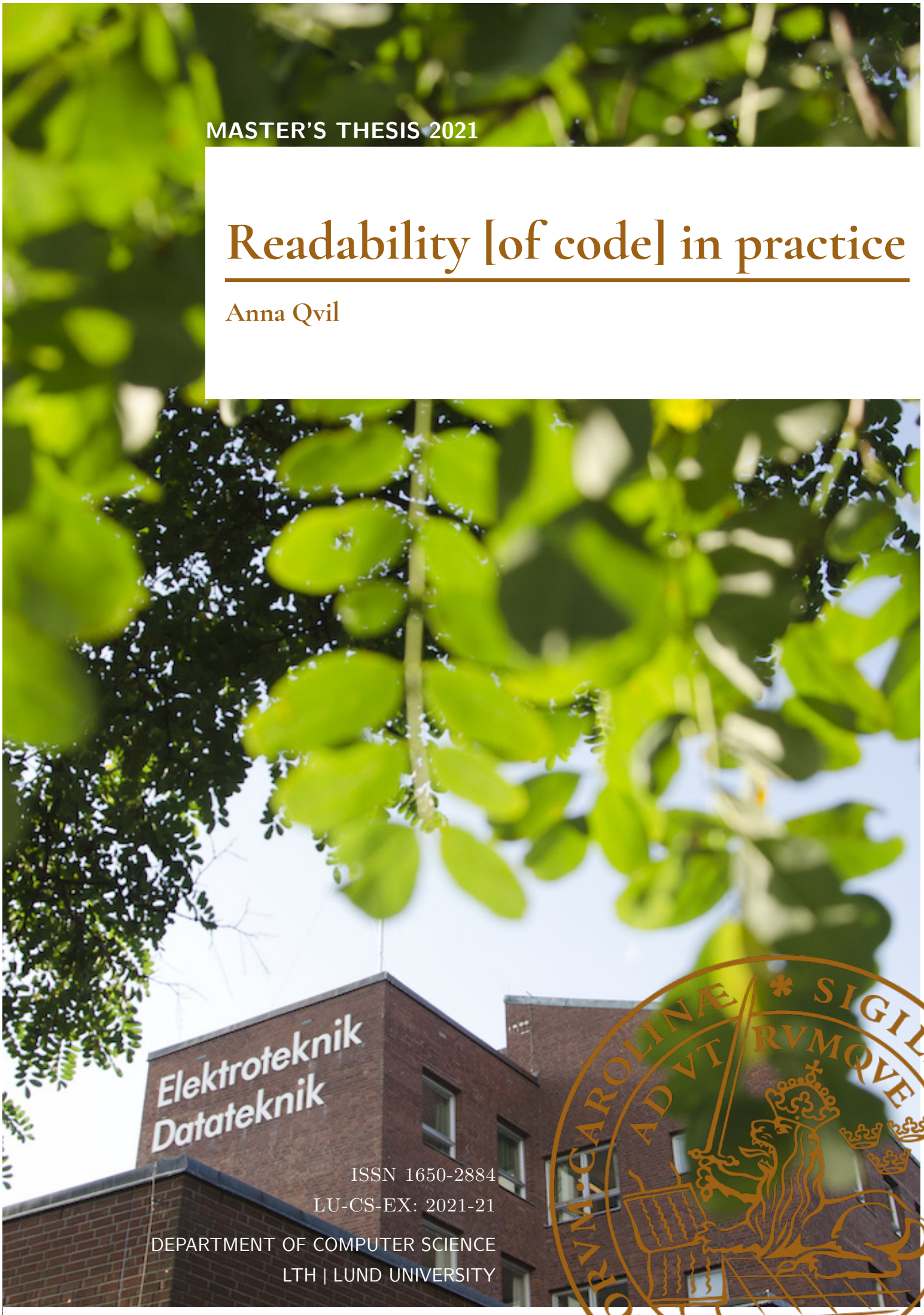


MASTER'S THESIS 2021

Readability [of code] in practice

Anna Qvil

ISSN 1650-2884
LU-CS-EX: 2021-21

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-21

Readability [of code] in practice

Läsbarhet av kod i praktiken

Anna Qvil

Readability [of code] in practice

(An exploratory study of software professionals' perception of readability)

Anna Qvil
anna@qvil.se

June 15, 2021

Master's thesis work carried out at Sony Nordic (Sweden), Local branch to Sony Europe B.V. (NL).

Supervisors:
Andreas Bexell, andreas.bexell@sony.com
Emma Söderberg, emma.soderberg@cs.lth.se
Luke Church, luke@church.name

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

Abstract

Readability of code is a central concern in software development. Different models for defining and measuring readability have been created but they have unfortunately been shown to poorly capture the changes developers do to enhance readability. In this study, the concept of readability is explored from the perspective of software professionals through interviews and a mining study. The results show that developers value properties such as consistency, descriptive naming and comments. The results from this study also show discrepancies from the existing models. One clear example of this is how some of the existing models claim that fewer lines of code enhance readability whereas three of the six interviewees said that they do not mind longer code.

Keywords: readability, mining software repositories, software development, code review, developers perception

Acknowledgements

There are a couple of persons I would like to thank for their incredible support and help with this master thesis.

Andreas Bexell at Sony for giving me this amazing opportunity and for all the countless hours he spent helping me through this process.

Emma Söderberg and Luke Church at the Faculty of Engineering at Lund University for their invaluable support and help to form and accomplish this study.

Last but not least, all the workers at Sony for welcoming me and participating in the interviews.

Contents

Chapter 1
Introduction

When developing software programs there is often a team of developers working together. As several developers work together there is a need to easily understand each others' code, and having readable code makes that a lot easier. In software engineering the term readability is used to discuss how readable the code is. Evidence suggests that readability is also closely connected to maintainability as, in maintenance, developers need to modify code already in place and therefore also need to read and understand it [5].

Currently, there exist different models meant to define and measure the readability of code. These models were developed as a tool to help developers when writing code. The models is presented in section 2. Recent studies have found that some of these models are unable to detect intended readability improvements made by developers [11]. This seeming gap between the state-of-art and practice will be explored in this study.

1.1 Objective

The goal of this thesis is to explore how software professionals reason about readability and what they think are important aspects of readability. A part of this objective is to also look at how developers treat readability in code and code reviews. Although this study does not aim to create a new model for code readability, it is trying to explore the gap between the existing models and software professionals view of readability. This gap was reported by Fakhoury [11] et. al, where they found that that the models did not seem to capture changes made in code meant to enhance readability.

The objective of this study is summarized in these two research questions:

- **RQ1:** What do software professionals consider to be major aspects of readability?
- **RQ2:** What changes do software professionals impose on code to enhance its readability?

Aspects of readability refer, in this study, to different elements of the code that are important for readability. The aspects can be whatever the software developers think of, such as lines of code, indentation, order of functions, etc. Software professionals refers to people who in some way work professionally with code, this could be e.g. developers, test developers.

1.2 Scope

When studying readability there are many aspects to take into consideration, for instance, the programming language used. As programming languages contain different constructs they can affect how developers read the code. In this study, only Java is used as it is a widely used language, both in the industry as a whole but also within Sony, the sponsoring company.

All the data for this project will be collected from Sony as it is easily accessible to the author.

Chapter 2
Related work

This section presents some of the existing models for code readability. These models are presented to later see how they correspond to findings from the interviews and the mining in this study. As a part of this study is to do a mining study (analyse data from a software repository) we also take a look at some other projects that have used mining as a methodology.

2.1 Readability models

When working with code there is often a question about how to measure the quality. For the sake of readability, there have been studies aiming to conduct measurements and models specifically for readability. One of the examples of this is the paper written by Buse and Weimer [7], in this paper, they asked 120 human annotators to score small code snippets according to their readability in an attempt to see if readability can be automatically scored. The authors found that readability can be automatically assessed with 19 lexical features in mind. The features taken into consideration were, for example, line length, number of identifiers, number of keywords. The full list of the lexical features considered by this model can be found in Table 2.1. Each feature represents either an average value per line or a maximum value for all lines.

The study done by Buse and Weimer was later simplified by Posnett et. al. in their paper "A simpler model of software readability" [13]. The simplification was done as the authors believed that the model proposed by Buse and Weimer used too many features. In their paper, they built a new model that relies on two main measures: size and entropy. They argue that as the snippets used by Buse and Weimer all ranged from $4 - 11$ lines of code they could not guarantee that the findings done by Buse and Weimer hold for larger pieces of code. They address this problem and also use Halstead's Metrics [3] to improve and simplify the model from Buse and Weimer. Posnett et. al. propose the readability measure seen in Equation 2.1.

$$z = 8.87 - 0.033V + 0.40Lines - 1.5Entropy \tag{2.1}$$

Avg.	Max.	Feature name
X	X	Line length (# characters)
X	X	# identifiers
X	X	identifier length
X	X	indentation (preceding whitespace)
X	X	# keywords
X	X	# numbers
X		# comments
X		# periods
X		# commas
X		# spaces
X		# parenthesis
X		# arithmetic operators
X		# comparison operators
X		# assignments (=)
X		# branches (if)
X		# loops (for, while)
X		# blank lines
	X	# occurrences of any single character
	X	# occurrences of any single identifier

Table 2.1: Metrics considered by Buse and Weimers model

In this measurement the V is Halsteads's metric for program volume [3]. Program volume is calculated with the use of *program length* (N),

$$N = TotalNumberOfOperators + TotalNumberOfOperands \tag{2.2}$$

and *program vocabulary* (n),

$$n = NumberOfDistinctOperators + NumberOfDistinctOperands \tag{2.3}$$

This gives the metric *program volume* (V),

$$V = N * \log_2 n \tag{2.4}$$

$Lines$ is Equation 2.1 is the total number of lines. Lastly there is entropy that is calculated using counts of terms (token and bytes) and the number of unique terms. In the equation for entropy X represent a document and x_i is a term in the document. $Count(x_i)$ is the number of occurrences of x_i in said document and $p(x_i) = \frac{count(x_i)}{\sum_{j=1}^{n} count(x_j)}$. Entropy $H(X)$ is then defined as can be seen in Equation 2.5

$$H(X) = -\sum_{i=1}^{n} p(x_i) * \log_2 p(x_i) \tag{2.5}$$

Another readability model was proposed by Dorn [10] where he argues for the need for visual, spatial, and linguistic features in a readability model. The study aimed to create a model that was generalizable and relied on a larger set of features. This study was based on the judgement

of over 5000 different human annotators and 360 code snippets, where each was up to 50 lines long. The results of the study included aspects such as syntax highlighting, variable naming standard and operator alignment as well as line length, loops and spaces. The results showed that this model corresponded 2.3 times better with the human judgement than the models from Buse and Weimer, and Posnett et. al.

Scalabrino et.al [17] later improved upon the formerly mentioned studies. They did this by proposing a model that combined the previously mentioned models with a set of textual features. The textual features measure the consistency between code and comments, specificity of the identifiers, usage of complete identifiers, etc. This model combined with the other models was shown to have higher accuracy than the ones more focused on structural features.

With all the presented models in mind, Fakhoury et. al. wrote the paper "Improving source code readability: Theory and practice" [11] aimed to evaluate how well the models capture when developers do a change to improve readability. They did this by mining 63 Java projects for commits where the commit message explicitly state that a change was made to improve readability. They then took the measurements of the models on the source code before and after the change to see if they got a higher score after the change. In this study, they found that the current readability models fail to capture the improvements made by the developers.

Other than the models presented above there exist a few more. One of them is a paper from DeYoung and Kampen [9] where they proposed a model based on three variables:

- VAR - the average normalized length of variables (average of all variables length divided by the number of lines between first and last use)
- NSL - number of lines containing statements
- CYCLO - the sum of program branches plus one

They contained these values by letting graduate and undergraduate students write a program in Algol 68 and then grade each other solutions based on readability. After this and some more testing with the solutions from the students, the authors proposed the following metric for scoring readability:

$$\hat{R} = 0.295 * VAR - 0.499 * NSL + 0.013 * CYCLO \tag{2.6}$$

This metric suggest that VAR have a positive impact, NSL have the most, and negative, impact and $CYCLO$ have the least but positive impact.

Jørgensen [12] proposed another readability metric based on the extensiveness of comments ($C1$), blank lines in the left margin ($C2$) and blank lines ($C3$) as well as the average length of variable names ($C4$), the average number of arithmetic operators per 100 lines ($C5$) and the average of goto-statements ($C6$). The proposed metric is as can be seen in Equation 2.7.

$$R' = 5.7 - 0.061 * C1 - 0.047 * C2 - 0.023 * C3 - 0.15 * C4 + 0.0065 * C5 + 0.21 * C6 \tag{2.7}$$

Jørgensens used a similar method as DeYoung and Kampen where he had students solve a programming issue, this time in Pascal, and then another set of students evaluated the different solutions based on a couple of questions.

Choi et. al. in their 2018 article [8] proposed another model for readability. To build their model they picked indicators on readability based on their experience and others' work. With those indicators, the writers then used sample code from open source repositories and made a questionnaire where undergraduate students, postgraduate students, doctoral researchers, and programmers in the software industry rated the samples on a scale of how easy they were to understand. From this, the writers then made a regression analysis and in the end ended up with the following metric for scoring readability:

$$\begin{aligned} Readability = & (-0.020) * LOC + (0.040) * numberOfComments \\ & + (0.0370) * numberOfBlanklines \\ & + (-0.755) * numberOfBitwiseOperators \\ & + (-0.153) * maxNestedContorlStatements \\ & + (-0.001) * programVolume \\ & + (-0.611) * Entropy \end{aligned} \tag{2.8}$$

Riberio and Travassos [15] later looked at the models from Buse and Weimer, Postnett et. al, Jørgensen and DeYoung et. al as well as other published papers about readability and comprehension and found contradictions among these papers. They analysed three contradictory source code attributes: indentation spacing, identifier length and code size. Riberio and Travassos did this by first performing a literature review where they, for example, found that Jørgensen's model has a positive contribution for indentation spacing while the others have a negative contribution for the number of spaces used for indentation. Riberio and Travassos also surveyed computer science undergraduates with different amounts of experience to see if the difference in experience could be a factor in why there are contradictions. They found that the participants in the study preferred 4-space indentation regardless of experience. There seemed to be a preference for long and complete words in identifiers, but participants with higher experience and low domain knowledge were more affected by the length of the identifiers. Lastly, in regards to code size, they found that for comprehension all participants showed positive results for more lines of code but for the readability they found that less experienced participants preferred longer code while more experienced preferred shorter. In the end, Riberio and Travassos claim that the contradictory results are a consequence of interchangeable use of the concepts of readability and comprehension.

Building on the findings from Fakhoury et. al. and Riberio and Travassos this study will take a further look into what aspects of readability that seems to be important for developers, to see how well that is linked to the presented studies. As Fakhoury et. al. also determined that changes made to enhance readability were not captured well by the presented models this study will also look at what kind of changes that are made to enhance readability.

2.2 Mining studies

As mentioned in the previous section Fakhoury et. al. [11] used mining when evaluating the readability models. In their paper they identified relevant commits by using the keywords 'readable', 'readability', 'easier to read', 'comprehension', 'comprehensible', 'understand', 'understanding' and 'clean-up'. They then manually went through the commits they found and excluded commits that did not explicitly reflect readability improvements (e.g commits that improved the readability of UI elements).

Bexell [6] also used mining as a method in his paper ”Software Source Code Readability: A Mapping Study”. In his paper, he looked for commit messages that contained the string ”reada”.

This study aims to perform mining in a similar way as the two mentioned above but using a larger set of keywords and looking at entire review instances from Gerrit, a code collaboration tool. That means to not only look for commit messages that use the specified keywords but also review comments. It is preferable to look at the entire review instead of just commit messages as that can give more results, as discussions about readability may be found in other places than commit messages.

Chapter 3
Method

The method of this study is divided into two parts: interviews and a mining study. The interviews and the mining study are connected though a set of keywords that are yielded by the interviews and then used in the mining. In the following sections, the two steps are explained.

3.1 Interviews

The interviews in this study serve two main purposes; to get material to answer RQ1, aspects of readability, and to get keywords to use in the mining study. For the keywords the interviews were analysed to look at how the interviewee talked about code and for the aspects it was looked at what elements of the code the interviewees discussed. The interview protocol was constructed to cover both RQ1 and the keywords, see Appendix B for the full protocol. This section outlines how the interviews and the analysis of the interviews were conducted.

3.1.1 Execution of the interviews

For this study, the semi-structured [16] approach to interviews was taken. This choice was taken as the interest for this study is in how the developers reason about the readability of code instead of them just grading a piece of code.

The interviews were divided into three sections. In the first part, the interviewee was asked in advance to bring an example of code that triggered thoughts of readability. The exact task given to the developers was formulated as follows:

> *I would appreciate it if you could find a Java-code example that somehow makes you think about readability. It doesn't matter if it's in your opinion is an example of good readability, bad readability or somewhere where you implemented changes to improve*

readability. The example can be of any length, the important part is that you feel that it is a representative example.

The example they brought was discussed as to what elements of that code made it readable or not readable. One of the initial question for the participants when looking at the example they brought was *Why did you decide to bring this example, what elements of the code makes it readable or not readable?*. This topic was then discussed to the point where the developers had gone through all their thoughts of the example. As can be seen in the full interview protocol, if the participants pointed out something they wanted to change they were also asked what they would write in a commit message if they were to commit this change. This question was specifically added to help gather keywords for the mining study.

In the second part, the interviewees were presented with an example selected by the author. With the new example, they were asked to go through the code and point out elements that they thought were examples of either good or bad readability. If needed, in both these parts, the discussion was limited to one function at the time to be able to get as specific elements of the code as possible.

The new example the interviewees were presented with was a code snippet containing a Sudoku example. The example was taken from [19] and modified to remove all things regarding the Java applet code as this is not something that is normally used in newer versions of Java. The final version of the code used in the interviews can be found in Appendix A. The Sudoku code was picked as it was not too long or complicated and most people are familiar with how Sudoku works. This made it so that the interviewees had an easier time focusing on the topic of readability and not figuring out what the code does. The same Sudoku-code was also used in the 2013 paper "Impact of programming features on code readability" [18] where it was used to test a readability tool. In the paper, they show that with a few modifications to the code they could receive a higher readability value from the tool. From this, a reasoning was made that while the original code was quite good there were things that could be improved. So in the interviews, the interviewee would have the opportunity to find both examples on good and bad readability.

Lastly in the third part, one finishing question was asked to capture if the developers had any other thoughts of readability that had not been covered by the two code snippets.

For the interviews, employees at Sony were asked to participate. To find participants the interviews were advertised in a newsletter in the hope to find willing participants. From the newsletter, only one developer reached out to volunteer. Another two were found by asking for volunteers in daily stand-up meetings. The last three were personally reached out to and asked to participate. Who to personally reach out to was based on getting a range in both gender and experience level within the group of volunteers. As all the persons reached out to agreed to participate this ranged was accomplished. Everyone who volunteered was made aware that they could drop out at any point. The selection of people to interview was chosen to get a range of both junior and senior developers as well as men and women. In the end, two women and four men were interviewed. The same devising can be found in the range of junior and senior developers where two had about one year of professional experience within software development and four had about 10 to 20 years of experience.

In total six interviews were carried out. The interviews were conducted online with the help of Microsoft Teams as the opportunity to have them in person were limited by a pandemic making it so all personnel were working from home [1]. The data from the interviews

[1] https://www.folkhalsomyndigheten.se/the-public-health-agency-of-sweden/

were collected by recording both screen and sound from the Teams meeting. All participants were asked before the recording started for consent to record. After the interviews, the recordings are only accessible to the author and one of the academic supervisors and in the report, the interviews are anonymised and no names of the participants will be presented. The anonymisation also meant that function names etc. from the interviewees' examples are redacted. The first three interviews were carried out with one of the academic supervisors present and then the last three were done with only the author as the sole interviewer. When present, the academic supervisor took notes and added a few questions to the discussion when needed.

3.1.2 Analysis of the interviews

After all the interviews were done the interview recordings were used for analysis. The keywords were extracted by looking at what types of words were used to talk about the code. A categorization scheme, a way of identifying and organizing themes from interviews [14], was used to extract keywords. The scheme arose from the data itself and looked at words or phrases, used by the interviewees, that; described the code, an action they wished to see to enhance the readability of the code or a specific element of the code they found either good or bad for readability.

For data to RQ1, the interviews were analysed to see what specific elements of the code were discussed in the interviews. Not only were the elements analysed but also how the interviewees reasoned about the elements. For example, if they mentioned consistency, what did they mean by that and was it positive or negative in regards to readability? The answers from the interviews were put into different categories as can be seen in Section 4.1. These categories were formed by the author seeing the elements specifically mentioned by the interviewees or the author's interpretation of what was being discussed. Things that were about similar topics were grouped. Below is a list of the groups that were formed including different element that share a common theme, all other elements stand alone in the results.

- Comments and documentation - grouped since comments are a form of documentation
- Indentation, blank space and curly brackets - grouped since they are all about formatting of code
- Consistency, familiarity, convention - grouped by the author's interpretation that using convention and being concise are both a way of adding familiarity to code.

3.2 Mining

The mining part can be broken down into five different steps:

1. Collection of data
2. Filtering on keywords
3. Keyword validation

`communicable-disease-control/covid-19/`

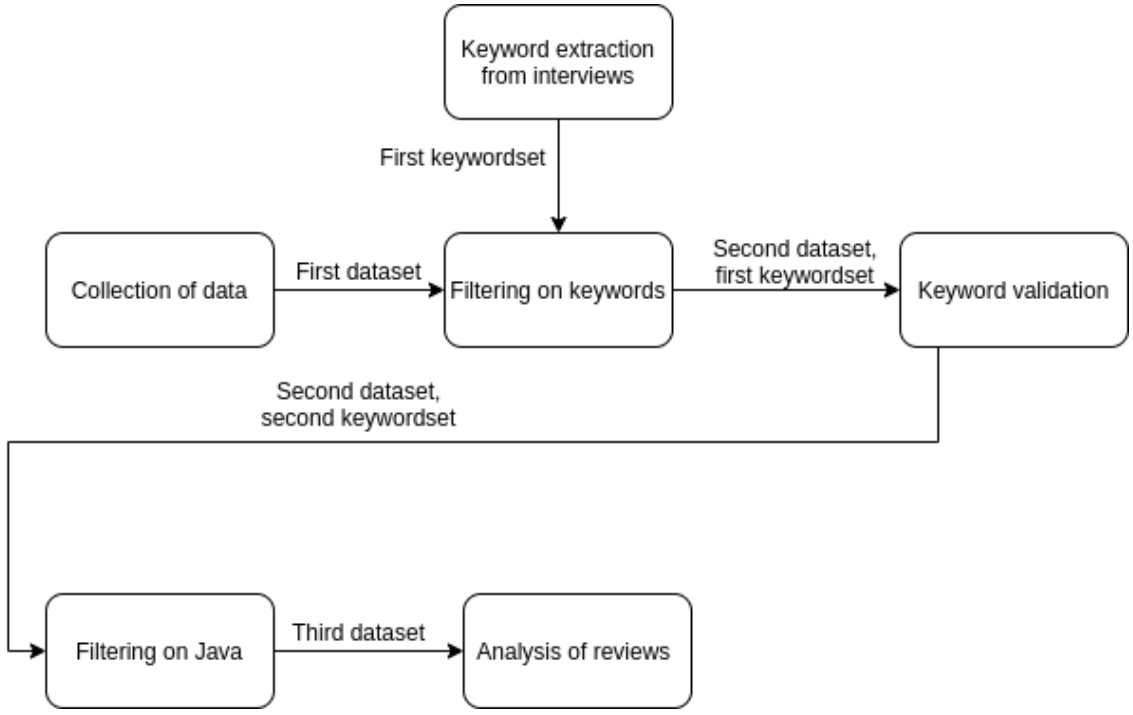


Figure 3.1: Flow chart of the mining study

```
1    for s in {0..223500..500};
2        do ssh -p <port> -l <user> <host> gerrit query --format=
     JSON --comments --patch-sets --files -S $s status:merged >
     comments.$s.json;
3        sleep 1;
4    done
```

Figure 3.2: Shell script to get reviews

4. Filtering on commits that include Java-files

5. Analysis of reviews

The parts are also shown in figure 3.1.
This section outlines what was done in each of these steps.

3.2.1 Collection of data

The first step was to collect all reviews that Sony had in their Gerrit. A review in this study refers to a place where developers discuss commits of code and upload new modification of it til both the reviewers and creator are satisfied. The collection of reviews was made using Gerrit query over ssh [2]. The full script used to collect all of the data is presented in figure 3.2.

In the end, 223 688 reviews were collected and sorted into files with 500 in each, this was made just to make the data easier to handle later. The 223 688 reviews were all reviews accessible from the script shown in Figure 3.2. To be able to load these JSON objects into Python

in a later stage the last row of each file was also removed as it only contained information about what had been collected and was not a review.

3.2.2 Filtering on keywords

The second step was to carry out initial filtering for each of the keywords. This filtering was done using a Python script and Regex. During the filtering, all keywords were evaluated on their own and all reviews that used the specific keyword were saved in a list. This means that if a review used more than one of the keywords it was saved in the list for both of them. For this filtering, only presence was considered, that is, a review was selected if the keyword was found at least once within it. The full script used is found in Appendix C.

3.2.3 Keyword validation

After filtering the next two steps were done to lower the number of reviews to be able to later pick relevant ones for further analysis. The first step in this reduction was to go through about ten reviews for each keyword to see if the reviews seemed to capture comments about readability. A commit was considered to be about readability if the comment were directly about the code and the code quality. If the keyword did not seem to indicate discussions or commits about readability, the keyword was removed from the set of keywords.

3.2.4 Filtering on commits including Java-files

From this new set of keywords, another filtering was done where only reviews that contained Java files were kept. Information about which files were changed for each patchset existed in the JSON-object [1]. If one Java file was found within the review the review was kept. This filtering was done in Python as well and the full script is found in Appendix D.

3.2.5 Analysis of reviews

The last step in the mining study was to pick one review from each keyword and examine them in more detail. The reviews were picked at random so that each keyword was included at least once. These reviews were then examined to analyse all the comments made (inline and patchset comments), how the code looked when the comment was placed and how the code looked after the comment. This was carried out to see what changes happen in code to enhance readability and with that answer RQ2, changes to enhance readability.

Chapter 4
Results

In this section results from the interviews and mining study are presented in relation to each research question. As the mining study used keywords from the interviews the extracted keywords are also presented.

4.1 RQ1: Aspects of readability

From the interviews the following aspects were found. In total six interviews were done and will below be referenced as P1-6. The quotes are slightly edited for grammar with carefulness to not change any meaning.

4.1.1 Naming

The naming of variables, functions and classes was something that was discussed in all of the interviews:

> *It starts with the naming. A class with the name describing what it does, a function with the name describing what it does. If you move away from that, cheating a bit or being lazy, you're going down.* (P3, general comment) (1)

> *The first function here is just called <function name> but that's not really what I'm doing. So you need to be careful when naming your methods. [...] It needs to be really clear what I'm doing and what the caller will be expecting in the result of the method.* (P4, interviewees example) (2)

For the Sudoku code, the interviewees presented different thoughts on the naming of variables and functions. When it came to the variable names "row" and "col", two of the interviewees mentioned that they liked them as variable names, the other interviewees did not

mention them at all. Then when it came to the names for the “checkBox”, “checkRow” and “checkCol” functions some different thoughts were presented:

> *I see from the names what they do so I don’t actually need to read them because I make an assumption and I hope that’s correct.* (P2) (3)

> *The best thing here is the naming, so the naming makes it perfectly clear what you’re doing, exactly what you’re doing.* (P5) (4)

> *The checkRow should be named something else. checkIfAlreadyExistOnRow or something more explanatory of what it does. What does it check? It checks the row, yes, but for what? [...] checkRow; true, what does true mean? What did I check?* (P4) (5)

> *The naming of this method doesn’t match the Java naming convention. Because the naming of a boolean method could be like a question instead. Like ’isNumber...’ This is not a good name.* (P6) (6)

Naming was also discussed when the interviewees expressed that it is important that the name has to match what the function does and that the function should not have side effects that are not reflected in the name:

> *So the activity started off with a name and its activity is pretty broad and if you start off with a class and you add a member that really shouldn’t be, doesn’t fit in with that. But you place them there because it’s convenient or you a bit lazy to create a separate class or a separate component* (P3, interviewees example) (7)

4.1.2 Comments, documentation

Comments were something that was highlighted as important in regards to readability in four of the interviews (in the other two comments was not mentioned):

> *I think that the comment is good because the comment in here say that you need to put some definition for the beginning so you can start to prepare in your mind, okay I see that there is some recursion and backtracking and I start to think okay, recursion what it it in my mind so I have some pre-preparing what I expect to read* (P1, Sudoku example, comment in the beginning of the class) (8)

> *Always comment when you leave the method to some other method why you’re doing this and explain. Like we see here every four-or fifth row there is an explanation* (P6, Sudoku example, solve function) (9)

> *It should have added code explanation, a small comment why they do col+1 and row +1 here* (P5, Sudoku example, next function) (10)

> *I'm not really sure how to validate that this works even. Maybe if we could have more steps and more comments about what is done* (P4, Sudoku example, solve function) (11)

When looking at the Sudoku code the interviewees were generally happy about the number of comments and the content of the comments. In some cases, the interviewee wished for more explanation of the code with more comments, as can be seen in Quote 10 and 11 above.

Regarding documentation, it was brought up that the comments in the Sudoku example were not in Java-doc standard as something negative. One interviewee also talked generally about why they think documentation is needed:

> *So documentation is needed for two reasons. The first reason is that it's information for those who use the code, and the second is to be able to edit the code later when you find bugs or you would like to evolve the code.* (P6, general comment) (12)

4.1.3 Magic values

Wikipedia explains a magic value within programming as "Unique values with unexplained meaning or multiple occurrences which could (preferably) be replaced with named constants" [4]. Magic values came up in the interviews as something negative in regards to readability:

> *Always when I see a number, perhaps not a zero or a one, but when I see an eight like in the top row or the ten. When I see it in a review I usually tell this should be a define, you should define it somewhere and describe it.* (P3, Sudoku example, solve function) (13)

> *Here's like magic constants, magic numbers, what that really means is that it's just there and hopefully correct. That is also something that I truly believe in: that you should use constants and read values from a place where it's easy to control what the meaning of the constant is* (P4, interviewees example) (14)

4.1.4 Organisation

Organisation of code can mean many things. In the interviews for this study, some of the interviewees talked about the subject in the forms of ordering functions and organising code into relevant classes/files:

> *I think the start method should be up by the init method. They are both public and they are closely related. And the comment for init as well is: this is called by the start method. So I would expect init and then the start method. Or maybe even the other way around but really close. Mainly all the public methods at the top because you would want to look at them first* (P4, Sudoku example) (15)

Before I check in code I see if I have changed its behaviour. Does it still do what it says it does? And very often I see that I have completely changed it and added a function to the class that has nothing to do with the class (P3, general comment) (16)

You see like you have a start function and then the end function is in a completely different class somewhere else and that certainly doesn't help readability (P5, interviewees code example) (17)

4.1.5 Indentation, blank space, curly brackets

The participants of the interviews displayed a few different ideas when it came to the use of indentation, blank lines and curly brackets in code. Some presented that they thought it was important to use them as dividers between code parts but was not sure if it was directly connected to readability.

P1 said that:

The code is quite spread out but it's not doing very much. And it being quite spread out is breaking the chain of thought when reading it. (P1, Sudoku-example, solve function) (18)

about the blank lines in the `solve` function in the Sudoku example. But also said:

for each step you make short break [...] make it easier to follow (P1) (19)

Furthermore it was said that:

The code is not formatted which is one of the biggest thing in code readability (P5, interviewees example) (20)

Rule number one always, absolutely always use the automatic formatting in your editor because it's such a simple thing, it's built in to every single editor yet it still something many people fail to do. (P5, general comment) (21)

I don't like the indentation already. [...] One space is not enough. It's inconsistent and I prefer four, some people prefer eight some people prefer two. I like the blocks being a bit more easy to, it's easier to separate them when there is a bigger difference. Then again I don't like eight because its takes to much space (P2, Sudoku example) (22)

In relation to Quote 22, P2 also commented on the use of spaces in the Sudoku code as something that made them stop because it looks unusual to them but later said that:

I'm actually getting used to the spaces (P2) (23)

Curly brackets were discussed concerning the Sudoku example:

> *Consistently I believe in curly braces. "Do you also believe that they help with the readability of the code?" Yeah because they are also used if you have a, no that's kind of more indentation, because if you have many lines Android Studio can help you close sections and the cursor you can find where this if-statement ends.* (P4, Sudoku example, question asked by interviewer in quotation) (24)

> *Usually, it is standard practice in Java to have the brackets on the same line as the function. But that is I think is a personal opinion and doesn't affect readability.* (P5, Sudoku example) (25)

4.1.6 Lines of code

Lines of code (LOC) were primarily discussed in two different ways in the interviews. First is the expression that the check functions in the Sudoku example were easy to read as they were small:

> *These are really really small. I think I would be fine with this. "Because of their size?" Yeah* (P3, Sudoku example, check functions, question asked by the interviewer in quotation) (26)

> *It's not complex code, it's very simple code it's very standard code, it's very short code.* (P5, Sudoku example, check functions) (27)

The other way LOC where discussed is that a few interviewees expressed that they do not mind longer functions if that meant easier code:

> *One thing that I do prefer that may be not so common is I actually don't mind long functions. Which is a tip that is often given. [...] if you have a lot of small functions that is only called from one place it can be very hard to follow the code. But if you have a long function that does this thing, then this thing, then it's clear that this is happening in sequence.* (P2, part of answer to the last question of the interview) (28)

> *You're writing Java so that probably means you don't have any concurrent super thing going on so we don't need to save code lines etc. So write more lines instead of fewer to add readability.* (P6, part of answer to the last question of the interview) (29)

> *Writing cool complex code is most often a mistake because readability comes in hand when updating the method. [...] Cool code is like writing things as few lines as possible.* (P6, part of answer to the last question of the interview) (30)

When discussing the solve function in the Sudoku example one of the interviewees mentioned that when they saw that a method had many LOC, they expected it to be more complex and contain a lot of information. When the function then mostly consist of blank spaces and comments it broke their line of thought as they were in the mindset that the function would be more complex:

> *because it's a super short function but they put it long because you don't get a lot of information from that function but they put it quite long so the feeling when you read something that is very long like that it needs to keep a lot of information.* (P2, Sudoku example, solve function) (31)

4.1.7 Inline code

In one of the interviews, the interviewee discussed an example of code that had an anonymous class definition inside of a function that also used threading.

> *This structure tripped me up when I was reading it and it would be better to have as a named class to avoid confusion to what runs where.* (P2, interviewees example) (32)

> *I would break it out to make the class not inlined or not anonymous* (P2, interviewees example) (33)

> *Since the piece of code you're going to schedule is inline here it's easy to confuse with something that is run inline.* (P2, interviewees example) (34)

4.1.8 Threading

One of the interviewees talked about readability problems when it came to threading:

> *You normally expect code to be linear in some way.* (P2, interviewees example) (35)

> *you have a synchronous so you have some kind of locking and you post something and just form that you may from a quick glance think that all this in in the synchronised block but it really isn't because it run on another thread or on this thread but at a later time. It's not obvious which thread it will run on.* (P2, interviewees example) (36)

4.1.9 Consistency, familiarity, convention

Consistency was discussed in the interviews as something important for readability:

> *I would never have a space here but it's used throughout the code in the same way [...] the consistency is good and clear.* (P6, Sudoku example, space between code ending on line and semicolon) (37)

> *I think this feels like you know if Sudoku lines and boxes and columns are all nine, here you are using both eight and ten which both deviates from nine and there is no reason to.* (P4, Sudoku example, solve function) (38)

Familiarity were both discussed in the form of using familiar names as well as familiar code style:

[...] Oracle they very much like defining the standard and there is a very clear standard on how Java code should be written and follow that it can very much help when the next developer comes and looks at the code because then their immediately familiar with the concepts (P5, general comment) (39)

This one makes me feel more comfortable, the checkRow, checkCol and checkBox and the first one I don't feel good as it's not familiar to me to put a lot something like that. (P1, Sudoku example, check functions and createModel function) (40)

You recognize the pattern (P2, Sudoku example, checkRow function) (41)

Following convention was also something that was expressed as important by the interviewees:

[...] you have a language like Java that is supposed to be object-oriented and then you have everything being static it makes it very difficult to follow what the program does. [...] you expect a Java program to be object oriented and you have these objects but then you have static variables that bypass the object oriented methodology and that's an area I've seen a lot a bugs appear (P5, interviewees example) (42)

4.1.10 For-each loops

In one of the interviews, it was discussed that the interviewee believed that using for-each loop were to prefer over normal for loops as you then could work with named objects:

I've highlighted that I'm working with <thing> and the <thing's> properties, and if we compare to the previous one it's only handled as a list and some properties of a list. So in this case it's easier even if I have the same two loops it's much easier because I say for a <thing> in some list of <things> I'm getting their properties and for each property I'm checking a status. Even if I don't actually check what the code does I'm getting the three main headings of what I'm doing. (P4, interviewees example) (43)

4.2 Keyword extraction

From the interviews, a set of keywords were extracted that can be seen in Table 4.1. In the table, some words are written with the syntax (word1 |word2) as they are variations of the same word/meaning and was treated together.

The keywords were extracted from the interviews by looking at what types of words the interviewees used to talk about the code. Below are a few quotes picked from various interviews to exemplify how the words were used when selected to be a keyword. The selected keywords are marked in bold.

For each step you make a short break or something to try to separate the steps from each other [...] and that make it ***easier to follow*** (P1) (1)

(easy|easier) to follow	(hard|harder|difficult) to follow
easy|easier	hard|harder
clear|clearer	needs to be (clear|clearer)
organize	(easy|easier) to read
(hard|harder|difficult) to read	(easy|easier) to understand
read	understand|understanding
confusing|confuse|confusion	misunderstand|misunderstanding
familiar	comfortable
spread out	compact
long	complex|complexity
comment|comments	name|naming
make sense	obvious
inline	easy to miss
break out	inconsistent|inconsistency
consistent|consistency	unusual
repeat|repetition	verbose
weird	(doesn't|don't) fit in
(shouldn't|should not) be there	should be there
indentation|indent	unnecessary
fit in	create separate
belong	magic (number|constant|string)
unreadable	readable
readability	standard
explanatory |explain	guideline
(code|coding) style	formatted|formatting |format
refactoring	practice
convention	system evolution
documentation	

Table 4.1: Set of keywords

Quote 2 below is an answer to the question "If you were to implement that change, what would you write in the commit message?".

> ***Refactoring*** *code and removing static* (P5) (2)

> *I don't like the* ***indentation*** *already. "What is it about the indentation that you don't like?" One space is not enough. It's* ***inconsistent*** *and I prefer four, some people prefer eight some people prefer two.* (P2, interviewers question in quotation marks) (3)

> *It's kind of not* ***obvious*** *piece of code* (P2) (4)

> *Maybe I would put a less than and equal to nine here to make it a little* ***clearer*** *in that case* (P2) (5)

> *My issue in this one, if I go to the actual reading, is that you can get things in many levels and I think* ***naming*** *and restricting what you do in one line of code could be very good to do. Even if you can do it in one row it doesn't necessarily mean that you should because it becomes very* ***unreadable****.* (P4) (6)

> *Write and use naming that make sense and follow a* ***coding guideline****. It could be any guideline as long as there is a guideline and the people you work with are in a agreement that one is used.* (P4) (7)

Quote 8 below was mentioned as something negative in regards to readability.

> *Maybe the checkBox have a lot of* ***magic numbers*** (8)

4.3 RQ2: Changes to enhance readability

In this section results found in the mining study aimed to answer RQ2, changes to enhance readability, are presented. First, the data used in this part is presented with some graphs to help better understand it. Then results from the qualitative work of analysing the reviews are presented.

4.3.1 Understanding the data

In the following section, the stages refer to the filtering part of the mining study (step two-four as can be seen in Section 3.2. This means that stage one below is "filtering on keywords", stage two is "keyword validation" and stage three "filtering on commits that include Java-files". Note that all graphs presented in this section do contain false positives. A false positive here means a review that does not contain comments about readability. The graphs are presented to help describe the context in which the study was carried out.

easy|easier	create separate	inline
belong	unusual	verbose
comfortable	inconsistency	obvious
guideline	spread out	standard
clear	unnecessary	should be there
format	read	repeat
system evolution	name	hard
long	comments	familiar
misunderstand		

Table 4.2: Removed keywords in step three of the mining study

In the mining, a total of 223 688 reviews were downloaded. These were then filtered on the presence of each keyword. In Figure 4.1 it is presented how many reviews existed for each keyword in the different stages of filtering. As some keywords had a much larger presence, the graphs are presented in a Log-scale.

From the graphs it is shown that although the number of keywords and number of reviews get smaller the distribution is between the words are just about the same.

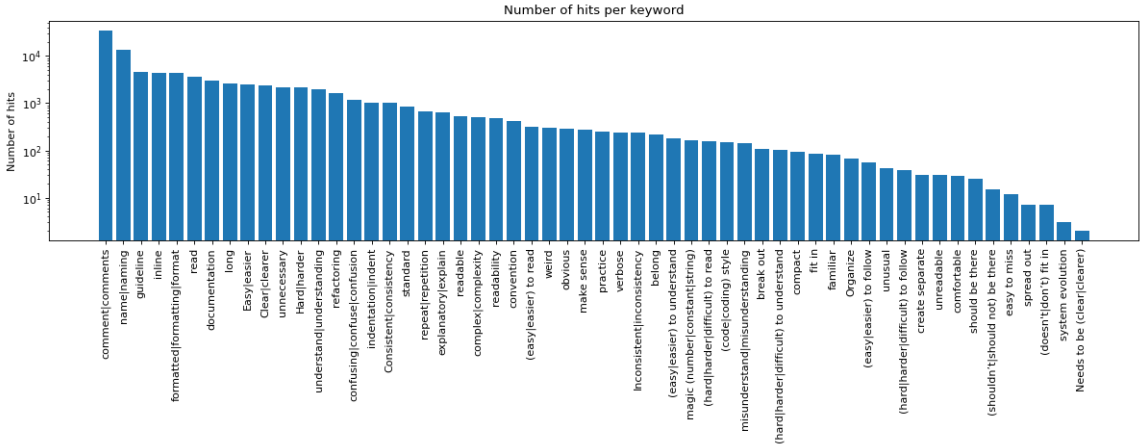
In the second stage, 24 of the keywords were removed. For example "create separate" was removed as discussions with the keyword was mostly about creating separate tickets/commit. The word "unusual" was removed as it mostly referenced unusual values and error codes received/passed to the code. All keywords that were removed can be found in Table 4.2.

The number of comments (inline comments, patchset comments and commit message) that were in a review in relation to how many keywords were in the same review was also considered. This measurement was done for each step in the filtering. The measurement was produced to see if there was any connection between more comments and more keywords. The result can be found in Figure 4.2. As can be seen in the figures there seem to be a slight increase of keywords when the number of comments is increased. The same thing holds even when the number of reviews get smaller. As some outliers existed in this data the graphs are presented in a Log-scale.

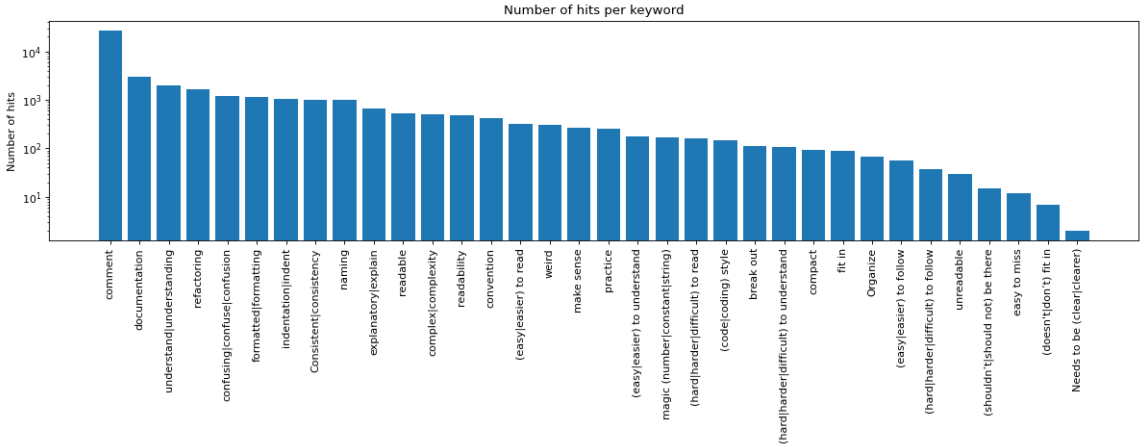
Lastly, there was a calculation of how many comments there were in a conversation in relation to how many keywords that conversation had. A conversation was defined as inline code comments that were about the same line of code in the same patchset. Like the rest, this measurement was done for each step in the filtering process and the results can be found in Figure 4.3. From the figures, there do not seem to be an increase in keywords even though the conversation is longer. The data presented in Figure 4.3 together with the data from Figure 4.2 could indicate that more comments are made about readability when the total number of comments are increasing, but there seem to only be short discussions about readability. No conclusions can though be drawn as the set of reviews are false positives.

4.3.2 Qualitative analysis of reviews

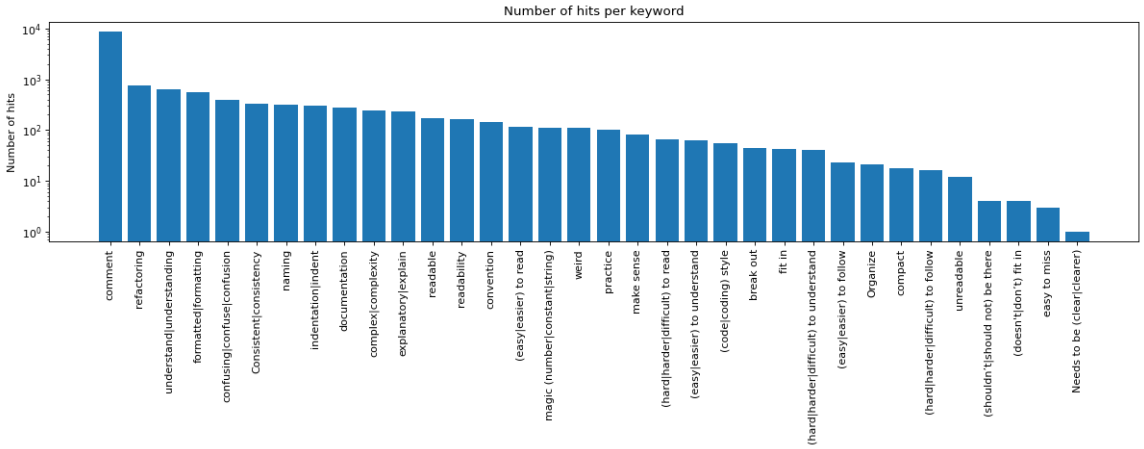
When analysing the reviews several comments were found that were about readability. Most of them were found by the keywords but there also existed some that were about readability but used different phrasing than the exact keywords used. In this section, some review comments will be presented and the change they imposed described, as well as some general



(a) Stage one

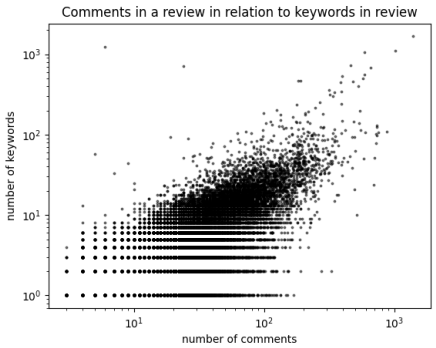


(b) Stage two

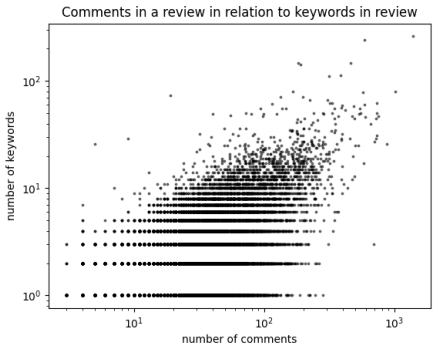


(c) Stage three

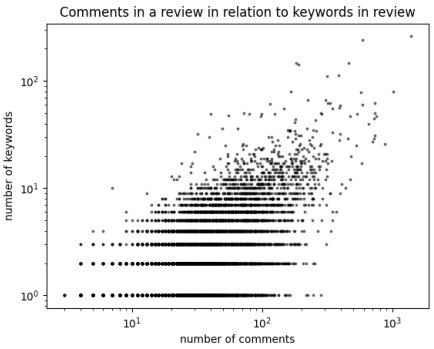
Figure 4.1: Number of reviews for each keyword, per stage in filtering process



(a) Stage one

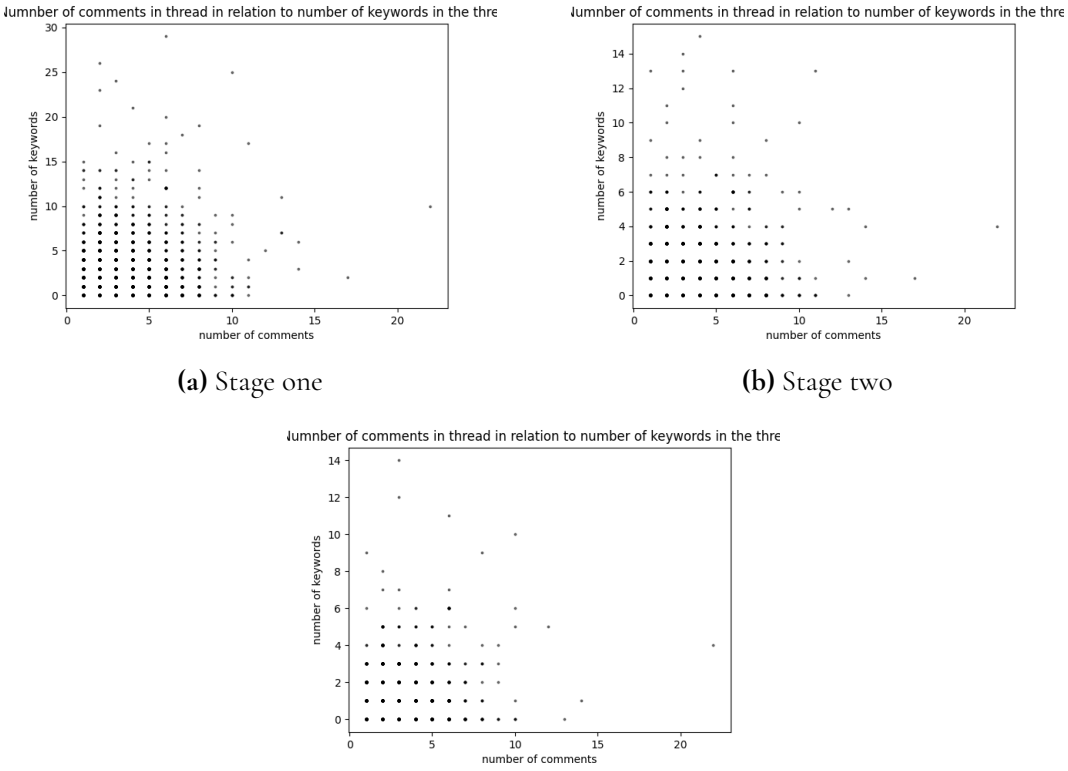


(b) Stage two



(c) Stage three

Figure 4.2: Number of comments and keywords in a review, per stage in filtering



(a) Stage one

(b) Stage two

(c) Stage three

Figure 4.3: Number of comments in a discussion and keywords in that discussion, per stage in filtering

findings.

The following presented formatting is used in this section: first, the comment from the review is presented then below is a box with a qualitative assessment about the content of the code when the comment was made on the left-hand side and how it was changed on the right-hand side. RX (e.g R1, R2) refer to different reviewers in the discussion.

input ->str, Since all the parameters are "input"-parameters, it does not make sense to call only one of them 'input'. If you insist on using 'input' as a parameter name, then all parameters should be called inputX, like 'input1', 'input2' etc. However, such naming is not very informative. In particular, if a method makes some general manipulation on a String in-parameter, as is the case here, a proper name of the parameter is 'str'. (1)

A function had a String as input named 'input', it also had another String input called 'algorithm'. It then did some manipulation of the string according to the specified algorithm	The parameter 'input' was renamed to 'str'

Naming was also commented in Review 2 in this way:

R1: *I would prefer more descriptive member variables names.*
R2: *I understand your concern. The reason here is (as you pointed out verbally) that we send these references into many places which will lead to problem 1 reported in this file (long lines). For that reason we have chosen a shorter representation. Would it be acceptable as is?* (2)

A variable was named using only two letters	No change was made

formatting (3)

This comment was made on a line where a function head with all its parameters passed the line limit	A line-break was added to the function head

Similar comments as in Review 3 could be found in other reviews where the reviewer commented on, for example, missing spaces between parameters, missing space when doing a cast, `(String)variable` instead of `(String) variable`, missing blank line between imports from different sources. Most times these things were fixed but sometimes they were left as is.

R1: *Is this really needed here?*
R2: *It's here for clearer code and improved code readability. Does not have other purpose than that* (4)

A function that only makes one call to another function that uses and returns the result from that call. The outer function takes no parameters the inner takes two.	No change was made

remove if we don't need this (5)

A constant declared empty as all lines in the declaration was out-commented	constant removed entirely

Similar comments as in Review 5 can be seen in other reviews. Functions, variables or constants that no longer is used is asked to be removed and then get removed. In some cases, the functions are kept but it is explained why they are needed.

Extract constant "GMT" (6)

The string "GMT" was being used in a function call	"GMT" was made to a constant and the constant used in the function call instead.

The extraction of constants as can be seen in Review 6 above was also a quite common phenomenon. If strings or numbers were used in many places there was a comment wishing that they instead become a constant.

Java docs for class, methods and members (7)

Javadoc was missing for a public function	Javadoc added

As in Review 7 above, Javadoc was often commented on if it was missing or if the functionality of the function had changed and the documentation was not updated. Spelling mistakes in Javadoc was also often commented on.

perhaps some refactoring (8)

Repetition of same lines of code nine times for different types of sensors	All sensor put in an array and the similar lines are done in for-loop instead, reducing the number of lines and removing duplicated code

Comments regarding repetition, as in Review 8 above, in code were also something that there were comments on in other reviews. Other fixes for this are, for example, breaking code out into functions.

Move the declaration of <variable> closer to the code that uses it (Comment from SonarQube[1]) (9)

A variable was declared on row two in a function but not used until row six	The declaration was moved closer to the usage of variable

For some reviews, SonarQube was used to help review the code. The comments made by SonarQube was sometimes fixed but often ignored.

R1: *Imo*[2] *easier to follow with one return*
R2: *Agree. Otherwise it just ridiculous to have the variable...* (10)

The function returned a value in an if-statement in the middle of the function while before initialising a variable meant for the return	The variable is set to correct value in the if-statement and function now only have one return-statement

else if () instead of else { if() {? (11)

An else-statement only contained an if-statement	Changed to an else if

please group constants at the top, it is easy to miss the other fields (12)

Constants were mixed with normal parameters at the top of the class	Constants now grouped together at the top

[1] `https://www.sonarqube.org/`
[2] in my opinion

Assign them before, so it is more readable. Readability is more important than micro-optimizations. (13)

Here two variables were assigned within the condition of the if-statement	The variables were assigned their value before the if-statement

R1: *format*
R2: *format mall says ok…* (14)
R1: *still think it's more difficult to read*

A line was long and therefore was divided into two lines with only the argument and the parenthesis around the argument on the second line	No change

Chapter 5
Discussion

In this section, the results from the study will be discussed. They are discussed from their relation to the readability models presented in Section 2.1 and some other interesting features will be presented.

5.1 Relation to readability models

In related work, it's stated by Fakhoury et. al. [11] found that four of the existing models for readability seem to fail to capture developers efforts to enhance readability. Riberio and Travassos [15] also found contradictions among four of the models. The findings from Fakhory et. al. and Riberio and Travassos overlap by both of them looking at the model from Buse and Weimer and Posnett et. al. The following section will look at how the results from this study compare to the models described in section 2.1.

In regards to the readability models, there are some things found by the interview study that are not reflected at all in the models. These things are the organisation of code, threading, inline code and familiarity/convention. Besides this there are some things that are not explicitly mentioned in any of the models but can be argued that they are present indirectly, as discussed below. Table 5.1 outlines which of the aspects presented in Section 4.1 that are explicitly represented in the models.

For-each loops and use of magic values were also discussed in the interviews and is something that can not be explicitly found in any of the models. It could be argued that for-each loops bring more identifiers and therefore can be seen as a negative aspect in regards to feature "number of identifiers" in the model from Buse and Weimer [7] This is problematic as it in the interview was lifted as something positive to get the extra identifier by using a for-each loop. The same reasoning can go for magic values if they are replaced by named constants (which was suggested in the interviews). The change would reduce the number of values that are used in the code but adds an identifier. The change from magic values to named constants, in regards to the model from Buse and Weimer, would not sufficiently enhance the

Mentioned	Not explicitly mentioned
Naming	Threading
Lines of code	Inline code
Comments, documentation	Organisation
Indentation, blank lines, curly braces	Magic values
	Consistency, familiarity, convention
	For-each loops

Table 5.1: Aspects from this study in relation to current readability models

readability according to that model.

Then some features are mentioned in the models but are inconsistent between how the model treats them as opposed to how the developers reasoned about them in the interviews. One of these features is lines of code (LOC). LOC is mentioned in four of the models discussed in Section 2.1. The models from Buse and Weimer, Choi et. al and Jørgensen state that a larger number of LOC reduces readability and the model from Posnett et. al. [13] have a positive indication for LOC. From the interviews, you can see two different views on the problem of LOC: one is that some of the interviewees expressed that they did not mind a longer function if that meant easier statements and the other is that some interviewees expressed that the check functions from the Sudoku example were easy to read as they were short. These two views are not deemed contradictory but are two interesting viewpoints on the issue of LOC. One interviewee also explicitly stated that if a function is only a few lines there should be no problem understanding it. It could be argued that that statement though leads more into understandability more than readability. None of the interviewees suggested that a greater number of LOC would negatively impact the readability of the code as three of the models suggest. Riberio and Travassos [15] say in their article that "*It is important to note that more lines tended to have worse readability perceptions for experts than for novices.* " in this study, however, no difference could be found for LOC in regards to the experience level of the interviewees.

Scalabrino et. al. [17] and Dorn [10] started to introduce textual features to the readability models. In Dorn's model, he introduced textual features and suggested that the use of natural language within identifiers was a way to enhance readability. Scalabrino et. al. then built on this and added more textual features. Two of the features from the model Scalabrino et. al. is a measurement of the overlap between words used in the comment and the identifiers in the function and the readability of the comments. In regards to the readability of the comments, it could be seen from the mining study that reviewers often commented on spelling mistakes in the comments. Other than spelling mistakes the findings from the interviews and the mining study found that developers seem to value the descriptiveness of the comments, something that the measurement of overlap between words used in the comment and the identifiers could partially be an indicator of.

Comments also exist in the other readability models, where Choi et. al.[8] and Buse and Weimer claim that more comments add to the readability and the model from Jørgensen claim that extensiveness of comments have a negative impact on readability. In the mining study it was often commented on if Javadoc was missing and in the interviews, some of the interviewees expressed that they wished for more comments in some parts of the code. This

supports the claim that more comments add to readability.

Another textual feature in code that is lifted in both the interviews and readability models are identifier names. Scalabrino et. al. and Dorn propose that using words that can be found in the dictionary in identifier names enhances readability. Buse and Weimer, DeYoung and Kampen and Jørgensen also include identifier/variable names in their models but only mention them in the form of the length of the identifiers. The results from this study stress the descriptiveness of the names rather than the use of words from the dictionary. From the mining study, you can see an instance where a reviewer commented on the fact that a parameter to a function was simply called "input" (Review 1). The word "input" is present in the dictionary and would therefore, according to the models, be enhancing the readability while it from a developers perspective is not descriptive enough.Concerning the length of the identifier names, you can also see one of the review comments about using two-letter abbreviations for variable names not being descriptive enough (Review 2). This correlates with the belief that longer variable names enhance readability, but it is worth stressing that the request was for a more descriptive name and not necessarily a longer. In the paper from Riberio and Travassos, they observed that less experienced developers had a bigger difficulty comprehending shorter names on identifiers than more experienced developers. No such difference between more and less experienced developers could be found in this study.

5.2 Readability vs. understandability

Posnett et. al. briefly discusses the distinction between readability and understandability [13]. They say that understandability is linked to the semantic issues of code while readability is linked to the syntactic. When looking at some of the answers and topics from the interviews in this study this difference does not seem to be clear to the developers. For example, the interviewee that discussed threading and inline code lean more into semantic issues of the code as the code does not seem to work as the interviewee first interpreted.

This uncertainty about what readability should be defined as is something that also can be found in the state-of-art and many papers about readability use slightly different definitions [6]. If the developers interviewed in this study had different definitions about readability in mind this might have lead them to highlight things that, for example, could lean more into understandability.

5.3 Expectations

One finding from this study also involves expectations of code. One of the interviewees mentioned (Quote 8) that when you read a comment for a class you can get an expectation of what will come in the class and then be prepared for that. Another way that it was discussed in the interview is that when the developer saw that a function had many lines they expected it to be more complex, when this expectation was not met it broke the chain of thought for the reader of the code.

You can also see discussion about expectations in two other places from the interview quotes in section 4.1, convention and naming. There it is discussed that when breaking convention (Quote 42) makes the code harder to follow. In regards to naming, the aspect of

expectations are brought up in the meaning that when reading a function name it should be clear what to expect from the function.

5.4 Visuals from working environment

When working with code, developers often use some sort of Integrated Development Environment (IDE). These IDEs can help the developers with things like formatting but is also responsible for syntax highlighting. In the interviews, one of the interviewees mentioned that the code is easier to look at using an IDE. The automated formatting from an IDE can help with things like indentation, spaces and more. This is something that developers seem to think is important from both the interviews and the mining study. Visual features provided by an IDE are a part of the model proposed by Dorn, in this study only one of the interviewees discussed the visual features from an IDE.

5.5 Threats to validity

In this section the threats to the study's validity is discussed.

5.5.1 Internal validity

In some of the interviews, the author was the sole interviewer. This means that the interpretation of the interviews was mostly done solely by the author and there is a risk for wrongful interpretations. This risk is somewhat mitigated by the fact that interviews were recorded and one of the academic supervisors for this study also looked at them. That means that the results are verified by one more person. Worth mentioning as well is that even if both the author and supervisor had taken part in all of the qualitative data gathering and analysis, it is still possible in qualitative analysis like this that a different grouping of people may have drawn other conclusions.

The reviews collected in the mining study might not contain the entire discussions as developers usually work closely together and can have discussions in person. This could affect the results by not giving the full picture of the problems found for RQ2.

The results from the interview study might also be affected by the example chosen by the author. If another example was chosen the discussions could have become about other elements of the code. This threat was mitigated by using two examples in every interview where the interviewee brought the second one.

5.5.2 External validity

This study was only made within the Sony company. This means that things like company culture and internal coding standards could affect the results. In one of the interviews, the interviewee said that it is important to follow the convention set by the company. If Sony or some teams then have any specific conventions regarding how to write code this could affect the results.

This study only considered the Java programming language. This means that there is no guarantee that the results found in this study are transferable to other programming languages. However, some of the results should be transferable between languages, e.g. the finding concerning naming of identifiers.

Many of the results from this study come from the six interviews performed. This means that no conclusions can be drawn about whether all aspects of code readability (RQ1) have come up in this study. The things these developers thought was important for readability might not be represent for a larger group of developers. As to the RQ2, it is worth noting that only a few reviews were analysed and the result presented are only a few examples of what changes happens in the code to address readability concerns.

Chapter 6
Conclusions

In this master thesis, the concept of readability was explored from developers' perspective. The results from the study show signs of a discrepancy between the existing models for code readability and how developers reason about readability. This discrepancy is shown through, for example, how developers highlight the importance of identifier names to be descriptive whereas the models focus more on the length of the identifier or if the identifier name consists of words existent in the dictionary.

The study also looked at what kind of changes happens in the code to enhance readability. In this study, it was found that changes like formatting, reducing repetition, etc. are some of the things developers do to enhance the readability of the code.

6.1 Future work

As this study only has been exploratory and on a small set of people and only within one company, more research needs to be done to see if any statistical conclusions can be drawn. To be able to create a more accurate model the aspects found in this study need to be examined in more detail to see if the same aspects can be found for a larger set of developers in different companies.

References

[1] Gerrit code review - json data. https://gerrit-review.googlesource.com/Documentation/json.html#change Retrieved at 2021-05-25.

[2] gerrit query. https://gerrit-review.googlesource.com/Documentation/cmd-query.html Retrieved at 2021-05-25.

[3] Halstead complexity measures. https://en.wikipedia.org/wiki/Halstead_complexity_measures, Retrieved at: 2021-05-27.

[4] Magic number (programming). https://en.wikipedia.org/wiki/Magic_number_(programming) Retrieved at: 2021-05-25.

[5] K.K. Aggarwal, Y. Singh, and J.K. Chhabra. An integrated measure of software maintainability. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 235–241, 2002.

[6] Andreas Bexell. Software source code readability: A mapping study. Master's thesis, Blekinge Institute of Technology, 2020.

[7] R.P.L. Buse and W.R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010.

[8] S. Choi, S. Kim, J.-H. Lee, J.A. Kim, and J.-Y. Choi. Measuring the extent of source code readability using regression analysis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10963 LNCS:410–421, 2018.

[9] G.E. DeYoung and G.R. Kampen. Program factors as predictors of program readability. In *Proceedings - International Computer Software and Applications Conference*, pages 668–673, 1979.

[10] Jonathan Dorn. A general software readability model. Master's thesis, University of Virginia, Charlottesville, 2012. http://www.cs.virginia.edu/~{}weimer/students/dorn-mcs-paper.pdf.

[11] S. Fakhoury, D. Roy, A. Hassan, and V. Arnaoudova. Improving source code readability: Theory and practice. In *IEEE International Conference on Program Comprehension*, volume 2019-May, pages 2–12, 2019.

[12] Anker Helms Jørgensen. A methodology for measuring the readability and modifiability of computer programs. *BIT Numerical Mathematics*, 20(4):393–405, 1980.

[13] D. Posnett, A. Hindle, and P. Devanbu. A simpler model of software readability. In *Proceedings - International Conference on Software Engineering*, pages 73–82, 2011.

[14] Rogers Y. Sharp H. Preece, J. *Interaction Design: Beyond Human-Computer Interaction*, pages 373–383. Wiley, Hoboken, NJ, 2 edition, 2007.

[15] T.V. Ribeiro and G.H. Travassos. Who is right? evaluating empirical contradictions in the readability & comprehensibility of source code. In *CIbSE 2017 - XX Ibero-American Conference on Software Engineering*, pages 99–112, 2017.

[16] Colin Robson and Kieran McCartan. *Real world research : a resource for users of social research methods in applied settings*. Wiley, Hoboken, fourth edition edition, 2016.

[17] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6), 2018.

[18] Y. Tashtoush, Z. Odat, I. Alsmadi, and M. Yatim. Impact of programming features on code readability. *International Journal of Software Engineering and its Applications*, 7(6):441–458, 2013.

[19] Peter Tellenbach. Backtracking to solve a sudoku puzzle. `https://www.heimetli.ch/ffh/simplifiedsudoku.html`, Retrieved at: 2021-03-02.

Appendices

Appendix A
Sudoku code example

```
/*************************************************************/
/*                                                           */
/* An applet to demonstrate recursion and backtracking       */
/* ==================================================        */
/*                                                           */
/* V0.3   18-MAR-2007  P. Tellenbach   www.heimetli.ch       */
/*                                                           */
/*************************************************************/

import java.awt.* ;

/**
 * Solves a sudoku puzzle by recursion and backtracking
 */
public class SimplifiedSudoku implements Runnable
{
   /** The model */
   protected int model[][] ;

   /** Creates the model and sets up the initial situation */
   protected void createModel()
   {
      model = new int[9][9] ;

      // Clear all cells
      for( int row = 0; row < 9; row++ )
         for( int col = 0; col < 9; col++ )
            model[row][col] = 0 ;

      // Create the initial situation
      model[0][0] = 9 ;
      model[0][4] = 2 ;
      model[0][6] = 7 ;
      model[0][7] = 5 ;
```

```
35
36          model[1][0] = 6 ;
37          model[1][4] = 5 ;
38          model[1][7] = 4 ;
39
40          model[2][1] = 2 ;
41          model[2][3] = 4 ;
42          model[2][7] = 1 ;
43
44          model[3][0] = 2 ;
45          model[3][2] = 8 ;
46
47          model[4][1] = 7 ;
48          model[4][3] = 5 ;
49          model[4][5] = 9 ;
50          model[4][7] = 6 ;
51
52          model[5][6] = 4 ;
53          model[5][8] = 1 ;
54
55          model[6][1] = 1 ;
56          model[6][5] = 5 ;
57          model[6][7] = 8 ;
58
59          model[7][1] = 9 ;
60          model[7][4] = 7 ;
61          model[7][8] = 4 ;
62
63          model[8][1] = 8 ;
64          model[8][2] = 2 ;
65          model[8][4] = 4 ;
66          model[8][8] = 6 ;
67      }
68
69      /** This method is called by the start-method  */
70      public void init()
71      {
72          createModel() ;
73      }
74
75      /** Checks if num is an acceptable value for the given row */
76      protected boolean checkRow( int row, int num )
77      {
78          for( int col = 0; col < 9; col++ )
79              if( model[row][col] == num )
80                  return false ;
81
82          return true ;
83      }
84
85      /** Checks if num is an acceptable value for the given column */
86      protected boolean checkCol( int col, int num )
87      {
88          for( int row = 0; row < 9; row++ )
89              if( model[row][col] == num )
90                  return false ;
```

```
91
92          return true ;
93      }
94
95      /** Checks if num is an acceptable value for the box around row
        and col */
96      protected boolean checkBox( int row, int col, int num )
97      {
98          row = (row / 3) * 3 ;
99          col = (col / 3) * 3 ;
100
101         for( int r = 0; r < 3; r++ )
102             for( int c = 0; c < 3; c++ )
103             if( model[row+r][col+c] == num )
104                 return false ;
105
106         return true ;
107     }
108
109     /** This method is called by main to start solving the sudoku */
110     public void start()
111     {
112         init() ;
113         // This statement will start the method 'run' to in a new
        thread
114         (new Thread(this)).start() ;
115     }
116
117     /** The active part begins here */
118     public void run()
119     {
120         try
121         {
122             // Start to solve the puzzle in the left upper corner
123             solve( 0, 0 ) ;
124         }
125         catch( Exception e )
126         {
127         }
128     }
129
130     /** Recursive function to find a valid number for one single
        cell */
131     public void solve( int row, int col ) throws Exception
132     {
133         // Throw an exception to stop the process if the puzzle is
        solved
134         if( row > 8 )
135             throw new Exception( "Solution found" ) ;
136
137         // If the cell is not empty, continue with the next cell
138         if( model[row][col] != 0 )
139             next( row, col ) ;
140         else
141         {
142             // Find a valid number for the empty cell
```

```
143          for( int num = 1; num < 10; num++ )
144          {
145             if( checkRow(row,num) && checkCol(col,num) && checkBox(
    row,col,num) )
146             {
147                model[row][col] = num ;
148
149                // Delegate work on the next cell to a recursive
    call
150                next( row, col ) ;
151             }
152          }
153
154          // No valid number was found, clean up and return to
    caller
155          model[row][col] = 0 ;
156       }
157    }
158
159    /** Calls solve for the next cell */
160    public void next( int row, int col ) throws Exception
161    {
162       if( col < 8 )
163          solve( row, col + 1 ) ;
164       else
165          solve( row + 1, 0 ) ;
166    }
167
168    public static void main( String[] args )
169    {
170     SimplifiedSudoku simpleSudoku = new SimplifiedSudoku();
171     simpleSudoku.start();
172   }
173 }
```

Listing A.1: Sudoku code example for inital interviews, used with permission [19]

Appendix B
Interview protocol

The interview protocol was constructed by using different topics to discussed and then having some prepared follow-up questions that could be asked.

- What is your professional background and role at Sony?
- In regards to the example they brought with them:
  - Could you explain shortly what this code does?
  - Why did you choose this specific snippet?
  - What specific elements of the code makes it readable/not readable?
  - If you were to implement that change, what would you write in the commit message? (In regards to if they mention something they would like to change)
- In regards to the Sudoku code example:
  - What are your general first impressions of the code? What makes it readable or not readable?
  - What elements of the code make it readable?
  - When looking at the solve-function, are there any elements of the code here that you think is examples of good or bad in regards to readability?
  - If you were to implement that change, what would you write in the commit message? (In regards to if they mention something they would like to change)
- If you were to talk with a person new to programming, what would you encourage them to think about when writing code to make it readable?

Appendix C
Filtering on keywords

```
1
2  def filterCommits(entry, rx):
3      ##Commit message
4      if rx.search(entry['commitMessage']):
5          return True
6
7      ##Inline code comments per patchset
8      for patch in entry['patchSets']:
9          if 'comments' in patch:
10             for comment in patch['comments']:
11                 if rx.search(comment['message']):
12                     return True
13
14     ##Comments on entire patchSets
15     for comment in entry['comments']:
16         if rx.search(comment['message']):
17             return True
18
19 keywordList = []
20 allKeywords = ""
21 with open('keywords', encoding='utf8') as words:
22     for line in words.readlines():
23         word = r'\b' + line[:-1] + r'\b'
24         keywordList.append(word)
25         allKeywords += "(" + word + ")|"
26 allKeywords = allKeywords[:-8]
27 keywordList = keywordList[:-1]
28
29 dataPerKeyword = dict((word,[]) for word in keywordList)
30
31 for filepath in glob.glob(os.path.join('<path_to_data>', '*.json'))
       :
32     with open(filepath) as f:
33         data = json.load(f)
```

```
34
35      for word in keywordList:
36          rx = re.compile(word, re.IGNORECASE)
37          result = []
38          changeIds = set() #check so no review exist more than once
        for each keyword
39
40          for entry in data:
41              if entry['id'] not in changeIds:
42                  changeIds.add(entry['id'])
43                  if filterCommits(entry, rx):
44                      dataPerKeyword[word].append(entry)
45
46  for key in dataPerKeyword.keys():
47      with open("final_res/" + key.replace(" ", "_")[2:-2] + ".json",
        "w") as outfile:
48          json.dump(dataPerKeyword[key], outfile)
```

Listing C.1: Python code to filter data on keywords, code is released to public domain

Appendix D
Filtering on commits including Java-files

```
1
2  def filter_java(entry):
3      rx = re.compile('.java$')
4      for patch in entry['patchSets']:
5          if 'files' in patch:
6              for file in patch['files']:
7                  if rx.search(file['file']):
8                      return True
9      return False
10
11 ## remove non java commits
12 for key in dataPerKeyword:
13     dataPerKeyword[key] = [e for e in dataPerKeyword[key] if
       filter_java(e)]
14
15 for key in dataPerKeyword.keys():
16     with open("final_res3/" + key.replace(" ", "_")[2:-2] + ".json"
       , "w") as outfile:
17         json.dump(dataPerKeyword[key], outfile)
```

Listing D.1: Python code to filter data to only include reviews containing Java-files, code is released to public domain

EXAMENSARBETE Readability [of code] in practice:
an exploratory study of software professionals' perception of readability
STUDENT Anna Qvil
HANDLEDARE Emma Söderberg (LTH), Luke Church (LTH), Andreas Bexell (Sony)
EXAMINATOR Görel Hedin (LTH)

Läsa kod, jobbigare än man tror?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Anna Qvil**

När man läser en artikel kan det ibland vara fullt av långa konstiga meningar som gör artikeln svår att läsa. På samma sätt kan utvecklare ibland skriva kod som är oläsbar för deras kollegor. Detta är en utforskande studie kring vad det är som gör kod läsbar.

En mjukvaruutvecklare som jobbar i ett projekt med andra lägger ofta en stor del av sin tid på att läsa kod. För att göra det lättare att läsa kod så har det utvecklats en mängd olika modeller ansedda att förklara vad det är som gör kod läsbar. Dessa modeller har senare visat sig inte vara anpassade efter vad utvecklarna själva anser är viktiga aspekter kring vad som ökar läsbarheten av kod. I denna studie har detta gapet mellan modeller och utvecklare utforskats genom intervjuer och genom att titta på de förändringar i kod som sker för att höja läsbarheten.

Resultaten från studien visar viss skillnad mellan modellernas och utvecklarnas tankar kring läsbarhet. Ett tydligt exempel på detta är att flera av modellerna säger att kod är mer läsbar ifall den innehåller färre rader. I några av intervjuerna uttrycktes dock att den intervjuade inte hade något emot längre kod utan föredrog fler rader kod om det betydde att raderna i sig blev mer lättlästa. I en av intervjuerna uttryckets detta som att man inte ska sträva efter att skriva ”cool” kod, d.v.s mer komplex kod för att skriva så få rader som möjligt.

En annan sak som kom upp från intervjuerna var hur läsbarhet påverkades av förväntningarna utvecklarna hade på koden. Dessa förväntningar kom från att läsa kommentarer, namn på variabler/funktioner eller genom att se längden på funktionen. Att ha en tydlig kommentar inför en metod eller klass belystes som något positivt då det gav utvecklaren av en idé av vad de kommer att få läsa och de då kunde sätta sig i rätt ”mindset” inför läsningen.

Utöver detta hittades en mängd olika koncept som utvecklarna ansåg var viktiga för att öka läsbarheten av kod. Detta inkluderade namngivning av klasser, funktioner och variabler, formatering av kod, att koden var konsekvent m.m.

Förändringar som skedde i koden följde delvis resultaten från intervjuerna. Formatering fixades, koden omorganiserades och repetition av liknade kod togs bort.