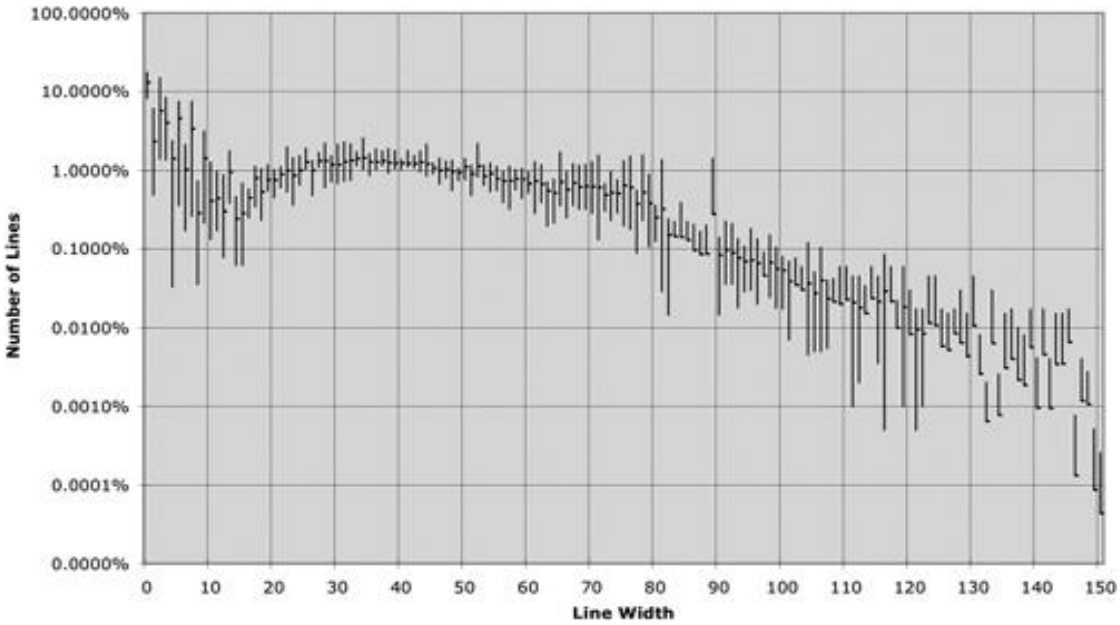

Figure 2: Java line width distribution

The research was done with Java projects. As can be seen from figure 2, the longer a line is, the less chance there is that it appears in a codebase. Most

lines spread between 20 to 60 characters, which takes up about 40% of the projects. This suggests that it is essential to keep code short. A good rule of thumb is to have a limit of 120 characters whenever code is produced, so that readers do not need to scroll horizontally (Martin and Feathers, 2009).

Horizontal white space is used to associate components that are strongly connected and that, on the other hand, disassociate things that are more weakly related. Similarly, to when writing a function, it is a must for developers to have white space after function arguments and the curly brace that is the beginning of what the function does. Meanwhile, function names should be closely positioned together.

As mentioned earlier, indentation plays an outstanding role in a codebase's readability. A source file is a hierarchy rather like an outline (Padolsey, 2020). Information within a source file contains individual classes and methods or blocks within the methods. Each level of this hierarchy is a scope into which names can be declared. To make the hierarchy of scopes more visible, line indentation is harnessed.

Statements and classes are not indented at all; they are positioned at the beginning of each line. Methods within a class are indented one level to the right of the class. Implementations of those methods such as functions are indented again one level to the right of the method declaration.

Programmers consider indentation to be something that they rely heavily on (Martin and Feathers, 2009). Indentation helps visualize lines of code from left to right and position code in order and scopes. Good visualization allows developers to quickly go through scopes by scanning. Without indentation, programming would be virtually unreadable by humans.

3.2.5 Error handling

Error handling is one of the things programmers and developers do when writing code. Inputs can be inconsistent, and programs can fail (Martin and Feathers, 2009). When such an event occurs, programmers are responsible for ensuring that the codebase can do what it needs to do. Therefore, within the current software development field, it is worth reviewing how programmers tackle various errors and exceptions.

When a person learns programming, it is believed that one has first a brush with syntax errors. A good compiler provides necessary assistance on how to tackle them. It is also observed that small programs are practically trouble free after testing and debugging. What about larger programs? The larger the program, the higher is the possibility of errors (Bhave and Patekar, 2012). Therefore, it is worth remembering that error handling is an important aspect of software development, and it needs to be properly handled.

Handling negative cases and returning early

When writing code, programmers are basically adding more code to the codebase. This can increase the complexity of the codebase in one way or another. Complexity generally negatively affects developer experience by making code harder to read and to maintain, thus, increasing the likelihood of bugs (Martin and Feathers, 2009). A good rule of thumb in handling such cases is to handle negative cases first and return early. This approach can significantly improve code readability and scalability, not to mention decrease in the need of using nested statements.

```
Const employeeJourneyData = useSelector((state) => state.home.data?.employeeJourney);
If (!employeeJourneyData) {
        Return null;
}

Return (
        <div classname=”employee-journey”>
                <h3 classname=”employee-journey__header”>
                        {intl.formatMessage({
                                Id: “employee_journey.header”
                        })}
                </h3>
                <ServiceJourneyContainer
                        journeySteps={employeeJourneyData}
                />
        </div>
);
```

Listing 1: Handling negative cases and returning early examples

Listing 1 shows a piece of code from the author. It is with clear intent that the author checks for the employeeJourneyData variable to be available first. If there are issues whilst fetching data, a null value is returned, letting the IDE know that there is an error in displaying data.

It is known that handling negative cases and returning early is a powerful technique to improve readability and reduce complexity of code (Martin and Feathers, 2009). The approach leads to simpler, more straightforward code that is easier to read, understand and maintain. By following this practice, developers

can enhance developer experience and reduce the risk of introducing bugs, ultimately leading to more efficient and productive development processes.

Error handling as documentation

Error handling in a codebase is like letting the IDE know what to expect (Padolsey, 2020). Moreover, not only is it worth informing the IDE, but it is also worth informing fellow developers and programmers of what is expected at a certain function if an error handling technique, such as try-catch. Therefore, handling exceptions is like having documentation about a certain function without having to write it down.

The purpose of clean code is to be readable; however, it must also be robust. Programmers can write robust clean code if error handling is seen as a separate concern. This ensures core functionality remains clear while systematically managing unexpected events.

Error handling serves as both a safeguard and implicit documentation. It prevents crashes, provides meaningful user messages, and logs detailed information for developers. Well-implemented error handling highlights potential failure points, indicating anticipated exceptions and their management.

In conclusion, treating error handling as integral documentation aligns with clean code principles. It clarifies potential issues, maintains resilience and improves user-friendliness. By considering error handling crucial, programmers can ensure their code is functional, robust and communicative.

3.2.6 Unit tests

In software development, ensuring the reliability and functionality of code is undeniably one of the utmost important factors. A unit test is a fundamental practice that assists developers in achieving that goal. By isolating individual

components of the software and verifying their correctness, unit tests play a critical role in a development process.

Three laws of Test-Driven Development (TDD)

Since a unit test plays such a vital role in software development, it is advisable to acknowledge the Three Laws of TDD in unit testing (Martin and Feathers, 2009). The first law states that programmers may not write production code until a failing unit test has been written, whereas the second law states that developers may not write more of a unit test than is sufficient to fail, and not compiling is failing. The third law states that developers may not write more production code than is sufficient to pass the currently failing test.

The three laws lock programmers into a cycle that is perhaps 30 seconds long. The tests and the production code are written together, with the tests just a few seconds ahead of the production code. Given that all laws are strictly followed, a huge number of test cases will be written over time. Theoretically, this approach would provide full coverage into the smallest details to the codebase.

Clean tests

Various clean code principles have been mentioned; however, it is vital that programmers do not forget an important clean test principle, which is readability. Readability in unit tests is perhaps more important than in production code (Martin, 2012). Followed by clarity, simplicity and density of expressions, readability helps developers to read and, more importantly, to understand the intention of a unit test.

A clean test should clearly express what is being tested, under what conditions, and what the expected outcome is. This clarity enables developers to quickly grasp the purpose of the test, facilitating easier maintenance and quicker debugging. Simplicity ensures that tests are not over-complicated with

unnecessary logic, focusing instead on the specific behaviour being validated. The density of expressions, meaning the conciseness and precision of the code, helps in making tests more manageable and less error prone.

3.2.7 Version control

Version control is a fundamental practice in software development that enables teams to manage changes to the codebase efficiently and collaboratively. It tracks modifications, facilitates coordination among developers and maintains a comprehensive history of the project. This chapter provides a brief overview of version control systems (VCS), their importance and core concepts.

Version control systems, such as Git, help prevent conflicts by allowing multiple developers to work on different parts of the project simultaneously. These systems support branching and merging, making it easier to develop features, fix bugs and experiment without disrupting the main codebase. As highlighted by Kent Beck in “Extreme Programming Explained”, effective version control is crucial for agile development, allowing teams to respond to change quickly (Beck and Andres, 2012).

The two main types of version control systems are centralized (e.g., Subversion) and distributed (e.g., Git). Distributed systems, as explained in “Pro Git” by Scott Chacon and Ben Straub, offer greater flexibility and robustness by allowing each developer to have a complete copy of the repository (Chacon, 2009).

Core concepts in version control include repositories, commits, branches, and merges. Understanding these concepts is essential for leveraging a VCS effectively. Regular commits capture the project's evolution, branches enable isolated development, and mergers integrate changes back into the main line.

3.2.8 Commit message

A commit message serves as the primary means of communication about the changes within a codebase (Chacon, 2009). A well-crafted commit message provides a clear, concise description of what has changed and why, which assists in understanding the history and evolution of the project.

Adhering to a commit message convention is essential for maintaining consistency and clarity across the codebase. One widely adopted convention is the Conventional Commits specification, which structures commit messages into a format that includes a type, scope and a brief description. A commit message from the case company could be taken as an example,

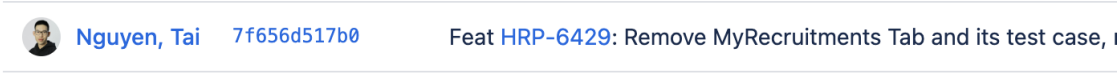

Figure 3: Example commit message

This approach not only enhances the readability of the commit history but also supports various automated tools and processes that rely on a structured commit message. Additionally, clear and consistent commit messages make it easier for team members to review changes, conduct code audits and track down the origins of specific modifications.

The implementation of commit convention within a development team encourages better practices and improves overall project maintainability. By following a set of structures, developers can ensure that every commit message conveys meaningful information, contributing to a more understandable and navigable codebase. Moreover, later within the study, the author presents a commit message template that will be harnessed within a software development team. The template is expected to create better readability and understanding of a commit, which allows other developers to fully grasp the set of changes within a commit.

3.2.9 Popular tools

Achieving clean, readable and maintainable codebase is every developer's responsibility (Martin, 2012). To support programmers in meeting the goal, a variety of tools have been developed to enforce certain coding standards. Among the most used tools are Prettier, Checkstyle and EsLint. Each of the mentioned tools serves a distinct role in assisting developers to follow guidelines and standards, which can contribute to a scalable and well-organized codebase.

Prettier

Prettier is a code formatter that automatically ensures styling consistency across projects. Prettier can be customized to suit developers' preferences, allowing adjustment for attributes such as line length, indentation or quote styles. Formatting tools like Prettier eliminates the need to discuss code style when working among team members, which reduces cognitive load (Padolsey, 2020). This allows developers to focus more on functionality and logic rather than code-style related preferences.

Checkstyle

While Javascript tends to harness Prettier's functionality to increase the codebase's readability, Checkstyle is mainly used in Java projects. Checkstyle is a static analysis tool designed to help keep maintainability of coding standards. Like Prettier, Checkstyle offers flexibility in customizing guidelines or rules in terms of naming conventions and indentation. Checkstyle is thought to be crucial for ensuring consistent code quality, especially in collaborative settings where multiple developers contribute to the same codebase (Bhave and Patekar, 2012). Automation is one of Checkstyle's strengths; it simplifies the process of checking for style violations, helping to keep consistency in the general structure.

ESLint

Another Javascript's formatting tool is ESLint, a linting tool which detects and aligns violations in styling based on pre-defined rules configured either by developers or default settings. ESLint also provides high levels of customization and supports plugins for different rule sets. ESLint helps maintain clean code, reduces technical errors and improves overall code structure (Crockford, 2008).

SonarQube

SonarQube is widely known software harnessed for its continuous code quality inspection feature. A comprehensive static code analysis is provided by the platform to detect bugs, code smells and security vulnerability. SonarQube helps smoothing development workflows, maintaining a clean and readable codebase (SonarQube, 2024).

One key feature provided by SonarQube is its ability to enforce quality gates, which act similar as thresholds for code quality metrics that must be met before code in a project can proceed to the next stage of the development pipeline. This ensures issues are caught early, promoting cleaner and more reliable code (Martin and Feathers, 2009). Clean code should be easily understandable and free from unnecessary complexity. These principles are supported by SonarQube, allowing developers to shift their focus to producing and delivering high-quality and maintainable code.

TypeScript

TypeScript is a typed programming language built on top of JavaScript, allowing developers to catch errors or uncertainties in the codebase during the development phase rather than at runtime or deployment. The approach enhances code quality, as potential bugs can be identified early in the software lifecycle (Padolsey, 2020).

By providing robust type-checking capabilities, TypeScript assists programmers in enforcing the clear code standard. TypeScript requires variables, functions and class members to clearly state what type they are in and what return type

they are expecting, which reduces ambiguity. TypeScript's type annotations improve code readability, maintainability, which is essential for effective collaboration in product development (Beck and Andres, 2012). TypeScript's integration to development environments like Visual Studio Code is one of the features allowing developers to quickly adhere to clean code principles when producing code.

Tern

Another lightweight Javascript analysis design platform designed to help developers by providing intelligent code completion, error detection and dependency analysis is TernJS. TernJS is a stand-alone application which offers real-time insights into Javascript code. It contributes to clean code principles by reducing complexity and errors and promoting clarity and maintainability of code (TernJS, 2024).

Within the JavaScript context, TernJS excels at parsing code to understand its structure and dependencies, allowing it to offer suggestions base on its understanding. TernJS's auto-complete function reduces cognitive workload for developers. Avoiding unnecessary complexity is one of the factors leading to clean code (Martin and Feathers, 2009). TernJS is also customizable, letting programmers adjust their needs according to specific projects, which helps to flexibly adhere to different code standard.

Code climate

Code climate is a comprehensive platform widely known within the software development field, especially in area where code quality and optimization are main goals. Code climate team members are the pioneers in software engineering intelligence, who assist engineering teams to strive for better code solutions. (Code Climate Documentation, 2024).

Code Climate offers two primary solutions: Code Climate Quality and Code Climate Velocity. Each of the solutions offers a different perspective on the

client codebase. While Code Climate Quality focuses on static code analysis, it helps detecting potential code complexity, duplication or bugs. There is also possibility to integrate with the Continuous Integration/Continuous Development (CI/CD) pipeline. Code Climate Velocity provides insight into team productivity and workflow. It can track critical metrics such as cycle time – time taken to complete tasks – and review speed, which monitor speed of code reviews between developers. This helps to identify bottleneck situations in processes to make improvements on workflow. Together these tools support clean code principles by enhancing maintainability, readability and collaboration (Martin and Feathers, 2009).

Jest

Jest is a widely known open-source testing framework developed by Facebook. It is designed to ensure code correctness and readability within Javascript and Typescript context. Jest is thought to be best suited for React applications; however, it is not limiting itself to only React but is also available to handle various Javascript frameworks. Jest contributes to clean code practices by promoting extensive testing, which helps maintain code quality, readability and reliability (Padolsey, 2020).

One of Jest's strengths is being easy of use. It simplifies the process of writing and running tests manually, and built-in configuration helps developers to reduce time spent on setting up test environments. Jest allows programmers to capture and validate the rendered output of User Interface (UI) components in JavaScript frameworks over time, ensuring changes do not behave in unexpected ways. Moreover, Jest offers to run tests in parallel in a default setting, optimizing speed and efficiency for a software project. Overall, Jest offers extensive testing features that support the clean code principle (Jest, 2024)

Coveralls

Coveralls is another popular code coverage tool that helps programmers track whether a part of their codebase is tested efficiently. By integrating with testing frameworks like Jest, Coveralls provides detailed reports of test coverage metrics. The reports help teams identify gaps in their testing strategy and ensure critical parts of their codebase is well-covered (Carullo, 2020).

Coveralls offers possibility to integrate with a CI/CD pipeline like GitHub or Bitbucket to automatically generate coverage reports during a development cycle. Not only does Coveralls highlight uncovered parts of code, making it easier for developers to focus on untested code, but it also supports various languages other than Javascript. Clean code emphasizes the importance of comprehensive testing, encouraging developers to focus on writing test cases, which ensures code is both robust and maintainable (Beck, 2004).

4 Empirical research

4.1 Introduction

Having explored the theoretical foundations of clean code and its principles, it is important to be aware of its critical role in enhancing developer experience. The next step is to connect these concepts with practical implementation. This chapter focuses on presenting an empirical aspect of the study, which allowed the author to applied clean code principles into practical scenarios. Specifically, the author emphasized on implementing a commit lint template and refactoring existing code at the case company based on the principles introduced in the theoretical part.

The first chapter focuses on the commit lint template which the author has been developing and implementing. By enhancing a communication channel such as a commit message, the author believes consistence and maintainability will improve within the code base (Martin and Feathers, 2009). Standardized commit messages not only make version control more understandable but also support efficient collaboration among developers.

Following the commit lint implementation, the empirical study will document the refactoring of legacy code. This process will showcase how clean code practices such as simplifying functions, function naming or enhancing error handling can affect directly the readability and maintainability of the codebase (Beck and Andres, 2012; Fowler, 2019). Through this practical application, the author aims to demonstrate how clean code principles elevate both code quality and developer experience.

4.2 Commitlint

Commitlint is a tool that enforces structured commit message conventions in software development. By ensuring consistency in commit messages, Commitlint helps developers create a clean and organized code history, which in turn supports better code quality and collaboration. Clear communication in codebases is a cornerstone of maintainable, scalable software, and commit messages play a pivotal role in the clarity of codebase by documenting the “why” behind changes (Martin and Feathers, 2009).

4.2.1 Role of commit messages in developer communication

Standardized commit messages serve as documentation that helps teams understand the purpose of changes. In large software projects, poor commit structures can lead to “commit message debt”, where uninformative messages reduce the ability of team members to understand the history of a project (Martin and Feathers, 2009). Commitlint addresses these issues by enforcing a structured commit message, which improves transparency and readability.

The ability to quickly grasp the intent of each change without reading code directly is essential for team efficiency (Sommerville, 2011). With Commitlint, commit messages adhere to a format that can communicate changes effectively, enabling smoother workflow and reducing misunderstandings in projects.

Moreover, having a clear commit history simplifies the debugging process. By following commit convention rules, for instance, using commit types like “Fix”, “Feat” or “Chore” helps programmers locate relevant commits during debugging. A descriptive change being mentioned in the commit message is also what makes developers’ lives easier. Being able to show the change intent and clearly describe it is key within the software development field.

4.2.2 Commit message template

By following Commitlint guidance and references, the author was able to configure a commit message template which adheres to commit convention rules which enables better communication between commit messages and workflow.

Structure

Commit conventions allow teams to add more semantic meaning to git history. Being able to include “type”, “scope” or “breaking changes” assists developers in adding more necessary information for peer programmers (Commitlint, 2024).

The most common commit conventions follow the structure below:

```text
type(scope?): subject
body?
footer?
```

type(scope?): subject

body?

footer?

Listing 2: Commit conventions structure

Listing 2 presents the template structure typically used by tools like Commitlint, which ensures that commit messages follow a consistent standard. The template consists of five main elements: type, scope, subject, body and footer. Each of these elements serves its purpose in conveying the content and context of the commit message.

The header of the commit contains three elements – type, scope and subject – which summarize the intent of the change in one single line. The “type” file categorizes the nature of the change, using labels like feat (feature), fix (bug fix) or docs (documentation). This helps developers quickly understand a certain type of updates or improvement, which is useful for tasks like debugging or code review.

The “scope” field indicates the specific area of the codebase that the change affects. However, most teams or projects are using a ticketing system when it comes to software development like Jira or Trello (Atlassian, 2024). It is also widely known that the “scope” field is meant for a ticket number. The case company could be taken as an example. Jira is a project management software that is being harnessed currently, for better case management in a bigger picture. All tickets within the author team at the case company start with “HRP”. Moreover, whenever a ticket is created, a ticket number will be appointed by Jira, which means a standard example of the author’s team is “HRP-1234”.

The “subject” element of the header provides a brief description of the changes in sentence-case format. Ideally, the description is concise and clearly states the purpose of the changes in a single line. A short, descriptive subject line helps developers quickly understand the key point of the commit without needing to examine the code itself (Martin and Feathers, 2009).

The ”body” and “footer” sections are both optional. They are recommended for more complex commit messages that require additional explanation. The mentioned sections allow developers to elaborate the change’s context and implementation details that might not be obvious from the header alone.

A standardized commit structure not only promotes clarity and readability but also supports consistent communication among team members. Aligning with clean code principles, the Commitlint approach reinforces clarity, readability and maintainability in software development (Fowler, 2019).

Example

This section explains how the rules are being applied on the standardized commit message template.

Figure 4: Example of commitlint rules

In the configuration file ‘commitlint.config.js’ that locates in a parent directory, a set of rules is configured to standardize the commit message format across the project. The setup aims to maintain a well-organized and clear commit history, which is vital for effective collaboration and long-term code maintainability and readability. Each rule of the configuration has a specific function, ensuring every commit adheres / to a set structure.

Given the structure of a commit message mentioned above, the configuration file sets a standard for three main components: type, scope and subject. According to official Commitlint documentation, ‘2’ means that the console will always raise whichever commit message does not satisfy one or more rules within the configuration file, as an error, which leads to programmers being

unable to commit. It is essential for a commit message to have such ‘type-enum’ property, which means the commit message should contain one in the given types (e.g. Chore, Feat, and Fix). While scope and subject rules are basic standard, the author intends to ‘ignore’ commit messages that contain ‘WIP’, which means ‘Work in Progress’. ‘Work in Progress’ or ‘WIP’ indicates that a developer is in the middle of a certain workflow. By marking a commit as ‘WIP’, developers can effectively signal to team members that the changes are still under development, which avoids confusion or unfinished code. Having commit rules ignored, ‘WIP’ commit messages will assist developers in strictly following commit rules while it is not yet necessary to do so.

A commit message example from the case company project can be seen below.

Figure 5: Example commit message

4.3 Advantages of commitlint

This section compares the custom commitlint template developed in this study with existing linting tools that allow users to enforce clean code principles into codebases. The analysis highlights the unique features of the proposed template, its alignment with clean code principles and its impact on developer experience compared to other tools. The comparison focuses on the following aspects: flexibility, usability and effectiveness in improving DX.

4.3.1. Flexibility

The custom commit lint message template prioritizes adaptability, allowing the case study team to define rules that align with specific project needs. For example, team members can agree to customize prefixes to reflect task types

(e.g fix, feat, chore and docs) and scopes that mirror the domain-specific components. This ensures that the template remains relevant across different projects and is flexible to changing requirements. Flexibility is also one of the core elements the case study team prefers/preferred (?) based on several discussions. Moreover, flexibility is also shown through the ability to integrate and combine with other tools such pre-commit hooks (pre-commit, 2024). It allows developers to freely select linting tools in their own preferences while still being able to get the best results out of many tools.

Existing tools such as Conventional Commits are powerful but tend to emphasize standardization over customization. While they allow configuration through JSON or YAML files, their predefined structures may not align fully with unconventional or custom workflows, which leads to incompatibility. Additionally, ability to combine different tools is what case study team evaluates highly, this gives the opportunity to take advantage of different tools for different purposes.

4.3.2. Usability

The proposed commit message template was designed to be straightforward and accessible, ensuring usability across all levels of expertise. Programmers reported that the provided examples create easy usage, even for less experienced team members. By integrating simple error messages when a set of rules are violated, the template offers real-time guidance, making it easy to navigate and complete a commit message. Moreover, installation the phase is said to be like usability in terms of being straightforward and easy to understand. The author set up several steps needed to complete the installation. Each step guides clearly what users need to do in order to successfully install and use the commit message template.

While the established tools offer extensive documentation and community support, the complexity can pose challenges for teams unfamiliar with configuration or command-line interfaces. Features such as integration with

continuous integration (CI) pipelines add functionality but may require additional setup and training. However, once they are implemented, their usability improves quickly significantly due to automated enforcement mechanisms (Martin and Feathers, 2009).

4.3.3. Effectiveness in improving developer experience

Empirical findings from the case study team demonstrated that the custom commit lint template had a significant positive impact on DX. Developers reported that the standardized format reduced ambiguity in commit messages, allowing better communication, letting teammates know the purpose of each change during code reviews. This clarity also enhanced debugging quality and improved collaboration by reducing the need for clarification or follow-up discussions. The template's simplicity ensured that it did not add unnecessary complexity to the development process. Such a feature is a key factor to a successful software development project (Martin and Feathers, 2009).

The existing tools are also effective in improving DX, particularly by enforcing consistency and providing detailed error feedback when commit message rules are not followed. However, their broader functionality can add overhead to the development process, especially for smaller-scale teams or projects. The effectiveness of improving DX often depends on the customization level and expertise of developers which harness them.

5 Discussion

5.1 Introduction

This chapter discusses the findings of the study in response to the research questions raised in Chapter 2. By addressing the set of questions, the chapter aims to explore the interconnection between clean code principles and developer experience in the modern software development field. Moreover, the

chapter’s intention is to explain how clean code practices impact developer productivity, team collaboration and overall system maintainability.

While the research questions guide the discussion, this chapter also explores broader implications of the findings. Specifically, this chapter examines how clean code contributes to improving workflow efficiency and promotes positive development culture. Additionally, practical insights from the empirical section are harnessed to provide the needed context for creating actionable recommendations for development teams.

5.2 What is developer experience (DX)?

Developer Experience is a critical dimension of software development which includes perceptions, emotions, and overall satisfaction of developers during the interaction with the tools, systems and workflows. Like User Experience, where the focus lies on end-user satisfaction, DX emphasizes the efficiency and ease of a programmer’s journey during the process of building, maintaining and deploying a software system, in other words, a software lifecycle (Sommerville, 2011).

DX is shaped by a variety of factors, including usability, efficiency, consistency and satisfaction (Martin and Feathers, 2009). The theoretical framework highlighted in Chapter 3 demonstrates that DX has direct impact on developer performance and team dynamics. It is believed that the tools and processes which simplify daily repetitive tasks reduce cognitive workload and promote transparency can significantly enhance DX, creating more room for efficiency to grow (Padolsey, 2020).

From the author’s point of view, developer experience can be determined as what a developer goes through during one working day and how the developer’s emotions and feelings throughout the day affect productivity and quality. The case study team member could be taken as an example. A typical working day involves a variety of activities, from taking the first cup of coffee of

the day to constantly being in meetings or doing ad-hoc tasks and performing code reviews. Such busyness does not support a developer in writing quality code. Therefore, the author believes that it is vital to ensure developers are at their best in terms of comfort when reading or writing code; thus, good developer experience will come in handy.

5.3 What are clean code principles?

Clean code principles form a foundation for writing software that is efficient, maintainable and easy to understand. They serve as a set of guidelines that programmers can follow to ensure functioning yet readable and scalable code. Clean code is often described as code that communicates clearly, avoids unnecessary complexity and adheres to consistent practices (Martin and Feathers, 2009).

The principles are built on several key factors, including naming, functions, error handling, formatting ,indentation, comments, version control and a commit message. While each of these factors serves a distinct purpose as stated in Chapter 3.2, they together can help developers create software that stands the test of time if adhered to; moreover, programmers can foster an environment where efficiency and satisfaction thrive.

The choice of naming plays a critical role in making code self-explanatory. Names should be indicative, avoid disinformation and be consistent. Functions are said to only be small, do only one thing and avoid unnecessary complexity. Error handling, indentation and comments are believed to help code become more robust, readable and protect the codebase from the test of time (Martin and Feathers, 2009).

5.4 What are the criteria for clean code?

Clean code is not solely about following a set of principles but also about meeting specific, measurable criteria that ensure code is not only functional but

also easy to maintain, scale and understand (Martin and Feathers, 2009). Several core criteria have been identified through both theoretical and empirical research. These criteria provide a framework for assessing code quality and ensuring long-term maintainability. The author believes clean code criteria is such a vast topic, and that one can adhere to those criteria in different ways according to priorities in values. The author has had discussions with case study teammates of certain elements of clean code principles being presented and discussed to find the best suited solutions for the case company team, which were readability, maintainability, scalability and consistency.

A developer will spend most of his or her time reading someone else's code during their careers, said Robert Martin. The author noticed the same during one of his first days working as a Junior Software Developer; therefore, the author strongly believes that readability should be one of the most essential values that contributes greatly to how clean a person's code can be. Readable code reduces the effort needed to understand one's intentions. Whether it is a simple function or a complex recursive function, writing readable code not only helps the author but also other peers. Clean code is like well-written prose, where each section is clear and the intention behind every function or variable is immediately apparent (Martin and Feathers, 2009).

Maintainability is another element which can be measured when referring to clean code. It refers to how easily code can be updated, modified or extended without introducing errors or complications (Fowler, 2019). Clean and maintainable code is designed to allow adjustments from other developers without the need of refactoring. It is emphasized that maintainability is a product of consistent practices, modular design and avoidance of complicated solutions (Padolsey, 2020).

Scalability is another element to take into consideration when trying to demonstrate clean code. Scalability involves the ability to accommodate new features or handle increased demands without major considerations. Clean code is designed to be scalable, which also minimizes complexity by avoiding

tightly coupled components and encourages modular design (Beck and Andres, 2012).

Consistency refers to following certain coding standards and conventions throughout the codebase. A consistent coding style allows developers to understand and modify code more easily. It is vital that consistency exists so that it is beneficial for teamwork as it reduces confusion and promotes development (Fowler, 2019).

5.5 How important is DX in software development?

The author believes having a good developer experience will greatly contribute to the work quality of a developer. At the case study team, by simply writing a clear, descriptive commit message, the impact is already huge as it helps other teammates to get a good understanding of whatever goal the author is trying to achieve. A small action like putting effort into a commit message is already having such a positive impact that the author is excited when imagining what effect an improved systematic developer experience will have on software development industry.

As the software development field keeps growing rapidly, complex systems are constantly being designed and developed, which places more emphasis on the importance of providing developers with efficient and supportive environments (Sommerville, 2011). This chapter explores the significance of developer experience in fostering productivity, collaboration and satisfaction within a team.

5.5.1 Productivity through optimized workflows

Optimized workflows ensure developers spend less time on repetitive, low-value tasks and place their minds on more critical tasks which require creativity or different solutions. Optimized workflows can exist in various forms, from a simple commit message template ensuring developers communicate correctly to each other via code and commit messages, to harnessing CI/CD pipelines

which automate testing and deployment, allowing developers to work on certain processes seamlessly (Beck and Andres, 2012).

The case company team can be looked at as an example. The team is using a pre-commit, commit lint and Bamboo CI/CD pipeline for formatting, commit messages, testing and deployment respectively. Whenever there are code changes, such as a new feature or a bug fix or documentation, developers run certain scripts which harness a pre-commit library to check, for example, imports, extra spaces, unused variables, and frontend issues. Moving on to the next step, developers will create a commit message describing their intentions using a/the commit message template adhering to a certain set of rules which ensure clear and descriptive communication via commit messages. Once a commit message is created and the working branch is up-to-date, developers create a pull request waiting for other peers to review it. Once the review process is completed and the pull request is approved and merged, programmers start the process of deploying the changes to different environments, like development or quality assurance, whereas during the deployment, various tests for the frontend using Jest and the backend using pytest will be conducted, ensuring there is no room for bugs. When the changes satisfy the tests, the programmer will manually make a final deployment – while closely monitoring the process – to production where it is available to all users.

Within the automated workflows mentioned above, developers are thought to place their focuses on writing maintainable, clean code and let the system handle automated actions. By embracing such workflows, it is said to reduce significantly the effort required for manual and repetitive tasks, which in turn boosts productivity (Fowler, 2019).

5.5.2 Minimizing frustration and reducing turnover

Frustration is a common yet overlooked challenge within software development. The issue comes from various reasons such as disorganized codebases, inefficient tools or unclear processes. When frustration is not being solved, it

can lead to developer burnout, decreased morale, which eventually can lead to turnover (Martin and Feathers, 2009). Therefore, it is essential to address this problem through focusing on developer experience.

There are many causes of developer frustration, which often stem from the inability to understand legacy systems or “spaghetti code”. Another reason can be that outdated and old-fashioned tools make it difficult to increase the workflow’s pace, which requires developers to spend more time on routine tasks. Vague commit messages are also one of the main reasons that create frustration as they can lead to misunderstandings and duplicated efforts (Fowler, 2019).

While the frustration level increases, it leads to turnover in personnel consequently. High developer turnover has several negative effects on companies or teams. When experienced programmers leave, it is not only critical knowledge about the codebase, workflows or other practices they are taking with them, but also it is about the costs of hiring and onboarding new members, as this requires a significant amount of time and resources, which companies cannot afford. If the codebase is poorly managed or lacks comprehensive documentation, the turnover effect is much greater (Sommerville, 2011).

5.5.3 Improving collaboration

Collaboration is the backbone of successful software development teams, and it is vital to foster such culture for delivering high-quality software efficiently. By acknowledging and improving developer experience, companies and teams can benefit from the culture, which encourages the developers, creates an environment where developers can collaborate seamlessly and share knowledge on a frequent basis (Beck and Andres, 2012).

A positive developer experience ensures software development teams and organizations operate under consistent standards and practices, while reducing

ambiguity and aligning effort between team members. By adopting clean code principles, it can be thought that a shared language is established. Such a shared language reduces cognitive workload during code reviews and simplifies collaboration across different parts of the project (Martin and Feathers, 2009).

An improved developer experience is essential for team communication as it reduces barriers and makes information easily accessible (Fowler, 2019). By harnessing clean code principles, developers are equipped to write better documentation, which ensures all team members, regardless of role or experience, can access clear guidelines and debugging steps. This reduces dependency on specific individuals for knowledge, promoting smoother collaboration. Moreover, tools like Bitbucket (Bitbucket, 2024), combined with DX-driven practices like commit lint, allow contributions to be tracked and documented properly, which promotes and fosters a culture of open communication and collaboration.

6 Conclusion

The study examined the relationship between clean code principles and developer experience, emphasizing how embracing clean code principles can improve software development processes and encourage a positive environment for developers. Through a combination of theoretical research and practical analysis, the study demonstrated the significance of clean code in enhancing productivity, collaboration and satisfaction within software development teams or organizations.

The findings in both theory and practice highlight DXs are a crucial factor in modern software development, and that they are often forgotten. A good DX is proven to reduce frustration and promote efficiency among programmers, directly impacting the quality of the code produced and the overall project success. Clean code practices such as readability, maintainability and scalability are seen as key factors contributing to DX. The empirical case study validated insights, showing that practices such as structured commit messages

and modular design improved communication, which enhanced workflows and reduced unnecessary errors.

The study also compared the custom commit message template developed during the research with existing tools, such as conventional commits or EsLint or Checkstype. The comparison revealed that while established tools provide solid automation and scalability, the custom template offers flexibility, simplicity and ease of integration, which made it effective for smaller teams or projects with customized workflows. By following the case study team’s needs, the proposed template is reported to enhance DX without unnecessary complexity.

Despite contributions, the research has limitations. The empirical findings were drawn from a single case study, which may not apply to all development environments or teams. Future research offers possibility to expand on these findings by exploring different team settings and languages to better understand the role between clean code and DX.

In conclusion, the study highlights the importance of clean code as a fundamental factor influencing positive DX. By enforcing clean code principles, software development teams and organizations can enable developers to work more effectively and collaborate seamlessly. As the software industry continues to move forward, a strong DX will remain critical for building high-performing teams capable of producing maintainable, scalable and readable code.

References

Atlassian, Trello (2024). Available at: https://www.atlassian.com/software/trello (Accessed: 11 November 2024).

Atlassian, Jira (2024). Available at: https://www.atlassian.com/software/jira (Accessed: 11 November 2024).

Atlassian, Bitbucket (2024). Available at: https://www.atlassian.com/software/bitbucket (Accessed: 4 December 2024).

Beck, K. and Andres, C. (2012) *Extreme programming explained: embrace change*. 2nd ed. Boston: Addison-Wesley (The XP Series).

Bhave, M. and Patekar, S. (2012) *Object-oriented programming with C++*. 2nd ed. New Delhi: Dorling Kindersley.

Carullo, G. (2020) *Implementing effective code reviews: how to build and maintain clean code*. New York, NY: Apress.

Chacon, S. (2009) *Pro Git*. Berkeley, CA: New York

Code Climate Documentation (2024). Improving code quality and engineering performance. Available at: https: www.codeclimate.com (Accessed: 18 November 2024).

Commitlint (2024) *Commit conventions*. Available at: https://commitlint.js.org/concepts/commit-conventions.html (Accessed: 31 October 2024).

Cortex (2024) ‘What is developer experience’. Available at: https://www.cortex.io/post/why-developer-experience-matters (Accessed: 15

April 2024).

Corckford, D. (2008). JavaScript: The good parts. Sebastopol, CA: O'Reilly Media.

Feathers, M.C. and Martin, R.C. (2005) *Working effectively with legacy code*. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference (Robert C. Martin series).

Fowler, M. (2019) *Refactoring: improving the design of existing code*. 2nd edition. Boston: Addison-Wesley (The Addison-Wesley signature series).

Jest (2024). Available at: https: www.jestjs.io (Accessed: 18 November 2024).
Martin, R.C. (2012) *Agile software development: principles, patterns, and practices*. Internat. Ed. Upper Sadle River, NJ: Pearson Prentice Hall (Alan Apt series).

Martin, R.C. and Feathers, M.C. (2009) *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, N.J.: Prentice Hall.

Padolsey, J. (2020) *Clean code in JavaScript: develop reliable, maintainable, and robust JavaScript*. Birmingham: PACKT Publishing.

Pre-commit (2024). Available at: https://pre-commit.com/ (Accessed: 9 December 2024).

Sommerville, I. (2011). Software Engineering. Addison-Wesley
SonarQube (2024). Static code analysis tool. Available at: https: www.sonarqube.org (Accessed: 18 November 2024).

TernJS (2024). Tern: A Javascript Code Analyzer. Available at: https: www.ternjs.net (Accessed: 18 November 2024)

Appendices

Appendix 1. Handling negative cases and returning early examples

```
Const employeeJourneyData = useSelector((state) => state.home.data?.employeeJourney);
If (!employeeJourneyData) {
        Return null;
}

Return (
        <div classname=”employee-journey”>
                <h3 classname=”employee-journey__header”>
                        {intl.formatMessage({
                                Id: “employee_journey.header”
                        })}
                </h3>
                <ServiceJourneyContainer
                        journeySteps={employeeJourneyData}
                />
        </div>
);
```

Appendix 2. Commit convention’s structure

type(scope?): subject

body?

footer?