International Journal of Academic Information Systems Research (IJAISR)
ISSN: 2643-9026
Vol. 6 Issue 12, December - 2022, Pages: 1-64

Clean Code in Practice Developers' perception of clean code

Mohammed Yousef Abu Hassan

E-mail: medo94125@gmail.com

Abstract: ***Context****. There is a need for developers to write clean code and code that adheres to a high-quality standard. We need developers not to introduce technical debt and code smells to the code. From a business perspective, developers that introduce technical debt to the code will make the code more difficult to maintain, meaning that the cost for the project will increase.* ***Objectives****. The main objective of this study is to gain an understanding about the perception the developers have about clean code and how they use it in practice. There is not much information about how clean code is perceived by developers and applied in practice, and this thesis will extend the information about those two areas. It is an effort to understand developers' perception of clean code in practice and what they think about it.* ***Realization (Method)****. To understand the state-of-the-art in the area of clean code, we first performed a literature review using snowballing. To delve into developers' perception about clean code and how it is used in practice. We have developed and sent out a questionnaire survey to developers within companies and shared the survey via social networks. We ask if developers believe that clean code eases the process of reading, modifying, reusing, or maintaining code. We also investigate whether developers write clean code initially or refactor it to become clean code, or do none of these. Finally, we ask developers in practice what clean code principles they agree or disagree with. Asking this will help identify which clean code principles developers think are helpful and which are not.* ***Results****. The results from the investigation are that the developers strongly believe in clean code and that it affects reading, modifying, reusing, and maintaining code, positively. Also, developers do not write clean code initially but rather refactor unclean code to become clean code. Only a small portion of developers write clean code initially, and some do what suits the situation, while some do neither of these. The last result is that developers agree with most of the clean code principles listed in the questionnaire survey and that there are also some principles that they discard, but these fewer.* ***Conclusions****. From the first research question, we know that developers strongly believe that clean code makes the code more readable, understandable, modifiable, or reusable. Also, developers check that the code is readable using code reviews, peer reviews, or pull requests. Regarding the second research question, we know that developers mostly refactor unclean code rather than write clean code initially. The challenges are that to write clean code initially, a developer must have a solid understanding of the problem and obstacles in advance, and a developer will not always know what the code should look like in advance. The last research question showed that most developers agree with most of the clean code principles and that only a small portion of developers disagree with some of them. Static code analysis and code quality gates can ensure that developers follow these clean code practices and principles.*

Keywords: clean code, code quality, technical debt, refactoring

Clean Code in Practice

Developers' perception of clean code

Mohammed Yousef Abu Hassan

1, January, 2023

Contact Information:
Author:
Name: Mohammed Yousef Abu Hassan
E-mail: medo94125@gmail.com

ABSTRACT

Context. There is a need for developers to write clean code and code that adheres to a high-quality standard. We need developers not to introduce technical debt and code smells to the code. From a business perspective, developers that introduce technical debt to the code will make the code more difficult to maintain, meaning that the cost for the project will increase.

Objectives. The main objective of this study is to gain an understanding about the perception the developers have about clean code and how they use it in practice. There is not much information about how clean code is perceived by developers and applied in practice, and this thesis will extend the information about those two areas. It is an effort to understand developers' perception of clean code in practice and what they think about it.

Realization (Method). To understand the state-of-the-art in the area of clean code, we first performed a literature review using snowballing. To delve into developers' perception about clean code and how it is used in practice. We have developed and sent out a questionnaire survey to developers within companies and shared the survey via social networks. We ask if developers believe that clean code eases the process of reading, modifying, reusing, or maintaining code. We also investigate whether developers write clean code initially or refactor it to become clean code, or do none of these. Finally, we ask developers in practice what clean code principles they agree or disagree with. Asking this will help identify which clean code principles developers think are helpful and which are not.

Results. The results from the investigation are that the developers strongly believe in clean code and that it affects reading, modifying, reusing, and maintaining code, positively. Also, developers do not write clean code initially but rather refactor unclean code to become clean code. Only a small portion of developers write clean code initially, and some do what suits the situation, while some do neither of these. The last result is that developers agree with most of the clean code principles listed in the questionnaire survey and that there are also some principles that they discard, but these fewer.

Conclusions. From the first research question, we know that developers strongly believe that clean code makes the code more readable, understandable, modifiable, or reusable. Also, developers check that the code is readable using code reviews, peer reviews, or pull requests. Regarding the second research question, we know that developers mostly refactor unclean code rather than write clean code initially. The challenges are that to write clean code initially, a developer must have a solid understanding of the problem and obstacles in advance, and a developer will not always know what the code should look like in advance. The last research question showed that most developers agree with most of the clean code principles and that only a small portion of developers disagree with some of them. Static code analysis and code quality gates can ensure that developers follow these clean code practices and principles.

Keywords: clean code, code quality, technical debt, refactoring

CONTENTS

ABSTRACT ..... **I**

CONTENTS ..... **II**

1 INTRODUCTION **4**

1.1 BACKGROUND 4
1.1.1 Clean code *4*
1.1.2 Code quality *4*
1.1.3 Code smells *4*
1.1.4 Technical Debt *5*
1.1.5 Refactoring *5*
1.2 PURPOSE 5
1.3 SCOPE 6

2 RESEARCH QUESTIONS **7**

3 RESEARCH METHOD **8**

3.1 LITERATURE REVIEW 8
3.2 QUESTIONNAIRE SURVEY 9
3.2.1 Participant recruiting and survey overview *9*
3.2.2 Data collection *9*
3.2.3 Data Analysis *9*
3.3 ALTERNATIVE METHODS 10

4 LITERATURE REVIEW RESULTS **11**

4.1 DEVELOPERS' BELIEF IN CLEAN CODE 15
4.2 WRITING CLEAN CODE INITIALLY OR REFACTORING CODE TO CLEAN CODE 15
4.2.1 Proactive versus Reactive refactoring *15*
4.2.2 The Reasons to Why Developers Refactor Code *15*
4.2.3 Refactoring Tools in Practice *16*
4.3 CLEAN CODE PRINCIPLES AND PRACTICES FOUND 16
4.3.1 Static Code Analysis and Quality Gates *19*
4.4 LITERATURE REVIEW CONCLUSION 19

5 SURVEY RESULTS **20**

5.1 DEMOGRAPHICS 20
5.2 DEVELOPERS' BELIEF IN CLEAN CODE 21
5.2.1 Thematic analysis *23*
5.3 CLEAN CODE INITIALLY OR UNCLEAN CODE FIRST 23
5.3.1 Thematic analysis *26*
5.4 PROMINENT CLEAN CODE PRINCIPLES 26
5.4.1 Thematic analysis *31*

6 ANALYSIS **33**

6.1 DEMOGRAPHICS 33
6.2 DEVELOPERS' BELIEF IN CLEAN CODE 33
6.3 CLEAN CODE INITIALLY OR UNCLEAN CODE FIRST 34
6.4 PROMINENT CLEAN CODE PRINCIPLES 34

7 CONCLUSION **35**

8 VALIDITY THREATS **36**

9 FUTURE WORK **37**

10 REFERENCES 38

11 APPENDIXES 41

11.1 APPENDIX A 41
11.2 APPENDIX B 43
11.3 APPENDIX C 49
11.3.1 Likert scale questions 49
11.3.2 Other question types 54
11.4 APPENDIX E 57
11.4.1 RQ1: Thematic analysis 57
11.4.2 RQ2: Thematic analysis 58
11.4.3 RQ3: Thematic analysis 60
11.5 APPENDIX F 61

1 INTRODUCTION

1.1 Background

1.1.1 Clean code

Clean code [1] advocates for writing readable code so that other people can know the code's intent almost directly. It should be easy to follow someone else's logic when reading clean code. Easily readable code will make other programmers understand the code better, leading to increased code maintainability [2]. If other programmers understand the code by just reading it, it will also be easier for the programmers to modify it when needed since they understand what it does. For example, clean code deals with naming, structuring, formatting, refactoring, testing, etc.

The "Clean Code" movement has defined principles and practices that will help programmers write improved code [3], such as using meaningful names, indentation, avoiding duplication, and many more. Most developers have probably been at the stage where they name a variable horribly, and it does not convey any meaning, and they are only using it at the current moment. The developers know what the name means right now, but they will probably not know why it is there if they had to read it about one year. It may not even be the same developers that re-read and modify the code. Some other developers may continue to maintain the code that the previous developers wrote. Hence, we must write code that is understandable to other developers. Developers should not have to think a lot to understand the code written by others. They should know what the code does just by reading it.

1.1.2 Code quality

Code quality is heavily related to clean code, but those two concepts are not the same. According to Börstler et al. [4], people perceive code quality as having software metrics to measure a system's quality. When talking about code quality, we mean code that is of high quality. What high-quality code is considered is dependent on the context and the team we are working within, and there is no definitive set of metrics we can combine to characterize what code quality is [5]. Everyone has different opinions and different perspectives of what is considered high-quality code, and we need to reach a consensus about this within the team. If we are not reaching a common consensus, the team members may follow what they consider good code. What is considered good or bad code depends on a developer's experience [4]. According to all participants of the study reported in [4], the top three most concerning code qualities were the code's readability, structure, and comprehensibility. In order to maintain high-quality code, we must ensure that we reduce or remove the code smells of it.

1.1.3 Code smells

Code Smells are symptoms of bad design and implementation choices [6], [7], and can introduce degradation to the code quality, making it more difficult to understand, change, and maintain [8]. When accidentally introducing code smells to a system, it can introduce faults that make the system more troublesome to maintain in the future. Programmers must try to avoid introducing code smells because of these reasons. Examples of code smells are Duplicated code, Long method, or Large class. Yamashita and Moonen [8] have studied whether developers know about code smells in practice, and they found out that about 32% of

developers did not know anything about code smells. Showing an evident lack of knowledge of code smells on the developers' side. To enable developers to produce *code-smells-free* code. We need to raise awareness on what code smells are first. If programmers do not know what a code smell is, they cannot assess whether a piece of code contains smells and try to remove them. Code smells have been around since the late mid-nineties, and developers have built tools that help developers identify the code smells. However, development teams are often unaware of the significant benefits that a refactoring tool has in practice [9].

1.1.4 Technical Debt

We can avoid introducing some of these code smells by applying the clean code principles and practices to make it easier to write high-quality code [3]. It will also help to reduce the technical debt of the code. Technical debt is a metaphor used to discuss (and often quantify) the long-term consequences of suboptimal decisions taken with the goal of speeding up the development [10]. It deals primarily with non-visible aspects and issues of software development and its maintenance. Mistakenly or knowingly introducing technical debt to the code in the project will make the code more difficult to maintain and enhance. It is leading to increased cost and time of the development of the product. The longer the project has been going on, and technical debt has increased, the harder it is to remove [3]. Technical Debt Issues (TDIs) are an atomic, measurable manifestation form of Technical Debt, and being Code Smells is one type of TDIs [11].

1.1.5 Refactoring

In case TDIs have already been introduced, then refactoring can be a solution. Refactoring is a technique used to modify the code without changing the system's external behavior [12]. External behavior means that the functionality of the system has not changed and still works as intended. In other words, the changes made to the code by a developer when performing a refactoring should not change the result of what the code or system did before. In practice, refactoring is defined differently but is not far from the state-of-the-art definition [12]. Developers do not relate a lot to refactoring to preserve the code's external behavior. Developers are more concerned about refactoring in terms of readability, maintainability, or performance. Refactoring operations are behavior-preserving, but we can refactor the code to make room for the development of new features, architectural or design changes [13]. Then the change will not be behavior preserving anymore.

1.2 Purpose

Professional developers do know how to get a program to work correctly and achieve functional correctness. However, this does not necessarily mean that all developers know how to write code that is easy to read, modify, reuse, and maintain, but some developers might know. Therefore finding out what the clean code principles experienced developers agree with can be helpful to support less experienced developers when writing code that is more readable and understandable. Companies need developers who can write code that is easy to be read, modified, reused, and maintained. Developers who can do this will help reduce friction in the development process, making it possible to smoothly introduce changes to the software while avoiding introducing technical debt into the code. Therefore, this thesis aims to determine the perception the developers have about using clean code in practice.

1.3 Scope

The scope of this thesis focuses on whether developers believe in clean code from a practical perspective. We will check if they believe clean code contributes to more readable, understandable, modifiable, or reusable code. Only a few attributes from the ISO 25010 standard are selected because most of the other quality attributes are not concerned with clean code. We also focus on analyzing how clean code is produced in practice: writing clean code initially or writing not so clean code to be refactored later. It is about if developers do refactor source code to become clean code. It is not in the scope of the thesis to analyzing refactoring code for other purposes than making the code clean. Refactoring operations (e.g., Rename, Extract Method, Move Method, Pull Up Members) will not be discussed in depth. However, tools such as static code analyzers and quality gates will be discussed in the context of its usage to write clean code. Finally, in the last section, we focus on the practices and principles of clean code.

2 RESEARCH QUESTIONS

The main objective of this study is to gain an understanding about the perception the developers have about using clean code in practice. To achieve this goal, we have defined the following research questions:

- **RQ1:** Do developers believe that clean code eases the process of reading, understanding, modifying, or reusing code?
- **RQ2:** Do developers initially write clean code or write unclean code that later on needs refactoring to become clean code?
- **RQ3:** What are the most prominent clean code principles developers need to use in practice to write clean code, which makes code easy to read, modify, reuse, and maintain?

The first research question is about whether developers believe that clean code affects easing the process of reading, understanding, modifying, or reusing code. We are investigating if developers believe in clean code or not. Developers will have a chance to state their opinions about why they do or why they do not believe in clean code.

The second research question aims to understand whether developers in practice can write clean code initially or if they write more messy code at first that will need refactoring later on to become clean code. We are investigating the challenges with writing clean code initially or refactoring unclean code to become clean code.

Finally, the third research question is about what prominent practices and principles developers use in practice. To answer the first research question, we will complete a systematic literature study to find relevant practices and principles and then use a questionnaire survey to see what the developers think about these. Developers will agree and disagree with some principles, which will hopefully help us sort out what principles most developers see as prominent.

3 RESEARCH METHOD

3.1 Literature Review

To understand the state-of-the-art in the area of clean code, we performed a literature review using snowballing, following the guidelines by Wohlin [14], combining a database search to define the start seed with snowballing iterations doing citation and references analysis.

To define the start set (seed), we carried out a database search using Google Scholar with the search string *allintitle: clean code* but realized that this resulted in very few relevant papers. Therefore, we used another search string: *"clean code" OR "code quality"* to find more relevant papers. We were looking for papers reporting if clean code is used in practice or papers that report the impact of clean code principles and practices on code quality.

We performed both forward- (citations analysis) and backward- (references analysis) snowballing iterations starting with the seed. Forward snowballing (citations analysis) is the process of searching for potential papers which cited a particular paper using a citation database, i.e., in our case Google Scholar. Backward snowballing is the process of looking at the references of a given paper to find new potential papers:
We defined the following inclusion criteria that we applied to consider a paper as relevant

- Is the paper published in an English journal, conference, or workshop proceedings indexed in Google Scholar?
- Is the paper published after 2010?
- Does the paper include the terms "clean code" or "code quality" in the title, abstract, or full text?
- Does the paper define principles and practices of clean code or report their usage in practice?

We applied the abovementioned acceptance criteria both to define the start set and during the snowballing iterations. We have excluded papers talking only about static analysis techniques unless there is a strong emphasis on their use in practice.

The publication date for the book about clean code is 2009, and we exclude papers published less than or equal to the year 2009. The publication date's acceptance criteria are that it has to be equal to or greater than 2010. Due to that the Clean Code book is the foundational work for clean code. Since it is was published year 2009, we set the publication date 2010, since many people might not have read it the year it was published.

In some rare cases, when doing forward snowballing, the citations were more than 100 for the current paper, so to remove noise, we filtered the results using the search string *"clean code" OR "code quality"* to find the candidate papers. In other words, we used Google Scholar to look for citations on the current paper and then search once deeper with the previous search string.

The snowballing procedure goes on until reaching saturation which means we do not find any more relevant papers.

Only papers that have been peer-reviewed such as conferences, journals, magazines, and workshops, are included as relevant papers. The only exception the Clean Code book [1] written by Martin, which we included as grey literature since it is the foundational work, and we will be using it as a reference.

3.2 Questionnaire Survey

3.2.1 Participant recruiting and survey overview

We used a questionnaire survey sent out to developers within different companies and shared it on social media and forums. The survey included questions related to the research questions about clean code and dividing the survey into sections mapped to these research questions. The setup for the thesis was to have sub-questions in the questionnaire to each of the research questions. The participants we aim for are developers that have practical experience in programming or experience in the software industry.

3.2.2 Data collection

The data collection was performed through an online questionnaire developed using the QuestBack survey tool. The questionnaire was distributed using social networks, but also spread out to contacts within some companies that redistributed the survey within their respective organizations, therefore we used convenience sampling.

Most of the questions in the survey consist of a seven-point Likert scale ranging from 1 to 7. We choose this Likert scale to avoid the central tendency due to cultural aspects (in Sweden, we can refer to it as *Lagom effect*), or the cultural tendency to "not too much, not too little" [15]. Questions from *strongly disagree* to *strongly agree*. Other questions were yes or no, short text, or drag and drop ranking.

Since some of the questions are regarding specific clean code principles, we included a short description of each of them, in case developers were unfamiliar with some specific principles. However, if this was not enough for the developer to understand the principle, we provided a link called "more info" behind the name of each principle. Developers could click on that link to get more info about a specific principle, and a new window would open to explain the principle further.

When designing the questions for the practices and principles, we decided to use a question matrix, to avoid having a question per principle. However, we noticed that having all principles in the same matrix question was unpleasant for the participants (i.e., the participants will have an extremely long question with more than 30 principles and will have to scroll down to provide the answers). Therefore, we divided this large question into sections according to which chapter the principles belonged to in the Clean Code book [1]. We grouped the principles into the general category for principles that were not clear which chapter they belonged to or were coming from other sources.

3.2.3 Data Analysis

The closed questions are analyzed using different types of diagrams (e.g., diverged stacked bar, pie chart) that summarize the developers' response to the question. To analyze whether to include or exclude a principle, the Wilcoxon p-value test will be used to check if the answers were significantly higher than the neutral value 5, if p-value is less than 0.05, means that it is statistically significant bigger [16]. If $p < 0.05$, then we include the principle, and otherwise, we exclude it.

The open-ended questions are analyzed using thematic analysis [17]. Thematic analysis is a systematic framework for coding analyzing qualitative data to identify patterns emerging from datasets in response to research questions [17]. We applied an inductive (bottom-up), in which the themes emerge and are linked to the data [18].

For the thematic analysis process, we began with coding and identifying themes that we could group, as illustrated in Figure 3.2.1. To ensure that the coding and identifying of themes have been adequately done, we have iterated through the process some more times. Once the themes seem to have been found and grouped correctly, the process is ended. Also, mind maps will

be used for each thematic analysis section to present the findings visually to the readers. Each mind map will have the research question it is investigating in the center.

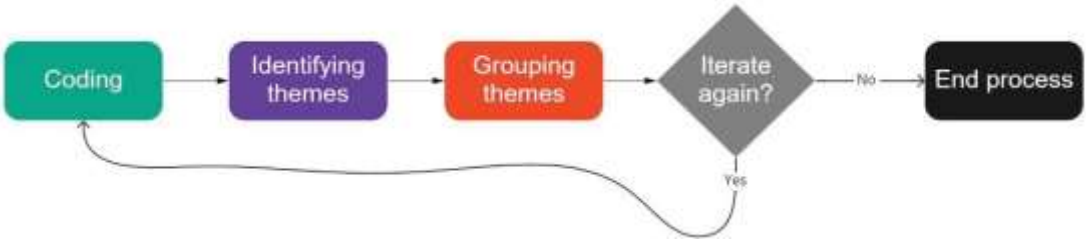


Figure 3.2.1: Thematic analysis process

3.3 Alternative methods

Our approach for searching for references was to use Google Scholar and then do a literature review using the snowballing procedure. We could also have had defined databases that we could use to search within. If using a database, we would get very few hits on clean code, and we would probably miss papers to include due to a strict search query. In contrast, we would get too many papers about clean code if defining a search query that is too general.

Our main research method was to create a questionnaire survey to collect information since this felt appropriate since we can send it via the internet to the developers, and they only have to answer it. This way, it is easier to get many participants. As an alternative, we could have interviewed developers via video conference calls and then recorded the interview to clean up later and transcribe it. Preferably recording both video and audio, but excluding video if participants are hesitant toward it. Only in case, we got permission from the participants, of course. The advantage of interviews is that we would have gotten a broader perspective of what the participants think about clean code and get more insight into it. The disadvantage is that it is harder to find participants that want to do an interview, and besides, after interviewing them, we do not know how much analysis we have to do.

Regarding data analysis, thematic analysis seemed suitable since the theory about clean code is not entirely new [19]. There is evidence of it in literature. Thematic analysis is more about grouping collected data to conclude when there is a context to the questions asked [19]. We, therefore, did not need to build an entirely new theory upon the open-ended answers like we could have done if using Grounded Theory. We also ask more specific questions that are not so general, and Ground Theory is more suitable for more general questions.

4 LITERATURE REVIEW RESULTS

We used the snowballing procedure to find literature about clean code. We did both forward and backward snowballing iterations to find relevant papers. We kept on doing this until we reached saturation, i.e., when we did not find any more relevant papers to add.

To identify papers in the start set iteration, we used the papers' with the criteria that the papers mentioned "clean code" and "code quality" either in the title, abstract, or full text. The papers must have been peer-reviewed and be of the type conference, journal, magazine, or workshop.

Since the search query *"clean code" AND "code quality"* on Google Scholar found about 723 papers, most of them did not apply to the type criteria, making it easier to exclude many of them. Papers in the start set iteration are often tricky to select. We looked carefully at the title. If not sure to include or exclude from reading the title, whether the title had "clean code" or "code quality" in it. Also, if the title had anything regarding clean code in practice, we opened the paper and read the abstract, introduction, and conclusion. Mainly to find whether any of the previously mentioned sections said anything about "clean code" or "code quality", but also clean code in practice. To decide whether to include the paper in the identified start set. In the start set, we found 9 papers that we included in the start set.

In iteration one of the snowballing procedure, we continued to adhere to the criteria we had set. In this iteration, we found a total of 11 papers to include. We did not find any more papers that we thought were relevant using the snowballing procedure for some of the papers' start set. In iteration two, we found six papers to include. However, some of these papers sometimes had way over 100 citations, and it is not easy to look at over 100 cited papers and decide whether to include or exclude them. It would take a long time to do so. We needed to search within the citations using a Google Scholar search query such as *"clean code" OR "code quality"* to make the citations manageable. Otherwise, the snowballing procedure could have gone on for a very long time. Adding the new criterion that if the papers have very many citations, then, in that case, the search query will be used in further snowballing iterations. In iteration three, we only found three papers that were relevant to include. We did not find any more relevant papers in iteration four of the snowballing procedure and reached saturation.

Table 4.1 shows the papers that we have found in the seed and iterations. The table shows the total papers that we found, including discarded ones. It also shows which papers that we included in which iteration.

Seed or iteration	**Number of citations and references screened**	**Included papers**
Seed		S01, S02, S03, S04, S05, S06, S07, S08, S09
Iteration 1	23 references and 6 citations	P1, P2, P3, P4, P5, P6, P7, P8, P9, P10, P11
Iteration 2	10 references and 6 citations	P12, P13, P14, P15, P16
Iteration 3	0 references and 3 citations	P17, P18, P19
Iteration 4	0 references and 0 citations	

Table 4.1: Snowballing iterations

See Table 4.2 below, which shows the papers found in the seed and the iterations. It shows how the papers are related and the connections between how a particular paper was found. The paper reference for a seed paper is, for example, S01. Papers are just given paper references in the way the list was ordered. The paper references for iterations is, for example, P1. The references column is if the new paper was found when looking at the references of the current paper, called backward snowballing. The cited column is if the current paper was searched for on Google Scholar and then clicked on how many cited the paper, called forward snowballing. After having included papers that might be used as references, we also denoted whether the

papers are empirical or not. Checking what research method the papers are using to do their study. Also, explaining with a short description what the main contribution of each paper is.

Rigor and relevance We used rigor and relevance method [20] to evaluate the papers included. Rigor and relevance scores were used to assess the quality of the papers. Rigor is a scale that discusses how well the context, study design, and validity of the papers are described with possible values 0, 0.5, and 1. To calculate the rigor value, we sum the numbers for context, study design, and validity to get a total rigor score for the paper. The context column is supposed to describe the context to the reader. The study design is how the study was planned, designed, executed. The study design is if the reader can replicate the same study from reading the paper. Finally, validity is about the threats to the paper's validity, such as internal validity, external validity, and construct validity, if applicable.

We identify how relevant a paper is by using the relevance score. In relevance, unlike rigor, we have four variables instead of three, and these four columns are subjects, context, scale, and research method. When evaluating the columns in relevance, we only use 0 to 1 as a score to put on a specific paper. Then we added these four values from the relevance columns to get the total relevance score. It may be unclear what subjects column are, but that is just if the users are the intended users (e.g., programmers, developers, software engineers) or not in the paper. The context column is about whether it was done in the right environment setting or not—the developers developing a big project from an industrial perspective. Scale is about the size of the application or code used in the paper, whether it is of realistic size or not.

The research methods contributing to relevance are the following:

- Action research
- Lessons learned
- Case study
- Field study
- Descriptive/Explorative study

Paper [ref]	Found in	Refs	Cited	Rigor	Relevance	Empirical	Main contribution
S01 [3]	Seed			1	1	No, experience report	Software platform prototype for SME. Trying to incorporate DevOps principles to the industrial domain
S02 [2]	Seed	P1, P2, P3		2,5	2	Yes, semi-structured interview	Focus more on people related aspects to write high quality code
S03 [4]	Seed	P4, P5, P6		3	2	Yes, survey and interview	Perception of code quality divided amongst students, educators, and professional developers
S04 [21]	Seed			2	4	Yes, case study	Refactoring game code in C# to remove code smells
S05 [22]	Seed		P7	3	4	Yes, case study	Add, modify, delete. Clean code to reduce Technical Debt Density
S06 [23]	Seed			2	1	Yes, experiment	Classify source code as bad smell, ambiguous, clean code
S07 [24]	Seed	P10	P8, P9	3	1	Yes, experiment	Impact of refactoring. Negative/Positive effects
S08 [25]	Seed		P11	1	4	Yes, case study	Using a refactor tool helps developers resolve issues with naming, unnecessary code, etc.
S09 [26]	Seed			1	3	Yes, experience report	Using practices from Agile, DevOps, Software Craftsmanship helping teams define requirements more accurately
P1 [27]	Iter 1			3	4	Yes, questionnaire survey	Focus on OOP concepts and design quality. Also, Clean Code principles such as SOLID.
P2 [28]	Iter 1			1	4	Yes, case study	Humans need involvement in the process of evaluating quality, and not
P3 [8]	Iter 1	P12, P13		3	3	Yes, exploratory, and descriptive survey	32% of developers did not know about code smells, and the majority of developers were moderately concerned about code smells.
P4 [29]	Iter 1		P14	3	1	No, experiment	For-loops harder than ifs, and flat structures are slightly easier than nested structures
P5 [30]	Iter 1			3	2	Yes, controlled experiment	Difficult to give meaningful names to methods, variables, etc. Comprehension of parameters and local variables.
P6 [31]	Iter 1			2,5	2	No, effect analysis	Classification model for more readable or less readable code, and coding violations

P7 [32]	Iter 1			2,5	3	Yes, evaluation and analysis	Adding new features with clean code takes 7% less effort compared to unclean code
P8 [33]	Iter 1			2	1	No, systematic literature review	Few studies about what refactoring techniques affects what software quality attributes
P9 [34]	Iter 1			2	1	No, literature review	Refactoring has positive effect on external and internal software quality attributes, and lack of empirical studies regarding refactoring techniques
P10 [12]	Iter 1	P15	P16	2,5	4	Yes, field study	Refactoring in practice is more than just behavior-preserving program transformations
P11 [35]	Iter 1			2,5	1	No, systematic literature review	Refactoring scenarios can have conflicting results on quality
P12 [36]	Iter 2			2,5	1	Yes, exploratory study	Use module decay index to calculate if a module is becoming smelly, and prevent it if so
P13 [9]	Iter 2	P16		2	4	Yes, survey	Few refactoring tools provides recommendations to developers, and testability of correctness after refactoring
P14 [37]	Iter 2	P18, P19		2	1	Yes, experiment	Python code snippets read by developers to determine if easy or hard to read
P15 [38]	Iter 2			3	3	Yes, large-scale study, quantitative/qualitative	Developers refactor for code readability, fault-proneness, testability
P16 [39]	Iter 2			2	4	Yes, semi-structured interviews	Appropriate and inappropriate uses of refactoring tools and configuration overhead for refactoring tools
P17 [40]	Iter 3			3	3	Yes, industrial case study	Refactoring done for many reasons, tests are needed to ensure correctness after refactoring
P18 [41]	Iter 3			3	1	Yes, field study	Experienced programmer reviewing and checking code readability using Code Readability Testing technique
P19 [42]	Iter 3			3	2	Yes, controlled experiment	Minimize nesting, avoid do-while loop to increase readability and understandability

Table 4.2: Papers found in SLR

4.1 Developers' belief in clean code

For RQ1, we have not found many papers from the literature review investigating whether developers believe in clean code and its effect at all. Therefore, this is not easy to answer. However, most papers such as [3], [32], [22] are talking about clean code, maintenance, and technical debt. The problem with this is that it does not tell whether developers believe in clean code. From the literature review, we do not have any proof that developers believe in clean code or not. This will have to be answered by the results from the questionnaire survey instead.

4.2 Writing clean code initially or refactoring code to clean code

In the snowballing procedure, we found many papers that talked about both refactoring and technical debt. To the best of the author's knowledge, not so many papers studied whether developers write clean code initially, but many papers talking about refactoring such as [12], [25], [32], [36], [40], could be found. Therefore, it was not easy to find many papers that talked precisely about writing clean code initially versus refactoring smelly- or unclean code to clean code. Most papers only talked about code that was written in a bad state already. Therefore the code had to be refactored. Arif and Rana [32] mention that writing clean-code initially would positively affect software developers in the long term since it saves time and effort. Most papers do not mention anything about writing clean code initially. A refactoring proposal is discussed in the next section, which may help developers write clean code initially.

4.2.1 Proactive versus Reactive refactoring

There are two types of refactoring which are *proactive refactoring* and *reactive refactoring*. Reactive refactoring is when the code already has code smells that need to be fixed to clean up the code, while proactive refactoring is when the code smells have not yet been introduced but are about to be introduced soon [36]. It would be positive if developers can avoid introducing code smells and sense when they are about to be introduced. However, developers need to measure if the source code is becoming smelly. According to Sae-Lim, Hayashi, and Saeki [36], the developers can calculate the module decay index (MDI) to determine if a module is becoming smelly. Using the proactive method instead of reactive refactoring, developers can foresee which modules are becoming smelly and take action in advance. The developers do not have to wait until a module has become smelly to remove the code smell. It would not be practical to do refactorings on each class or module. Therefore, we have the MDI that helps with determining what a decaying module is. A decaying module is a module that does not have any code smells, but the module's quality is about to degrade, possibly introducing code smells [36]. The percentage of the decaying modules between each release seemed to be about 19% on average. A machine learning approach was suggested to more efficiently predict whether a module is heading toward being decayed in the software's next release. This suggestion could be one way for the developers to notice that the code they are writing is becoming less clean and have a chance to avoid it, meaning that they may have it somewhat easier to write clean code initially in that case. Then the developers do not need to refactor the code and can fix it almost directly instead.

4.2.2 The Reasons to Why Developers Refactor Code

There are many reasons to perform refactorings, and we will only name a few of them that are mentioned in [25], [32], [39]. One of the main reasons is related to the maintainability of the code [32]. Maintainability is vital for a software system to be maintained in the future. Reducing the maintainability cost is done by repaying the technical debt introduced in the code. It is not always that the original developers of the system are maintaining it. It could be

other developers, so it is also crucial to write readable and understandable code, which is another reason for refactoring. Code can sometimes be confusing to read and understand. The last reason to refactor mentioned by authors is to reduce the human errors made when programming.

To summarize why developers refactor, some of the reasons are: maintainability, readability, understandability, reduce technical debt in code, reduce human errors. There are probably more reasons than only these, but these are some of the reasons developers perform refactorings. Moving on from here, we investigate how developers perform their refactorings. Whether developers perform refactoring manually, use a refactoring tool, combining the two options, or do neither of these.

4.2.3 Refactoring Tools in Practice

Vakilian et al. mention that [39], researchers have found that most developers do not prefer to use refactoring tools and instead do small changes manually. Introducing large-scale changes when doing refactoring is often error-prone. Therefore, such manual refactorings and automated refactorings are avoided by the developers. Developers fear that the refactoring operations performed by them might break functionality that was working as intended before. Therefore, Kim, Zimmermann, and Nagappan [12] and Latte, Henning, and Wojcieszak [3] suggest that developers should write unit tests to confirm whether any functionality is no longer working—trying to detect if the program's behavior is still correct and has not changed because of the refactoring operation. According to the study, it can be understood that developers do other modifications to the code while refactoring, possibly introducing behavior changes. It is more error-prone to do refactoring manually since humans are more likely to make errors than a refactoring tool. A refactoring tool might introduce behavior changes in the code, but it is more automated and should help the developer avoid introducing as many errors while refactoring. As shown by studies such as [12], [39], developers do not use the refactoring tools a lot but are aware that these tools do exist and what refactoring operations the refactoring tools support. Vakilian et al. [39] mention that 90% of developers performed refactorings manually. There are many reasons behind developers not using refactoring tools. Some of the identified reasons developers do not use the refactoring tools as much as possible are due to the issues with naming, trust, predictability, and configuration [39]. Refactoring operations sometimes have too complex names that the developers do not understand. The names lead to the distrust in refactoring tools and the need for predictability to see what a particular refactoring operation does. Finally, the refactoring tools also have a configuration overhead, and developers are usually not the ones who configure the refactoring tools.

Refactoring tools in practice also have the problem that they do not have a graphical user interface design that is simple enough to use [39]. The user interfaces can be too complex for the developers and make it difficult for them to see how to use the refactoring tool. This problem makes it unclear to developers if the refactoring tool is worth using. Regarding usability, the refactoring tool designers need to investigate this so that developers will have it easier to use the refactoring tools. Usability is important because if a refactoring tool is too complicated, no one will use it. Even though the refactoring tool could support the programmers if used within the proper context. Vakilian et al. [39] mention that some developers may overuse the refactoring tool in the wrong context. A developer that does not have much experience with programming may be trusting the refactoring tool too much and cannot verify if the refactoring was performed correctly without introducing behavior changes.

4.3 Clean Code Principles and Practices Found

For RQ3, we used the Clean Code book [1] to check what practices and principles existed. In the iterations, we found some of these clean code practices and principles. From these practices and principles, we try to pick out the ones that are related to the themes that Robert Martin has defined. We needed to look for papers talking about the same principles as Martin has

mentioned or new ones strongly related to clean code. This chapter lists the principles found. Most of the principles and practices are from the Clean Code book [1] and named in Table 4.3 below. Below is a summarization of the principles that we have found in the literature review and what papers reported them. The papers did not report much evidence of practical use for most of the principles. An extended table with a short description of the principles can be found in Appendix B.

Table of General principles			**Table of Naming principles**		
Principle	**Literature**	**Evidence of its use in practice**	**Principle**	**Literature**	**Evidence of its use in practice**
Boy scout rule	B, P2, S05		Use Meaningful Names	B, S01, S02, S04, S06	
Minimize nesting	P18		Use Intention-Revealing Names	B	
KISS – Keep It Simple, Stupid!	B, P1, S02		Pronounceable Names	B	
OCP – Open Closed Principle	B, P1, S06		Searchable Names	B	
Separate Constructing a System from Using it	B		Avoid Disinformation	B	
			Avoid Mental Mapping	B	
Table of Function and Method principles			**Table of Comment principles**		
Principle	**Literature**	**Evidence of its use in practice**	**Principle**	**Literature**	**Evidence of its use in practice**
Do One Thing	B, P1, S06		Amplification	B	
Command Query Separation	B		Clarification	B	
Extract Try-Catch Block	B		Explain Yourself In Code	B	
Have No Side Effects	B, S06		Explanation of Intent	B	
DRY – Don't Repeat Yourself	B, S02, S07, P9, P11, P15, P18		TODO Comments	B	
Function Arguments	B, S06		Warning of Consequences	B	
Structured Programming	B				
Methods/Functions should be small	B, S06, P1	P1			
Table of Formatting principles			**Table of Object and Data Structure principles**		
Principle	**Literature**	**Evidence of its use in practice**	**Principle**	**Literature**	**Evidence of its use in practice**
Team Coding Standards	B, S03, S08, S09, P18		Data/Object Anti-Symmetry	B	
Horizontal Formatting – Indentation	B, P14		Law of Demeter	B	

Dependent Functions	B				
Vertical Distance and Ordering	B, P14				
Organizing for Change	B				
Table of Error Handling principles			**Table of Unit Test principles**		
Principle	**Literature**	**Evidence of its use in practice**	**Principle**	**Literature**	**Evidence of its use in practice**
Prefer Exceptions to Returning Error Codes	B		Keeping Tests Clean	B	
Don't Pass Null	B		One Assert per Test	B	
Don't Return Null	B		Single Concept per Test	B	
Write Your Try-Catch Statement First	B				
Table of Class principles					
Principle	**Literature**	**Evidence of its use in practice**			
Class Organization	B				
High Cohesion	B, S02, S03, S04, S05, P1, P8, P11, P12, P15, P17	P1			
Low Coupling	B, S02, S03, S04, S05, S06, S08, P1, P8, P9, P11, P15, P17	P1			
Encapsulation	B, P11, P15				
Isolating from Change	B				
SRP – Single Responsibility Principle	B, S02, S06, S07, S08, P1, P3, P17	P1			
Minimal Classes and Methods	B				
One Level of Abstraction per Function	B, P4, P15, P17	P4			
Classes should be small	B, S02, S06, P1	P1			

Table 4.3: Clean code principles and practices found

4.3.1 Static Code Analysis and Quality Gates

Static analysis tools are tools that can help programmers with following code quality standards and rules. It can help programmers adhere to some of the principles in clean code. Human memory is not perfect, and the adherence to writing high-quality code and clean code is sometimes easily violated, whether intentional or not. Some static analysis tools can detect syntax errors, coding mistakes, and security vulnerabilities [3]. Which of these the static analysis tool can do depends on what the developers have implemented. Latte, Henning, and Wojcieszak [3] advise developers to integrate the static analysis tool if they can do so since it will help with syntax highlighting coding violations. This helps the developers avoid violating some coding practices and principles as would probably have been violated otherwise if they had to keep the practices and principles in memory. These tools give the developers feedback directly about the code quality regarding if something needs to be improved. The developers will know straight away.

Even if developers have a static analysis tool, they can still commit unclean code to the master branch that may violate some of the coding guidelines. Therefore there is a need to prevent the developers from knowingly or mistakenly committing such code to a branch. A quality gate can be configured to check for coding violations and alike and prevent developers from committing code that does not adhere to the code quality standard [3]. A quality gate can be used in combination with CI/CD. The suggestion is to use a pipeline script instead of manually configuring the CI/CD interface. In that case, all developers will have the configuration immediately instead of manually changing it.

4.4 Literature Review Conclusion

For RQ1, we did not find any empirical evidence in the literature review, as mentioned the previous section 4.1. One of the reasons in regard to this could be because of only using peer-reviewed content. Therefore, we cannot conclude whether developers believe in clean code. As mentioned, it will have to be answered by the survey instead.

Regarding RQ2, it is somewhat the same that we did not find much empirical evidence about if developers write clean code initially or unclean code initially. Arif and Rana [32] argued that it would be beneficial to write clean code initially in the long-term since it saves time and effort for the developers, however. A proactive- versus reactive refactoring approach was discussed in section 4.2.1. It also discusses a machine learning approach, which can help developers detect if code is becoming smelly and may help somewhat with writing clean code initially. We also investigated the reasons why developers refactor code. We found out that some of the reasons are maintainability, readability, understandability, technical debt, human errors. Finally, developers do not heavily use the refactoring tools, but developers are aware of their existence. The reason being the refactoring tools graphical user interfaces being complex and difficult to understand.

The main findings from the literature review regarding RQ3 are that we did not find many new clean code principles. We mostly found the principles that Robert Martin had already defined in the Clean Code book [1] to help developers write high-quality code. The only principle that we found that is new is the minimize nesting principle [42]. We also looked for clean code principles and whether developers used them in practice, but very few papers reported clean code principles used in practice, but some papers did mention the principles that Martin had mentioned.

We also found static analysis tools that can help avoid coding mistakes and syntax errors. Then developers can detect the errors that they make and avoid them. The last founding mentioned is using a quality gate that can help developers prevent committing unclean code that does not adhere to the code quality standard.

5 SURVEY RESULTS

Most of the questions in the survey consist of a 7 item Likert-scale from strongly disagree to strongly agree, and some questions are open-ended answers. The open-ended answers are analyzed using thematic analysis. We also map questions to an identifier in Appendix A, when summarizing the thematic analysis. In the paper by Börstler et al. [4], they used diverging stacked bar diagrams to analyze the Likert-scale questions, which we will adapt. At first, we will talk about the demographics of the participants.

5.1 Demographics

The demographics are that we have a total of 38 participants that have entirely completed the survey, whereby 35 are male, and 3 are female. We then ask participants about their age, as shown in Figure 5.1.1. We can see that most participants are between 31 and 40 years old. We are also trying to identify if participants may have worked in practice in software engineering for a while.

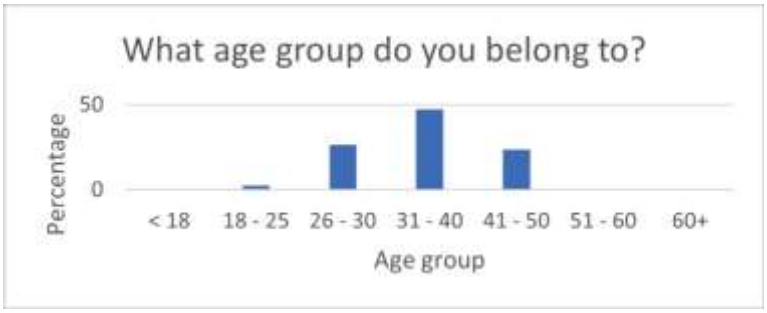


Figure 5.1.1: Age

Since the previous question does not exactly tell us that, we also asked about the years of programming experience that each participant had. As we can see in Figure 5.1.2, most of the respondents have more than 20 years of experience in programming, meaning that most developers have worked with programming for quite some time.

Figure 5.1.2: General programming experience

We also wanted to know the highest education degree that the participants had completed. Most had completed a Bachelor's degree, while most other participants had completed Bachelor's degree and a Master's degree, as seen in Figure 5.1.3. It is essential to mention that in the survey, it said that Bachelor's degree was up to 3 years of university education and that Bachelor's plus Master's degree was up to 5 years of university education.

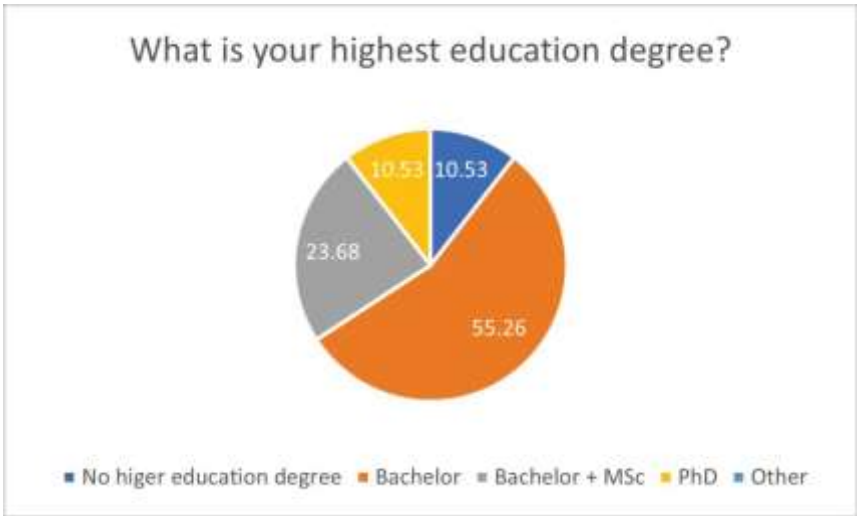
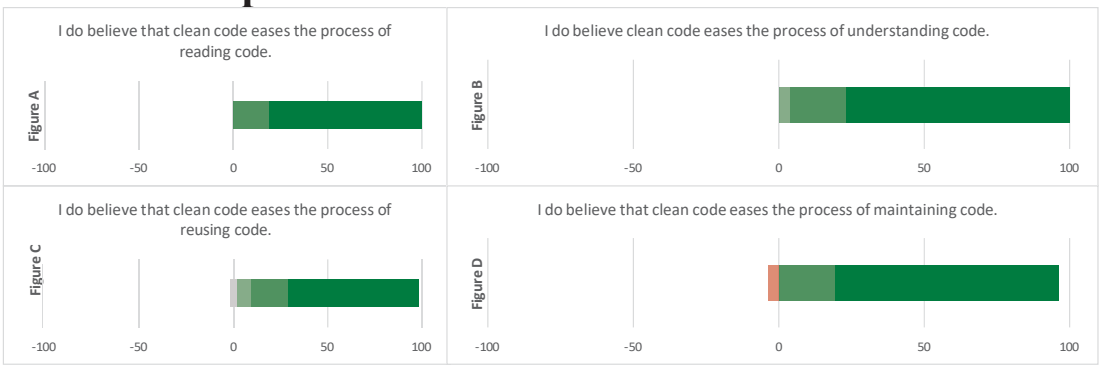


Figure 5.1.3: Highest education degree

5.2 Developers' belief in Clean Code



We begin by investigating whether developers believe in the effect of clean code. As illustrated in Figure A, Figure B, Figure C, and Figure D. As we can see, we have asked developers whether they believe that clean code eases the process of reading, understanding, reusing, and maintaining code. It is pretty clear that developers strongly agree with this. Only very few developers somewhat disagree that clean code eases maintaining the code in Figure D.

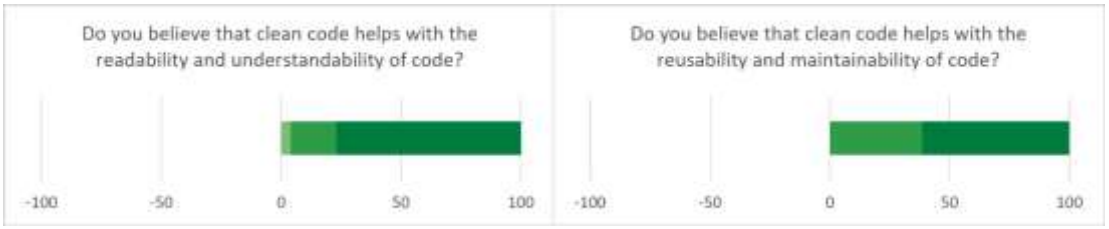


Figure E: Developers believe that clean code helps

We investigated if developers believe that clean code helps with readability, understandability, reusability, and maintainability. As illustrated in Figure E, we see that the result is that developers strongly agree to believe that clean code helps with these issues.

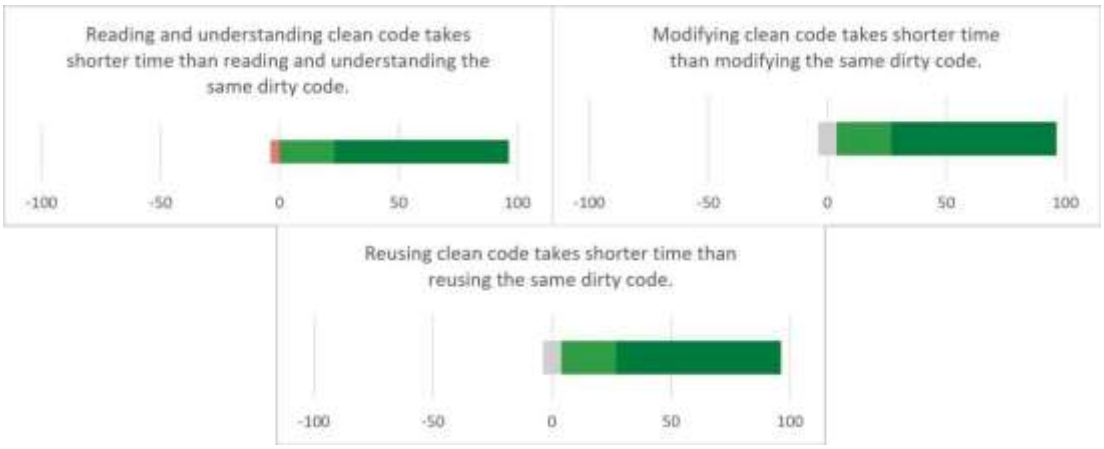


Figure F: Developers think clean code takes shorter time than dirty code

We then asked developers if they think it would take a shorter time to read and understand clean code than with dirty code, which means unclean code. We asked the same question regarding modifying code and reusing code. As illustrated in Figure F, we see that the result is that the developers believe that it would take a shorter time to read, understand, modify, and reuse clean code compared to unclean code.

Figure G: Clean code does not waste time from completing other tasks

In Figure G, we see that the developers strongly disagree with the statement. The statement mentions that writing readable and understandable code wastes time and prevents a developer from being productive. This statement shows that the developers think it is very untrue and that it would not be counterproductive to write clean code, and that it would take time away from completing tasks.

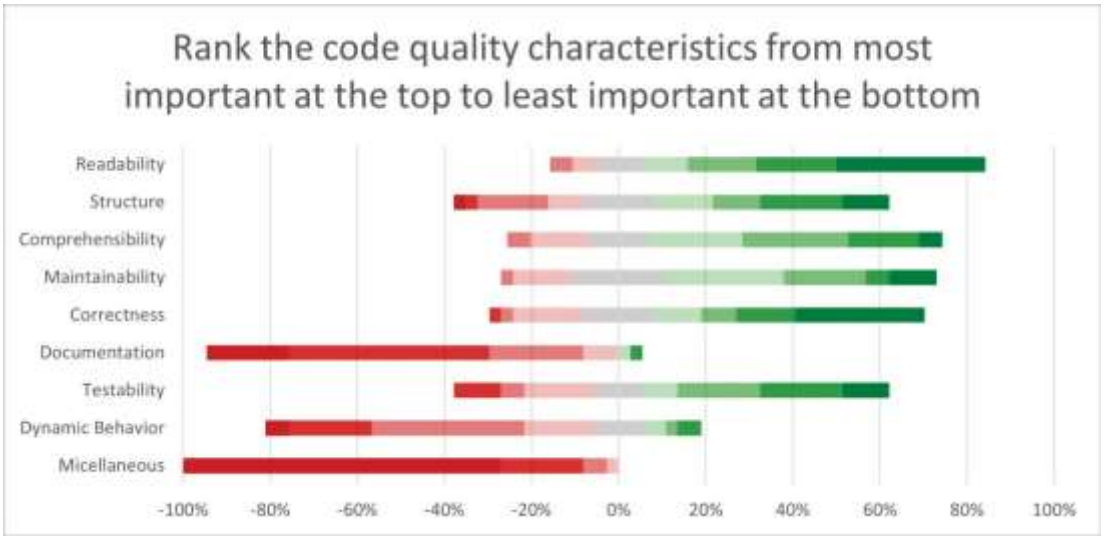


Figure H: Ranked code quality characteristics

As shown in Figure H, we can see a ranking of the code quality characteristics. This question was adapted from Börstler et al. [4], to rank the code quality characteristics from top to bottom using drag n drop ranking. The top three ranked are readability, comprehensibility, and maintainability. The ones in the middle by the most important in the middle first are correctness, structure, and testability. The bottom three are dynamic behavior, documentation, and miscellaneous. We must also mention that we used an explanation from the same paper we adapted the question from so that the code quality characteristics were explained further.

5.2.1 Thematic analysis

Since we asked about readable and understandable code, we also asked developers how they check that the code is readable and understandable (Q12a), as visualized in the green branch in Figure RQ1 Part 2. All figures regarding thematic analysis for RQ1 can be found in Appendix E. The developers answered that checking this is mainly done via code reviews, peer reviews, or pull requests. Some developers mentioned taking a short break from the code they wrote to clean the current state of their minds and then re-read it. After re-reading their code, they will ask another developer if they have time to discuss the code and review it.

Another question that we asked is somewhat related to the previous questions about readability and understandability. That question was why or why not developers believe clean code helps with readability and understandability. The point of view that is interesting is that one respondent mentioned that clean code has a lack of halt criterion and fears overusing it, as visualized in the red graph of Figure RQ1 Part 1. In comparison, other participants go back to the definition of clean code or what clean code is. For the most part, not answering why they think it helps. However, one respondent mentions that clean code should naturally be easier to reason than unclean code since it is easier to read and understand. Developers also mention that developers should create good source code since that will typically be the documentation. Documentation will not exist for all code, and therefore the source code needs to be clean, making it harder to create bugs in the code. We then go on to ask the same question about reusability and maintainability instead (Q12e), as visualized in the yellow graph in Figure RQ1 Part 1. Some developers agree that the code is more likely to be reusable if the code is easy to understand. Therefore, the developers can extend the code more easily if they understand what the code does and be more confident in doing code changes. Developers have also mentioned that they believe that clean code helps with reducing coupling and increasing cohesion. However, one developer argued that, for example, using a library that maintainers clean often could be less reusable since the maintainers clean it up and that an unclean library, in that case, would be more reusable. Developers mention that if the code is easy to read, it will be easy to understand. Therefore, the code will be easier to maintain.

5.3 Clean code initially or unclean code first

Figure A: Most developers refactor unclean code

The first question (Q9a) that we asked developers regarding RQ2 is RQ2 itself, as illustrated in Figure A. It shows that most developers tend to write messy code first and refactor it later to become clean code. In contrast, some of the developers write clean code initially, and some do neither.

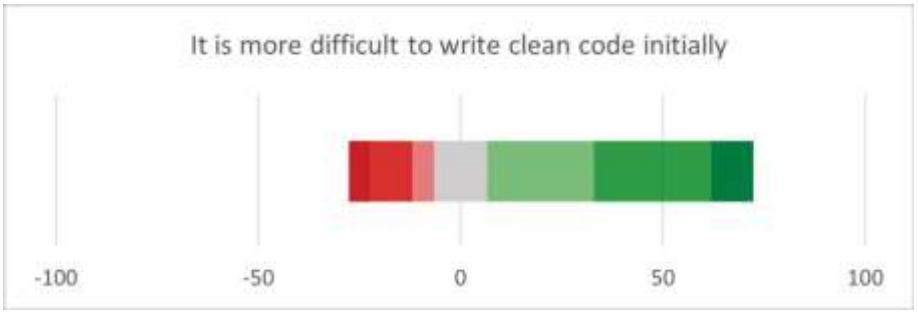


Figure B: Writing clean code initially is more difficult

According to Q9b illustrated in Figure B, developers seem to think differently about whether it is more difficult to write clean code initially or not. The majority of developers agree with the statement, meaning it is more difficult to write clean code initially, while some developers disagree with the statement. Before making a conclusion, we have to look at the open-ended answers since they will initially discuss the challenges with writing clean code. We will come back to this later.

Figure C: Requirements need to be clearly specified to write clean code

Concerning the previous question, we also asked if developers think the requirements need to be clear to write clean code. As illustrated in Figure C, most developers agreed with this, and only a few developers disagreed or had a neutral response. We can say that the requirements, for the most part, have to be specified clearly in order to write clean code for most developers.

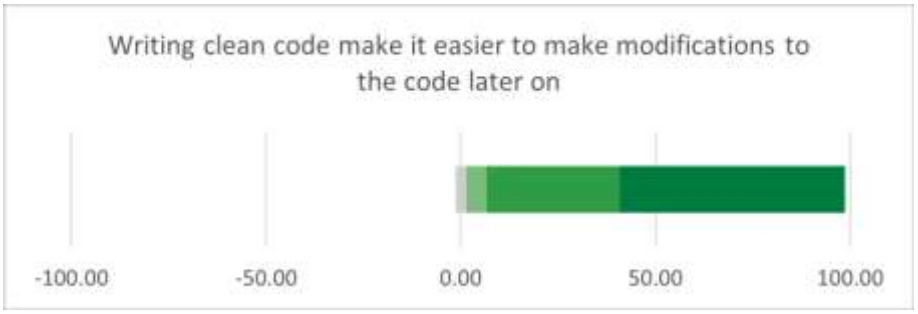


Figure D: Clean code makes it easy to make modifications to the code

We then moved on to ask developers if clean code makes code easy to modify later on. As illustrated in Figure D, the result is that developers think that writing clean code will make it easier to modify the code later on. There is no more to say about the result than that.

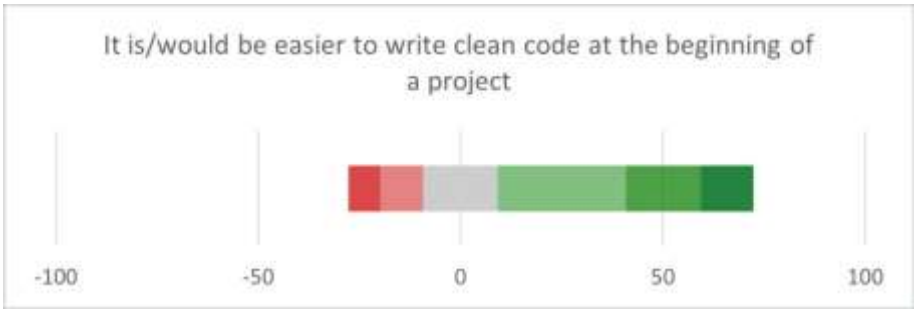


Figure E: It is easier to write clean code at the beginning of a project

Since we also wondered whether it is easier to write clean code at the beginning of a project, we asked developers about it, as illustrated in Figure E. It is not always the case that developers write new code at the beginning of a project, but it can sometimes be that way. Many developers agree with the statement in the question, but there is also a pretty large portion of neutral and disagreeing responses. It is possible to say that it is easier to write clean code at the beginning of a project since most developers agreed with the statement.

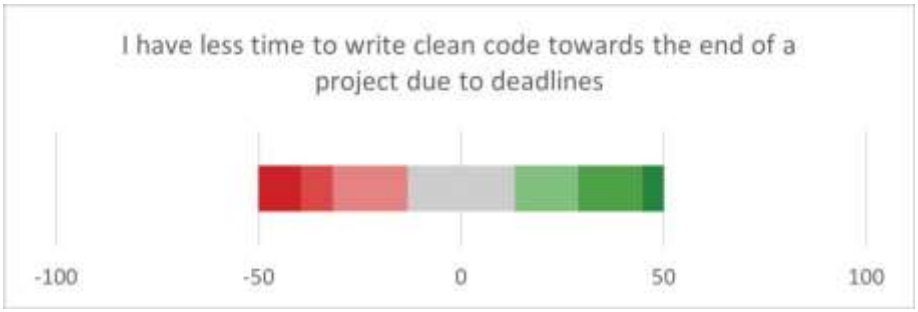
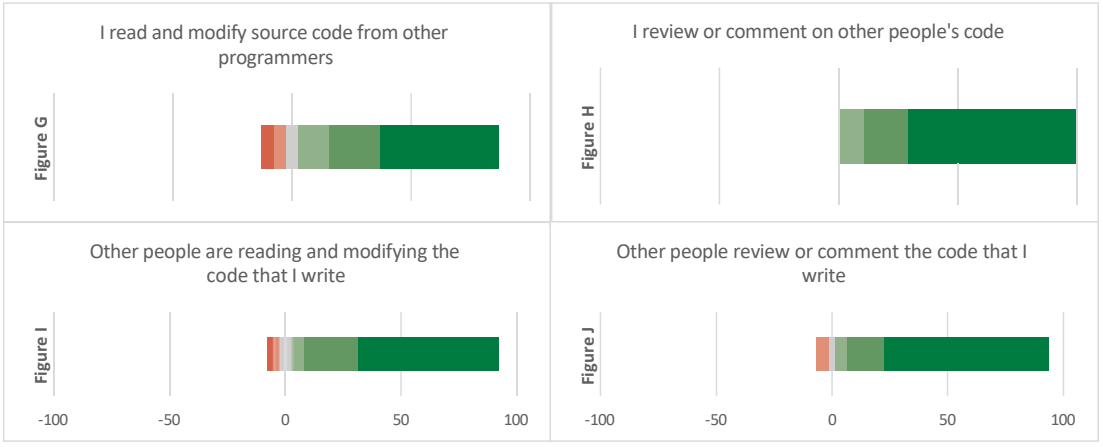


Figure F: Less time to write clean code towards the end

Somewhat related to the previous question, we asked if developers think they have less time to write clean code toward the end of a project due to deadlines, to which about half of the developers both agreed and disagreed. Figure F illustrates these responses. It is worth noting that more developers strongly disagree than developers that strongly agree. We do not think this is enough proof and will therefore leave it inconclusive.



The questions above illustrated in Figure G, Figure H, Figure I, and Figure J are precisely replicated from the paper by Börstler et al. [4]. We also adapted the visualization by using a diverged stacked bar diagram as shown from the paper. The result is quite clear that developers read, modify, and review other people's code and get their code read, modified, and reviewed by other people for the most part.

5.3.1 Thematic analysis

We also asked the respondents to give a short text describing what they think are the main challenges with writing clean code initially (Q9c), as shown in the pink graph in Figure RQ2 Part 1, found in Appendix E. From reading the answers and grouping them, developers seem to say that the one of the challenges is requiring a solid understanding of the problem and identifying the obstacles in advance. Developers do not always know what the code should look like until beginning to write it. According to some of the developers, it is more important for the developers to achieve functional correctness than having the code clean in the first place. However, code might need to change when working with it to find the solution, and in case that happens, the developer might need to restructure his or her code again. Hence, it is more difficult to foresee how to write the code in advance.

We also asked about challenges refactoring unclean code to become clean code (Q9d), as shown in the red graph in Figure RQ2 Part 3. According to participants, the challenge with refactoring unclean code is understanding the unclean code so that no functionality breaks when changing the code, so it still works as it was intended. Also, if there is a lack of tests, it will not be easy to verify that the code behaves the same way. The developers need sufficient amounts of tests to verify that the functional correctness is still intact. The obstacles mentioned are that developers are sometimes inexperienced and that the code size is large. In terms of schedule obstacles, it is related to the time it takes to refactor the code and the deadlines that have a particular impact on whether to refactor or not, shown in the green graph in Figure RQ2 Part 2.

Some developers also added that they do neither write clean code initially nor refactor unclean code, as illustrated in the purple graph of Figure RQ2 Part 1. One developer mentioned that he or she writes simple code first, and in case the code did not become simple enough, he or she would go back and refactor it. Another developer mentioned that it depends on the complexity of the task. In case the task is simple it is easier to write clean code initially, and vice versa.

We also asked developers about the refactoring techniques and operations that they use. According to the developers, some refactoring operations or techniques are: renaming things, apply SOLID principles, fowler refactoring techniques, common sense, mob programming, separate refactoring commits, extracting code blocks, functions, and classes, as illustrated in the orange branch Figure RQ2 Part 3. Some participants mention using inbuilt IDE features to do specific refactorings, while others do not mention any IDE. Presumingly doing the refactoring manually. However, we asked if they use any IDE or tool that helps with refactoring (Q9f). The IDE and tools mentioned for refactoring help were IntelliJ, Eclipse, Visual Studio Code, PyCharm, and ApoCode, as visualized in the blue graph in Figure RQ2 Part 2. Other participants mentioned that they do it manually sometimes or that it depends on the complexity of the refactoring task, whether they use a feature for refactoring from an IDE or tool.

5.4 Prominent Clean Code Principles

The third question, RQ3, is investigating which practices and principles that developers think are most prominent. Prominent in the sense of whether developers agree with a particular clean code principle a lot. We have tried only to pick the main clean code principles since including all principles would have extended the survey a lot and would not have been pleasant for the participants. We have divided the principles into different categories such as general, naming, function and methods, comment, formatting, object and data structure, error handling, unit test, and class. Instead of having one large question in the survey about these principles, it is more pleasant to divide them. There are three different levels of the opacity of green and red. The darker the green color is to the right, the more the developers agree, and the darker the red color, the more the developers disagree. In case the color is gray, then developers do neither agree nor disagree. The negative percentages to the left in the diagrams should be read as typical percentages without any minus sign. The exact percentages for the closed questions are

shown in Appendix C. To scientifically include or exclude a principle, the Wilcoxon test with p-value was used. If $p < 0.05$, then the result is statistically significant, meaning that the answers by developers are significantly higher than the neutral value equals to 4, and therefore we include the principle and otherwise exclude it. The table for all the p values for each principle can be seen in Appendix F.

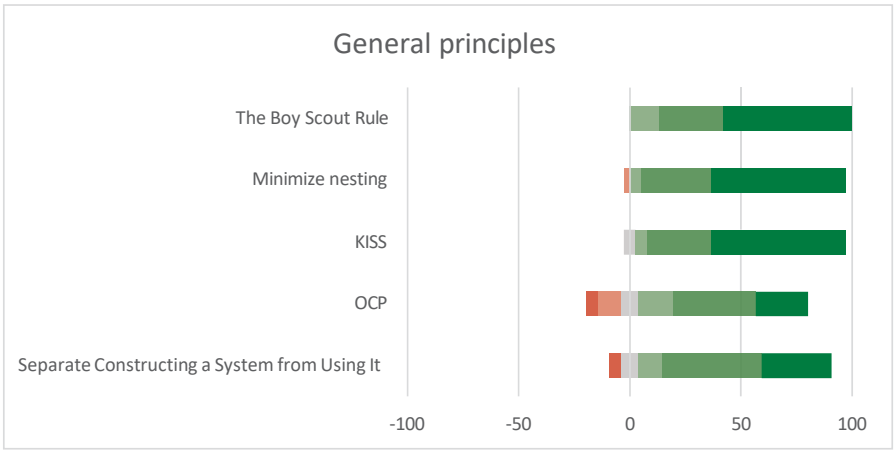


Figure A: General principles

As illustrated in Figure A, many developers agree with The Boy Scout Rule, Minimize nesting, and KISS principles. While many developers agree with the OCP and Separate Constructing a System from Using It, there are also disagreeing or neutral responses. Neutral meaning people that do neither agree nor disagree. According to the Wilcoxon test, $p < 0.05$ for all principles, meaning that the answers by developers are higher than the neutral value equals to 4, and therefore, we include each one of them.

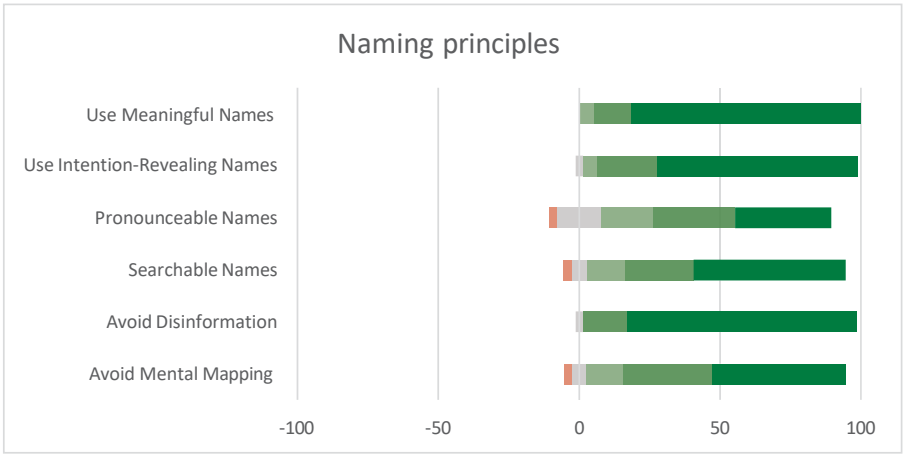


Figure B: Naming principles

Some developers seem to have some neutrality and disagreement towards Pronounceable Names, Avoid Disinformation, and Avoid Mental Mapping. However, all naming principles are statistically significant according to the Wilcoxon p-value, so we will include all of them.

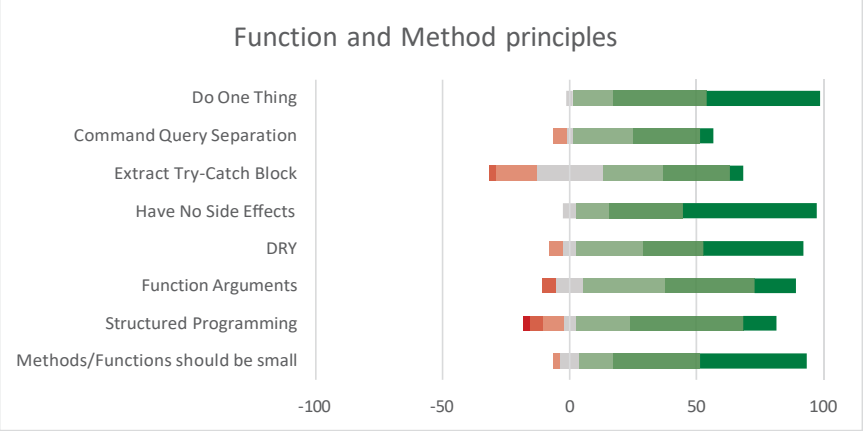


Figure C: Function and Method principles

Regarding the function and method principles illustrated in Figure C, there is more variance. Developers mostly agree with Do One Thing, Command Query Separation, Have No Side Effects, DRY, and Methods/Functions should be small. Only a small portion of developers somewhat disagree or have a neutral response towards Extract Try-Catch Block, Structured Programming, and Function Arguments. The Wilcoxon test states that $p < 0.05$ for all function and method principles, therefore we include all of them.

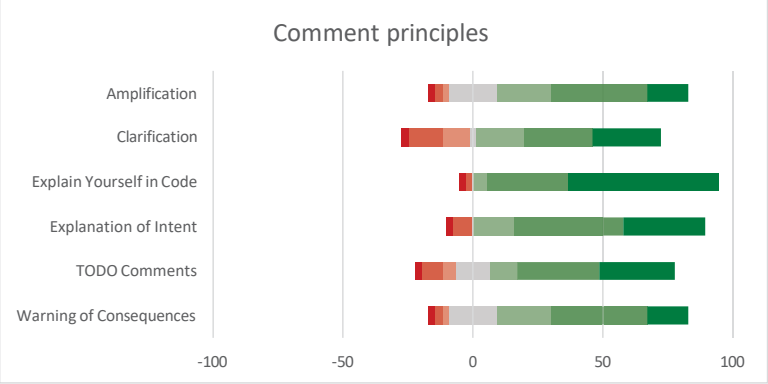


Figure D: Comment principles

As illustrated in Figure D, we can see no principle that is entirely free from disagreement by developers for the comment principles. The developers' responses have some neutrality or disagreement towards each comment principle. The primary response is that developers seem to agree with the principles, though. The scientific test also tells us that $p < 0.05$ for all principles, meaning we include all of them.

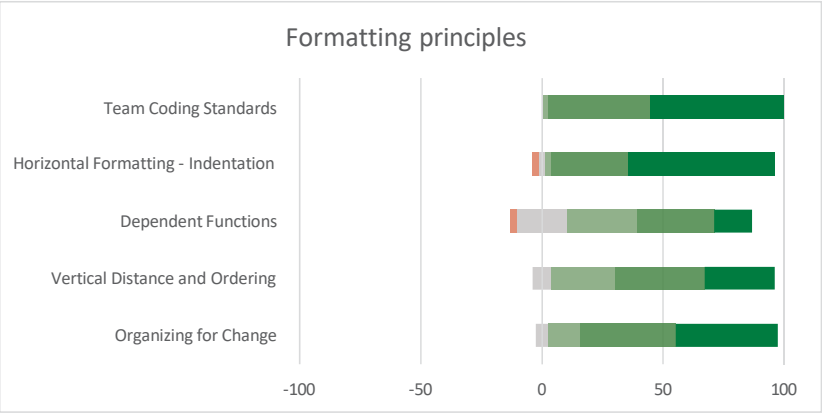


Figure E: Formatting principles

The only formatting principle that developers seem to neither agree nor disagree with is Dependent Functions, as illustrated in Figure E. Most other principles they seem to agree with. However, the Wilcoxon test also states that this is statistically significant and $p < 0.05$ for all formatting principles, so we in include all these principles.

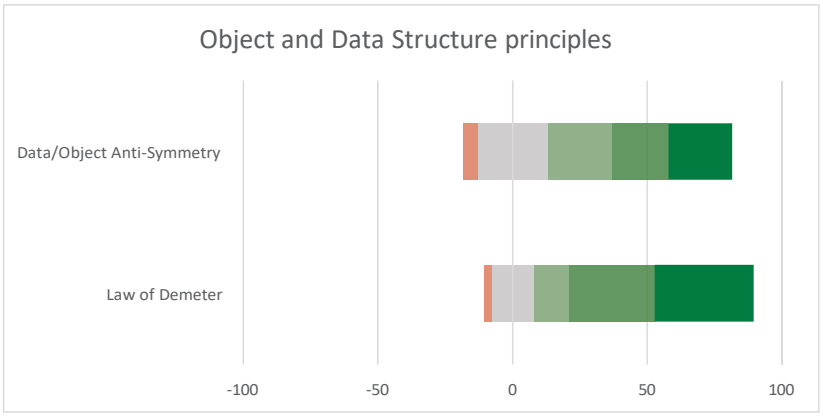


Figure F: Object and Data Structure principles

These two principles showed in Figure F quite many developers agree with. It seems like the developers have some uncertainty about these. As high as about 26% neutral for Data/Object Anti-Symmetry, and about 14% for the Law of Demeter. Due to the p-value still being less than 0.05, we also need to include these principles.

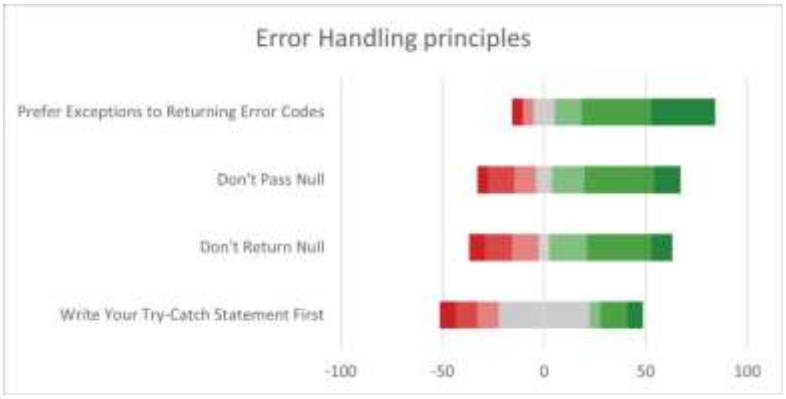


Figure G: Error Handling principles

Moving on to error handling principles as illustrated by Figure G. There is much disagreement with these principles. According to the Wilcoxon test, the p-value for the Write Your Try-Catch Statement First is not statistically significant. Therefore, we exclude that principle while including the remaining principles.

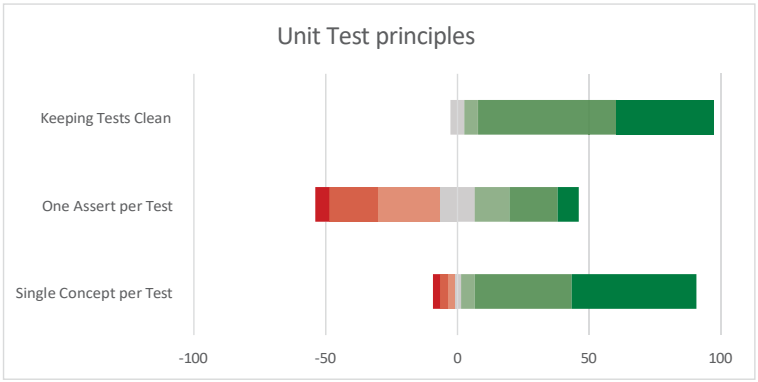


Figure H: Unit Test principles

Now moving on to unit test principles, we see that the only one to exclude is One Assert per Test, aligned with the Wilcoxon test. Single Concept per Test is almost the opposite to One Assert per Test illustrated in Figure H. However, we can see that the developers agree with the principle of Keeping Tests Clean, and single concept per test. The Wilcoxon test shows to include the other two principles.

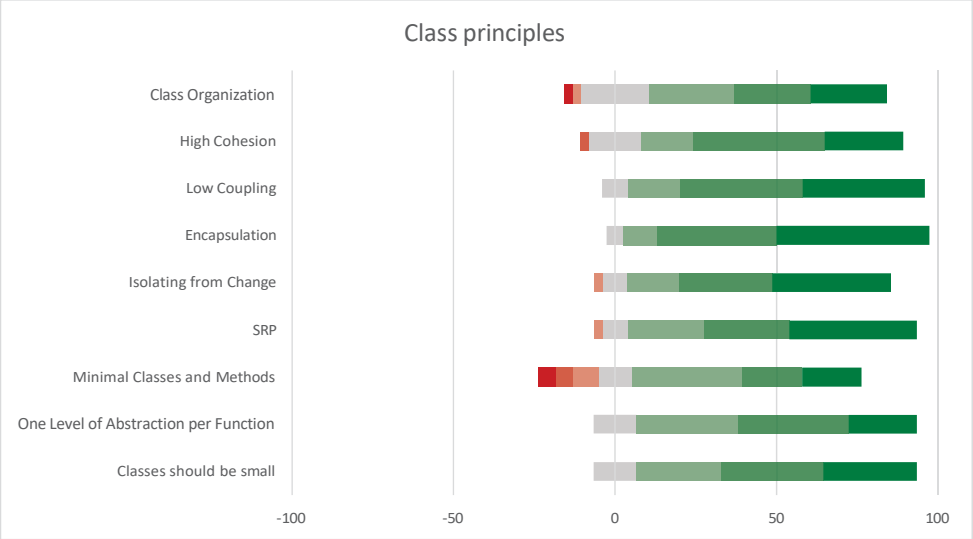


Figure I: Class principles

In Figure I, we are showing class principles. Although there are many neutral and some disagreeing responses to Class Organization, High Cohesion, One Level of Abstraction per Function, and Classes should be small. According to the Wilcoxon test, the p-value is statistically significant for all class principles, which means that we include all of them, and exclude none.

Now when the prominent principles have been sorted out. We move on to the following questions about refactoring, static analysis tools, and quality gates. These tools may help with following the prominent clean code principles or make the code clean again.

Figure J: Refactoring to keep code clean and refactoring tools

We then asked participants about static analysis- and refactoring tools, as shown in Figure J. We asked developers whether they believe that refactoring is a helpful tool to make the code clean and use refactoring as a technique to keep the code clean. Developers in practice think that refactoring is a useful tool that helps keep the code clean and that over 90% of developers use refactoring as a technique to keep the code clean.

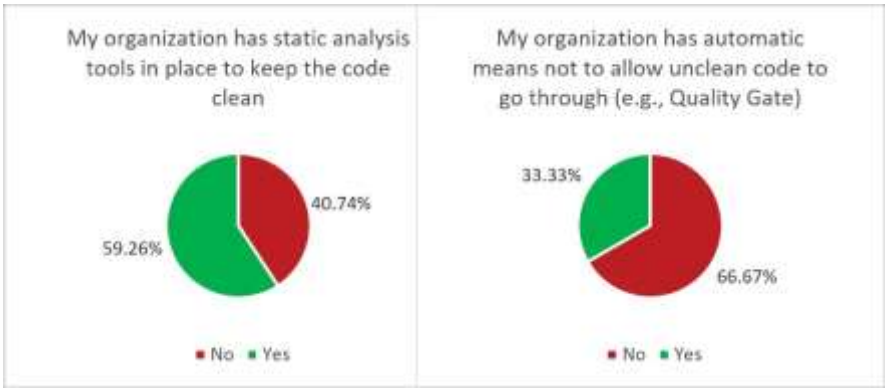


Figure K: Static analysis tools and automatic means to keep code clean

As shown in Figure K. The other questions in this section are if developers' organizations have any static analysis tools that help developers keep the code clean or if the organization has any automatic means such as a quality gate not to allow committing unclean code to the repository. Surprisingly, almost half of the developers replied with both yes and no, meaning that some organizations use static analysis tools and some organizations do not. However, in regards to the question on the right in Figure K, most developers have stated that their organization does not have any automatic means that prevents developers from committing unclean code, such as a quality gate.

5.4.1 Thematic analysis

For the question (Q8a) about principles that will help developers write better quality code, as mentioned in the pink branch of Figure RQ3 Part 1 and Figure RQ3 Part 2, which can be found in Appendix E. Some of the answers we got mentioned KISS, The Boy Scout Rule, SRP, Functions/Methods should be small, etc. Also, to use coding standards and appropriate naming to increase the code's readability and understandability. Refactoring was also mentioned as a technique to do combined with unit tests to check that the external behavior does not change.

One of the other questions was about if developers wrote self-explanatory code and how they did so or if they needed to use comments (Q8b). The developers responded that they try to write self-explanatory code by naming things appropriately. One developer mentioned that self-explanatory code should read like a natural language. In contrast, another developer mentioned that comments on the function, class, and package level might be necessary to explain performance and security issues but still agreed to avoid comments on the code block level. As a follow-up question, we asked what the developers think are the challenges with writing self-explanatory code (Q8c), as seen in the red graph of Figure RQ3 Part 4. The developers responded that they must think about naming things appropriately, which can be time-consuming in some instances, and developers need to meet the deadline. Another hindrance was that the management is preventing it and prioritizing business requirements as more critical to implement in particular cases. Also, the tasks or requirements to implement can sometimes be complex, and a developer needs enough understanding of the problem or problem domain to write self-explanatory code then.

From a previous question, we know that developers use refactoring as a tool to keep the code clean. As a follow-up question, we asked how they used refactoring to keep the code clean (Q8f), as shown in the purple graph in Figure RQ3 Part 3. Developers mention that whenever the code is unclean, too complex, or has other issues, they will refactor it manually or using IDE refactoring features, thus following The Boy Scout Rule. After making the code clean, some developers mention that they verify that the external behavior has not changed by using unit tests. However, some developers do refactoring that is not behavior preserving and introduces changes while refactoring.

We have also asked in a previous question about static analysis tools and automatic means to keep the code clean, as shown in the blue graph in Figure RQ3 Part 4. We asked participants

about the name of these tools that they use. The mentioned static analysis tools by participants are SonarQube, SonarLint, PMD, SpotBugs, ESLint (Q8h). Although most organizations did not have any automatic means not to allow unclean code to go through (Q8i), the organizations that use a quality gate or alike use SonarQube or Jenkins (Q8j).

As the last question for this section, we asked the developers if they found anything missing from a prominent principle or practice (Q8k), illustrated in the yellow graph in Figure RQ3 Part 3. They responded that: code reviews, keeping security in mind, clean commits, they thought were essential to think about also.

6 ANALYSIS

6.1 Demographics

We know that 35 of the participants are male, and 3 are female. We also see from the demographics that most participants are between 31 to 40 years old and have more than 20 years of general programming experience. Therefore, being relevant participants to the survey.

6.2 Developers' belief in Clean Code

From the results, we found out that developers do believe in clean code and believe it can help write more readable, understandable, modifiable, or reusable code. No papers discuss whether developers believe in clean code and whether they apply it in practice or not, but papers such as [2], [3], have reported that teams need to reach a common mindset or culture of clean code.

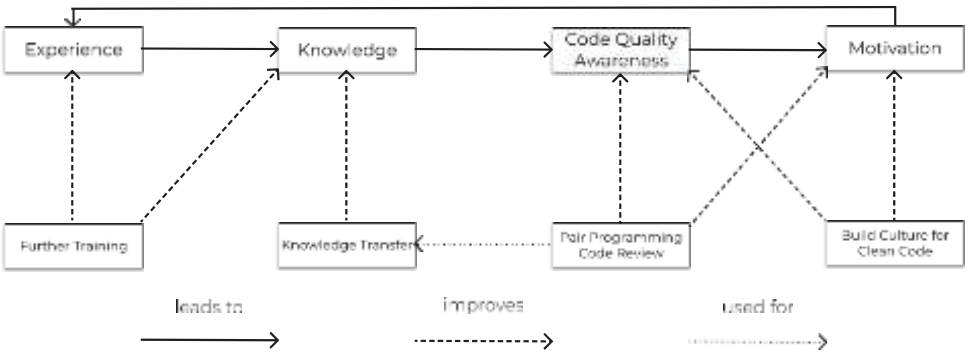


Figure 6.2.1: Showing how we can establish a culture of clean code using the main four people-related aspects (adapted from [2, p. 135])

According to Rachow, Schröder, and Riebisch [2], developers need to be aware of clean code, and this paragraph will talk about what Figure 6.2.1 illustrates. Further training will *improve* the developers' experience, and it will also *improve* the knowledge that the developers have. Knowledge and time can *lead to* the awareness of code quality, which *leads to* the motivation to extend their knowledge about clean code.

In the survey results, developers mention that they use code reviews to check if the code is readable and understandable. A code review is a process of communication between the author and the reviewer [2]. Code reviews can also transfer knowledge between developers to improve their code quality awareness and check that code is readable and understandable. Code reviews are an effective practice that does affect code quality positively.

Rachow, Schröder, and Riebisch [2] mention that pair programming or code reviews are *used for* transferring knowledge about code quality and clean code, as shown in Figure 6.2.1. Pair programming or code reviews are *used for* transferring knowledge about code quality and clean code. In pair programming, the developers get immediate feedback from each other. While in a code review, the author gets feedback from the reviewer about the problems with the code written. This approach has the potential to establish solid trust and openness within the team. Pair programming or code reviews will help *improve* code quality awareness for the developers. Adhering to clean code will require developers to establish a culture for clean code. There are many steps to get to that point, but once code quality awareness and motivation exist within the team, building the culture for clean code should be easier. Some developers have the urge to learn about code quality and clean code, but some developers have a disinterest in extending their knowledge about the topic.

We also asked our participants in the questionnaire survey if they believe that it takes a shorter time to read, understand, modify, or reuse clean code compared to unclean code. The respondents strongly agreed with this. Only some developers disagreed that clean code would take a shorter time to read and understand than unclean code. Arif and Rana [32] mention that

if developers remove code smells in advance and make the code clean, it will take 7% less effort to add new features to the code than with unclean code. Also, Digkas et al. [22] argue that writing clean code contributes to decreasing the technical debt density in the code, and we can decrease it by writing new cleaner code. In this study, Digkas et al. analyzed 27 open-source projects by the Apache Software Foundation. They found that 77% of all revisions had lower technical debt density, so it is reasonable to argue that the technical debt density is decreased by writing clean new code.

6.3 Clean code initially or unclean code first

The results regarding if developers write clean code initially are clear. Most developers do write unclean code initially and refactor it later. Developers responded that they fear breaking the code's functionality. As mentioned in [8], [22] developers are concerned with breaking the functionality, especially if an API is used, then developers would avoid refactoring code. The reason is that developers realized that breaking code that an API is using could affect other client applications using that API [8].

From the results, we also see that developers in practice use refactoring as a technique to keep the code clean. According to Digkas et al. [22], one of the more popular techniques to repay technical debt is refactoring. We can avoid introducing as much technical debt if we do not introduce as many code smells. According to Sae-Lim, Hayashi, and Saeki [36], proactive refactoring is an alternative to reactive refactoring. Proactive refactoring can help developers avoid introduce code smells since developers then can detect if a module is becoming smelly. This proactive refactoring technique could potentially contribute to more developers writing clean code initially instead of refactoring unclean code. They will already know that a module is toward becoming smelly.

6.4 Prominent Clean Code Principles

In the results from the first research question, we investigate the most prominent clean code principles. We see that developers agree with most of these principles. Only a few principles developers disagree against. Lucena and Tizzei [26] have extended Scrum with practices and principles from Agile Modeling, DevOps, and Software Craftsmanship. Software Craftsmanship includes clean code practices and principles. They mention that following Martin’s clean code principles is a practical way of showing how to adhere to the quality code principles of Software Craftsmanship [26].

As shown in our results, we also found that developers agreed that the requirements must be specified clearly to write clean code. Only a few developers disagreed with the previous statement. Therefore, the findings from both the literature and the results seem to be aligned. Lucena and Tizzei [26] also mention that integrating all of the previously mentioned methodologies can help the development team write down the requirements more precisely. It also showed that the development had a more sustainable velocity and could deliver a more valuable project to the customer when applying these practices.

A participant added that developers need to have clean commits when asking if they felt like any practice or principle was missing. Digkas et al. [22] also mention that the average commits were cleaner if providing code quality guidelines or recurring board meetings talking about code quality.

From the survey result, we know that most organizations do not have automatic means such as a quality gate to prevent committing unclean code, unfortunately. On the other hand, most organizations do have a static analysis tool that they do use to help developers adhere to coding guidelines, which is aligned with the suggestion in [3]. Doing so can sometimes be forgotten, and then we can have static code analyzers, quality gates, and continuous integration systems that will help us with that. It also showed that writing new clean code can help reduce TDIs and be more efficient and cost-effective [22].

7 CONCLUSION

This thesis investigates what developers in practice think about clean code. The first research question is about if developers believe in clean code. Regarding if clean code can help with readability, understandability, modifiability, and maintainability. We found out that developers do believe in the effect of clean code in practice and that they quite strongly seem to believe in it. We also asked developers how they check that the code is readable, to which they responded that they use code reviews, peer reviews, or pull requests. Developers also mentioned that they take a short break from the current code and then read it later to have a clear state of mind. Therefore, it should not be a problem to establish a common mindset or culture for clean code if most developers believe in it, meaning that most developers would follow the clean code paradigm.

The second research question is whether developers in practice write clean code initially or prefer to write unclean code first and then refactor it to become clean code. We found that most developers do usually not write clean code initially because it would require them to find the solution and obstacles in advance. Also, developers mention that they do not always know how the code should look like before beginning writing it, making it difficult to write clean code initially. Some developers do write clean code initially, but these are fewer than those that write unclean code. Some developers do both depending on the complexity and difficulty level of the task. If the task is simple, it is easier to write clean code initially than if the task was complex. Some developers also do neither of these three.

The last research question investigates the most prominent clean code principles that developers think are prominent and which principles we should discard, if any. We found out that most developers see most clean code principles as prominent and discarding only a few of them. Also, the developers added a practice about code review as being essential to check the readability and understandability of the code. Further also explaining that developers should use coding standards and appropriate naming to increase the code's readability and understandability in the open-ended answers. We also asked developers if they write self-explanatory code instead of using comments, to which most developers responded that they do. However, in some instances, like for security and performance issues, a comment might be needed, but developers agreed to use self-explanatory code instead of comments on the code block level.

8 VALIDITY THREATS

Here we will describe the threats to the validity of the literature review and the survey. We are also describing how we mitigated some threats to validity partially or entirely if any.

Internal validity. A threat to the internal validity is that we only considered some practices and principles, which means that we may have missed principles and practices that the developers would otherwise have found prominent. We mitigated this threat by: 1) doing an SLR to identify principles and practices, and 2) including an optional question asking whether we missed any practice or principle that the developer considers prominent. This fix only partially mitigates this risk. Another threat is that no respondents disagreed with some of the questions, such as The Boy Scout Rule: a developer should leave the code cleaner than they found it [1]. Developers know that this is something they should do, so they might have agreed with it because of that, and not because that they follow The Boy Scout Rule. The same goes for another question which was about if they use refactoring to clean up the code.

External validity is about whether it is possible to generalize the results. The survey was shared using social networks and internally within some companies. Unfortunately, we only have 38 respondents who entirely completed the survey, which means that the results cannot be generalized. However, 30 of the respondents are from different companies and have worked within the software engineering field, having more than three years of experience as developers. Therefore, the results are still relevant. Another threat is that the completion rate of the survey is only 6.11% which is relatively low.

Construct validity is concerned with the relationship between the theoretical and empirical parts. Another threat to validity is that a respondent did not understand the clean code principle that we showed. When showing the clean code principles, we display its name and then a short description of it underneath its name. We also had a link that participants could click on to get more info about a specific principle if needed if a participant did not understand the principle. This risk was further mitigated by sending the survey to a beta tester with experience within surveys and then correcting the survey according to the beta tester's feedback and using the feedback to fix questions that were difficult to interpret or had other issues. Hopefully, this helped mitigate the risk of a participant not understanding a principle. However, there is always a risk that a participant read the explanation too fast and interpreted a principle differently.

It is also that participants might be reluctant to give answers that do not represent professional or ethical attitudes. For example, if participants do not care about some questions, they might not want to answer that they do not care since this does not represent professionalism even if they think it does not matter.

It is also not certain that the participants are native English speakers, so it can generate unclarities in the responses, making it more difficult to interpret the answer to the question.

Reliability validity was improved by assessing the seed and iterations in the literature review. The researchers did this separately in parallel and then discussed which clean code principles should be included after reaching a consensus.

9 FUTURE WORK

From the analysis of the second research question. We learned that developers either write clean code initially, refactor unclean code, do both depending on the context, or do neither. We do not know anything about how the developers achieve to write clean code initially. We only asked about the challenges with writing clean code initially. A possible area of further research might focus on expanding upon what developers do to write clean code initially to achieve this. We could also investigate when developers find it suitable to write clean code initially and when they find it more suitable to write unclean code and refactor it to become clean code later — investigating the trade-off between which way works best scenarios.

Another alternative for future research regarding the second research question is to use a machine learning approach that was suggested in the paper written by Sae-Lim, Hayashi, and Saeki [36]. This approach was suggested to identify if a module is becoming smelly and prevent it in advance more efficiently using a machine-learning algorithm. If developers can use a machine-learning algorithm to prevent code smells, this may help to decrease the technical debt introduced to the code. A machine-learning algorithm might predict what will happen to the code if it has enough data collected to analyze. Therefore, it may help developers write clean code initially without needing to refactor unclean code as much if that is the case.

Regarding the last research question, the future questions that can be studied as follow-ups to this thesis are how developers would use the prominent practices and principles in practice, gaining a better understanding of why these practices and principles are prominent to developers' in practice. Understanding how the developers use these principles and practices is the next step to see how it can help, preferably interviewing developers about these principles and practices or conducting an experiment on how developers use these principles and practices.

Another option is that we can continue investigating the clean code principles using an experiment as future research. Code snippets can be given to participants to rate which code snippets are cleaner. To investigate whether participants know what clean code and unclean code are and see their differences. We can also ask them to clean up unclean code to identify the strategy that they use. First, we identify the strategy they use to clean up the code without introducing the clean code principles and then analyzing how they fixed it. Then we do the same thing again but introduce the clean code principles this time to compare if it has a positive effect on cleaning up the code.

We could also investigate how the code gets cleaner. A possible approach to do this would be to use either case studies or static analyzers to assess the code quality. We will need developers that have experience in development to continue with this investigation. Using static analyzers, we can probably check if the commits are becoming cleaner and automate the process and avoid researcher bias.

10 REFERENCES

[1] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Stoughton, MA, USA: Pearson Education, 2009.

[2] P. Rachow, S. Schröder and M. Riebisch, "Missing Clean Code Acceptance and Support in Practice - An Empirical Study," in *2018 25th Australasian Software Engineering Conf. (ASWEC)*, Adelaide, SA, Australia, 2018.

[3] B. Latte, S. Henning and M. Wojcieszak, "Clean Code: On the Use of Practices and Tools to Produce Maintainable Code for Long-Living Software," in *EMLS 2019: 6th Collaborative Workshop on Evolution and Maintenance of Long-Living Systems*, Stuttgart, 2019.

[4] J. Börstler, H. Störrle, D. Toll, J. Assema, R. Duran, S. Hooshangi, J. Jeuring, H. Keuning, C. Kleiner and B. MacKellar, "“I know it when I see it” – Perceptions of Code Quality," in *Proc. 2017 ITiCSE Conf. Working Group Reports*, Bologna, Italy, 2017.

[5] P. J, Lanza, M and B. G, "Improving Code: The (Mis)perception of Quality Metrics," in *2018 IEEE International Conf. Software Maintenance and Evolution (ICSME)*, Madrid, Spain, 2018.

[6] W. H. Brown, R. C. Malveau, H. W. McComick and T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, Wiley, 1998.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[8] A. Yamashita and L. Moonen, "Do developers care about code smells? An exploratory survey," in *2013 20th Working Conf. Reverse Engineering (WCRE)*, Koblenz, Germany, 2013.

[9] S. Tushar, S. Girish and S. Ganesh, "Challenges to and Solutions for Refactoring Adoption: An Industrial Perspective," *IEEE Software*, vol. 32, no. 6, pp. 44-51, 2015.

[10] P. Avgeriou, P. Kruchten, I. Ozkaya and C. Seaman, "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)," *Dagstuhl Reports*, vol. 6, no. 4, Oct., pp. 110-138, 2016.

[11] P. Kruchten, R. Nord and I. Ozkaya, Managing Technical Debt: Reducing Friction in Software Development, Pittsburgh, Pennsylvania, US: Pearson, 2019.

[12] M. Kim, T. Zimmerman and N. Nagappan, "A Field Study of Refactoring Challenges and Benefits," in *FSE '12: Proc. ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, Cary, North Carolina, US, 2012.

[13] E. Zabardast, J. Gonzalez-Huerta and D. Šmite, "Refactoring, Bug Fixing, and New Development Effect on Technical Debt: An Industrial Case Study," in *2020 46th Euromicro Conf. Software Engineering and Advanced Applications (SEAA)*, Portoroz, Slovenia, 2020.

[14] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *EASE '14: Proc. 18th International Conf. Evaluation and Assessment in Software Engineering*, 2014.

[15] L. Cohen, L. Manion and K. Morrison, Research Methods in Education, 7th ed. Reading, NY: Routledge, 2011. [E-book] Available: ProQuest Ebook Central.

[16] A. Bhatia and C. Yu-Wei, Machine Learning with R Cookbook, Birmingham, UK: Packt Publishing, 2017.

[17] V. Braun and V. Clarke, "What can “thematic analysis” offer health and wellbeing researchers?," *International Journal of Qualitative Studies on Health and Well-being* , vol. 9, no. 1, pp. 1-2, 2014.

[18] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology* , vol. 3, no. 2, pp. 77-101, 2008.

[19] A. Sundelin, J. G. Huerta, K. Wnuk and T. Gorschek, "Towards an Anatomy of Software Craftsmanship," *Transactions on Software Engineering Methodology*, accepted, to appear, 2021.

[20] M. Ivarsson and T. Gorschek, "A method for evaluating rigor and industrial relevance of technology evaluations," *Empirical Software Engineering*, vol. 16, pp. 365-395, 6 Oct 2011.

[21] P. Lerthathairat and N. Prompoon, "An Approach for Source Code Classification to Enhance Maintainability," in *2011 Eighth International Joint Conf. Computer Science and Software Engineering (JCSSE)*, Nakhonpathom, Thailand, 2011.

[22] G. Digkas, A. Chatzigeorgiou, A. Ampatzoglou and P. Avgeriou, "Can Clean New Code reduce Technical Debt Density?," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, accepted, to appear, 2020.

[23] P. Lerthathairat and N. Prompoon, "An Approach for Source Code Classification Using Software Metrics and Fuzzy Logic to Improve Code Quality with Refactoring Techniques," in *International Conf. Software Engineering and Computer Systems*, 2011.

[24] E. Ammerlaan, W. Veninga and A. Zaidman, "Old Habits Die Hard: Why Refactoring for," in *2015 IEEE 22nd International Conf. Software Analysis, Evolution, and Reengineering (SANER)*, Montreal, QC, Canada, 2015.

[25] C. Dibble and P. Gestwicki, "REFACTORING CODE TO INCREASE READABILITY AND MAINTAINABILITY: A CASE STUDY," *Journal of Computing Sciences in Colleges*, vol. 30, no. 1, pp. 41-51, 2014.

[26] P. Lucena and L. P. Tizzei, "Applying Software Craftsmanship Practices to a Scrum Project: an Experience Report," in *Proc. 2016 Workshop on Social, Human and Economics Aspects of Software*, 2016.

[27] J. Stevenson and M. Wood, "Recognising object-oriented software design quality: a practitioner-based questionnaire survey," *Software Quality Journal*, vol. 26, pp. 321-365, 2017.

[28] D. Steidl, F. Deissenboeck, M. Poehlmann, R. Heinke and B. Uhink-Mergenthaler, "Continuous Software Quality Control in Practice," in *2014 IEEE International Conf. Software Maintenance and Evolution*, Victoria, BC, Canada, 2014.

[29] S. Ajami, Y. Woodbridge and D. G. Feitelson, "Syntax, predicates, idioms — what really affects code," *Empirical Software Engineering*, vol. 24, pp. 287-328, 2018.

[30] E. Avidan and D. G. Feitelson, "Effects of Variable Names on Comprehension: An Empirical Study," in *2017 IEEE 25th International Conference on Program Comprehension (ICPC)*, Buenos Aires, Argentina, 2017.

[31] T. Lee, J.-B. Lee and H. P. IN, "Effect Analysis of Coding Convention Violations on Readability of Post-Delivered Code," *IEICE TRANS. INF. & SYST*, Vols. E98-D, no. 7, pp. 1286-1296, 2015.

[32] A. Arif and Z. Rana, "Refactoring of Code to Remove Technical Debt and Reduce Maintenance Effort," in *2020 14th International Conf. Open Source Systems and Technologies (ICOSST)*, Lahore, Pakistan, 2020.

[33] A. Almogahed, M. Omar and N. Zakaria, "Categorization Refactoring Techniques based on their Effect on Software Quality Attributes," *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, vol. 8, no. 8S, pp. 439-445, 2019.

[34] A. Almogahed, M. Omar and N. Zakaria, "Impact of Software Refactoring on Software Quality in the Industrial Environment: A Review of Empirical Studies," in *Knowledge Management International Conf. (KMICe) 2018,*, Miri Sarawak, Malaysia, 2018.

[35] J. A. Dallal and A. Abdin, "Empirical Evaluation of the Impact of Object-Oriented Code Refactoring on Quality Attributes: A Systematic Literature Review," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 44, no. 1, pp. 44-69, 2018.

[36] N. Sae-Lim, S. Hayashi and M. Saeki, "Toward Proactive Refactoring: An Exploratory Study on Decaying Modules," in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, Montreal, QC, Canada, 2019.

[37] M. Hansen, R. Goldstone and A. Lumsdaine, "What Makes Code Hard to Understand?," *https://arxiv.org/abs/1304.5257*, 2013.

[38] P. Jevgenija, Z. Fiorella, S. Simone, P. Valentina and O. Rocco, "Why Developers Refactor Source Code: A Mining-based Study," *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 4, pp. 1-30, 2020.

[39] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey and R. E. Johnson, "Use, Disuse, and Misuse of Automated Refactorings," in *2012 34th International Conf. Software Engineering (ICSE)*, Zurich, Switzerland, 2012.

[40] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni and M. Kessentini, "Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox," in *2021 IEEE/ACM 43rd International Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Madrid, Spain, 2021.

[41] T. Sedano, "Code Readability Testing, an Empirical Study," in *2016 IEEE 29th International Conf. Software Engineering Education and Training*, Dallas, TX, USA, 2016.

[42] J. Johnson, S. Lubo, N. Yedla, J. Aponte and B. Sharif, "An Empirical Study Assessing Source Code Readability in Comprehension," in *2019 IEEE International Conf. Software Maintenance and Evolution (ICSME)*, Cleveland, OH, USA, 2019.

[43] E. Avidan and D. Feitelson, "Effects of variable names on comprehension an empirical study," in *Proc. 25th International Conf. Program Comprehension*, Buenos Aires, Argentina, 2017.

11 APPENDIXES

11.1 Appendix A

Survey with the questions.

Demographics

ID	Question text	Answer format
Q1	To which gender identity you most identify?	M / F / Other
Q2	What age group do you belong to?	List of number groups
Q3	What is your highest education degree?	List of education degrees
Q4	How many years of programming experience do you have?	List of number groups
Q5	I am comfortable programming in [language]	7-item Likert-scale matrix
Q6	What area do you work within?	Short text (Optional)

RQ1

ID	Question text	Answer format
Q7a	This survey is about Clean Code. When you write your answers about Clean Code: What programming language do you have in mind?	List of programming languages + other programming language
Q7b	I have heard of some of the essential Clean Code practices and principles before.	7-item Likert-scale
Q7c	General principles Do you agree with the clean-code [principle]	7-item Likert-scale
Q7d	Naming principles Do you agree with the clean-code [principle]	7-item Likert-scale
Q7e	Function and Method principles Do you agree with the clean-code [principle]	7-item Likert-scale
Q7f	Comment principles Do you agree with the clean-code [principle]	7-item Likert-scale
Q7g	Formatting principles Do you agree with the clean-code [principle]	7-item Likert-scale
Q7h	Object and Data Structure principles Do you agree with the clean-code [principle]	7-item Likert-scale
Q7i	Error Handling principles Do you agree with the clean-code [principle]	7-item Likert-scale
Q7j	Unit Test principles Do you agree with the clean-code [principle]	7-item Likert-scale
Q7k	Class principles Do you agree with the clean-code [principle]	7-item Likert-scale

RQ1a

ID	Question text	Answer format
Q8a	What are the essential clean code practices and principles you think will help developers write better quality code, and why?	Short text
Q8b	What do you do to write self-explanatory code? Or do you need to use comments to explain the code?	Short text
Q8c	What are the challenges or hindrances with writing self-explanatory code, if any?	Short text
Q8d	I think refactoring is a useful tool to make code clean.	7-item Likert-scale
Q8e	Do you use refactoring as a technique to keep the code clean?	Yes / No

Q8f	How do you use refactoring as a technique to keep the code clean?	Short text (Optional)
Q8g	My organization has static analysis tools in place to keep the code clean	Yes / No
Q8h	Describe your organization's static analysis tool or tools in place to keep the code clean	Short text (Optional)
Q8i	My organization has automatic means not to allow unclean code to go through (e.g., Quality Gate).	Yes / No
Q8j	Which is the automatic tool within the organization used to prevent developers from committing unclean code?	Short text (Optional)
Q8k	Any practice or principle that you see as essential for keeping the code clean, and that has not been mentioned?	Short text (Optional)

RQ2

ID	Question text	Answer format
Q9a	Do you write clean code initially or write "messy" code that you refactor later?	Initially, Refactor it, None of these
Q9b	It is more difficult to write clean code initially	7-item Likert-scale
Q9c	What do you think are/would be the challenges with writing clean code initially?	Short text
Q9d	What do you think are/would be the challenges with refactoring unclean code to become clean code?	Short text
Q9e	What operations or techniques do you usually use to refactor code when needed?	Short text
Q9f	Do you use any IDE/tool that helps with refactoring?	Short text
Q9g	I do believe refactoring has a positive effect on the quality of code.	7-item Likert-scale
Q9h	To write clean code initially, the requirements have to be clearly specified	7-item Likert-scale
Q9i	It is/would be easier to write clean code at the beginning of a project	7-item Likert-scale
Q9j	Writing clean code make it easier to make modifications to the code later on	7-item Likert-scale
Q9k	I have less time to write clean code towards the end of a project due to deadlines	7-item Likert-scale

RQ2a

The questions in the table below are replicated exactly from the reference [4]

ID	Question text	Answer format
Q10a	I read and modify source code from other programmers	7-item Likert-scale
Q10b	I review or comment on other people's code	7-item Likert-scale
Q10c	Other people are reading and modifying the code that I write	7-item Likert-scale
Q10d	Other people review or comment the code that I write	7-item Likert-scale

RQ3

The first question in the table below are adapted from the reference [4]

ID	Question text	Answer format
Q11a	Rank the code quality characteristics from most important at the top to least important at the bottom	Drag and Drop Ranking
Q11b	I do believe clean code eases the process of reading code.	7-item Likert-scale
Q11c	I do believe clean code eases the process of understanding code.	7-item Likert-scale
Q11d	I do believe clean code eases the process of reusing code.	7-item Likert-scale

Q11e	I do believe clean code eases the process of maintaining code.	7-item Likert-scale
Q11f	Writing readable and understandable code is wasting time, and prevents you from being productive and completing tasks.	7-item Likert-scale

RQ3a

ID	Question text	Answer format
Q12a	How do you check that your code is readable and understandable by others?	Short text
Q12b	Do you believe that clean code helps with the readability and understandability of code?	7-item Likert-scale
Q12c	Why or why not do you believe clean code helps with readability and understandability?	Short text
Q12d	Do you believe that clean code helps with the reusability and maintainability of code?	7-item Likert-scale
Q12e	Why or why not do you believe clean code helps with reusability and maintainability?	Short text
Q12f	Reading and understanding clean code takes shorter time than reading and understanding the same dirty code.	7-item Likert-scale
Q12g	Reusing clean code takes shorter time than reusing the same dirty code.	7-item Likert-scale
Q12h	Modifying clean code takes shorter time than modifying the same dirty code.	7-item Likert-scale

11.2 Appendix B

Most of the explanations for the short description we reference the Clean Code book [1] denoted as B in the table. The exception is the minimizing nesting principle which we reference by using the paper from Johnson et al. [42].

Table of General principles				**Table of Naming principles**			
Principle	**Short description (2 lines)**	**Literature**	**Evidence of its use in practice**	**Principle**	**Short description (2 lines)**	**Literature**	**Evidence of its use in practice**
Boy scout rule	Leave the code cleaner than it was when you arrived	B, P2, S05		Use Meaningful Names	Names should mean something to the developers	B, S01, S02, S04, S06	
Minimize nesting	Try not to use nested code blocks	P18		Use Intention-Revealing Names	Reveal the intent of what it does or how it is used	B	
KISS – Keep It Simple, Stupid!	Keep the code simple, avoid complexity	B, P1, S02		Pronounceable Names	Easy to say orally	B	
OCP – Open Closed Principle	Function, class, module, etc. open for extension but closed for modification	B, P1, S06		Searchable Names	Find the name when searching for it	B	
Separate Constructing a System from Using it	The logic for creating objects, and logic for using objects, is separated	B		Avoid Disinformation	False names, obscuring the logic	B	
				Avoid Mental Mapping	Do not use another name for concepts developers already know	B	
Table of Function and Method principles				**Table of Comment principles**			
Principle	**Short description (2 lines)**	**Literature**	**Evidence of its use in practice**	**Principle**	**Short description (2 lines)**	**Literature**	**Evidence of its use in practice**

Do One Thing	Do one thing, and not multiple things	B, P1, S06		Amplification	Explain why a specific change is significant	B	
Command Query Separation	Do something or answer something, but not both [7]	B		Clarification	If it is unclear what the code does, then use a comment to explain what it does	B	
Extract Try-Catch Block	Put the try-catch block in a function of its own	B		Explain Yourself In Code	Try to explain in code without using comments if possible	B	
Have No Side Effects	Do one thing, and not multiple other things, also	B, S06		Explanation of Intent	When the intent is not self-explained by the code, write a comment	B	
DRY – Don't Repeat Yourself	No duplication or alike (e.g., almost identical blocks of code)	B, S02, S07, P9, P11, P15, P18		TODO Comments	Leave TODO comments when you think something should be done, but you cannot implement it at the moment	B	
Function Arguments	Only 1 – 3 arguments passed to a function	B, S06		Warning of Consequences	Warn other developers about running the code for some reason	B	
Structured Programming	Large functions have one entry, one exit. Avoid break, continue, and goto	B					
Methods/Functions should be small	The size/length of a function or method should be small	B, S06, P1	P1				
Table of Formatting principles				**Table of Object and Data Structure principles**			

Principle	Short description (2 lines)	Literature	Evidence of its use in practice	Principle	Short description (2 lines)	Literature	Evidence of its use in practice
Team Coding Standards	Reach a consensus about the coding standard	B, S03, S08, S09, P18		Data/Object Anti-Symmetry	Objects should hide implementations, but data structures should expose their data	B	
Horizontal Formatting – Indentation	Use indentation (e.g., tabs or spaces)	B, P14		Law of Demeter	Do not invoke more than one method upon a method that returns an object. If a method returns an object [7], then do not call methods on that object if it is possible to do so.	B	
Dependent Functions	"The caller should be above the callee, if at all possible." [7]	B					
Vertical Distance and Ordering	Blank lines, close related lines, order of lines	B, P14					
Organizing for Change	Classes should not be sensitive to code changes	B					
Table of Error Handling principles				Table of Unit Test principles			

Principle	Short description (2 lines)	Literature	Evidence of its use in practice	Principle	Short description (2 lines)	Literature	Evidence of its use in practice
Prefer Exceptions to Returning Error Codes	Throw exceptions rather than returning error codes	B		Keeping Tests Clean	Tests needs to be kept clean, because they become sort of a mess otherwise	B	
Don’t Pass Null	Do not pass NULL as an argument to a function	B		One Assert per Test	One assert in each test, not multiple asserts	B	
Don’t Return Null	Do not return NULL from a function	B		Single Concept per Test	One or multiple asserts, but for single concept is OK	B	
Write Your Try-Catch Statement First	Begin with writing your try-catch statement to think about error handling	B					
Table of Class principles							
Principle	Short description (2 lines)	Literature	Evidence of its use in practice				

Class Organization	Throw exceptions rather than returning error codes	B					
High Cohesion	High cohesion is concerned with how closely related the connections are within a module or a class	B, S02, S03, S04, S05, P1, P8, P11, P12, P15, P17	P1				
Low Coupling	Low coupling refers to a module or class that does not have many connections to other classes or modules	B, S02, S03, S04, S05, S06, S08, P1, P8, P9, P11, P15, P17	P1				
Encapsulation	A class should hide some of its behavior, and data should be kept private (unless it is a data class)	B, P11, P15					
Isolating from Change	Create interfaces or abstract classes to cope with change	B					
SRP – Single Responsibility Principle	A class or module should only have one responsibility	B, S02, S06, S07, S08, P1, P3, P17	P1				
Minimal Classes and Methods	Do not create too many classes and methods	B					
One Level of Abstraction per Function	High-abstraction or low-abstraction. Do not intermix these.	B, P4, P15, P17	P4				

Classes should be small	The responsibilities of a class should be kept low	B, S02, S06, P1	P1					

11.3 Appendix C

Exact percentages or numbers from the closed-questions.

11.3.1 Likert scale questions

Strongly disagree = SD **Strongly agree = SA**
Disagree = D **Agree = A**
Somewhat disagree = SWD **Somewhat agree = SWD**

Neither agree or disagree = NAD

11.3.1.1 RQ1

ID	Question text	SD	D	SWD	NAD	SWA	A	SA
Q7b	I have heard of some of the essential Clean Code practices and principles before.	0%	0%	0%	3.7%	18.52%	22.22%	55.56%

	General principles Do you agree with the [principle]	SD	D	SwD	NAD	SwA	A	SA
Q7c	Separate Constructing a System from Using It	0	5.26	0	7.89	10.53	44.74	31.58
	OCP	0	5.26	10.53	7.89	15.79	36.84	23.69
	KISS	0	0	0	5.26	5.26	28.95	60.53
	Minimize nesting	0	0	2.63	0	5.26	31.58	60.53

	The Boy Scout Rule	0	0	0	0	13.16	28.95	57.89
Q7d	**Naming principles** **Do you agree with the [principle]**	**SD**	**D**	**SwD**	**NAD**	**SwA**	**A**	**SA**
	Avoid Mental Mapping	0	0	2.63	5.26	13.16	31.58	47.37
	Avoid Disinformation	0	0	0	2.63	0	15.79	81.58
	Searchable Names	0	0	2.7	5.41	13.51	24.32	54.05
	Pronounceable Names	0	0	2.63	15.79	18.42	28.95	34.21
	Use Intention-Revealing Names	0	0	0	2.63	5.26	21.05	71.05
	Use Meaningful Names	0	0	0	0	5.26	13.16	81.58
Q7e	**Function and Method principles** **Do you agree with the [principle]**	**SD**	**D**	**SwD**	**NAD**	**SwA**	**A**	**SA**
	Methods/Functions should be small	0	0	2.63	7.89	13.16	34.21	42.11
	Structured Programming	2.63	5.26	7.89	5.26	21.05	44.74	13.16
	Function Arguments	0	5.41	0	10.81	32.43	35.14	16.22
	DRY	0	0	5.26	5.26	26.32	23.68	39.47
	Have No Side Effects	0	0	0	5.26	13.16	28.95	52.63
	Extract Try-Catch Block	0	2.63	15.79	26.32	23.68	26.32	5.26
	Command Query Separation	0	0	5.26	2.63	23.68	26.32	5.26
	Do One Thing	0	0	0	2.63	15.79	36.84	44.74
Q7f	**Comment principles** **Do you agree with the [principle]**	**SD**	**D**	**SwD**	**NAD**	**SwA**	**A**	**SA**
	Warning of Consequences	2.63	2.63	2.63	18.42	21.05	36.84	15.79
	TODO Comments	2.63	7.89	5.26	13.16	10.53	31.58	28.95
	Explanation of Intent	2.63	7.89	0	0	15.79	42.11	31.58

	Explain Yourself in Code	2.63	2.63	0	0	5.26	31.58	57.89
	Clarification	2.63	13.16	10.53	2.63	18.42	26.32	26.32
	Amplification	2.7	8.11	2.7	5.41	35.14	27.03	18.92
Q7g	**Formatting principles Do you agree with the [principle]**	**SD**	**D**	**SwD**	**NAD**	**SwA**	**A**	**SA**
	Organizing for Change	0	0	0	5.26	13.16	39.47	42.11
	Vertical Distance and Ordering	0	0	0	7.89	26.32	36.84	28.95
	Dependent Functions	0	0	2.63	21.05	28.95	31.58	15.79
	Horizontal Formatting - Indentation	0	0	2.63	2.63	2.63	31.58	60.53
	Team Coding Standards	0	0	0	0	2.63	42.11	55.26
Q7h	**Object and Data Structure principles Do you agree with the [principle]**	**SD**	**D**	**SwD**	**NAD**	**SwA**	**A**	**SA**
	Law of Demeter	0	0	2.63	15.79	13.16	31.58	36.84
	Data/Object Anti-Symmetry	0	0	5.26	26.32	23.68	21.05	23.68
Q7i	**Error Handling principles Do you agree with the [principle]**	**SD**	**D**	**SwD**	**NAD**	**SwA**	**A**	**SA**
	Write Your Try-Catch Statement First	7.89	10.53	10.53	44.74	5.26	13.16	7.89
	Don't Return Null	7.89	13.16	13.16	5.26	18.42	31.58	10.53
	Don't Pass Null	5.26	13.16	10.53	7.89	15.79	34.21	13.16
	Prefer Exceptions to Returning Error Codes	5.26	0	5.26	10.53	13.16	34.21	31.58
Q7j	**Unit Test principles Do you agree with the [principle]**	**SD**	**D**	**SwD**	**NAD**	**SwA**	**A**	**SA**
	Single Concept per Test	2.63	2.63	2.63	2.63	5.26	36.84	47.37

	One Assert per Test	5.26	18.42	23.68	13.16	13.16	18.42	7.89
	Keeping Tests Clean	0	0	0	5.26	5.26	52.63	36.84
Q7k	**Class principles** **Do you agree with the [principle]**	**SD**	**D**	**SwD**	**NAD**	**SwA**	**A**	**SA**
	Classes should be small	0	0	0	13.16	26.32	31.58	28.95
	One Level of Abstraction per Function	0	0	0	13.16	31.58	34.21	21.05
	Minimal Classes and Methods	5.26	5.26	7.89	10.53	34.21	18.42	18.42
	SRP	0	0	2.63	7.89	23.68	26.32	39.47
	Isolating from Change	0	0	2.63	7.89	15.79	28.95	36.84
	Encapsulation	0	0	0	5.26	10.53	36.84	47.37
	Low Coupling	0	0	0	8.11	16.22	37.84	37.84
	High Cohesion	0	2.7	0	16.22	16.22	40.54	24.32
	Class Organization	2.63	0	2.63	21.05	26.32	23.68	23.68

11.3.1.2 RQ1a

ID	Question text	SD	D	SWD	NAD	SWA	A	SA
Q8b	I think refactoring is a useful tool to make code clean.	0%	0%	0%	2.63%	10.53%	21.05%	65.79%

11.3.1.3 RQ2

ID	Question text	SD	D	SWD	NAD	SWA	A	SA
Q9b	It is more difficult to write clean code initially	5.26%	10.53%	5.26%	13.16%	26.32%	28.95%	10.53%
Q9g	I do believe refactoring has a positive effect on the quality of code.	0%	0%	0%	0%	5.26%	28.95%	65.79%
Q9h	To write clean code initially, the requirements have to be clearly specified	2.63%	5.26%	5.26%	10.53%	18.42%	31.58%	26.32%

Q9i	It is/would be easier to write clean code at the beginning of a project	0%	7.89%	10.53%	18.42%	31.58%	18.42%	13.16%
Q9j	Writing clean code make it easier to make modifications to the code later on	0%	0%	0%	2.63%	5.26%	34.21%	57.89%
Q9k	I have less time to write clean code towards the end of a project due to deadlines	10.53%	7.89%	18.42%	26.32%	15.79%	15.79%	5.26%

11.3.1.4 RQ2a

ID	Question text	SD	D	SWD	NAD	SWA	A	SA
Q10a	I read and modify source code from other programmers	0%	5.26%	5.26%	5.26%	13.16%	21.05%	50%
Q10b	I review or comment on other people's code	0%	0%	0%	0%	10.53%	18.42%	71.05%
Q10c	Other people are reading and modifying the code that I write	0%	2.63%	2.63%	5.26%	5.26%	23.68%	60.53%
Q10d	Other people review or comment the code that I write	0%	0%	5.26%	2.63%	5.26%	15.79%	71.05%

11.3.1.5 RQ3

ID	Question text	SD	D	SWD	NAD	SWA	A	SA
Q11b	I do believe clean code eases the process of reading code.	0%	0%	0%	0%	0%	19.23%	80.77%
Q11c	I do believe clean code eases the process of understanding code.	0%	0%	0%	0%	3.85%	19.23%	76.92%
Q11d	I do believe clean code eases the process of reusing code.	0%	0%	0%	3.85%	7.69%	19.23%	69.23%
Q11e	I do believe clean code eases the process of maintaining code.	0%	0%	3.85%	0%	0%	19.23%	76.92%

Q11f	Writing readable and understandable code is wasting time, and prevents you from being productive and completing tasks.	69.23%	15.28%	3.85%	0%	0%	7.69%	3.85%

11.3.1.6 RQ3a

ID	Question text	SD	D	SWD	NAD	SWA	A	SA
Q12b	Do you believe that clean code helps with the readability and understandability of code?	0%	0%	0%	0%	3.85%	19.23%	76.92%
Q12d	Do you believe that clean code helps with the reusability and maintainability of code?	0%	0%	0%	0%	0%	38.46%	61.54%
Q12f	Reading and understanding clean code takes shorter time than reading and understanding the same dirty code.	0%	0%	3.85%	0%	0%	23.08%	73.08%
Q12g	Reusing clean code takes shorter time than reusing the same dirty code.	0%	0%	0%	7.69%	0%	23.08%	69.23%
Q12h	Modifying clean code takes shorter time than modifying the same dirty code.	0%	0%	0%	0%	7.69%	11.54%	80.77%

11.3.2 Other question types

11.3.2.1 RQ1

ID	Question text	Python	JavaScript	Java	C++	C	C#	Kotlin	Swift	Go	Scala	Other
Q7a	This survey is about Clean Code. When you write your answers about Clean Code: What programming language do you have in mind?	2.63%	13.16%	55.26%	0%	0%	7.89%	2.63%	2.63%	7.89%	0%	7.89%

11.3.2.2 RQ1a

ID	Question text	Yes	No
Q8e	Do you use refactoring as a technique to keep the code clean?	96.30%	3.70%
Q8g	My organization has static analysis tools in place to keep the code clean	59.26%	40.74%
Q8i	My organization has automatic means not to allow unclean code to go through (e.g., Quality Gate)	66.67%	33.33%

11.3.2.3 RQ2

ID	Question text	Initially	Refactor	None of these
Q9a	Do you write clean code initially or write “messy” code that you refactor later?	26.32%	26.32%	52.63%

11.3.2.4 RQ3

Q11a. Rank the code quality characteristics from most important at the top to least important at the bottom.
The more 1’s a code quality characteristic has, the higher up it is in the list. The more 9’s it has, the lower it is in the list.

Rank the code quality characteristics	(	)	*	+	,	-	.	/	0
Readability	34%	18%	16%	11%	11%	5%	5%	0%	0%
Structure	11%	19%	11%	14%	16%	8%	16%	3%	3%
Comprehensibility	5%	16%	24%	22%	14%	14%	5%	0%	0%
Maintainability	11%	5%	19%	27%	22%	14%	3%	0%	0%
Correctness	30%	14%	8%	11%	16%	16%	3%	3%	0%
Documentation	0%	3%	0%	3%	0%	8%	22%	46%	19%
Testability	11%	19%	19%	8%	11%	16%	5%	11%	0%

Dynamic Behavior	0%	5%	3%	5%	11%	16%	35%	19%	5%
Micellaneous	0%	0%	0%	0%	0%	3%	5%	19%	73%

11.4 Appendix E

11.4.1 RQ1: Thematic analysis

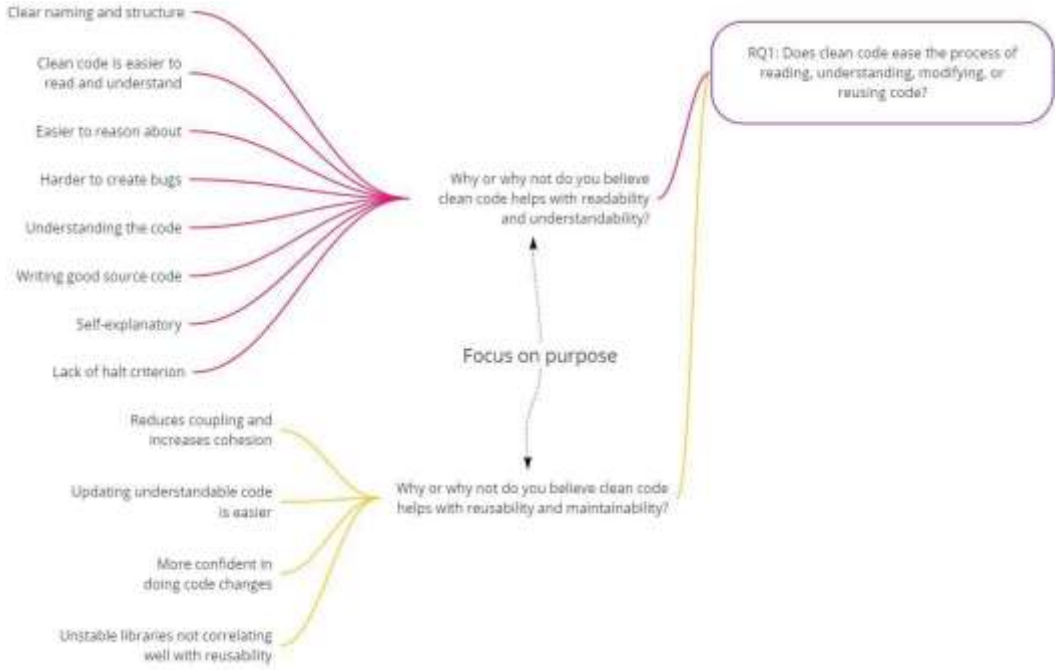


Figure RQ1 Part 1: Thematic analysis mind map of RQ1

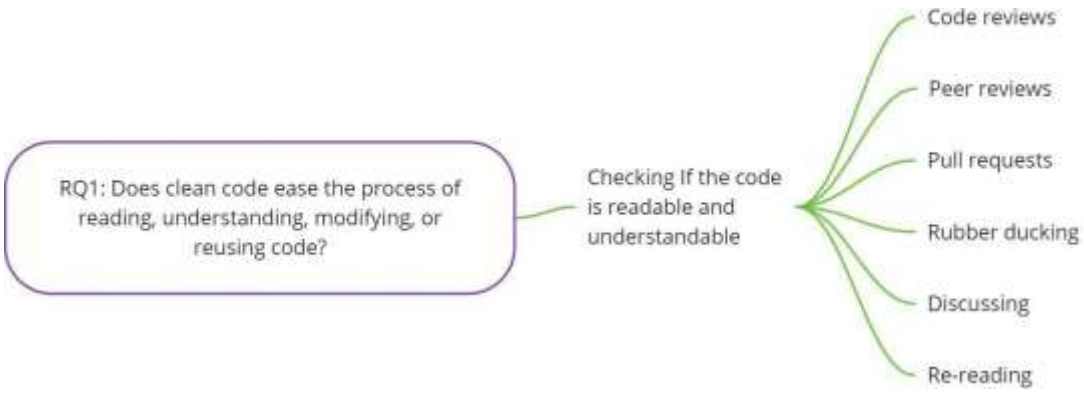


Figure RQ1 Part 2: Thematic analysis mind map of RQ1

11.4.2 RQ2: Thematic analysis

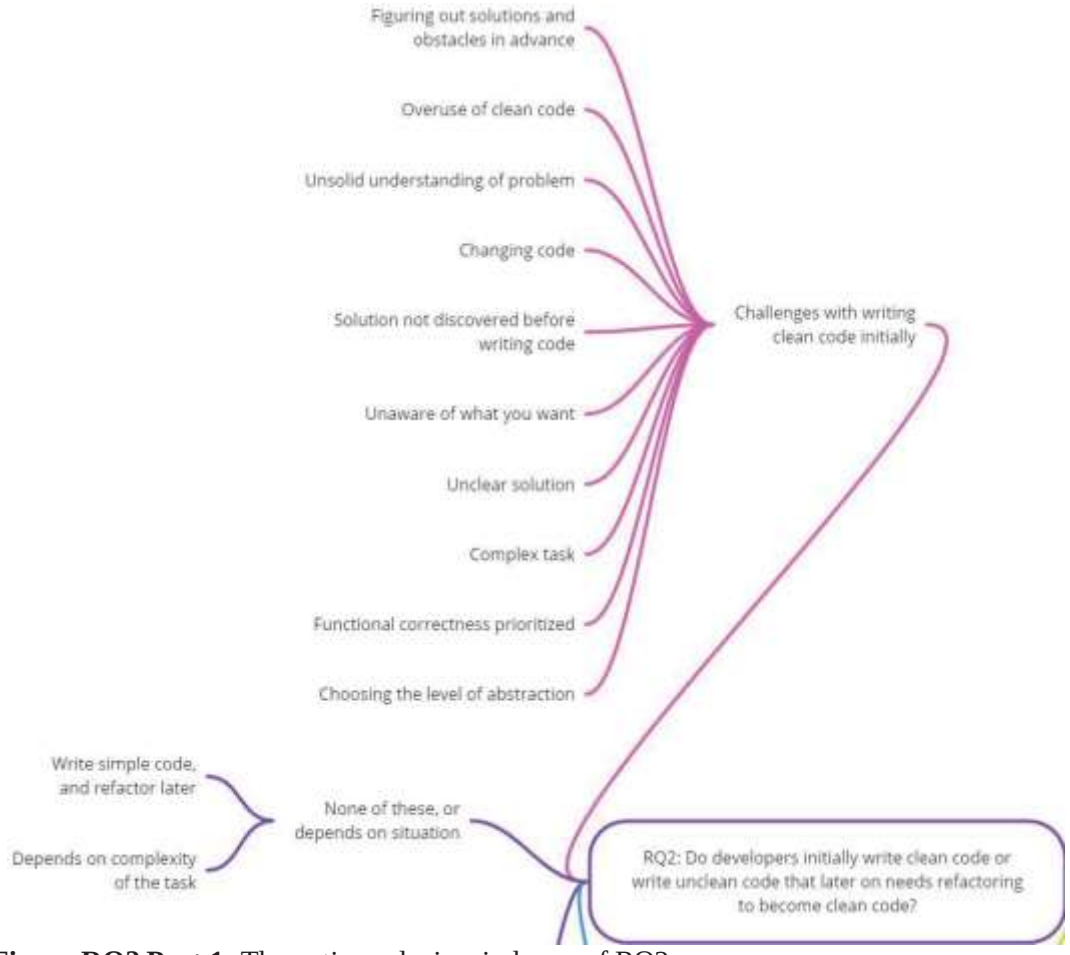


Figure RQ2 Part 1: Thematic analysis mind map of RQ2

Figure RQ2 Part 2: Thematic analysis mind map of RQ2

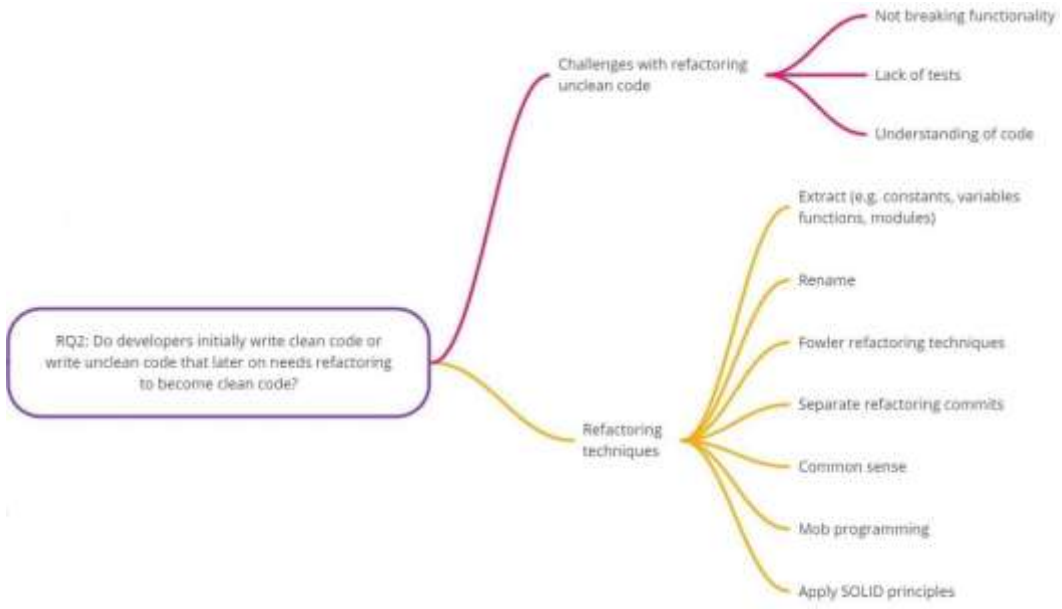


Figure RQ2 Part 3: Thematic analysis mind map of RQ2

11.4.3 RQ3: Thematic analysis

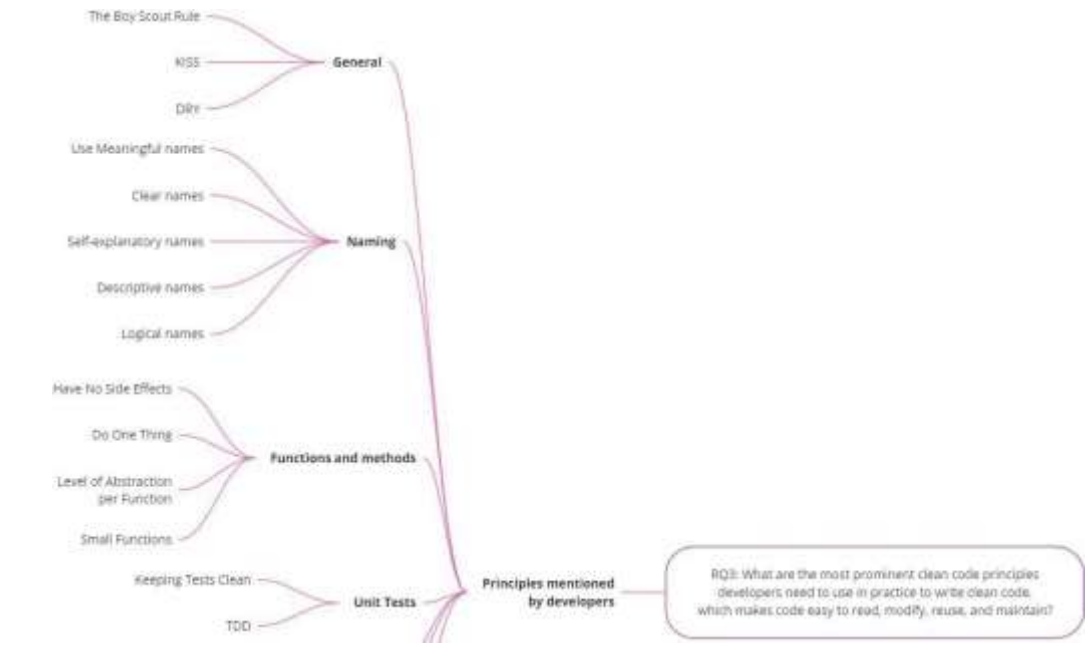


Figure RQ3 Part 1: Thematic analysis mind map of RQ3

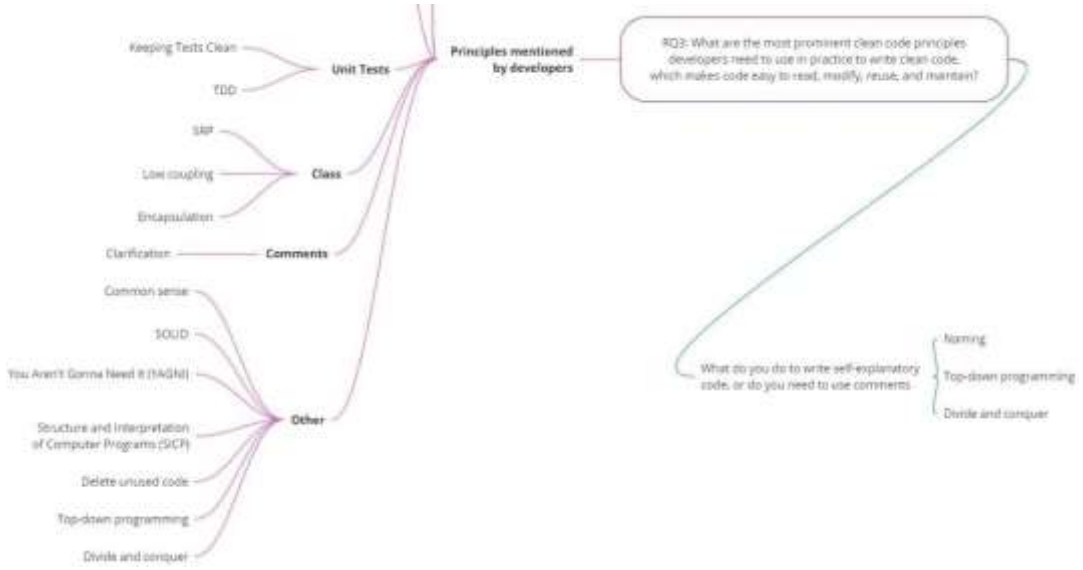


Figure RQ3 Part 2: Thematic analysis mind map of RQ3

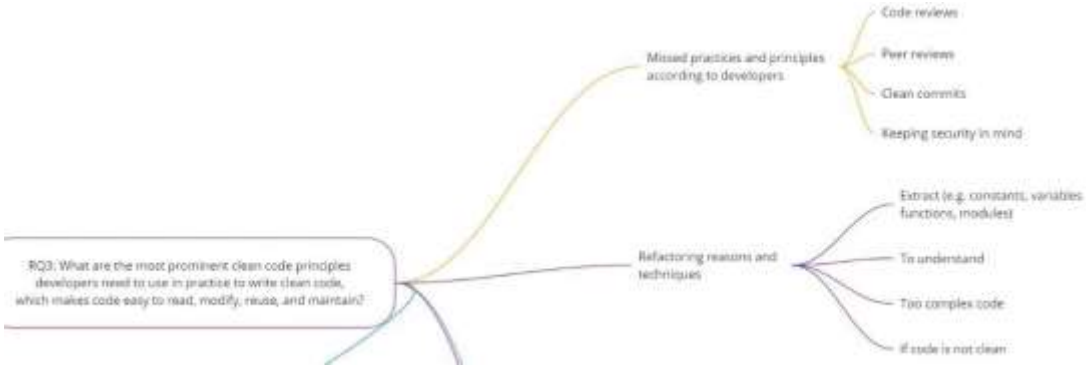


Figure RQ3 Part 3: Thematic analysis mind map of RQ3

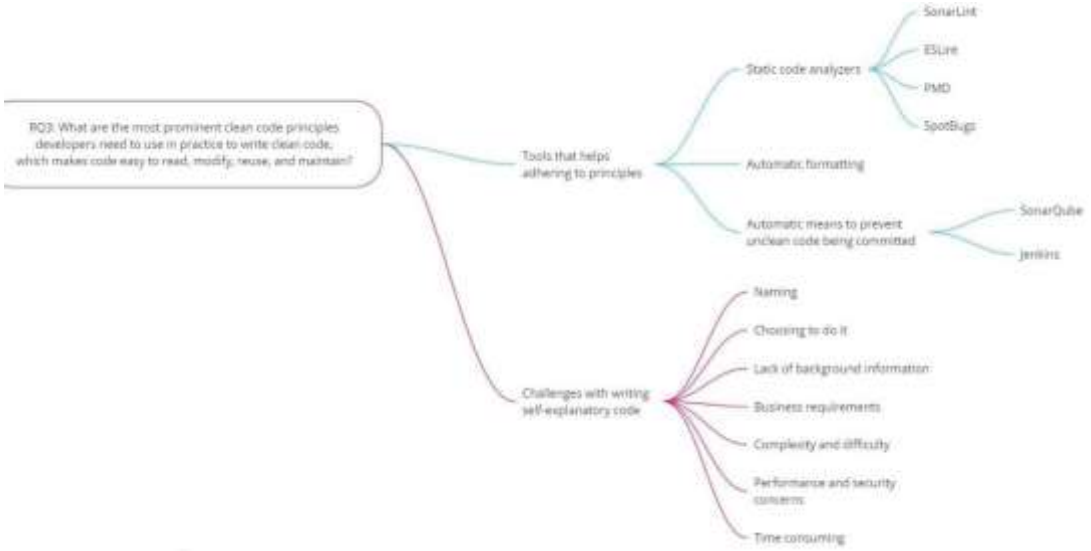


Figure RQ3 Part 4: Thematic analysis mind map of RQ3

11.5 Appendix F

The Wilcoxon p-value test is used to check if the answers were significantly higher than the neutral value 5, if p-value is less than 0.05, means that it is statistically significant bigger [16]. If $p < 0.05$, then we include the principle, and otherwise, we exclude it.

General principles	**p < 0.05**	**INCLUDE OR EXCLUDE**
The Boy Scout Rule	3.90E-16	INCLUDE
Minimize nesting	1.06E-14	INCLUDE
KISS	5.10E-15	INCLUDE
OCP	6.91E-07	INCLUDE
Separate Constructing a System from Using It	1.82E-11	INCLUDE

Naming principles	**p < 0.05**	**INCLUDE OR EXCLUDE**
Use Meaningful Names	2.20E-16	INCLUDE
Use Intention-Revealing Names	7.36E-16	INCLUDE
Pronounceable Names	3.45E-11	INCLUDE
Searchable Names	4.85E-12	INCLUDE
Avoid Disinformation	2.92E-16	INCLUDE
Avoid Mental Mapping	2.29E-13	INCLUDE

Function and Method principles	**p < 0.05**	**INCLUDE OR EXCLUDE**
Do One Thing	2.20E-15	INCLUDE
Command Query Separation	1.81E-12	INCLUDE
Extract Try-Catch Block	0.001383	INCLUDE
Have No Side Effects	2.29E-13	INCLUDE
DRY	6.20E-12	INCLUDE
Function Arguments	1.13E-9	INCLUDE
Structured Programming	2.50E-07	INCLUDE
Methods/Functions should be small	8.65E-13	INCLUDE

Comment principles	**p < 0.05**	**INCLUDE OR EXCLUDE**
Amplification	2.69E-07	INCLUDE
Clarification	0.0003011	INCLUDE
Explain Yourself in Code	3.18E-13	INCLUDE
Explanation of Intent	1.83E-10	INCLUDE
TODO Comments	4.33E-06	INCLUDE
Warning of Consequences	2.62E-08	INCLUDE

Formatting principles	**p < 0.05**	**INCLUDE OR EXCLUDE**
Team Coding Standards	3.48E-16	INCLUDE
Horizontal Formatting - Indentation	3.97E-14	INCLUDE
Dependent Functions	3.20E-10	INCLUDE
Vertical Distance and Ordering	3.73E-14	INCLUDE
Organizing for Change	8.42E-15	INCLUDE

Object and Data Structure principles	**p < 0.05**	**INCLUDE OR EXCLUDE**
Data/Object Anti-Symmetry	4.08E-08	INCLUDE
Law of Demeter	3.29E-11	INCLUDE

Error Handling principles	**p < 0.05**	**INCLUDE OR EXCLUDE**

Prefer Exceptions to Returning Error Codes	1.56E-08	INCLUDE
Don’t Pass Null	5.12E-03	INCLUDE
Don’t Return Null	0.03262	INCLUDE
Write Your Try-Catch Statement First	0.8072	EXCLUDE

Unit Test principles	**p < 0.05**	**INCLUDE OR EXCLUDE**
Keeping Tests Clean	6.46E-15	INCLUDE
One Assert per Test	0.05157	EXCLUDE
Single Concept per Test	2.95E-11	INCLUDE

Class principles	**p < 0.05**	**INCLUDE OR EXCLUDE**
Class organization	5.48E-09	INCLUDE
High cohesion	6.10E-10	INCLUDE
Low coupling	9.10E-13	INCLUDE
Encapsulation	7.71E-15	INCLUDE
Isolating from Change	5.45E-09	INCLUDE
SRP	9.50E-13	INCLUDE
Minimal Classes and Methods	2.95E-11	INCLUDE
One Level of Abstraction per Function	4.60E-13	INCLUDE
Classes should be small	4.74E-13	INCLUDE