

## **Code Readability and Reusability**

Date	18 November 2022
Team ID	PNT2022TMID17220
Project Name	Signs with Smart Connectivity for Better Road Safety

# Contents

Abstract .....	2
Sammanfattning .....	3
Introduction .....	6
Problem Statement.....	6
Background .....	7
Origins and Use of Eye-tracking in Understanding Humans.....	7
Applications of Eye-tracking technology throughout his-	
tory .....	7
Visual Attention.....	8
Code readability .....	8
readability Metrics.....	9
The effects of code styling on readability.....	9
Eye-tracking .....	9
Code reading.....	10
Method .....	10
Tools.....	10
Tobii EyeX .....	10
Unity 5 .....	11
Forms .....	11
Data gathering.....	12
Code samples .....	12
Stage 1 - recognising output.....	13
Stage 2 - Identifying errors .....	15
Analyzing the data .....	18
Heatmapping.....	18
Results.....	20
Pre-testing form .....	20
Explain the code 1.....	22
Explain the code 2.....	22
Explain the code 3.....	23
Find the bug 1 .....	25
Find the bug 2 .....	26
Find the bug 3 .....	27
Post-testing form.....	29
Discussion .....	32

Commenting .....	32
Variable naming .....	34
Comparing the "explain the code" results .....	35
Comparing the "find the bug" results .....	36
Conclusion.....	37
Indentation and Syntax highlighting .....	37
Logical naming.....	37
Comments .....	37
Limitations .....	37
Number of testers.....	38
Tobii EyeX .....	38
Future research .....	38
Appendixes .....	38
Pre-test form .....	38
Post-test form.....	41
Code .....	44

## Introduction

Code readability can be defined as a way of determining how easily code can be comprehended by the reader and is closely related to the life-cycle of any software product [1]. It revolves around aspects such as code maintainability, reuse and situations in which a programmer familiar with a code passes it on to another, perhaps less familiar, programmer; a situation in which it is important that this transition is smooth and that the new person can resume work with the code as smoothly as possible.

Using eye-tracking technology it is possible to analyse the readability of code by having subjects with various knowledge in programming participate in a range of experiments and by observing their behaviour and way of thinking towards some particular piece of code. Such knowledge is interesting as it allows for programming languages to further become more interactive and more easily understood by assigning them higher readability. A higher readability, in turn, improves the maintainability, reuse and general functionality of the code. [2]

This report aims to show the correlation between how code appears regarding styling, syntax and various other features that may affect readability and how easily the code may be interpreted by a programmer with concern to his/her level of programming skill. The skill level of the testing programmers will vary from 1 to 5+ years of programming studies/work experience. In order to collect and examine data and obtain relevant results, Eye tracking technology will be used to perform the experiments. More information about the tools and equipment used will be provided in the Methods section.

## Problem Statement

The aim of this report is to explain and identify the correlations between how well a code is written with regards to code formatting, syntax, etc. and how easily the code can be read and interpreted by a programmer. By gathering data and information using forms and eye-tracking technology, the objective is to reach a conclusion as well as make a clear statement about whether code becomes easier to read and interpret or not. Thus, this report aims to answer the following query;

*Can the readability and interpretation of code be visibly improved by adding features such as syntax highlighting, code indentation, comments and/or logical variable naming?*

## **Background**

This section will provide a background from previous, closely related work.

### **Origins and Use of Eye-tracking in Understanding Humans**

The subject of Eye-tracking refers to the method of studying and observing the behaviour of the eyes and gaze, and examining the movement of the eyes. It is an old field of research that was introduced more than 100 years ago. Back then researchers relied on electrodes mounted to the skin around the eye which measured differences in electrical potential to detect eye movement [3]. There were also methods which used large contact lenses. A metal coil installed around the edge of the lens was used to cover up both the clear membrane that covers the front of the eye, as well as the white part of the eye as seen from the outside. The eye movements were then detected by fluctuations in an electromagnetic field when the metal coil moved according to the eyes. Nowadays however, most systems track where someone is looking by using video images of the eye [3]. Eye tracking is useful in many different areas; the most essential ones will be discussed in the sections following this one. [4]

### **Applications of Eye-tracking technology throughout history**

Eye-tracking technology has found its use in many different areas as of today. Whilst Eye-tracking has garnered attention as a tool for gaming and visual entertainment recent years, it has since long been considered a useful tool in medical and psychological research [4]. Within mainstream psychological research, eye-tracking has been used to provide insight into patients ability to problem solve, mental imagery, reasoning and search strategies. HCI (Human computer interaction) uses eye-tracking as a research tool because it provides information about many aspects of cognition. Usability research finds eye-tracking useful to maximise the efficiency of menu-based interfaces, search strategies and features of web sites. To optimise advert design and web banner placement the commercial sector is increasingly utilising eye-tracking technology. Air-traffic-control training uses eye-trackers to make the design of the cockpit controls as fail safe as possible. [3]

## Visual Attention

Visual attention is defined by the psychologist William James as;

*Everyone knows what attention is. It is the taking possession by the mind, in clear and vivid form, of one out of what seem several simultaneously possible objects or trains of thought. It implies withdrawal from some things in order to deal effectively with others.’ [5]*

Eye-tracking is a form of determining visual attention. This phenomenon, visual attention, has been studied for over 100 years, long before eye-tracking technology existed. Back then simple ocular observations and peoples introspective were the basis of the studies. Visual attention has had different definitions according to different researchers, which then can has been combined to try to fully explain the term.

In 1925 Von Helmholtz observed that visual attention had a tendency to wander to new things, he claimed that studying visual attention was mainly concerned with "where" we look. On the contrary, William James, believed that "what" we looked at played a bigger role in determining visual attention. An image can be split up into regions, and James thought that these regions all had different levels of attraction. These two concepts, "what" and "where" formed the basic metaphor for visual attention. In the 1940's Gibson proposed a new, third factor of visual attention generally called "how". Gibsons "how" addresses the ability to "trick" someone by for example making them expect words describing animals and hence making them automatically read "sael" as "seal". Basically "how to react" to certain visual objects based on the persons predefined assumptions.

There are a few more terms that have come up, but they are not as relevant, they are more specific and detailed. The three consisting of "where", "what" and "how" are sufficient and make up the ground pillars of the definition. [5]

## Code Readability

Seeing as different programming languages work and look differently, they should also differ in readability. This was looked into in the report [6], where the programming languages Java and C# were compared. With multiple features and tests as well as input from 15 expert programmers, Java proved to be the more readable of the two. Not only does the actual code style or language affect readability. In [7], the authors looked at what would be the optimal choice for readability in terms of documentation. Documentation in terms of code comments and self-documenting code were compared.

The results showed that experienced professional programmers relied more on the self-documenting code and not so much on internal code comments while students were pretty much the opposite, hence liked having comments throughout the code.

### **Readability Metrics**

A study involving 120 computer science students was conducted in an attempt to come up with a technique for modelling code based on its metrics of readability. The focus of the study was to assign a certain piece of code to the test subjects after which they completed a series of snippets, each snippet graded and corresponding to a unique aspect of the code. For instance, a typical snippet may query the subject about the difficulty of some specific part of the code, and request a grading on a scale from one to ten. The conclusion of the study was that it is very much viable to develop one such technique for modelling code and, in this case, one that shows a significant correlation with more conventional metrics. [2]

### **The effects of code styling on readability**

File organisation, indentation, white space and naming conventions are just a few factors that can be grouped up and defined as code styling, with its purpose to make code more readable and easier to understand. Over half of the lifetime of a software is spent maintaining it and its usually maintained by more than one developer. This is all claimed in [1] where research was done on several open-source Java projects to find out whether the number of violations of a chosen code convention correlated with the projects readability score. They used the metric described in the previous section.

### **Eye-tracking**

There are generally two different ways of monitoring eye movement using an eye-tracker, either measure the position of the eye relative to the head or the orientation of the eye in space. Eye-tracking technology has been used to analyse the way we look at things in a computer environment before. A study comparing three graph representation (force-directed, layer-based and orthogonal) layouts showed that a force-directed approach provided the best visualisation experience [8]. Another study looked at how different representation of number display (percentages versus fractions, digits versus words and rounding versus decimals) affected people with dyslexia [9].

## Code reading

As far as code reading has been researched, [10] looked at how camelCase and underscore\_style (these terms can be read into detail at [10]) impacted human comprehension, with the hypothesis that the choice of style does affect how fast and accurate code is read. Multiple tests were ran on different levels of code complexity. This study showed that experienced programmers were barely affected by the choice but beginners benefited more from camel case.

Another important aspect of code readability is the classification of written code into specific tokens, and observing in what ways the programmer interacts with these. [11] performed a series of studies in which test subjects were to evaluate and recognise written code using eye-tracking technology, and in which the various components of the code were divided into different types, or *tokens*. The purpose of this study was to conclude whether or not programming languages are improving as mediums for human communication. By examining and assembling the *gaze time*, i.e. the time that the test subjects spent reading each specific token, along with other factors, such as the density of the code or subject's skill level in a particular language, etc, relevant comprehension scores may be calculated. Dubochet concluded that it was difficult to perform these experiments as the number of participating test subjects were few, but that the retrieved results were nevertheless useful. The conclusion, although vague, was that programming languages are indeed evolving into better mediums for human communication [11], but the methods of gathering and assembling the data were interesting and may prove useful in this report.

## Method

### Tools

The tools that will be used to gather and assemble the data used in this study will be discussed in this section. The tests will be performed on a computer running Windows 10.

### Tobii EyeX

Tobii EyeX is a device used for eye tracking, mostly aimed at using alongside video games. It will be used to capture how the test subjects look at different code examples, as eye tracking has previously been used in modern psychology to determine patients ability to problem solve. [3]



The eye tracker will record coordinates from the screen that will then be used to determine patterns. The eye tracking device will be used to gain insight into how the test subjects examine the code snippets and if the added code styling features can be visibly noticed. If the results would instead have been simply based on how well each assignment was solved with regards to time and the amount of correct answers, it could be due to the specific test design rather than as an effect of using code styling. Using an eye tracker will thus provide an additional aspect to solidify the results, making it possible to check whether, for example, code comments actually helped the subject out visibly.

## **Unity 5**

Unity is software primarily used to make video games, however, Unity will be used to display the code examples and interact with the Tobii EyeX. [12] Unity was chosen simply because of prior knowledge within the framework as well as Tobii having an already made Unity SDK for the Eyex device. Furthermore, the Unity library and scripts that is provided by Tobii to be used with the EyeX proved to be useful as it gave a deeper understanding of how the Eye tracking works. This allowed for more specific, tailored scripts to be written solely for this research. A link to the repository containing the written code will be provided in the appendix.

## **Forms**

In order to compare results between different users with varying level of coding experience a form will be used before the tests. A second feedback form will be used after the tests. The platform of choice will be Google Forms, for its convenience.

The first form asked the user about their programming experience and what languages they considered their best. Seeing as all the tests will be presented with Java code, they were also asked about their skill/confidence within Java programming. Finally it asked what the user considered the most important aspect(s) of code readability.

The second form had the user "rank" the four different types of improvements used in the tests. These four included syntax highlighting, indentation, proper function/variable naming as well as comments, each given a score from 1 to 10. After the ranking the user chose which step of improvement was more important (from nothing to syntax highlighting and indentation as in step one to two in the tests or going from having indentation and syntax highlighting and adding proper naming and comments) or if both of the

changes had the same significance.

## **Data gathering**

First the user will be presented with the first form. We chose to have a form instead of basing it on how long someone has studied/worked for simply because people may have much or no experience prior to their time in school/work. Next up the user will be presented with code examples within Unity. While the user is looking at the examples, the Tobii EyeX will record the users eye movement. Once the user feel like they recognise what the code example does, they press a button. Unity will then save the recorded eye tracking coordinates to a file. This file will then be used to generate a heatmap. After all the tests have been ran, the user will be presented with a second form to give some feedback.

## **Code samples**

Each test subject will be presented with two types of code assignments. The first of these will ask the subject to determine the output of the given code sample, whilst the other type will ask the user to find bugs or errors hidden within the code. Each code sample will be presented in three stages. Each of these stages are described below. All of the tests will be written in Java and will be presented by the text editor Atom with the font Menlo. The tests were written in Java seeing as it is a very popular language [13] and is being taught as the introductory language here at KTH, where most of the tests will be made. Atom and the Menlo font were also chosen based on popularity.

1. The first stage consists of a piece of code without any indentation or syntax highlighting or any styling that makes the code easier to read.
2. The second stage introduces syntax highlighting as well as code indentation to the code, and represents the styling and help offered by the editor in which a programmer writes the code.
3. The third stage represents the readability improvements in the code that is added by the programmer; comments and variable names.

In the following sections the two assignments will be further explained along with these three stages explained above.

## Stage 1 - Recognising output

Each stage in the code sample provided here will feature a unique functionality. More information about these functionalities will be found below.

```
long foo(int n)
{
    if(n <= 1)
    {
        return n;
    }
    else
    {
        return foo(n-1) + foo(n-2);
    }
}
void main(String[] args)
{
    for(int i = 1; i <= n; i++)
    {
        System.out.println(foo(i));
    }
}
```

Figure 1: An example of a Java program outputting the Fibonacci sequence with no indentation or syntax highlighting and entirely without documentation.

```

int foo(int n) {
    if(n <= 1) {
        return n;
    }
    else {
        return foo(n-1) + foo(n-2);
    }
}

float bar(int[] seq) {
    int bzzrt = 0;

    for(int i = 0; i < seq.length; i++) {
        bzzrt += seq[i];
    }
    return (float) bzzrt/seq.length;
}

void main(String[] args) {
    int[] zop = new int[25];
    for(int i = 0; i < 25; i++) {
        zop[i] = foo(i);
    }
    System.out.println(bar(zop));
}

```

Figure 2: The same Fibonacci code but with added indentation and syntax highlighting provided by the text editor (Atom in this case). Furthermore, this code sample features a function called bar that takes a list of integers as input and calculates the mean value of these.

```

/**
 * Returns the n'th fibonacci number
 * @param n
 * @return the n'th element in the fibonacci sequence
 */
int fibonacci(int n) {
    if(n <= 1)
        return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}

/**
 * Checks whether a given number is prime or not
 * @param n
 * @return true if n is prime, false otherwise
 */
boolean isPrime(int n) {
    if(n <= 1)
        return false;
    int sqrt = (int) Math.sqrt(n) + 1;
    for(int i = 2; i < sqrt; i++) {
        if(n % i == 0)
            return false;
    }
    return true;
}

void main(String[] args) {
    for(int i = 1; i <= 25; i++) {
        if(isPrime(fibonacci(i)))
            System.out.println(fibonacci(i));
    }
}

```

Figure 3: The final stage with indentation, syntax highlighting, function name and proper use of comments. Aside from the above functionality from the earlier stages, this stage features a function called `isPrime` that checks whether a given integer is prime or not. The output of this program snippet are those integers that count as both primes and elements of the Fibonacci sequence.

## Stage 2 - Identifying errors

Each stage in the code sample provided here will feature some sort of bug or problem that will cause the code exhibit unexpected behaviour or to not function at all. It is important for a programmer to easily recognise such flaws in code, and depending on the readability of the code and the skill of the programmer the difficulty of finding these errors may vary. It is especially difficult for a person that has recently been assigned to a code to recognise such errors.

Furthermore, it was deemed more relevant to have the majority of errors being run-time errors, as the flaws in the code as these can not be detected

by the compiler and must therefore be detected by the programmer. For instance, missing brackets or uninitialised variables may be detected instantly by the compiler whilst out-of-reach-attempts with arrays or arithmetic errors such as division by zero are impossible for the compiler to predict, as there is no telling whenever a variable may hold a value that will cause such an error. However, some errors will be compile time errors to not make the tests too predictable.

```
void foo(int[] a) {  
  for (int i = 1; i <= a.length; i++) {  
    int tmp = a[i];  
    for(int j = i - 1; j >= 0 && tmp < a[j]; j--) {  
      a[j + 1] = a[j];  
    }  
    a[j + 1] = tmp;  
  }  
}
```

Figure 4: An example of the Insertion Sort algorithm with no indentation, no syntax highlighting and no documentation. It contains two bugs, one being an access outside of the list range and the second one with trying to access the variable `j` outside its scope (the for loop).

```

int[] bar(int[] arr) {
    int i, j, zot;
    int n = arr.length;
    for (i = 0; i < n - 1; i++) {
        zot = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[zot])
                zot = j;
        }
        if (zot != i)
            tmp = arr[i];
            arr[i] = arr[zot];
            arr[zot] = tmp;
    }
}

```

Figure 5: An example of the Selection Sort algorithm which now also features syntax highlighting and indentation provided by the text editor. This code sample contains a bug in which a multi-lined if statement doesn't contain brackets, the tmp variable is never declared and the function is missing a return value.

```

/**
 * Sorts a list of integers in ascending order
 * @param arr a list of integers to be sorted
 * @return the same list but sorted in ascending order
 */
void bubbleSort(int[] arr) {
    boolean swapped = true;
    int j = 0;
    int tmp;
    while (swapped) {
        swapped = false;
        j++;
        for (int i = 1; i < arr.length - j; i++) {
            if (arr[i] > arr[i + 1]) {
                tmp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = tmp;
                swapped = true;
            }
        }
    }
    return arr;
}

```

Figure 6: An example of the Bubble Sort algorithm which now has syntax highlighting, indentation, proper function names and comments. This stage has a bug where the bubbleSort function is said to return an array of integers, which it does in the end. However, the function declaration says void (return nothing), there's a missing bracket for the while loop and the for loop starts at index 1.

## Analyzing the data

Here the method of analysing the data will be presented and discussed. As shown in the previous section, a test result will be a set of pictures, three for each stage making it 6 in total per person. These pictures will be filled with dots representing the subjects gaze points during the test. The gaze points will vary in opacity making it so that a spot that has been gazed upon several times is less opaque (more "colourful") than a spot that has only been gazed upon once. These dots will then be clustered into smaller regions to be able to determine which parts of the code was looked at most and compare them between testers. Clustering will be explained more below.

## Heatmapping

By presenting the subject with these types of code and allowing them to solve the given assignments, data about the gaze points will be gathered and



examined with concern to the time taken to solve them, how correct the results are, and by also observing how densely the gaze points are packed into clusters.

For instance, a dense cluster of gaze points would indicate that the subject required more time to interpret that particular piece of code. Similarly, a less densely populated cluster would imply an area that would either be more trivial in understanding or less relevant in order to complete the given assignment.

Whilst the time taken to complete the assignments will be presented in the results section, these results as well as extracts of the heatmaps will be discussed in the Discussion section.

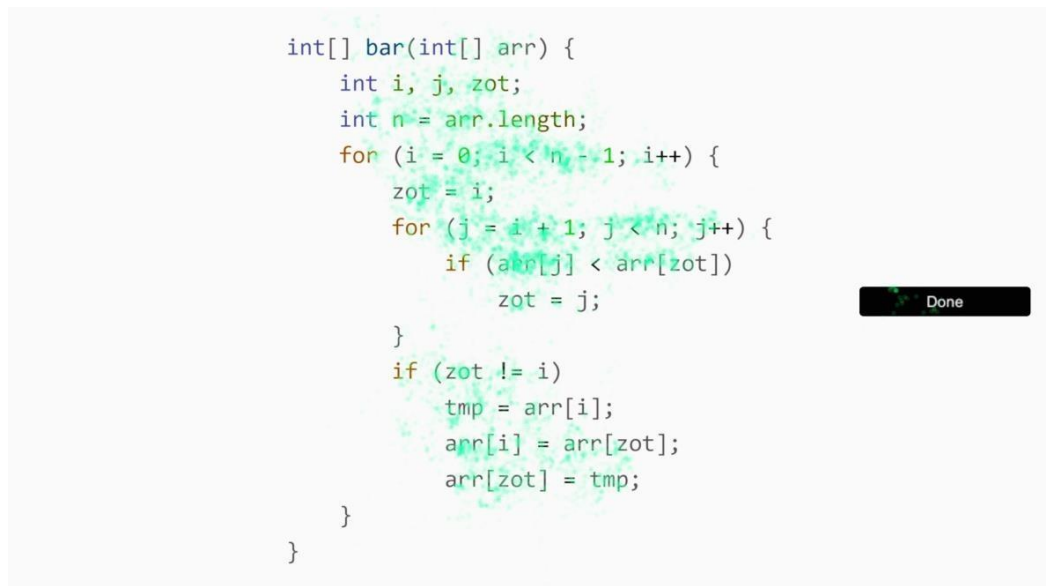


Figure 7: Each gaze point (represented as pink dots) serves as a snapshot of where the eyes were focused on the screen at a specific point in time. The gaze points are assembled to file and then plotted so that the data can be examined.

A heat map such as the one illustrated in figure 7 will be used in order to examine how a user interacts with a given sample of code, and in order to measure and study its readability. By varying the amount code features the user may exhibit different levels of difficulty in understanding the code. The direct results, e.g. the time it took for the subjects to finish each assignment, will be provided in the results section. An extract of some generated heatmaps and eye-tracking data will be discussed in the discussions section, in order to compare and link the results to the actual gaze points generated.

## Results

This section will present the results gathered from the forms and the 6 different code samples.

### Pre-testing form

This form was presented to each test subject before doing the tests to get an overview of their programming skill. It also asked the subjects what they initially thought was the most important part of making code readable. The results will be presented in separate diagrams below.

Hur länge har du arbetat/studerat med programmering?

10 svar

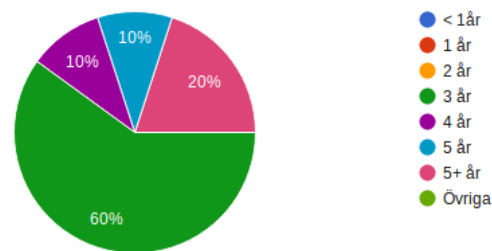


Figure 8: The testers were asked how long they've worked/studied programming, where 3 years dominated with 60%.

Vilket/vilka är det programspråk du känner dig mest säker i? (Fyll i språket i Övrigt om det ej finns med bland svarsalternativen)

10 svar

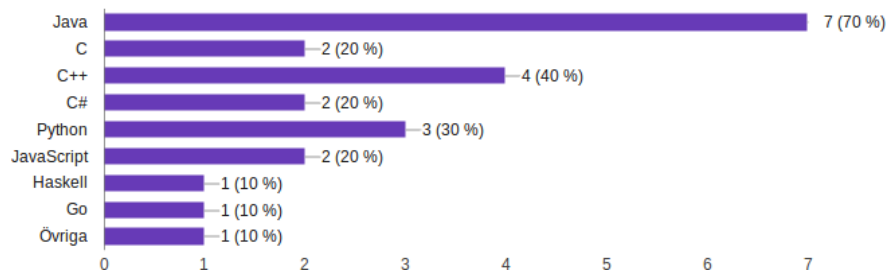


Figure 9: Which language(s) the testers considered most familiar with (multiple answers possible). Java, followed by C#.

Hur säker känner du dig i Java?

10 svar

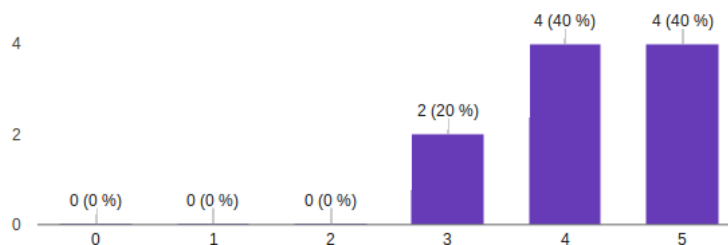


Figure 10: How prominent they considered themselves with Java.

The last question was a written question regarding the subjects thoughts on what the most important aspect of code readability is. The answers were mixed, some subjects thought that structure and uniformity was the most important while three subjects had comments as their top priority. One answer was that code should be split up in well-defined modules, each having a clear purpose and never to repeat the same code. All in all it seems like they all had different preferences. The form used can be found in the appendix.

## Explain the code 1

This is the first code example which is just in plain text. It consists of a recursive Fibonacci function as well as a main function that prints the numbers. The output is a printing of the 'n' first Fibonacci numbers.

That this n was never initialised was pointed out by 3 subjects. This was never intended by the authors and thus served as a "bonus" point", for the amusement of the test subject. Regarding the results for this code snippet, 80% of the subjects managed to answer correctly implying that two subjects got the wrong answers.

```
long foo(int n)
{
    if(n <= 1)
    {
        return n;
    }
    else
    {
        return foo(n-1) + foo(n-2);
    }
}

void main(String[] args)
{
    for(int i = 1; i <= n; i++)
    {
        System.out.println(foo(i));
    }
}
```

Figure 11: The first code example to be explained.

The following table shows the time taken for either solving the example, giving the wrong answer or simply giving up and moving on (in seconds). It also shows the total % of correct answers.

Average	Fastest	Slowest	Correct %
96.6	21	247	80%

## Explain the code 2

Now the same Fibonacci code from the first example has had indentation and syntax highlighting added. A lot of the subjects gave a sigh of relief once they entered this example because of the added syntax highlighting and indentation.

There's now a new added function that evaluates the average of all the values in a given array. The program generates the first 25 Fibonacci numbers and puts them in the array zop, which is then passed to the function bar. It then

prints out the result, which is going to be the average of the elements in zop, that is, the average of the first 25 Fibonacci numbers.

The results here were pretty much the same as for the first example. The same people that managed to solve the first one also solved this one, and the ones that didn't solve the first one had trouble with this one.

```
int foo(int n) {
    if(n <= 1) {
        return n;
    }
    else {
        return foo(n-1) + foo(n-2);
    }
}

float bar(int[] seq) {
    int bzzrt = 0;

    for(int i = 0; i < seq.length; i++) {
        bzzrt += seq[i];
    }
    return (float) bzzrt/seq.length;
}

void main(String[] args) {
    int[] zop = new int[25];
    for(int i = 0; i < 25; i++) {
        zop[i] = foo(i);
    }
    System.out.println(bar(zop));
}
```

Figure 12: The second code example to be explained.

The following table shows the time taken for either solving the example, giving the wrong answer or simply giving up and moving on (in seconds). It also shows the total % of correct answers.

Average	Fastest	Slowest	Correct %
109.8	51	256	80%

### Explain the code 3

In the final "explain the code" example there are added function names and comments.

The function that previously computed the average is now replace by a function called isPrime, which as it sounds, checks if the given input is a prime

number or not. First off it loops from 1 to 25 and generates each of the first 25 prime numbers. While generating the Fibonacci numbers, an if statement continuously checks whether the Fibonacci number is prime or not. If the Fibonacci number is prime, it is printed out.

For this example 70% managed to guess the right output. However, only one of the two that failed the previous ones also failed this one. The remaining two that didn't manage to guess right were previously 2/2.

We didn't hear as much praise for the comments but the naming got some good feedback.

```
/**
 * Returns the n'th fibonacci number
 * @param n
 * @return the n'th element in the fibonacci sequence
 */
int fibonacci(int n) {
    if(n <= 1)
        return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}

/**
 * Checks whether a given number is prime or not
 * @param n
 * @return true if n is prime, false otherwise
 */
boolean isPrime(int n) {
    if(n <= 1)
        return false;
    int sqrt = (int) Math.sqrt(n) + 1;
    for(int i = 2; i < sqrt; i++) {
        if(n % i == 0)
            return false;
    }
    return true;
}

void main(String[] args) {
    for(int i = 1; i <= 25; i++) {
        if(isPrime(fibonacci(i)))
            System.out.println(fibonacci(i));
    }
}
```

Figure 13: The third code example to be explained.

The following table shows the time taken for either solving the example, giving the wrong answer or simply giving up and moving on (in seconds). It also shows the total % of correct answers.

Average	Fastest	Slowest	Correct %
98.9	37	218	70%

## Find the bug 1

This is the first code example where the subject is supposed to look for bugs. Again starting with nothing but plain text.

The code represent an insertion sort algorithm. Below there will be two pictures, firstly one which is the same that the subjects saw and then followed by the same but with added comments to reveal the bugs. The latter one was never shown to the subjects.

Two bugs were embedded in this first example. The first one being that the outer for loop starts at 1 and never considers sorting the first element in the list. This first bug was found by 6 out of 10 subjects. The second bug is that the code tries to access the variable `j` outside its scope. This bug was found by 7 out of 10 subjects. Every subject found at least one of the bugs. So far there are still 3 subjects that are 100% correct.

```
void foo(int[] a) {  
    for (int i = 1; i <= a.length; i++) {  
        int tmp = a[i];  
        for(int j = i - 1; j >= 0 && tmp < a[j]; j--) {  
            a[j + 1] = a[j];  
        }  
        a[j + 1] = tmp;  
    }  
}
```

Figure 14: The first code example which contain bugs.

```
void foo(int[] a) {  
    for (int i = 1; i <= a.length; i++) { // Bug  
        int tmp = a[i];  
        for(int j = i - 1; j >= 0 && tmp < a[j]; j--) {  
            a[j + 1] = a[j];  
        }  
        a[j + 1] = tmp; // Bug  
    }  
}
```

Figure 15: Same example but with the two bugs commented.

The following table shows the time taken for either finding all the bugs or simply giving up and moving on (in seconds). It also shows the total % of found bugs.

Average	Fastest	Slowest	Total bugs found
302	90	785	65%

## Find the bug 2

The second find the bug example is again a sorting algorithm, selection sort. It now has added syntax highlighting and indentation. Three bugs exists in this code. Firstly the code claims to return an int array but there is no return value. Half of the subjects, 5, managed to find this bug. This eliminated one of the subjects that were 100%. The second bug, however, was found by 8 out of the 10 subjects. This bug is that tmp is never declared as a variable and was often found last out of the three. Finally, the second if statement needs to have brackets if it has more than one lines. This was also caught by 8 of the subjects.

```
int[] bar(int[] arr) {
    int i, j, zot;
    int n = arr.length;
    for (i = 0; i < n - 1; i++) {
        zot = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[zot])
                zot = j;
        }
        if (zot != i)
            tmp = arr[i];
            arr[i] = arr[zot];
            arr[zot] = tmp;
    }
}
```

Figure 16: The second code example containing bugs.



```

int[] bar(int[] arr) {          // Bug
    int i, j, zot;              // Bug
    int n = arr.length;
    for (i = 0; i < n - 1; i++) {
        zot = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[zot])
                zot = j;
        }
        if (zot != i)           // Bug
            tmp = arr[i];
            arr[i] = arr[zot];
            arr[zot] = tmp;
    }
}

```

Figure 17: Same example but with the three bugs commented.

The following table shows the time taken for either finding all the bugs or simply giving up and moving on (in seconds). It also shows the total % of found bugs.

Average	Fastest	Slowest	Total bugs found
316	106	633	70%

### Find the bug 3

The last example is a version of bubblesort with added comments and variable/function names. Just as the previous example, this code contains three bugs. 70% caught the first one, the function declares void but in the end it returns an array. The for loop starts at  $i = 1$  hence missing out on sorting the first element. This was found by 80% of the subjects. Finally, the code is missing a `}` for the while loop which was pointed out by 90%, making it the highest finding-rate of any of the bugs.

```

/**
 * Sorts a list of integers in ascending order
 * @param arr a list of integers to be sorted
 * @return the same list but sorted in ascending order
 */
void bubbleSort(int[] arr) {
    boolean swapped = true;
    int j = 0;
    int tmp;
    while (swapped) {
        swapped = false;
        j++;
        for (int i = 1; i < arr.length - j; i++) {
            if (arr[i] > arr[i + 1]) {
                tmp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = tmp;
                swapped = true;
            }
        }
    }
    return arr;
}

```

Figure 18: The third and final code example containing bugs.

```

/**
 * Sorts a list of integers in ascending order
 * @param arr a list of integers to be sorted
 * @return the same list but sorted in ascending order
 */
void bubbleSort(int[] arr) {          // Bug
    boolean swapped = true;
    int j = 0;
    int tmp;
    while (swapped) {                  // Bug
        swapped = false;
        j++;
        for (int i = 1; i < arr.length - j; i++) { // Bug
            if (arr[i] > arr[i + 1]) {
                tmp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = tmp;
                swapped = true;
            }
        }
    }
    return arr;
}

```

Figure 19: Same example but with the three bugs commented.

The following table shows the time taken for either finding all the bugs or simply giving up and moving on (in seconds). It also shows the total % of found bugs.

Average	Fastest	Slowest	Total bugs found
249.6	860	431	80%

## Post-testing form

This form was presented to each test subject right after they were done with the tests, with the purpose of finding out if what they thought was the most/least important feature (Syntax highlighting, indentation, variable/function names or comments).

The four were treated one by one and rated from 0 (not important at all) to 10 (super important). Each features result will be presented in a separate diagram below.

First off is syntax highlighting.

### Syntax highlighting

10 svar

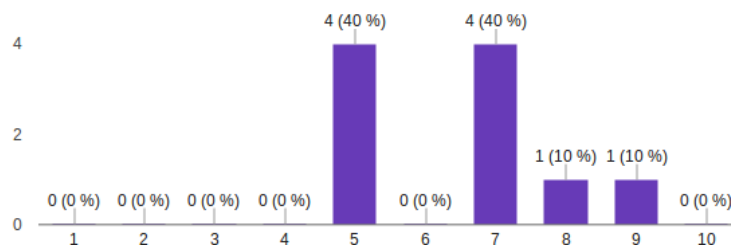


Figure 20: Syntax highlighting results.

It got 4 votes for 5, 4 votes for 7, 1 vote for 8 and 1 vote for 9.  
Next up is indentation.

### Indentering (indrag)

10 svar

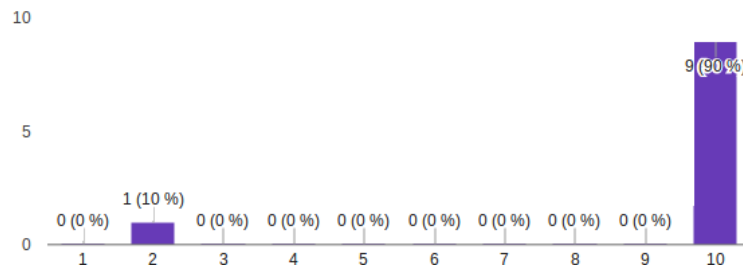


Figure 21: Indentation results.

It got 9 votes for 10 and 1 vote for 2. A clear leader so far.  
Third up is proper variable/function naming.

### Lämpliga variabel/funktionsnamn

10 svar

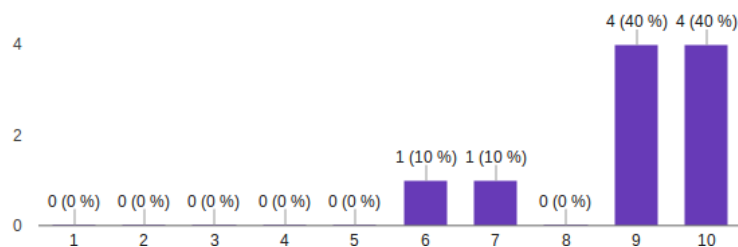


Figure 22: Proper variable/function naming results.

6 and 7 got 1 vote respectively and 4 voted for 9 and another 4 voted 10.  
Finally the subjects had to determine the importance of comments.

## Kommentarer

10 svar

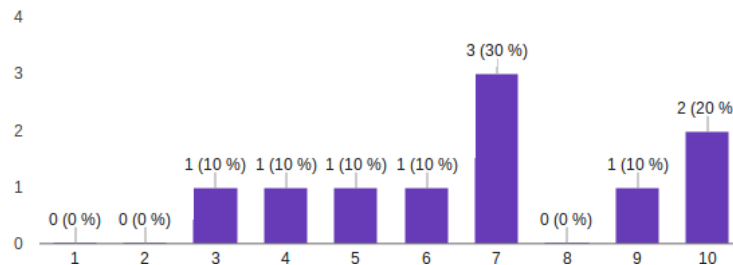


Figure 23: Comments results.

This got the most varied results yet. One subject voted for 3, 4, 5, 6 and 9 respectively. 7 got 3 votes and 10 got voted twice. The average score for each one is shown in the table below.

The table below shows the average of how important each test subject considers different code styling features to be. For instance, code indentation appears to be deemed most important by most subjects, with a score of 9.2.

Syntax highlighting	Indentation	Variable/function naming	Comments
6.5	9.2	8.9	6.8

After grading each of the four features the subjects were asked which type of improvement they considered to be most important.

**Step one:** from plain text to having syntax highlighting and indentation.  
or

**Step two:** having syntax highlighting and indentation and adding comments and proper variable/function names. They could also choose that they had the same amount of effect or that they were unsure which one they preferred.

Vilken av följande förbättringar upplevde du gjorde mest skillnad för läsbarheten?

10 svar

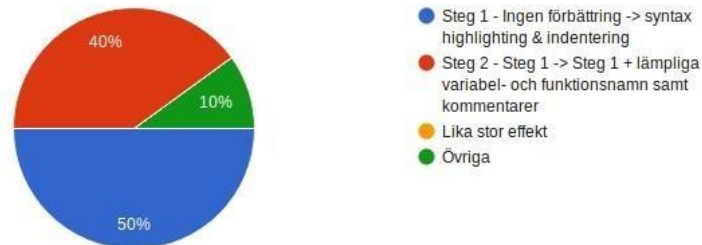


Figure 24: Which type of improvement considered the most important.

The results were close. 50% considered step one more important while 40% considered step two. The remaining answer was unsure.

## Discussion

In this section the results will be discussed.

### Commenting

Half of the test participants failed to find the bug related to the missing return statement in "Find the bug 2". Interestingly, the reason for this seems related to the introduction of code comments with "Find the bug 3". It appears that since there is no content above the method signature the test subject ignored the header and jumped straight into the code, assuming that no errors existed with the header. The introduction of commenting seemingly increased the likelihood of the person to look at the signature of the method, which also seems to be correlated with the raised amount of subjects finding the bug related to the void-return bug in "Find the bug 3". The "@return" piece of text in the comment along with the "void" return implied by the signature just below seemingly looked conspicuous in the eyes of the programmer.

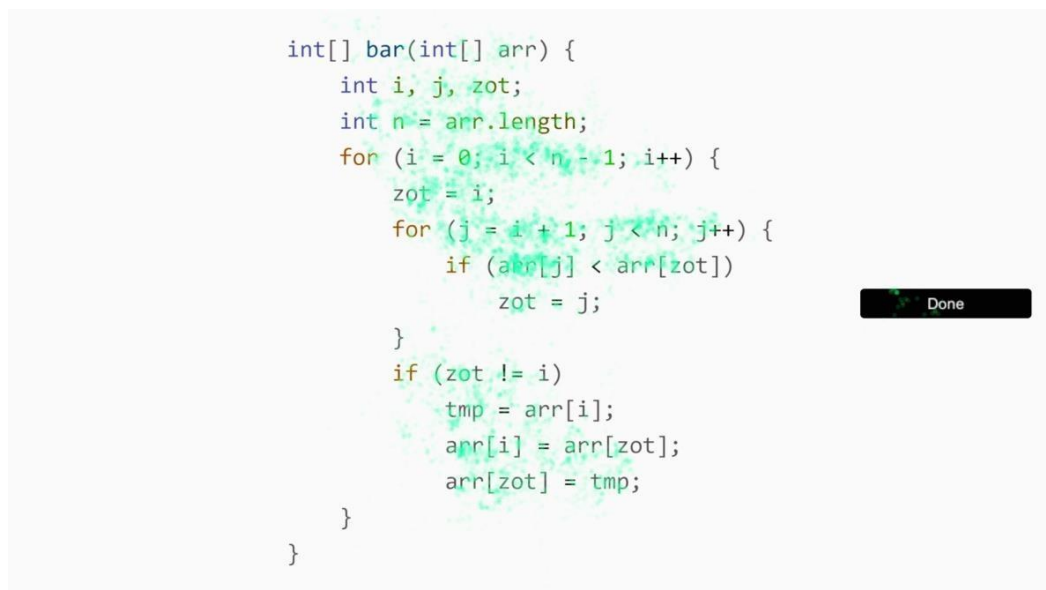
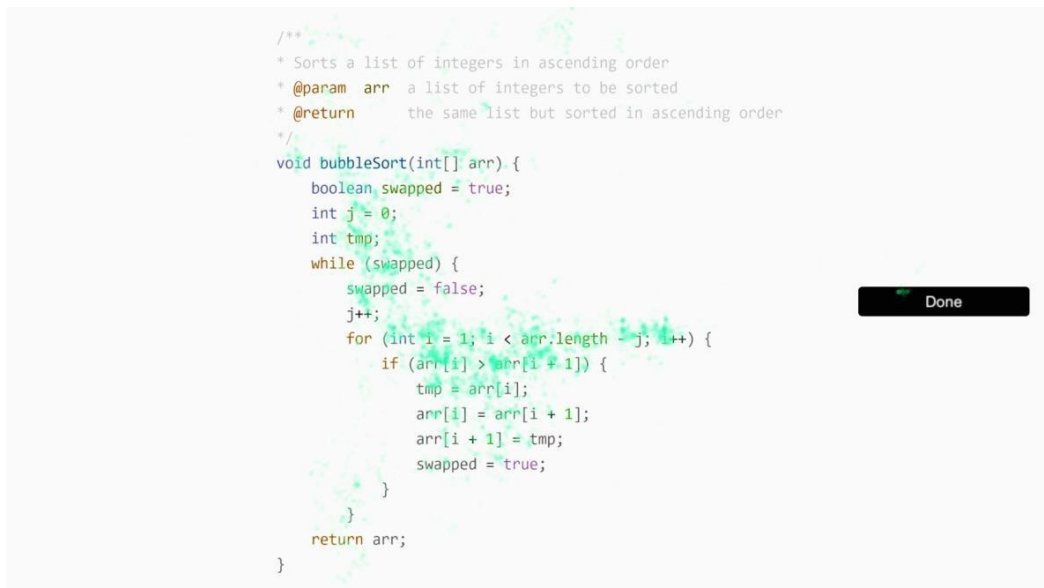


Figure 25: The heatmap for one subject in which it is clearly visible that a considerably less amount of gaze points are focused around the **int[]** statement in the method signature when compared the rest of the code. Most of the subjects showed similar behaviour, none lingering on this spot, and this is most likely correlated with the high number of subjects failing to spot the missing return statement in this particular assignment. This subject failed to do so as well.

A heatmap visualization is overlaid on a Java code snippet. The heatmap uses a color scale from blue (low performance) to red (high performance). The highest performance (red) is concentrated on the function signature 'void bubbleSort(int[] arr)' and the 'return arr;' statement. Other areas of moderate performance (yellow/green) include the parameter documentation '@param arr' and the loop 'for (int i = 1; i < arr.length - j; ++i)'. The rest of the code, including the 'while (swapped)' loop and the swap logic, shows lower performance (blue/green). A 'Done' button is visible on the right side of the code editor.

```
/**
 * Sorts a list of integers in ascending order
 * @param arr a list of integers to be sorted
 * @return the same list but sorted in ascending order
 */
void bubbleSort(int[] arr) {
    boolean swapped = true;
    int j = 0;
    int tmp;
    while (swapped) {
        swapped = false;
        j++;
        for (int i = 1; i < arr.length - j; ++i) {
            if (arr[i] > arr[i + 1]) {
                tmp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = tmp;
                swapped = true;
            }
        }
    }
    return arr;
}
```

Figure 26: This heatmap shows the same subject's performance on the task following the previous; *Find the bug 3*. However, despite failing to spot the missing return statement in the previous assignment, the user managed to spot the void-return error in this code. This is most likely due to the introduction of comments at the top of the code which implied that this function shouldn't actually return as the signature yields a **void** return type.

## Variable naming

The introduction of variable naming made its most prominent appearance within the "Explain the code 3". First and foremost, the subjects did not view the Fibonacci code as they noticed it was the same method provided in the previous examples, the only difference was that it was now named "fibonacci()". The most gaze points recorded here was from the main method which changed appearance, even though the isPrime() method contained almost three times as many rows of code. The reason for this seems to be that the subjects understood what the purpose of the isPrime() method was just by looking at the signature and thus it was not necessary to spend any prolonged periods of time within that method. Instead, most of the time was spent by examining the main method's rows of code, the purposes of which was not as clear unlike isPrime(), as isPrime() had a logical name.





Figure 27: The heatmap for one of the subjects in which it can be clearly observed that the purpose of the `isPrime()` function is explained by the name of the method. In contrast, the content of the `main()` method is also newly introduced yet this content is not explained by the `main()` method signature. Note the varying amount of gaze point between the two methods. Also note that the newly introduced comments in the top of the code also has a considerable amount of gaze points.

## Comparing the "explain the code" results

This section will only look at comparing the three "explain the code" examples and their results, in other words test 1-3.

The first "explain the code" example consisting solely of the Fibonacci function in plain text and without any formatting had an average time to finish of 96.6 seconds. 80% of the subjects managed to solve it. If we look at the second "explain the code" example, with additional syntax highlighting and indentation, the average time was 109.8 seconds. That is a little bit slower than the first example, however, it contains a considerable amount of additional code, including the mysterious function "bar" that has to be inspected. The main function is also expanded a bit. In many of the cases the subjects that had got the previous code example right quickly recognised that the function "foo" was also the Fibonacci function and spend most of their time looking through "bar". It only differed 13.2 seconds between the two even though the second example has a lot more code and logic to understand and

it had the same 80% correct ratio. This shows that indentation and syntax highlighting affects code readability in a positive way. There were also a few sighs of relief once they moved from test one to two because of the added indentation and syntax highlighting.

The third and final "explain the code" example took an average of 98.9 seconds to finish. That is 10.9 seconds faster than the second example and 2.3 seconds slower than the first example. Here it is obvious that the first function is a Fibonacci function because of the added function/variable naming as well as comments. The second function is called `isPrime()` and we saw a clear pattern that the subjects pretty much skipped going through the actual code of the `isPrime()` function after reading its name. This shows that proper naming plays a role in code readability. We are not sure why the number of correct results dropped to 70% compared to the previous two examples.

## **Comparing the "find the bug" results**

This section will only look at comparing the three "find the bug" examples and their results, in other words test 4-6.

The first "find the bug" example contains two bugs. The average time to finish was 302 seconds. If we compare this to the second "find the bug" example where it has syntax highlighting and indentation the average time to finish was 316 seconds. However, the second example contains three bugs. Essentially it only took 14 more seconds for the subjects to finish the second bug test, making the average time to find one bug 151 seconds for the first example and 105.3 seconds for the second. The second one also contained an extra function without lowering the "difficulty" of the bugs (adding complexity and more actual code to read through). The total amount of found bugs were increased by 5%. This shows that indentation and syntax highlighting play a big role in code readability. One could argue, however, that the subjects have adapted and gained more skill in looking for faults after doing the first "find the bug" example.

If we then turn our heads to the third, final, "find the bug" test, we see that it took 249.6 seconds on average to finish. That is 52.4 seconds faster than the first test and 66.4 faster than the second test, making it an average of 83.2 seconds per bug found (seeing as this example also contains three bugs). The total amount of bugs found were the highest yet at 80%, compared to the previous 70% in the second test. The commenting and naming added seem to have affected the subjects in a positive way. Not only was it the fastest and most successful test yet, it also contained the most code. Again, it could be argued that the subjects have adapted and gained some more

insight from the previous test.

## **Conclusion**

We can now answer the research question;

*Can the readability and interpretation of code be visibly improved by adding features such as syntax highlighting, code indentation, comments and logical variable naming?*

Yes. A visible improvement regarding the code readability and interpretation could be observed after adding the code styling features named above. Each addition gave different improvements and will be presented below.

### **Indentation and Syntax highlighting**

The addition of indentation and syntax highlighting did not show any major visual improvement, however, it provided faster understanding relative to the amount of code increased and the subjects expressed joy when it was added.

### **Logical naming**

With the addition of logical function and variable naming many of the subjects heatmaps showed that they skipped reading through code of functions such as `isPrime()` which leads to a quicker understanding of the overall code.

### **Comments**

Many subjects had problem finding return type errors in the second stage (with syntax highlighting and code indentation, but no comments or logical names). With the introduction of comments the subjects was seen looking at the function headers more and thus in many cases, despite missing the return type error in the second stage quickly found the same type of error in the third stage.

### **Limitations**

When looking at this project there are a few limitations that needs to be brought up.

## **Number of testers**

We had a total of 10 people doing the tests. This gave us relevant and good results. Having more testing would have been beneficial to gain more samples to the results, but this was never accomplished. The reason for this was mostly due to time constraints; it took longer time than initially expected to set up the test environment as well as to write the code necessary to perform the experiments. It was also slightly more difficult than initially planned to gather and recruit test subjects.

## **Tobii EyeX**

The Tobii EyeX eye-tracking device is one of their cheaper variants and is mostly intended for gaming use. We thought that this would be a problem, since we were interested in small areas of text. However, this turned out not to be a problem, the EyeX is more accurate than we expected. There were only issues in one of the tests where we think the subject moved his/her head too much, resulting in very few actual gaze points. Aside from that, the gaze points were sufficient and accurate enough for the data to be relevant enough to be extracted and studied without any hindrances.

## **Future research**

The experiments conducted for the research for this report were aimed towards people with varying skills in programming. An interesting approach would be to do similar testing for a group of students during their first year, and then do the same testing with the same group of students during a later stage of their studies to see if there are improvements.

Another approach could be to split the test subjects up into three different groups and have one group do the tests for stage 1 without any styling, have one group do the tests for stage 2 with added syntax highlighting and indentation, and have the third group do the stage 3 tests with comments and logical naming. The three groups results could then be compared and analysed.

## **Appendixes**

### **Pre-test form**

# Inför användartest

Detta formulär agerar grund till testen. Vi genererar ett id åt dig så att du förblir anonym

**\*Obligatorisk**

## 1. id

---

## 2. Hur länge har du arbetat/studerat med programmering? \*

Markera endast en oval.

- ☐ < 1år
- ☐ 1 år
- ☐ 2 år
- ☐ 3 år
- ☐ 4 år
- ☐ 5 år
- ☐ 5+ år
- ☐ Övrigt: \_\_\_\_\_

## 3. Vilket/vilka är det programspråk du känner dig mest säker i? (Fyll i språket i Övrigt om det ej finns med bland svarsalternativen) \*

Markera alla som gäller.

- ☐ Java
- ☐ C
- ☐ C++
- ☐ C#
- ☐ Python
- ☐ JavaScript
- ☐ Haskell
- ☐ Go
- ☐ Övrigt: \_\_\_\_\_

## 4. Hur säker känner du dig i Java? \*

Markera endast en oval.

	0	1	2	3	4	5	
Inte alls	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Jättesäker

**5. Vad tycker du är viktigast när det kommer till läsbarhet i kod? \***

---

---

---

---

---

---

Tillhandahålls av



## **Post-test form**

# Efter användartest

Lite feedback och frågor.

Rangordna följande element utefter hur mycket de påverkar läsbarheten i kod (1 för låg effekt, 10 för hög effekt).

**\*Obligatorisk**

## 1. Id \*

---

## 2. Syntax highlighting \*

Markera endast en oval.

	1	2	3	4	5	6	7	8	9	10	
Ingen påverkan	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Mycket hög påverkan

3.

## Indentering (indrag) \*

Markera endast en oval.

	1	2	3	4	5	6	7	8	9	10	
Ingen påverkan	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Mycket hög påverkan

## 4. Lämpliga variabel/funktionsnamn \*

Markera endast en oval.

	1	2	3	4	5	6	7	8	9	10	
Ingen påverkan	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Mycket hög påverkan

## 5. Kommentarer \*

Markera endast en oval.

	1	2	3	4	5	6	7	8	9	10	
Ingen påverkan	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Mycket hög påverkan



**6. Vilken av följande förbättringar upplevde du gjorde mest skillnad för läsbarheten? \****Markera endast en oval.*

- ☐ Steg 1 • Ingen förbättring •> syntax highlighting & indentering
- ☐ Steg 2 • Steg 1 •> Steg 1 + lämpliga variabel• och funktionsnamn samt kommentarer
- ☐ Lika stor effekt
- ☐ Övrigt: \_\_\_\_\_
- 

Tillhandahålls av



## **Code**

<https://github.com/emilppp/kex>

# Bibliography

- [1] Taek Lee, Jung Been Lee, and Hoh Peter In. A study of different coding styles affecting code readability. *International Journal of Software Engineering and its Applications*, 7(5):413–422, 10 2013. ISSN 1738-9984. doi: 10.14257/ijseia.2013.7.5.36.
- [2] R. P. L. Buse and W. R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, July 2010. ISSN 0098-5589. doi: 10.1109/TSE.2009.70.
- [3] Claude Ghaoui. *Encyclopedia of Human Computer Interaction*. Idea Group Reference, illustrated edition edition, 2006. ISBN 1591405629,9781591405627,9781591407980,1591407982.
- [4] Päivi Majaranta and Andreas Bulling. *Eye Tracking and Eye-Based Human–Computer Interaction*, pages 39–65. Springer London, London, 2014. ISBN 978-1-4471-6392-3.
- [5] Andrew T. Duchowski. *Eye Tracking Methodology: Theory and Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 1846286085.
- [6] Aihab Khan Aisha Batool, Muhammad Habib ur rehman and Amsa Azeem. Impact and comparison of programming constructs on java and c source code readability. *International Journal of Software Engineering and Its Applications*, 9(11):79–90, 2015.
- [7] David Tollemark Sebastian Nielsen. Code readability: Code comments or self-documenting code : How does the choice affect the readability of the code? *Blekinge Institute of Technology, Department of Software Engineering*, 2016.
- [8] Markus Schmitt Mathias Pohl and Stephan Diehl. Comparing the readability of graph layouts using eyetracking and task-oriented analy-

sis. *Computational Aesthetics in Graphics, Visualization, and Imaging*, 2009.

- [9] Luz Rello, Susana Bautista, Ricardo Baeza-Yates, Pablo Gervás, Raquel Hervás, and Horacio Saggion. *One Half or 50%? An Eye-Tracking Study of Number Representation Readability*, pages 229–245. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-40498-6. doi: 10.1007/978-3-642-40498-6\_17. URL [http://dx.doi.org/10.1007/978-3-642-40498-6\\_17](http://dx.doi.org/10.1007/978-3-642-40498-6_17).
- [10] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013. ISSN 1573-7616. doi: 10.1007/s10664-012-9201-4. URL <http://dx.doi.org/10.1007/s10664-012-9201-4>.
- [11] Gilles Dubochet. Computer Code as a Medium for Human Communication: Are Programming Languages Improving? In Chris Exton and Jim Buckley, editors, *Proceedings of the 21st Working Conference on the Psychology of Programmers Interest Group*, pages 174–187, Limerick, Ireland, 2009. University of Limerick. ISBN 978-1-905952-16-8. URL <http://www.csis.ul.ie/PPIG09/>.
- [12] The tobii eye tracking sdk. <http://developer.tobii.com/eyex-sdk/>.
- [13] Pypl popularity of programming language index. <http://pypl.github.io/PYPL.html>.

