

How programmers read regular code: a controlled experiment using eye tracking

Ahmad Jbara · Dror G. Feitelson

the date of receipt and acceptance should be inserted later

Abstract Regular code, which includes repetitions of the same basic pattern, has been shown to have an effect on code comprehension: a regular function can be just as easy to comprehend as a non-regular one with the same functionality, despite being significantly longer and including more control constructs. It has been speculated that this effect is due to leveraging the understanding of the first instances to ease the understanding of repeated instances of the pattern.

To verify and quantify this effect, we use eye tracking to measure the time and effort spent reading and understanding regular code. The experimental subjects were 18 students and 2 faculty members. The results are that time and effort invested in the initial code segments are indeed much larger than those spent on the later ones, and the decay in effort can be modeled by an exponential model. This shows that syntactic code complexity metrics (such as LOC and MCC) need to be made context-sensitive, e.g. by giving reduced weight to repeated segments according to their place in the sequence. However, it is not the case that repeated code segments are actually read more and more quickly. Rather, initial code segments receive more focus and are looked at more times, while later ones may be only skimmed. Further, a few recurring reading patterns have been identified, which together indicate that in general code reading is far from being purely linear, and exhibits significant variability across experimental subjects.

1 Introduction

Although there is a general agreement on the importance of code complexity metrics, there is little agreement on specific metrics and their accuracy [42]. Syntactic metrics like lines of code (LOC) and McCabe's cyclomatic complexity (MCC) are

A. Jbara
School of Computer Science and Engineering, Hebrew University, 91904 Jerusalem, Israel, and
School of Mathematics and Computer Science, Netanya Academic College, 42100, Netanya,
Israel. E-mail: ahmadjbara@cs.huji.ac.il

D. G. Feitelson
School of Computer Science and Engineering, Hebrew University, 91904 Jerusalem, Israel.
E-mail: feit@cs.huji.ac.il

commonly used mainly because they are simple. These metrics are additive and myopic: they simply count source code elements without considering their type and context. Therefore, they do not necessarily reflect the *effective* complexity of source code, that is, that attribute of code which makes it hard to understand. In particular, they lead to inflated measurements of well-structured long functions that are actually reasonably simple to comprehend [20].

In previous work [20, 18] we introduced *regularity* as a new factor that questions the additivity of the classical syntactic metrics. Regularity is the repetition of code patterns (e.g. a certain pattern of nested control statements), where repeated instances of the pattern are usually successive. Figure 1 contrasts a regular and a non-regular function of about the same length from Linux, and Figures 5 and 6 show the regular functions used in our experiments with the repeated instances indicated by rectangles.

Regular code is generally longer than non-regular code implementing the same functionality, and if measured by metrics like MCC it is also more complex, as there is a strong correlation between LOC and MCC. However, our experiments showed that long “complex” regular code is not harder to comprehend than the shorter non-regular alternative. The speculation was that regularity helps because repeated instances of a pattern are easier to understand once the initial ones are understood [18].

To investigate this idea, we conducted a controlled experiment that uses eye tracking to explore how programmers read regular code, and to quantitatively measure the time and effort invested in the successive repetitions in such code. The results indeed show that time and effort are focused on the initial repeated instances, and reduced as later instance are considered. This reduction can be modeled by an exponential function.

As a consequence additive syntax-based metrics like LOC or MCC may be misleading, because repeated instances contribute less to complexity and comprehension effort. This observation was made already by Weyuker in the context of her famed work on desirable properties of code complexity metrics [45], where she writes “Consider the program body P ; P (that is, the same set of statements repeated twice). Would it take twice as much time to implement or understand P ; P as P ? Probably not.” Our results enable us to take an additional step, and suggest a specific weighting function which can be applied to repeated code segments so as to reflect their reduced effect. This adds a degree of context sensitivity to previously oblivious syntactic metrics.

But overall effort modeling does not tell the whole story: it is also interesting to observe the subjects’ reading pattern. We used the eye tracking data to analyze the subjects’ scanpaths, namely how they scan the code they are reading. This shows that the way programmers read regular code is far from the conventional mostly-linear order employed in reading natural language texts. Instead, reading code appears to be done in a sequence of patterns such as scanning it, jumping ahead to look for ideas, jumping back to verify details, and so on.

However, the patterns employed and their order are highly individualistic. It is therefore necessary to collect much more data in different contexts before a general picture of code reading will emerge. Such future work can be based on the methodological foundations which we laid in our analysis of reading regular code, including the identified basic patterns and the use of smoothing to remove noise from the original eye tracking data and make the patterns more evident.

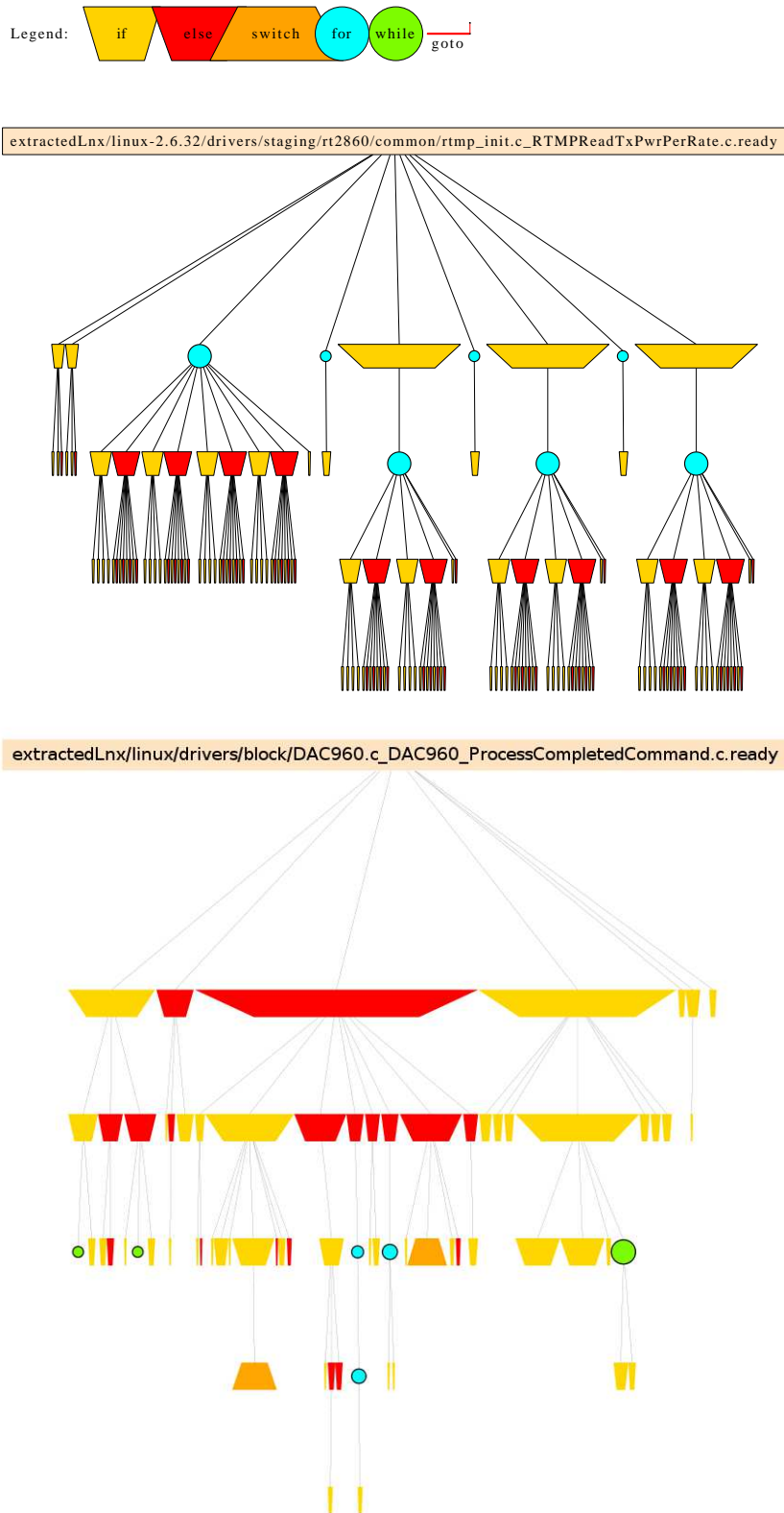


Fig. 1: Code-structure diagrams of two functions from the Linux kernel. Both have similar MCC values (117 and 127, respectively), but the first is obviously much more regular. See [19] for an explanation of the diagrams.

It is expected that these methodological innovations will be relevant not only for regular code but also for studying code reading in general.

2 Motivation and Research Questions

An important aspect of code complexity is the complexity of individual functions, because developers typically focus on a small set of functions for each programming or debugging task. A large number of metrics for measuring the complexity of a function have been proposed, but none of them is capable of fully reflecting the complexity of source code [10,26]. In previous work we have suggested *regularity* as an additional factor that affects code comprehension, especially in long functions, and provided experimental evidence for its significance [20,18]. Specifically, we conducted several experiments where developers were required to understand functions and to perform maintenance tasks on them, where different subjects were actually working on different versions of the same function. Thus we could evaluate the relationship between developer performance and the style in which the function was coded.

Before we continue, a clarification about terminology is in place. Regularity refers to sequences of code segments that have the same structure in terms of (possibly nested) control statements. Such sequences are most probably produced by developers who use copy-paste and then edit each instance in the sequence for its specific role. This can be considered a special case of code cloning. Code cloning is often considered bad practice because of the need to maintain the separate clones consistently, and because it may lead to code bloat. These concerns are irrelevant for regular code, because the instances are typically relatively small and co-located. We therefore argue that regularity should be considered as a distinct coding practice, and not be confused with cloning.

Regular functions by definition contain repetitive code segments, which usually come one directly after another. This suggests that understanding one of these segments would help understanding subsequent ones. Based on this we hypothesize that the cognitive effort needed for the second segment is lesser than that for the first, and as the developer proceeds in the sequence of repetitive segments the effort needed becomes smaller. After several segments it may be expected that the additional effort would even be negligible.

Our main purpose is therefore to study the way developers read regular code, and whether they invest equal effort in repetitive code segments. Moreover, if the efforts are indeed not the same and decreasing, we want to find a model that reflects the relation between the serial location of the segment and the amount of effort needed to comprehend it. Making the reasonable assumption that the invested effort reflects the encountered difficulty, such a model can then serve as a good context-dependent weighting function for metrics that consider all repeated segments to have equal impact and hence yield exaggerated measures.

In addition, we use this study to verify that our previous work is reproducible. Specifically, this is an “internal replication”, as it is being conducted by the same researchers [4]. We also used exactly the same functions as in our previous study, and compared the performance of subjects when reading regular programs versus their non-regular counterpart. However, the experimental context is different (one-

on-one experiments using an eye tracker versus a paper-based group experiment), and of course the experimental subjects themselves are different.

To recap, the specific research questions this paper addresses are:

1. Do developers follow any pattern when they are required to comprehend regular code? In particular, are their efforts equally divided among regular segments?
2. Assuming there is a pattern that governs the investment of effort, which model might fit and describe it?
3. Does the distribution of effort tell the whole story? In other words, is code read linearly and only the time spent on repetitions perhaps changes, or is the reading pattern more complicated?
4. In terms of correctness and completion time, are the results consistent with those of our previous work [18]?

Based on answering these research questions, the paper makes the following major contributions. The first and perhaps the most obvious is the methodological contribution in relation to code complexity metrics, and specifically common metrics like LOC and MCC. In this context we make two innovative contributions:

- To the best of our knowledge our work on regularity (in this and the previous paper) is the first empirical study of MCC and LOC in terms of explicitly measuring comprehension and its relation with such metrics.
- This paper also suggests the first empirically-grounded revision to just counting constructs. Specifically, if the code is regular and includes repetitions of the same pattern, we suggest that their contribution to the overall function complexity should have diminishing weights based on their location in the sequence.

Second, in terms of a take-home message and how practitioners can leverage our results, we note that regular code often “looks bad” and seems to defy the accepted best practices of writing short functions, reducing clones, and extracting functions. But we demonstrated experimentally that such code does not necessarily have adverse implications. As a result practitioners can feel confident that a regular style (if they believe that this is the natural style for their given programming task) may be used without ill-effects on code comprehension. Contrariwise, in those cases where the programming task suggests a solution with regular style, if programmers are convinced that regular solutions are not recommended they might feel compelled to use a non-natural non-regular one instead, which might contribute to the difficulty of comprehending the code in future maintenance tasks.

Third, on a scientific level, this paper augments our previous work on regularity in partly explaining how developers cope with high MCC functions in real systems. Very long functions do in fact exist (the largest we have observed so far is a function with MCC=1316 in the FreeBSD operating system). We now know that developers can handle them, inter alia, by not scrutinizing all the code rigorously when the code includes repeated patterns.

And fourth, we make advances in the field of studying general code reading patterns. This includes two independent contributions:

- One is to reconfirm the occurrence of various code reading patterns that have been identified before, and suggest some additional ones. This is an additional step in the direction of being able to describe code reading using a basis of primitive patterns.

Table 1: Attributes of the two versions of the programs used in the experiment. “Reg.” reflects regularity using the compression ratio.

Program	Regular version			Non-regular version			Description
	LOC	MCC	Reg.	LOC	MCC	Reg.	
Median	53	18	79.3%	34	13	60.7%	Calculate medians of all 3×3 neighborhoods
Diamond	46	17	82.8%	26	14	43.8%	Find max Manhattan-radius around point with all same value

- The other is to suggest smoothing of the raw eye tracking data to bring out the patterns more clearly. The smoothing procedure is non-arbitrary, and based on analyzing dwell times in distinct areas of interest.

3 Methodological Approach

3.1 Test Programs

We use two programs from the image processing domain (Table 1). Each program has two versions: regular and non-regular. The specifications of the programs used are: *calculate the medians of 3×3 neighborhoods around all pixels* and *find the maximal Manhattan-radius around a point such that all pixels within this radius have the same value*. The programs were taken from our previous work [18], and they meet the following experimental criteria:

- Realistic programs of known domain.
- Reasonable regular and non regular implementations of the same specification.
- Non trivial specifications.

We could use one program with its two implementations, but we prefer two programs to avoid program-specific conclusions. We do not use more because then it becomes hard to enlist enough experimental subjects for each version.

To quantify the level of regularity of the different versions, we use an operational definition that is based on compression. This choice is based on the recognition that compression algorithms identify repetitions in their inputs and replace them with shorter encodings. We have systematically investigated different compression schemes and code preprocessing levels [17,20], and found that different combinations yield different results. The combination that gave the best correlation with perceived complexity (how developers subjectively rate the complexity of a function) was to strip the code down to a skeleton of keywords and formatting, and use the *gzip* compression routine. Regularity is then quantified by the compression ratio. Possible alternative metrics and their evaluation are left to future work.

3.2 Eye-tracking Apparatus

We use the *Eye Tribe* eye tracker (www.theeyetribe.com) in this work. The device uses a camera and an infrared LED. It operates at a sampling rate of 60Hz, latency less than 20ms at 60Hz mode, and accuracy of 0.5° – 1° . The device supports 9,

12, or 16 points for the calibration process. We used 9 points mode. The screen resolution was set to 1280 by 1024, and the source code was presented in a full screen window using the Consolas 11 font. The left margin was about 120 pixels and the top one is about 35.

The *Eye Tribe* is a remote eye tracker and as such it provides the subjects a non-intrusive work environment which is essential for reliable measurements. Furthermore, the device allows head movements during the real experiment but not while calibrating.

To analyze the tracking data we use *OGAMA* (www.ogama.net). It is an open source software designed for analyzing eye and mouse movements. *OGAMA* supports many commercial eye trackers like *Tobii*. In its last version (4.5) support for the *Eye Tribe* has been added. This builtin support makes the process easier and saves the import of the data between systems.

3.3 Task Design

Basically, we adopted the programs and the task of experiment 2 from our previous work [18] with one difference. In our previous work, each subject sequentially performed the same task (*understanding what does a program do*) for the regular version of one program and the non-regular version of the other. In this work we follow a between-subject design where each subject performs the task on one version only. This design decision has been taken on the basis of a pilot study where subjects claimed that performing two programs is hard especially when you have to keep your gazes within the screen for a long time [43].

In addition to answering the comprehension question *what does the program do*, the subjects were asked to evaluate the difficulty of the code on a 5-point scale, and state the reasons for their evaluation.

A post-experiment question was presented to each participant regarding the way they approach the programs, with the goal of understanding how their effort was distributed in the code and why. Retrospectively, it turned out that this post-experiment question was important as there were cases where the eye tracking data did not fit the participant's opinion.

3.4 Grading Solutions for Correctness

In grading the solutions of the subjects we followed [22,7,38]. In particular, we adopted a multi-pass approach where three evaluators were involved. Initially, the first author evaluated the answers according to a personal scale. In the second pass another colleague evaluated the answers. However, in a few cases there were large gaps between the two evaluations. To resolve this, the second author made a third pass on these cases.

The final grade for each of the cases was computed as the average of the three evaluations when these were close enough (≤ 10 pts). Otherwise, we computed the average of the two closest grades and the outlier was excluded. It should be noted that in all cases where we chose two grades of the three, these two grades were always very close to each other.

Table 2: Experiment design and the subjects in the different groups.

Group	Function	Style	Subjects*	Course grades	
				All	Programming
1	Median	Regular	6	84.0±7.9	86.5±11.1
2	Diamond	Regular	5	86.6±9.0	87.0±9.8
3	Median	Non-regular	4	82.2±11.0	83.2±9.9
4	Diamond	Non-regular	5	85.6±8.1	86.2±7.7

3.5 Subjects

The subjects in this experiment are 18 3rd year students at the computer science department of Netanya Academic College, and two faculty members. In total we had 20 subjects. All participants except three were males. The average age is 24.8 (SD=8.7), and subjects are without industrial experience except one subject who had 3 years experience before his academic studies.

To ensure fair comparisons we asked the subjects about their average grades in general and in programming courses. Initially assignment was random, but later we assigned subjects to groups so as to reduce the variability in grades. Table 2 shows the average grades of the 4 groups. According to this table we see that in terms of groups and style the averages are quiet similar.

3.6 Procedure

The first author was the experimenter of all subjects. The experimenter initially gave a general overview about the experiment and the eye tracker. Participants were told that the experiment is about comprehension but were not told the specific goal. The experimenter showed each participant how the eye tracker operates and let him practice that by himself. In particular, the experimenter asked each participant to notice the track-status window that shows the subject’s eyes and their gazes. This is important because when the participant moves his head it is reflected in this window allowing the participant to learn about the valid range of his head’s movements.

Once the participant felt satisfied with the system, the experimenter asked him to calibrate. The system notification about the calibration results uses a five-level scale. Table 3 shows the different levels, their accuracy, and the number of subjects at each level. The subject who failed the calibration process was tracked manually (he was requested to move the mouse to show the code he is looking at). Luckily he was assigned to a non-regular function, so was not needed for the detailed analysis of regular ones.

After the calibration phase the subject started the experimentation. The first screen presents a general overview and instructions, and the second screen presents the program to comprehend. The participant is allowed to study the program as much time as he wants and then answers the question. While studying the program he is allowed to use off-computer means to trace the variables even if this forces him to disconnect his gaze from screen.

A post-experiment question was asked by the experimenter about the way the subject studied the program. The initial question was “how did you approach the

Table 3: Accuracy levels of the calibration process and how many subjects fall into each of these levels.

Level	Accuracy	subjects
Perfect	< 0.5°	12
Good	< 0.7°	5
Moderate	< 1.0°	2
Poor	< 1.5°	0
Re-calibrate	bad	1

program”. In the ensuing discussion subjects were also asked where they invested effort. They were also shown the heatmap of their gazes trying to learn more about the process, and asked to comment on it — specifically, whether it reflects what they think they did. Finally, they were asked to rate the function on a five-point difficulty scale.

3.7 Study Variables

The dependent variables of this study are *correctness*, *completion time*, and *visual effort*. The *correctness* variable is the score a subject achieves for answering the “what does the function do?” question. The *completion time* variable measures the time a subject spent in the function stimuli including answering the question. The rationale of considering the time of writing the answers is that subjects may re-consider the stimuli while writing their answers.

The *correctness* and *completion time* variables are not the main variables we want to analyze in this study as they have been studied already in a previous work for comparing the comprehension of regular and non-regular implementations of the same program. Thus we use them for replication and for generating a challenging environment to get a realistic measure for the *visual effort* variable.

The *visual effort* variable measures, in terms of eye movements, the effort a subject needs to invest to get an answer. It is a *latent* variable so it is measured indirectly using *observable* variables related to fixations.

Fixation is one of two types of data that are considered when using the eye tracking technique. It occurs when the eyes stabilize on an object. The other type of data is called *saccade*. It describes the rapid movements between fixations.

We derive our observable variables from fixations rather than saccades as two important mental activities occur during fixation. These activities are derived from two assumptions that relate fixation to comprehension. The *eye-mind* assumption states that processing occurs during fixation, and the *immediacy* assumption posits that interpretation at all levels of processing are not deferred [21].

The observable variables that are measured to represent visual effort are *fixation count* and *total fixation time*. While it has been shown that there is also a positive correlation between cognitive effort and pupil dilation, measurements of pupil dilation are very sensitive to lighting conditions and require substantially more data. We therefore do not report results on pupil dilation.

3.7.1 Fixation Count

This metric counts the number of fixations in a predefined area of interest (AOI).

3.7.2 Total Fixation Time

This metric measures the total fixation durations in a predefined AOI.

3.7.3 Fixation Locations

Finally, we also record fixation locations, to enable a reconstruction of the scan path. The scan path is the path that the subject's gaze traverses over the code being read.

4 Results and Analysis

4.1 Regular vs. Non-Regular Versions

4.1.1 Correctness and Time

This work is in the context of a larger study of regular code and how developers deal with it. Therefore we start by replicating our previous experiments [18] and verifying that the results are consistent. We use the following hypotheses to test the differences between regular and non-regular versions of the same program:

- H_0 : Programmers achieve similar scores and time in understanding non-regular versions as in the regular counterpart.
- H_1 : The regular versions are easier and faster to understand even though they are longer and have higher values of McCabe's cyclomatic complexity.

To test our hypotheses we initially look at the means of all regular and non-regular scores for each program, then consider the whole distribution of regular scores against the whole distribution of non-regular ones.

The four groups' scores met the normality assumption which was tested by the Shapiro-Wilks test. The *diamond* groups did not meet the equality of variance assumption so we did not assume that. As the groups are unrelated we used the independent t-test. Comparing the means of the regular and non-regular groups of the *diamond* programs yielded a significant difference between these two groups ($t(4.967) = -3.211, p = 0.012$). So we can reject the null hypothesis and accept the alternative one. When examining the groups of the *median* program the difference between the means was not significant. Thus we cannot reject the null hypothesis in this case.

One explanation for the similar scores in the median program is that the difference between the values of the regularity measure for the regular and non-regular versions is not large enough. Furthermore, the non-regular version contains a code segment that computes the median by partial sorting. As sorting is a programming plan [40] it might serve as a strong clue for the whole function understanding.

Taken together, the results show that the regular versions are not more difficult, contradicting the naive expectation that subjects of the regular version achieve lower scores due to high values of LOC and MCC.

We also compared the whole distribution of regular scores (of the two programs) to the distribution of non-regular scores. We did not use the independent t-test as the groups failed the normality assumption even under transformation. In such

Table 4: Correctness and completion time results for all implementations.

Style	Correctness average	Completion time average
Regular (median)	66.5±30.0	26.0±12.9
Regular (diamond)	92.0±9.8	25.7±12.3
	80.2±25.4	25.9±12.0
Non-regular (median)	65.0±28.7	25.2±7.4
Non-regular (diamond)	49.1±27.0	31.5±21.6
	56.1±26.7	28.7±16.3

cases it is recommended to use the Mann-Whitney non-parametric test. This test is used to compare differences between two unrelated groups when their dependent variable is not normally distributed. By running this test it was found that the regular group achieved significantly better scores than the non-regular group ($U = 24, p = 0.028$).

In terms of completion time, we also applied the independent t-test as the four groups were normally distributed and each pair also met the *equality of variance* assumption. For the two programs there was no significant difference in the means, so we cannot reject the null hypothesis.

According to Table 4 the results are quite similar for the two styles in the two programs (with slight advantage for the regular style despite its long implementations when compared to the non regular style), except for one non-regular implementation (*diamond* program) where one subject in this group spent much time and as a result the average got a relatively high value.

These results (correctness and completion time) follow those of our previous work where we used the same functions as in this work [18].

4.1.2 Difficulty of Programming Style

Besides the “what does the function do?” question, we also asked the subjects to rank the function difficulty on an ascending 5-point scale. Figure 2 shows the distribution of the subjects’ answers. In particular it shows that a third of the subjects of the non-regular implementation ranked their functions as very hard while none of the regular-implementation subjects used this level. On the opposite side, 2 subjects ranked the regular implementations as easy while not even one subject of the non-regular group used this rank.

Moreover, about 55% of the regular group ranked their functions as easy or moderate while about 78% of the non-regular group ranked their functions as hard or very hard. These results are a bit more extreme than those we obtained previously [18].

4.2 Visual Effort

4.2.1 Heat Map

One way to identify regions which garner special attention is using *heat maps*. These are designed to visualize the concentration of fixations, and can represent data from one or many subjects. Using this we can answer questions like *what locations of the stimulus are noticed by the average subject?* We use this technique to

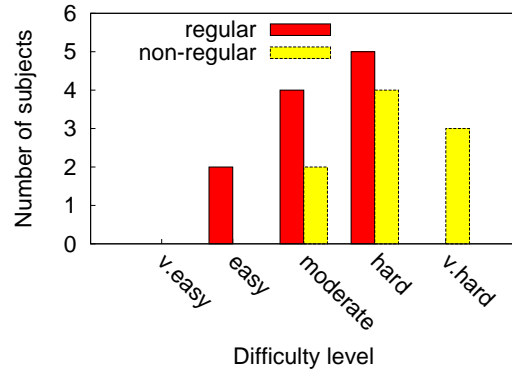


Fig. 2: Distribution of perceived difficulty ratings.

investigate whether subjects follow an obvious pattern in terms of effort allocation, and by this we answer our first research question.

Figure 3 shows the heat maps of the regular implementations (*diamond* and *median* programs). Both maps show that the average subject largely fixates on the first instance of the repeated pattern. The innermost red spot indicates the region that received the largest attention, and as we move downward the color becomes colder and regions get less attention. These figures show an aggregation of all subjects of each regular group.

The conclusion is that subjects spend more effort in the initial instances. When it comes to the last instances the examined area gets minimal focus.

Importantly, subjects did refocus on the final processing that comes after the regular repeated instances in the *median* program. This shows that attention is not just reduced with length, and subjects do not just tend to ignore the end of the function. Thus it strengthens the above result concerning reduced attention to repeated segments. The *diamond* program does not have such a final processing part.

There is no such obvious behavior in the non-regular counterparts as shown in Figure 4. Subjects generally focus on the inner-loop of the functions.

4.2.2 Areas of Interest

Heat maps show the dominant areas in the code without clear separation between repeated segments. Areas of interest are geometric areas defined by the experimenter for the sake of between-area and within-area analyses.

In both regular implementations we are interested in the repeated instances. In the *median* version we identified 8 areas of interest as shown in Figure 5 (one AOI for each instance), and in the *diamond* version we identified 4 areas of interest (Figure 6). For each AOI we count the *number of fixations* of each subject in this AOI, and the *total time* of all fixations of each subject in this AOI. We then calculate the distribution of effort across the AOIs for each subject. In other words, we find the fraction of time and fixations the subject spent on the first AOI, the fraction spent on the second AOI, and so on. This normalization allows us to

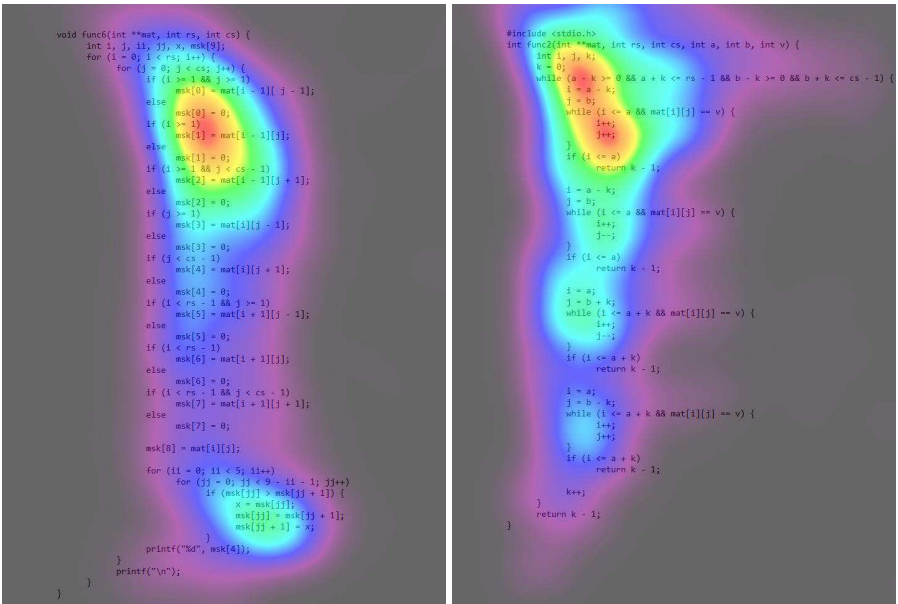


Fig. 3: Left: heat map of the regular implementation of the *median* program based on 6 subjects. Right: heat map of the regular implementation of the *diamond* program based on 4 subjects (we excluded the fifth subject due to a contradiction between his heat map and think-aloud results).

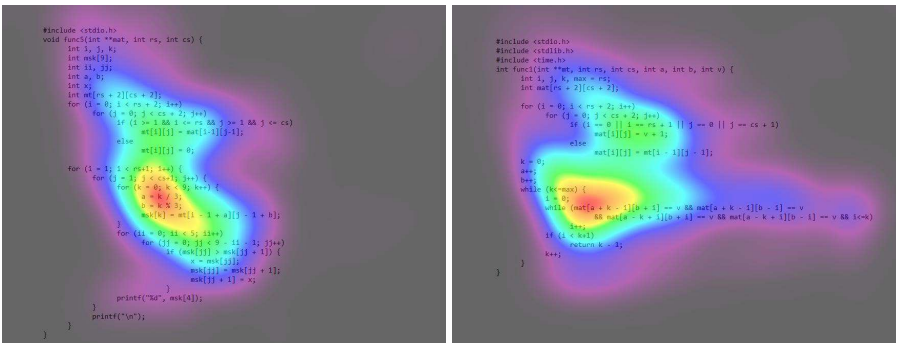


Fig. 4: Left: heat map of the non-regular implementation of the *median* program based on 3 subjects (we excluded the fourth subject as he failed the calibration process). Right: heat map of the non-regular implementation of the *diamond* program based on 5 subjects.

```

void func6(int **mat, int rs, int cs) {
    int i, j, ii, jj, x, msk[9];
    for (i = 0; i < rs; i++) {
        for (j = 0; j < cs; j++) {
            if (i >= 1 && j >= 1)
                msk[0] = mat[i - 1][j - 1]; AOI1
            else
                msk[0] = 0;
            if (i >= 1)
                msk[1] = mat[i - 1][j]; AOI2
            else
                msk[1] = 0;
            if (i >= 1 && j < cs - 1)
                msk[2] = mat[i - 1][j + 1]; AOI3
            else
                msk[2] = 0;
            if (j >= 1)
                msk[3] = mat[i][j - 1]; AOI4
            else
                msk[3] = 0;
            if (j < cs - 1)
                msk[4] = mat[i][j + 1]; AOI5
            else
                msk[4] = 0;
            if (i < rs - 1 && j >= 1)
                msk[5] = mat[i + 1][j - 1]; AOI6
            else
                msk[5] = 0;
            if (i < rs - 1)
                msk[6] = mat[i + 1][j]; AOI7
            else
                msk[6] = 0;
            if (i < rs - 1 && j < cs - 1)
                msk[7] = mat[i + 1][j + 1]; AOI8
            else
                msk[7] = 0;

            msk[8] = mat[i][j];

            for (ii = 0; ii < 5; ii++)
                for (jj = 0; jj < 9 - ii - 1; jj++)
                    if (msk[jj] > msk[jj + 1]) {
                        x = msk[jj];
                        msk[jj] = msk[jj + 1];
                        msk[jj + 1] = x;
                    }
            printf("%d", msk[4]);
        }
        printf("\n");
    }
}

```

Fig. 5: The areas of interest (AOIs) of the *median* regular implementation.

average across subjects without giving more weight to subjects that just spend more time or have more fixations.

Tables 5 and 6 show the resulting average measures of all areas of interest for the regular versions of both programs. As may be expected the distributions in terms of number of fixations and total fixation times are very similar. Obviously these results show that subjects spent more time (and thus effort) in the earlier segments, and the time spent is sharply reduced as we progress to later segments. The only exception is in the *median* results where the maximum is attained on the second AOI and not on the first. This may be a result of the fact that the AOIs

```

#include <stdio.h>
int func2(int **mat, int rs, int cs, int a, int b, int v) {
    int i, j, k;
    k = 0;
    while (a - k >= 0 && a + k <= rs - 1 && b - k >= 0 && b + k <= cs - 1) {
        i = a - k;
        j = b;
        while (i <= a && mat[i][j] == v) {
            i++;
            j++;
        }
        if (i <= a)
            return k - 1;
        i = a - k;
        j = b;
        while (i <= a && mat[i][j] == v) {
            i++;
            j--;
        }
        if (i <= a)
            return k - 1;
        i = a;
        j = b + k;
        while (i <= a + k && mat[i][j] == v) {
            i++;
            j--;
        }
        if (i <= a + k)
            return k - 1;
        i = a;
        j = b - k;
        while (i <= a + k && mat[i][j] == v) {
            i++;
            j++;
        }
        if (i <= a + k)
            return k - 1;
        k++;
    }
    return k - 1;
}

```

The code is annotated with four Areas of Interest (AOIs) indicated by pink boxes and labels to the right:

- AOI1** (top box): Contains the first while loop and its associated if statement.
- AOI2** (second box): Contains the second while loop and its associated if statement.
- AOI3** (third box): Contains the third while loop and its associated if statement.
- AOI4** (bottom box): Contains the fourth while loop and its associated if statement.

Fig. 6: The areas of interest (AOIs) of the *diamond* regular implementation.

are only 3 lines long, so attention may “spill over” to neighboring AOIs, but the first one does not have another preceding AOI.

If subjects spend more time in one area rather than others that would normally mean that this area is more complex than others. But in our study, given that the segments are pretty similar, a better interpretation is that once one segment is learned it is easier to comprehend the others.

4.2.3 AOI Transitions

Heatmaps and fixations data provide us with clues about the areas in the code where the programmers spend the most effort. However, we do not get any information about the way they progress while reading. In particular, we are interested to know how they move between AOIs. From this we can learn about their read-

Table 5: Averages relative investment in the AOIs of the *median* regular implementation. Numbers do not sum to 1 due to rounding.

	Average fraction of	
	number of fixations	total fixation time
AOI1	0.204	0.198
AOI2	0.233	0.235
AOI3	0.171	0.180
AOI4	0.103	0.104
AOI5	0.093	0.103
AOI6	0.075	0.072
AOI7	0.064	0.062
AOI8	0.053	0.042

Table 6: Averages relative investment in the AOIs of the *diamond* regular implementation. Numbers do not sum to 1 due to rounding.

	Average fraction of	
	number of fixations	total fixation time
AOI1	0.424	0.440
AOI2	0.271	0.268
AOI3	0.169	0.175
AOI4	0.134	0.115

Table 7: average relative transitions between AOIs of the *median* regular implementation.

	AOI1	AOI2	AOI3	AOI4	AOI5	AOI6	AOI7	AOI8
AOI1	<i>0.755</i>	<i>0.199</i>	0.019	0.006	0.006	0.006	0.002	0.004
AOI2	0.160	<i>0.668</i>	<u>0.146</u>	0.011	0.003	0.002	0.005	0.001
AOI3	0.024	0.181	<i>0.652</i>	<u>0.115</u>	0.009	0.002	0.005	0.007
AOI4	0.020	0.040	0.162	<i>0.590</i>	<u>0.151</u>	0.012	0.005	0.016
AOI5	0.005	0.012	0.059	0.183	<i>0.576</i>	<u>0.134</u>	0.023	0.004
AOI6	0.012	0.014	0.031	0.032	0.150	<i>0.582</i>	<u>0.142</u>	0.033
AOI7	0.016	0.024	0.014	0.024	0.035	0.183	<i>0.559</i>	<u>0.140</u>
AOI8	0.014	0.022	0.038	0.017	0.021	0.061	0.177	<i>0.647</i>

ing pattern and whether regularity affects the supposed story order in natural languages and semi-linear order in source code [5].

According to [15] a transition is a saccade from one AOI to another one. A traditional transition matrix contains the frequencies of direct transitions between all pairs of AOIs. However, as our data contains frequencies from different subjects we are exposed to possible bias as a result of having a subject with relatively high transition rates. To overcome this we normalize the data as in Tables 5 and 6. Tables 7 and 8 show the average normalized transition frequencies between AOIs of the median and diamond programs, respectively. The main diagonal in each table contains transitions within the same AOI. Generally a saccade within an AOI is not really a transition but rather a *within-AOI saccade* [15]. We included them in the transition matrix to compare with the transition rates.

The first thing to notice when examining these tables is that the most transitions occur within an AOI (main diagonal, in italics). And in both tables we see that the probability for the within-AOI saccades decreases as we progress to higher

Table 8: Transitions frequencies between AOIs of the *diamond* regular implementation.

	AOI1	AOI2	AOI3	AOI4
AOI1	<i>0.813</i>	<u>0.163</u>	0.019	0.003
AOI2	0.147	<i>0.712</i>	<u>0.114</u>	0.02
AOI3	0.025	0.144	<i>0.708</i>	<u>0.121</u>
AOI4	0.017	0.054	0.117	<i>0.809</i>

AOIs. This again indicates that subjects face more difficulty in initial AOIs and comprehension becomes easier in the repeated instances later. But this behavior does not hold for the last regular AOI. One possible explanation for this is that participants try to conclude before starting a new iteration and therefore they stay longer in last AOIs. Indeed, it seems that the last repetitive line/statement/AOI may be a special case. Beller et al. also show that the last line/statement in repetitive instances of the same code are more erroneous, and state that the psychological reasons are still an open issue [2].

The next observation is that most transitions occur between each AOI and its two adjacent AOIs. The diagonal below the main diagonal, indicated in bold, reflects transitions to the previous (upper) AOI. Similarly, each cell in the upper diagonal (underlined) reflects transitions to the next lower AOI. Interestingly, there is an advantage for the upper AOI, meaning going *back* in the code. For example, according to Table 7 the probability that the next transition from AOI2 goes to AOI1 is 0.160 whereas the probability for AOI3 is 0.146. This property is preserved for all AOIs in both tables. It is reasonable that a programmer frequently moves to the previous AOI while studying the current one as these segments are similar and it is natural that he tries to compare between them and infers about the current one based on the previous one. The interesting point is that as the programmer progresses to next AOIs the number of transitions decreases (except in few cases).

If we add AOIs for other non-repeated parts of the code, we find that the within-AOI saccades in the end block that finds the median value in the *median* program is the highest value along the main diagonal. This follows our observation from the heat map above regarding the renewed focus on non-regular segments that follow regular ones.

4.2.4 Verification of Eye Gaze Results

A post-experiment question was asked by the experimenter of each of the subjects about their approach and effort allocation to the different parts of the function. During the conversation they were presented with the heat map of their session and were asked whether this map matches their subjective impression. In particular the focus was on the subjects of the regular implementations. We summarize their responses in Table 9. According to this table more than 72% of the subjects stated clearly that they spent more time on the first instances. One subject just stated that instances are similar without any statement regarding effort allocation. Two subjects did not express awareness of the regularity issue.

The responses of *Subject20* and *Subject19* were particularly interesting. *Subject20* did not agree with his heat map and said that he did not investigate the program this way. His heat map shows one spot on the last inner while and one

Table 9: Subject opinions regarding their effort allocations in the regular implementations.

Subject	Version	Response
Subject1	diamond	I realized that once I understood the first segment, it will be easier to understand the rest due to similarity.
Subject2	median	Do not know why there is more focus on the first segment compared to others.
Subject5	median	Passed over all the code but focused on the first <i>if</i> more than others. I saw that the segments are similar so spent less efforts in the later. If the later segments were different I would spent more effort there.
Subject7	diamond	Inner loops were similar.
Subject9	median	Passed over all ifs, but it was enough to focus on a few to understand others.
Subject12	diamond	Spent much efforts at the beginning, tried to understand the loops at the beginning because I saw that they repeat themselves. In particular I realized that the differences are very small so it is easy to infer about other.
Subject13	diamond	Spent more efforts on the first inner loop because it is new for me and the rest are similar.
Subject16	median	Passed over all loops and ifs. Spent much efforts on the ifs.
Subject17	median	Most of the time in the ifs. Thought about one <i>if</i> and infer about others.
Subject19	median	I was panicked of the <i>if...else</i> statements but once saw they all similar I spent much time on those at the beginning. (She was surprised from the fact that her attention map follows the pattern of the others and said that she always thinks in a different way than others.)
Subject20	diamond	Most of the efforts were spent on the inner loops in particular the first one because it “jumps to the eyes” the similarity with others. I do not agree with the heat map (it shows he spent much efforts on the last loop), it does not reflect the real efforts I spent.

before the outermost loop. We believe that something went wrong while recording the gazes. It could be that the device was unintentionally moved by the subject or the subject himself moved.

Subject19 was surprised from the perfect matching between her mind and its heat map. She was even more surprised when she realized that her pattern follows the aggregated pattern of all other subjects. She said that she always thinks differently and it is interesting to see that this time she broke that.

4.3 Modeling Effort in Repeated Instances

We claim that not all code segments in a program should have equal weights, specifically if they are repetitions of the same pattern. The rationale is that once the developer understands one instance it is easier to understand the other instances and therefore needs less effort.

Based on this claim we observe that many widely used complexity metrics present inflated measurements of a given code. For example, the McCabe cyclo-

Table 10: Results of curve fitting to fixation data as a function of instance number in regular implementations.

Version	Measure	Equation	Model	Sig.	R^2
median	complete fixation time	Linear	$y = -0.0271x + 0.247$	0	0.558
		Logarithmic	$y = -0.0903 \ln(x) + 0.244$	0	0.509
		Quadratic	$y = 0.00112x^2 - 0.0372x + 0.264$	0	0.561
		Cubic	$y = 0.00131x^3 - 0.0166x^2 + 0.0305x + 0.199$	0	0.580
		Exponential	$\ln(y) = -0.253x - 1.190$	0	0.586
		Power	$\ln(y) = -0.819 \ln(x) - 1.245$	0	0.504
		Inverse	$y = 0.0631 + \frac{0.182}{x}$	0.002	0.361
	number of fixations	Linear	$y = -0.0262x + 0.242$	0	0.597
		Logarithmic	$y = -0.0893 \ln(x) + 0.243$	0	0.572
		Quadratic	$y = 0.00223x^2 - 0.0462x + 0.276$	0	0.615
		Cubic	$y = 0.00123x^3 - 0.0143x^2 + 0.0171x + 0.215$	0	0.633
		Exponential	$\ln(y) = -0.228x - 1.257$	0	0.616
		Power	$\ln(y) = -0.755 \ln(x) - 1.281$	0	0.558
		Inverse	$y = 0.0621 + \frac{0.184}{x}$	0	0.428
diamond	complete fixation time	Linear	$y = -0.1065x + 0.516$	0.002	0.516
		Logarithmic	$y = -0.235 \ln(x) + 0.436$	0.001	0.545
		Quadratic	$y = 0.0281x^2 - 0.247x + 0.657$	0.006	0.545
		Cubic	$y = -0.00776x^3 + 0.0863x^2 - 0.377x + 0.738$	0.02	0.546
		Exponential	$\ln(y) = -0.405x - 0.563$	0.001	0.584
		Power	$\ln(y) = -0.857 \ln(x) - 0.894$	0.001	0.568
		Inverse	$y = 0.0329 + \frac{0.416}{x}$	0.001	0.534
	number of fixations	Linear	$y = -0.0973x + 0.493$	0.001	0.558
		Logarithmic	$y = -0.215 \ln(x) + 0.421$	0.0	0.595
		Quadratic	$y = 0.0295x^2 - 0.245x + 0.641$	0.003	0.599
		Cubic	$y = 0.00277x^3 + 0.00878x^2 - 0.199x + 0.612$	0.01	0.599
		Exponential	$\ln(y) = -0.375x - 0.595$	0.0	0.633
		Power	$\ln(y) = -0.806 \ln(x) - 0.893$	0.0	0.632
		Inverse	$y = 0.0504 + \frac{0.383}{x}$	0.001	0.586

matic complexity is based on the number of conditions in the code where all conditions are treated the same. Conditions in the 10th instance of a pattern are counted just like those in the first instance. But this is misleading. As we showed, developers do not need to invest the same effort in repeated instances.

We therefore wish to build a model that reflects the effort needed to understand a repeated instance on the basis of its ordinal number. To do so we use the normalized fixation data for all the subjects and check the fit of candidate functions to this data (initially we use the raw data for all subjects, without averaging as in Tables 5 and 6). The natural candidates are various decreasing functions. Table 10 shows the models found by the SPSS curve fitting procedure for the different measures (complete fixation time and number of fixations) as a function of AOI for our two regular implementations. According to the table all the models are significant.

Overall, the best model appears to be the *exponential* model, which explains between 58.4%–63.3% of the observed variation. Not far behind it is the *cubic* model which explains about 54.6%–63.3% of the observed variation. The *quadratic* model is also a promising candidate, and captures 54.5%–61.5% of the variation. The *linear*, *power* and *logarithmic* models also explain more than 50% of the observed variation, but they are not good as the previous models. The worst is the *inverse* model which in two cases explained less than 50% of the variation.

Table 11: Results of curve fitting to averaged fixation data as a function of instance number in regular implementations.

Version	Measure	Equation	Model	Sig.	R^2
median	complete fixation time	Linear	$y = -0.0271x + 0.246$	0.001	0.885
		Logarithmic	$y = -0.0902 \ln(x) + 0.244$	0.002	0.806
		Quadratic	$y = 0.00111x^2 - 0.0371x + 0.263$	0.004	0.890
		Cubic	$y = 0.00134x^3 - 0.0170x^2 + 0.03205x + 0.196$	0.011	0.921
		Exponential	$\ln(y) = -0.241x - 1.151$	0	0.941
		Power	$\ln(y) = -0.772 \ln(x) - 1.214$	0.003	0.793
		Inverse	$y = 0.0627 + \frac{0.181}{x}$	0.03	0.57
	number of fixations	Linear	$y = -0.0261x + 0.242$	0.001	0.875
		Logarithmic	$y = -0.0893 \ln(x) + 0.242$	0.001	0.839
		Quadratic	$y = 0.00225x^2 - 0.0464x + 0.276$	0.003	0.901
		Cubic	$y = 0.00123x^3 - 0.0144x^2 + 0.0172x + 0.214$	0.009	0.928
		Exponential	$\ln(y) = -0.220x - 1.227$	0	0.947
		Power	$\ln(y) = -0.725 \ln(x) - 1.256$	0.001	0.848
		Inverse	$y = 0.0616 + \frac{0.184}{x}$	0.019	0.628
diamond	complete fixation time	Linear	$y = -0.107x + 0.516$	0.027	0.946
		Logarithmic	$y = -0.235 \ln(x) + 0.436$	0.001	0.999
		Quadratic	$y = 0.028x^2 - 0.246x + 0.656$	0.042	0.998
		Cubic	$y = -0.00766x^3 + 0.0855x^2 - 0.374x + 0.737$	0	1.0
		Exponential	$\ln(y) = -0.445x - 0.397$	0.001	0.998
		Power	$\ln(y) = -0.945 \ln(x) - 0.760$	0.013	0.975
		Inverse	$y = 0.0321 + \frac{0.416}{x}$	0.011	0.978
	number of fixations	Linear	$y = -0.0972x + 0.492$	0.035	0.931
		Logarithmic	$y = -0.215 \ln(x) + 0.421$	0.004	0.993
		Quadratic	$y = 0.0295x^2 - 0.244x + 0.640$	0.016	1.0
		Cubic	$y = 0.00266x^3 + 0.0095x^2 - 0.200x + 0.612$	0	1.0
		Exponential	$\ln(y) = -0.393x - 0.505$	0.01	0.981
		Power	$\ln(y) = -0.845 \ln(x) - 0.816$	0.008	0.984
		Inverse	$y = 0.0502 + \frac{0.382}{x}$	0.012	0.977

When examining the models in terms of programs, the *exponential* and *cubic* models are the best for the *median* program, while the *exponential* and *power* models are the best for the *diamond* program. The *inverse* model is the worst by far for the *median* program, but achieved noticeably better results for the *diamond* program.

The reason for the relatively low R^2 values of all the different models is that there are two separate sources of variation. For example, for the *median* version we have 8 AOIs. So one source of variation is the instance number of the AOI, and this is what we are trying to model. But in addition there is the variation among experimental subjects. Hence it is impossible to explain all the variation using a function of only the instances.

But as we are interested in the *average user* on the long term we can perhaps do better if we fit a model to the average value (across users) for each AOI. Thus the data is reduced to a single vector with 8 values for each measure in the *median* program, and 4 values for the *diamond* program. In fact these values are the ones shown in Tables 6 and 5.

The results of fitting to the averaged data are shown in Table 11. All model equations are pretty much similar (up to fractional digits) to those of Table 10, except for the *exponential* and *power* models where differences may be a bit bigger. All models are also statistically significant. The substantial change was in their

R^2 values. In particular, in the number of fixations measure of the median program, the *exponential* model explained about 95% of the variation while the worst model explained about 63%. Similarly, in the complete fixation time measure, the *exponential* model explained a bit more than 94% of the variation while the worst model explained 57%.

As for the *diamond* program the results show that the *cubic* model shows a perfect fit for both measures, and the *quadratic* model shows a perfect fit in for the number of fixations measure. The other models show a very high fit. However, note that with only 4 data points a cubic function can indeed pass through all the points, so this may be an overfit.

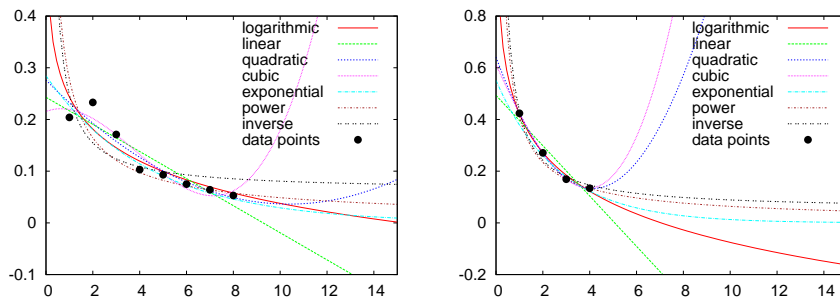


Fig. 7: Extrapolation of the model functions for the number of fixation measure from Table 10. Data points are from Tables 5 and 6. Left: *median* version. Right: *diamond* version.

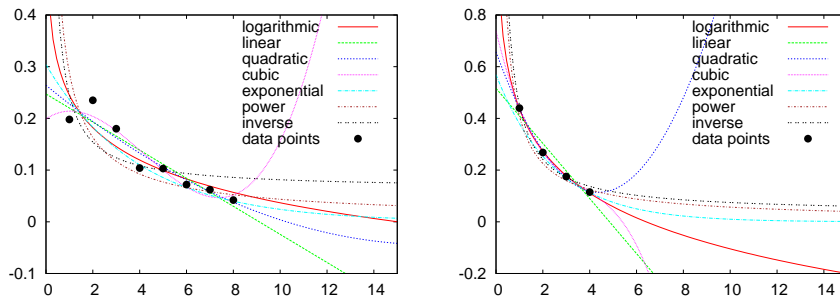


Fig. 8: Extrapolation of the model functions for the complete fixation time measure from Table 10. Data points are from Tables 5 and 6. Left: *median* version. Right: *diamond* version.

But when selecting a model one should consider the characteristics of the function and not only the R^2 of the fit. Specifically, while our data is about functions with only 4 or 8 repeated segments, actual functions (e.g. from Linux) may have many more. We therefore want a model that will have a defensible general shape,

and will at least not lead to unreasonable results if extrapolated. The seven model functions for the number of fixations on the *diamond* program from Table 10 are shown in Figure 7 (right). This shows that if we extrapolate to larger xs , the quadratic and cubic models grow to infinity, while the linear and logarithmic models attain negative values, all of which are unreasonable. As for the *median* version (Figure 7 left) the linear, logarithmic, quadratic, and cubic models behave as in the *diamond* program although not as steeply, and the *inverse* model converges to a positive value. For both programs, the exponential and power models have the more appropriate attribute of tending asymptotically to zero.

When considering the *complete fixation time measure* models which are shown in Figure 8, we see that the *logarithmic*, *linear*, *inverse*, *exponential* and *power* models behave as in Figure 7 for both programs. However, the *quadratic* model attains negative values at its minimum for the *median* program, and the *cubic* model tend to minus infinity for the *diamond* program.

Together with the previous results on goodness of fit (the R^2 values) this suggests that the exponential model has the best characteristics and this model should be preferred. Note, however, that extrapolation is always risky and therefore the reservations regarding functions that grow to infinity or to negative values for larger xs may be unfounded. Theoretically that is right, however, the number of repeated instances in the code does not grow to very large values. Therefore, for some thresholds other models could be a good fit.

4.4 Scanpath Analysis

A scanpath is defined as a set of fixations and directed saccades. They can be studied in either of two ways: by superimposing them over the stimuli, or by graphing the AOIs visited as a function of time. The second approach has the added value of adding temporal data over heatmaps and transition matrices used earlier.

We start the analysis by manual visual inspection (traditional approach) of the data at the granularity of AOIs. This is good for checking the quality of the data and providing very initial observations. We then suggest two improvements. First, we smoothed the scanpaths to get rid of distracting noise. Second, we identify recurring patterns which represent scanpath events, and analyze the scanpaths according to these events.

4.4.1 Traditional Approach

In this study the average number of fixations is relatively high, therefore showing them directly superimposed over the code would be overwhelming. To learn about the temporal aspect we adapted the traditional approach and created figures that show fixations in AOIs as a function of start time of each fixation point (Figures 9 and 10). Moreover, these figures include more AOIs than we depicted in Figures 5 and 6. These added AOIs capture the non-regular parts of the code. For the *median* program we added AOI0 for the code above AOI1. AOI9 is the single line right beneath AOI8. The two loops in the end are captured by AOI10 and the rest by AOI11. For the *diamond* program AOI0 captures the code above AOI1 and AOI5 the code at the end.

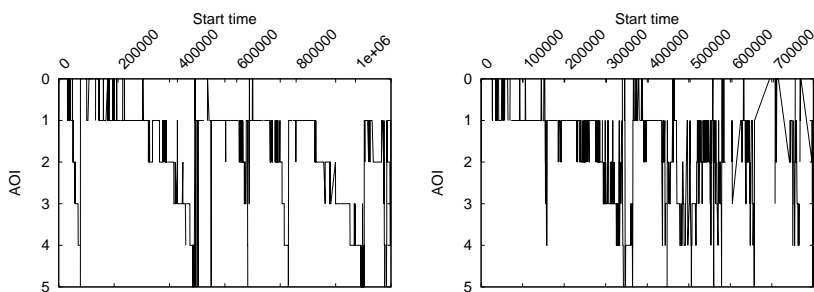


Fig. 9: Left: The fixations of *subject1* on AOIs of the *diamond* program over time. Right: The fixations of *subject12* on AOIs of the *diamond* program over time.

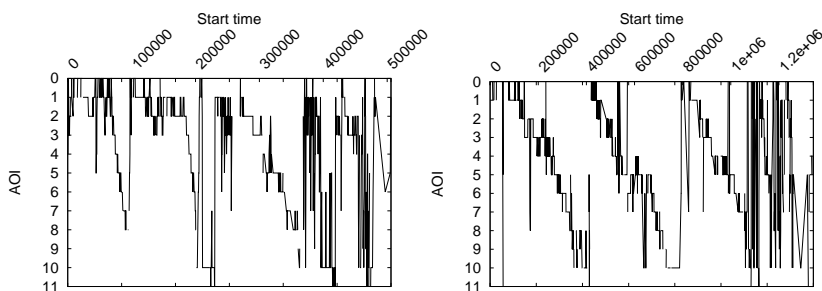


Fig. 10: Left: The fixations of *subject9* on AOIs of the *median* program over time. Right: The fixations of *subject16* on AOIs of the *median* program over time.

Figure 9 left shows that *subject1* started by looking for a while back and forth at AOIs 0 and 1, then made a quick scan of all AOIs with a very short fixation in each one, then jumped back to AOI 0 for a long session of moving back and forth between AOI 0 and 1 with many fixations in AOI 1 (horizontal lines). It is interesting to see that in this session the transitions between AOI 0 and 1 decrease as time goes on. At some point there is a jump to AOI 2 and the subject starts a new pattern where he jumps back and forth between AOIs 1 and 2 with short fixations between transitions. The same behavior is largely repeated for the pairs 2, 3 and 3, 4. For AOIs 4 and 5 there were very few back and forth moves without consecutive fixations in 5. The subject then moved to back to AOI 0, and similar patterns of traversing all the AOIs in sequence were repeated three more times.

Behaviors similar to that of *subject1* can be easily identified also in Figure 9 (right) and Figure 10. In particular, going back and forth while progressing towards lower AOIs occurs more than once within a subject's complete scanpath.

4.4.2 Scanpath Smoothing

Figures 9 and 10 provide some insights about the way programmers read regular code. However, it is quite evident that these figures are noisy in several areas. One explanation for this noisy data is probably the very large number of consecutive fixation points in a condensed areas. Another cause could be the use of

off-computer means for tracing which disconnect the gazes from the screen and re-connect them after a while. A third source of noise could be blinking, as this action moves the pupil and may be interpreted by the eye tracker as shifting the gaze. An additional problem is that these figures are based on discrete AOIs.

The purpose of providing scanpaths is to identify trends in reading regular code and not to know what happens in a specific point of time. To make these figures more clear one acceptable technique is smoothing. And to improve resolution we apply such smoothing to the raw gaze data rather than to the discretized AOI data.

Smoothing is a technique primarily used in the signal processing domain to reduce high-frequency noise in the signal. In this process points with abnormally high values compared to their adjacent points are reduced, and those with abnormally low values are increased. This process leads to a smoother signal. Another way to look at smoothing is in the frequency domain: smoothing is then achieved by low-pass filtering, which suppresses the high-frequency transitions up and down. The simplest smoothing algorithm is the *rectangular* where each point is replaced by the average of m adjacent points where m is the *smooth width*.

We applied the *rectangular* smoothing algorithm to our raw gazes of the fixation points with a smooth width of 7000 milliseconds. Thus every smoothed point is the average of a set of adjacent points that were sampled in a range of 7000 milliseconds. Setting the value of the *smooth width* to 7000 was not arbitrary. Initially we created the graphs for all subjects using smoothing widths of 1000, 3000, 5000, 7000, 10000, and 30000 milliseconds. As expected the higher the smooth width the clearer the graphs will be. However, there is a tradeoff and we may lose data.

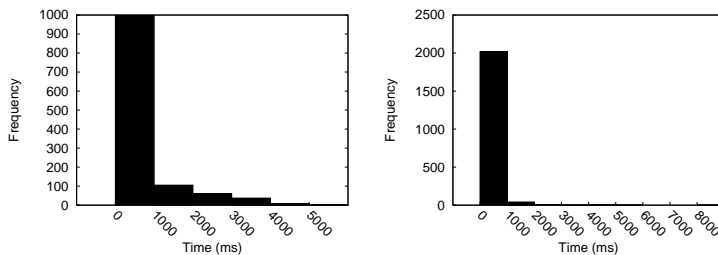


Fig. 11: Left: The distribution of dwell time of *subject1* on the *diamond* program. Right: The distribution of dwell time of *subject9* on the *median* program.

To make an intelligent choice we looked at the distribution of *dwell times*. A *dwell time* is defined as the duration of one visit to an AOI, from entry to exit (dwells of only one fixation were discarded). The most frequent dwell time can be an indicator for the appropriate smooth width. Figure 11 show two histograms of the dwell times of two subjects, one from each program. The histograms of the other subjects are pretty similar. The majority of the dwells are found to be within 1000 milliseconds. This fact already invalidates higher values such as 10000 and 30000 as candidates for the *smooth width*. As for the other candidate values we realized, by manual investigation, that the differences between the figures of all these values are not so large therefore we took the highest value we could.

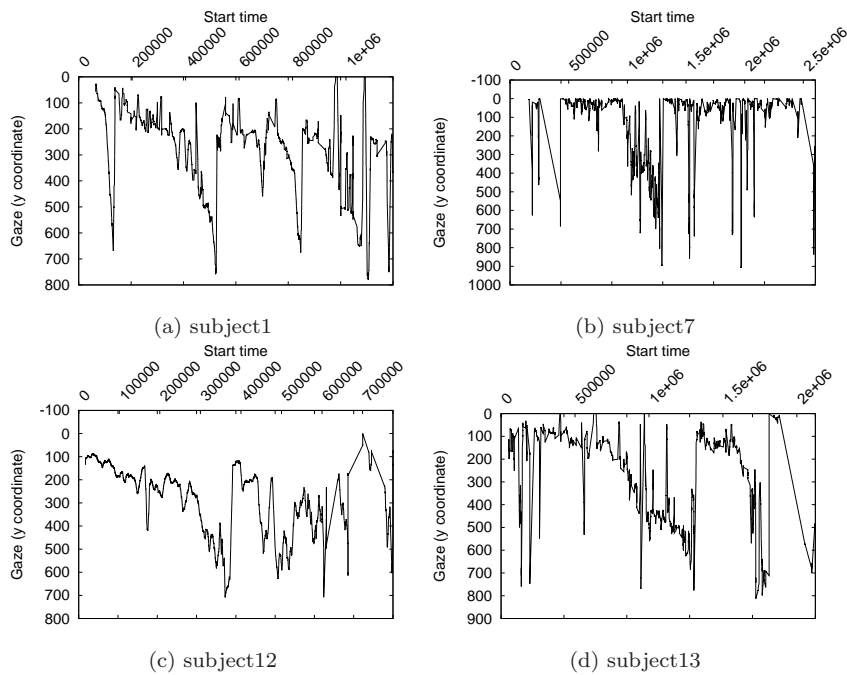


Fig. 12: Smoothed scanpaths of the *diamond* program subjects.

Figures 12 and 13 are smoothed versions of Figures 9 and 10 with additional subjects added. Note that as opposed to the noisy figures, in the smoothed ones the y axis is the y coordinate of the gazes and is not discretized into AOIs. We do not consider the x coordinate (location in the code line) as we are interested in the vertical transitions rather than horizontal ones. This is justified because the gazes nearly always remain within the scope of the code lines (and AOIs), so the pattern is captured by the y values.

According to Figure 12a *subject1* made a very quick scan of the code, and then restarted with a slow scan that includes very short back and forth moves. The progress was very slow at the beginning and then became successively quicker. This was followed by a shorter third scan that ends with a very quick move to the end, and a fourth scan which is quite similar to the third one. One key point to notice is that the start point of each new inner scan always moves forward.

Other subjects behaved differently. In Figure 12c *subject12* slowly read the beginning of the code then made a quick scan of the rest of the code. He then goes back and forth to different parts of the code in an unclear pattern. *Subject13* (Figure 12d) starts with three quick scans of most of the code, and then starts a very long period of reading almost all the code interspersed with back and forth moves. After this a new scan starts that again covers all the code but is shorter than the previous one.

Figure 12b shows the scanpath of *subject7*. This scanpath is largely different from the other scanpaths of the *diamond* program. It is true that it starts, like others, with a quick scan over the code, but then he performs a very long session

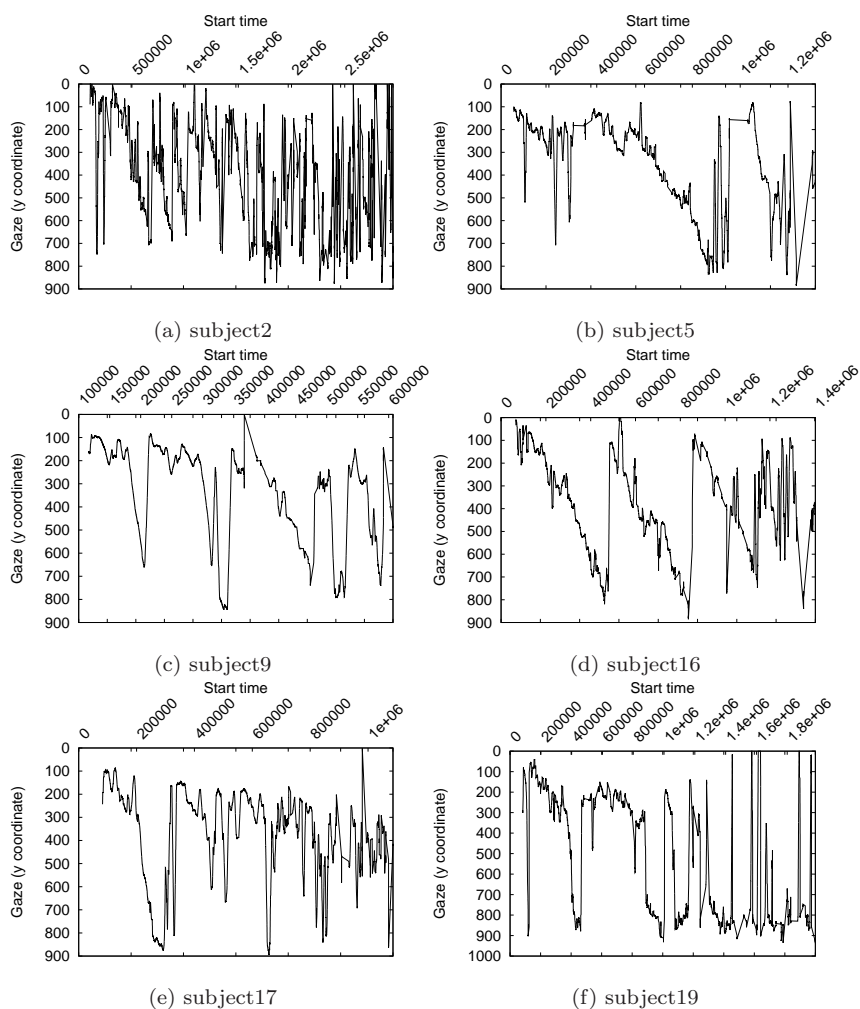


Fig. 13: Smoothed scanpaths of the *median* program subjects.

of small back and forth in the very initial parts of the code, followed by a very noisy scan to the end, and then returning to focusing on the beginning with some very quick jumps to the end. This might be a clue of comprehension difficulty as reflected by the grade this subject achieved (76.5 compared with the average of 90.4) and the very high time spent (41 min, compared to average of 26 min). A further point that might explain this is the low GPA of *subject7* which was reported at the pre-experiment questionnaire. This reflects the general methodological issue of variability among subjects in controlled experiments.

When examining the figures of the *median* program subjects we see that *subject2* scanpath is relatively noisy with an endless number of back and forth moves. The unclear trend can be explained by the very low score (10, average=66.5) he achieved and the very long time spent (48 min, average=26 min). Consistently

with other subjects, he made a quick scan at the beginning. *Subject9* initially focuses on the beginning, and then performs a quick scan of most of the code. He then repeats this pattern, this time with a slightly wider and longer scan. The third scan, however, is slower and seems to cover all the code methodically at a constant rate. *subject16* has a three similar scans where the third one does not cover all the code. As for *subject5*, he starts with a long period of back and forth moves at the initial parts in the code, then switches to a methodical scan similar to the previous two subjects. Many of the subjects end with relatively wide fluctuations going back and forth.

To summarize, it seems that subjects spent some time for a quick scan of the code (or part of it) probably to draw an overall picture about its size and structure before starting a real comprehension process. This preliminary scanning has been identified by Uwano et al. who argued that there is a correlation between the first scan time and the defect detection time [41]. Many of them also spend considerable time reading the initial part of the code, and perform slower methodical scans of all the code (or nearly all of it) later. In addition, short back and forth moves is a property that exists in all scanpaths.

4.4.3 Scanpath Events

One way to identify the reading patterns of subjects is by analyzing their scanpaths events. *Scanpath events* are temporal patterns that occur in eye-movement sequences [15].

In this section we analyze the scanpaths according to a set of events that have been published and reviewed in [15] as well as a few new events we introduce. Before delving in the analysis we introduce the events and describe them in Table 12. Some of the events were tagged as “new” which means that they were not listed in [15] and we are not aware of studies that define them as such. A possible exception is the *prescan* event, which is similar to the more specific *header scan* event identified by Uwano et al. [41]. In the related work section we provide details on other events.

We believe that the need for these new events reflects the fact that reading code is different from conventional reading [6,8]. Further, at this stage, where eye tracking is relatively new to software engineering, it is better to suggest more events and patterns to facilitate study by other researchers, which will eventually lead to convergence on an agreed set. Note that two of the new events (*prescan* and *verify*) are location dependent, occurring only at the start and the end of the session, respectively.

Interestingly, some events interact with their neighbors, which means that the behavior before the event and after it is expected. For example, a *scan* is most often followed by a *return*. The before-event and the after-event of *look ahead* are quite similar and generally *fixations*. Likewise for the *look back* event.

As noted in Table 12, each scanpath event can be represented by a single letter code. The entire scanpath can then be encoded by a string of these letters, where the size of each letter reflects the duration of the event (this was inspired by the *sequence logos* used in the bioinformatics domain [33]). For this purpose we define 4 levels: tiny, small, large, and huge. An alternative representation of duration could be repetition of the letter representing the event. However, this requires

Table 12: Scanpath events that occur in eye-movement sequences.










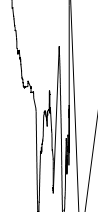
Event	Code	Description	Suggested Meaning	Illustration
Reading	R	A slow progress along the y axis (moderate slope).	Comprehension	
Fixation	F	Reading in one place.	Comprehension	
Scan	S	A fast progress along the y axis (steep slope).	Overview of code	
Look ahead	A	Jump ahead along the y axis then back.	Hypothesis testing, look for ideas	
Look back	B	Jump back along the y axis then return.	Recall definitions, verify details	
Prescan (<i>new</i>)	P	Scan for preview at $t = 0$.	Initial orientation	
Return (<i>new</i>)	T	Set y to a low value and start over.	Failure to conclude	
Forward jump (<i>new</i>)	J	Set y to a high value and continue.	Continue at a new location	
Fumbling (<i>new</i>)	M	No clear pattern or event.	Do not know what to do	
Verify (<i>new</i>)	V	multiple varied jumps at t=end	Verification	

Table 13: Event-coding strings of the scanpaths of the subjects.

Subject	Symbol representation of scanpaths
subject1	$P R_{S_B} S_T F A S A R B S / M_{F A}$
subject7	$P F R_{T F A F A F A} F_{A F A F A F A F}$
subject12	$R A R S_T R A J F_{T R V}$
subject13	$P R_A R_{B A B F_B} R B T F V$
subject2	$P_{F A A} S_T S_T S_T F A S_T F S M$
subject5	$P R A F A R T S_{B B B T F S V}$
subject9	$F A R A_{J F T F T} S T F_{J F T} F S B$
subject16	$S_T S_T S M A S_T V A$
subject17	$R S_B B T R A F A F A R A F A S T F / M A$
subject19	$P R S F_{T F A} R A F J R B F B B R B F B F B F B F B F_{B F}$

dividing the scanpath into equal units of time which may produce segments that are composed of different events.

Table 13 shows the event coding strings of the subjects' scanpaths. According to this table, the most frequent event is *Fixations* which are 22.5% of all events, followed by the *look ahead* event with 18.3%. The *look back*, *return*, *reading*, and *scan* events are at the same rank with about 12% each. The *forward jump*, *prescan*, *fumbling*, and *verification* events are relatively rare.

These frequencies should be taken with caution. Some events are visually similar and this makes it difficult to choose between them. For example, the difference between *fixation*, *reading*, and *scan* is based on the extent of steepness and this is not easily determined. In the case of fixation it is not so critical because even if some fixation events were considered as reading it still semantically belongs to comprehension. The second problematic point is that frequency simply counts items and does not take into account the duration of the event which means that not all counted events have the same contribution. For example, the *fixation* event occurrences come in all sizes, and in fact are common in each size category. So *fixation* events also have the highest total duration. When combined with the value of the *reading* events, which is reasonable as both are semantically similar, we find that comprehension is the most common event type.

An interesting pair of events are the *look ahead* and *look back*. The former is nearly equally divided between subjects, while the latter does not occur in 40% of the subjects. However, in 50% of the subjects it is divided nearly equally, and one subject contributes 42% of its total count. This means that the *look back* event is rare, and was ranked high thanks to an extreme value for one subject.

Another interesting aspect to examine is recurring patterns. The first one to notice is the *prescan* event that occurs in 60% of the subjects and is always followed by *reading* or *fixation*. A semantically similar event is *scan* which is followed by the *return* event in 55% of its occurrences. As for the *look ahead* related patterns, it seems that after a comprehension event subjects look ahead. This is evident in the coding strings as 80% of the *look ahead* occurrences are preceded by *reading* or *fixation*.

The coding strings in most cases end with *M* or *V*. Both letters indicate unclear behavior. However, in the former this behavior points to a lack of knowledge and control and in the latter it points to a verification process. We make the distinction partly based on how the subjects performed, assigning *M* to subjects who failed the mission and *V* to subjects who did better.

The ultimate purpose of event coding is to define a framework that enables pretty accurate representation of behavior to enable automatic manipulation and analysis, for example, similarity between strings and pattern identification within those strings. As we stated earlier event coding is debatable and it is subject to further work. For example, our coding so far totally ignores code coverage, which might be a very important parameter which describes how much code a *prescan* event covers, or how far in the code a *look ahead* reaches.

4.4.4 Reading Regular Code

Using the scanpath events we can also characterize the reading patterns used in regular code, and more specifically, the patterns employed in successive instances of the repeated pattern. Based on our results, it appears that the initial part of the regular functions is indeed read thoroughly, with reading or fixation events. This includes the initial repeated instances of the repeated pattern. But the later instances are actually not read, but only scanned, as evident for example in the scanpaths of subjects 5, 9, 17, and 19 in Figure 13.

5 Threats to Validity

The results of this work are subject to several threats to validity, in particular in the experimental part.

There is an obvious advantage to using a remote eye tracker over a head mounted device, especially when considering intrusiveness and how natural is the experiment environment. Yet, it is still somewhat restrictive and may influence subjects' behavior and affect their performance. For example, one subject noted a fear to move his head too much which prevented him from fully tracing the function.

Furthermore, the affective state of a participant might have an effect on the recorded data and as a result on the conclusions. For example reduced interest (due to fatigue or boredom) is indicated by smaller pupil size or less activity in the eye movements [43]. To cope with this we reduced the load relative to our previous experiments, and had participants deal with only one program style rather than both styles one after the other.

The small number of subjects in each group is another threat to validity. It is hard to avoid because of the need to conduct personal experiments with the eye

tracker, and our total of 20 is relatively high in this context when compared to other works that use eye tracking [1, 36, 37, 46].

In this work we only used two different programs and our conclusions rely on them. The hope is to generalize to additional examples. The reason we stuck to these programs is because we already used them in our previous work, and they appear to be non-trivial and realistic.

Furthermore, this work basically uses undergraduate students which could limit its generalization. However, the real question is not whether the subjects are students or professionals, but whether they are qualified for the task [9]. As our task was to understand a single function we feel students were adequate, and this is justified by the results where the vast majority indeed completed the task, even if they thought it was hard.

Two more threats are related to the areas of interest (AOIs). In our analysis each area of interest captures one repeated instance. However, repeated instances may form a continuum, therefore, areas of interest may span over two successive instances. Moreover, we used the same margins around the code of each instance, and created rectangular areas, but other options and geometric shapes are possible and may lead to slightly different results.

6 Related Work

A large body of work has been done in the area of syntactic complexity metrics. Lines of code (LOC) is a very straightforward metric that simply counts lines. Halstead defined the software science metrics including one which measures programming effort [14]. This is built on the basis of operator and operand occurrences. McCabe introduced the cyclomatic complexity metric which effectively counts the number of conditions in the code [23].

These metrics and others simply count syntactic elements. But are all lines in the code of equal importance? Do all operators or operands have the same effect? Do all constructs and conditions have the same intrinsic complexity? A few works have considered these questions and introduced weight-based metrics. For example, the cognitive functional size (CFS) metric is based on cognitive weights of the different control structure [34]. Oman et al. defined the maintainability index on the basis of three other syntactic metrics [27, 44].

Admittedly, these works have taken the syntactic metrics one step forward, but they still ignore the *context* of source code elements. In particular, repeated structures are based on the same elements but require different cognitive effort for the comprehension process. As far as we know we are the first to empirically quantify the effect of context on complexity as anticipated by Weyuker [45].

There have been other works that study repetitions in code. Vinju et al. empirically showed that the cyclomatic complexity metric overestimates understandability of Java methods. They introduced *compressed control flow patterns* (CCFPs) that summarizes consecutive repetitive control flow structure, which helps in identifying where and how many times the cyclomatic metric overestimates the complexity of the code [42]. But their focus was not on complexity or regularity, but rather on the question of whether people understand control flow by recognizing patterns. Nevertheless, in the analysis they assert that “code that looks regular is easier to chunk and therefore easier to understand”.

Sasaki et al. were even closer to our work. They recognized that one reason for large values of the MCC metric is the presence of consecutive repeated structures, and suggested that humans would not have difficulty in understanding such a source code. They then proposed performing preprocessing to simplify repeated structures for metrics measurement [32]. But both these works lack quantitative experimental evidence, and we are not aware of such evidence also in the context of clones in source code.

Repeated code has also been considered in the context of error proneness whenever modifications are required. As a first step for supporting modifications Imazato et al. investigated how repeated code is modified [16]. They revealed that more than 73% of the repeated code is modified at least once and 31-58% of the modifications on repeated code are needed for all elements.

Furthermore, Beller et al. recently introduced the concept of a *micro-clone* as an extremely short block of almost identical repeated lines or statements[2]. By analyzing many open source projects they concluded that the last repeated line (statement) is more error prone than other lines in the same block without providing a clear psychological reason for this effect.

Eye tracking has recently been used in several code comprehension studies. Sharif et al. have used eye tracking in multiple works. In [36] eye tracking was used to capture quantitative data to investigate the effect of identifier-naming conventions on code comprehension. The use of eye tracking was a better alternative to traditional means that were used in a previous similar work [3]. Likewise, in [37] they also replicate a previous work where traditional means were used. The replication uses eye tracking to extend the results and determine the effect of layout on the detection of roles in design patterns. Yusuf et al. used eye tracking to identify the most effective characteristics of UML class diagrams that support software tasks [46].

Rodeghero et al. used eyetracking for extraction of contextual information that aids in weighting different parts of code. Specifically, they present a tool that produces code summarization by extracting keywords from a given code. To teach the tool which keywords are better than others and which areas in the code get more focus by programmers they conducted a controlled experiment that uses eyetracking for studying the places the programmers read more closely than others. They concluded that programmers focus on method's signatures more than on method's invocations, and that control flow receives the least focus. This work shows the importance of context and that the same keyword may get different attention depending on its place in the code [31].

As for events (patterns) in eye tracking, Holmqvist et al. reviewed a large set of patterns in the domains of reading in general and computer use [15]. They defined *backtrack* as two saccades where the second goes in the opposite direction of the first. This event has been defined and used in computer domains [13,30] and in reading research [25]. A similar family of events are *regressions* where the saccades move in the opposite direction of the text rather than the previous saccade. The *look-back* event has been also introduced and defined as saccades to AOIs that were visited previously. Many works have considered look-back events in some way such as their relation to working memory [12] and time windows [24]. *Look-ahead* events are saccades that go forward and will be soon used or become part of a future plan. This type of event was considered by [24,28]. Two more events were reviewed in [15]: reading and scanning. The difference between them in the

reading domain was studied in [29]. In the field of program comprehension Crosby et al. and Uwano et al. identified the *scan* pattern in subjects' eye movements [8, 41]. Uwano et al. defined it as a preliminary scan of the source code, and showed that there is an inverse correlation between time spent scanning the code and the time for finding defects. Sharif et al. replicated the Uwano study but with more participants and additional eye-tracking measures [35]. This work also investigated how programmers find defects in source code and arrived at the same results regarding scan time and defect finding time. Furthermore they concluded that a correlation exists between scanning time and visual effort on relevant defect lines.

In addition to eye tracking, there have been works that use psycho-physiological sensors and functional magnetic resonance imaging in the context of program comprehension measurement [39,11]. Fritz et al. propose a novel approach for classifying task difficulty (as perceived by the developer) using data from psycho-physiological sensors. They aim at providing a way for detecting difficulties developers might experience and stop their work before they can introduce bugs into the code [11]. Siegmund et al. explored the feasibility of using fMRI for measuring program comprehension. In a controlled experiment it was shown that different patterns occurred in different regions in the participants' brains. These regions are associated with functions such as working memory, attention, and language processing [39].

These new techniques (EEG, EDA, and fMRI) are well established in other domains such as cognitive neuroscience and the above works show that they can be applied in measuring understandability. In particular, a new technique has the potential of measuring parameters that others do not, and these additional parameters can shed light on new aspects or at least make things easier to measure. Such additional techniques could be interesting also in the context of code regularity to better understand what really goes on in developers' brains while reading such code and eventually building better predicting models for the effort invested.

Our work is unique in using the results of eye tracking (specifically, the fixations data) to derive a quantitative model of effort investment. We know of no previous work that used eye tracking to quantify the parameters of a complexity model.

7 Conclusions

We conducted an eye tracking experiment to see how programmers read code when they try to understand it, for regular and non-regular versions of the same programs. Results show that in the repeated segments the programmers tend to invest more effort on the initial repetitions, and less and less on successive ones. Specifically, the time and number of fixations seem to drop of exponentially (although other models, e.g. cubic, are also possible).

One may claim that the fact that programmers invest less effort in the later repeated instances is a natural behavior which stems from fatigue or lack of interest. However the heat map of the *median* version showed that subjects renewed focus on the last segment of the function which is not part of the repetitive segments. This is also supported by the higher number of within-AOI saccades (relatively a high probability of moving within this AOI) of this segment compared with the previous ones. Thus we can claim that the reduced attention is indeed a function of the repetitions.

The reduced attention is related to the fact that repeated patterns can be anticipated and are easier to understand, as was verified by post-experiment debriefing with participants. The above observations therefore indicate that syntactic complexity metrics, which just count the number of appearances of various syntactic constructs, should be modified with context-dependent weights. For example, assuming an exponential model with a base of 2, a modified version of the MCC metric would add the full MCC of the first instance, but only $\frac{1}{2^{i-1}}$ of the MCC of the i th instance. This shows how syntactic measures can be reconciled with Weyuker’s suggestion that complexity metrics reflect context [45]. Future work is needed to see if such a weighted MCC is better than the simple version and in what contexts.

However, the current experiments are not extensive enough to enable a full model to be formulated. Additional measurement with more programs and subjects are needed in order to converge on a general model, or alternatively, to identify when different models are appropriate.

Moreover, the reduced total effort invested in successive repetitions of a code segment does not imply that all the repetitions are read in sequence at an ever increasing rate. On the contrary, we find that the way in which code is read is highly non-linear, and can be described as a sequence of recurring basic patterns such as fixation on a certain line, a linear scan of a large fraction of the code, a temporary jump back to previously read code, and more. But while the patterns themselves seem to be shared by different subjects, their use is inconsistent, with each subject using a different sequence of such patterns to read the same code. We further suggest that these patterns are likely to be used in reading all types of code, not only regular code. A lot of additional work is needed to better characterize the different patterns and how they are used.

In conducting such research, we suggest that several methodological innovations we introduced may be useful. First, we focus exclusively on the vertical dimension of the code, and ignore the location within a line of code. This allows us to plot the vertical location as a function of time. But plotting all fixations leads to very noisy graphs that are hard to interpret. The common solution is to plot dwells in AOIs instead of individual fixations. As an alternative we suggest to use smoothing, as achieved by computing a moving average. This retains the full resolution of the original data (instead of discretizing using AOIs) and enables the patterns to be seen more clearly.

In addition, we leave to future work the challenges of performing eye tracking in large scale systems (with scrolling and tabs), and a deep study into the impact of individual differences between experimental subjects.

Acknowledgments

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13). Many thanks to the reviewers of this extended version who helped to improve the paper considerably in terms of analysis and presentation relative to the original conference version.

References

1. R. Bednarik and M. Tukiainen, “An eye-tracking methodology for characterizing program comprehension processes”. In *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications*, pp. 125–132, ACM, New York, NY, USA, 2006, ISBN 1-59593-305-0, DOI: 10.1145/1117309.1117356.
2. M. Beller, A. Zaidman, and A. Karpov, “The last line effect”. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pp. 240–243, IEEE Press, Piscataway, NJ, USA, 2015.
URL <http://dl.acm.org/citation.cfm?id=2820282.2820317>
3. D. Binkley, M. Davis, D. Lawrie, and C. Morrell, “To camelCase or under_score”. In *IEEE 17th International Conference on Program Comprehension*, pp. 158–167, May 2009, DOI: 10.1109/ICPC.2009.5090039.
4. A. Brooks, J. Daly, J. Miller, M. Roper, and M. Wood, *Replication’s Role in Experimental Computer Science*. Tech. Rep. EFOCS-5-94 [RR/94/172], University of Strathclyde, 1994.
5. T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, “Eye movements in code reading: Relaxing the linear order”. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pp. 255–265, IEEE Press, Piscataway, NJ, USA, 2015.
URL <http://dl.acm.org/citation.cfm?id=2820282.2820320>
6. T. Busjahn, C. Schulte, and A. Busjahn, “Analysis of code reading to gain more insight in program comprehension”. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, pp. 1–9, ACM, New York, NY, USA, 2011, ISBN 978-1-4503-1052-9, DOI: 10.1145/2094131.2094133.
7. B. Cornelissen, A. Zaidman, and A. van Deursen, “A controlled experiment for program comprehension through trace visualization”. *IEEE Trans. Softw. Eng.* **37**(3), pp. 341–355, May–June 2011, DOI: 10.1109/TSE.2010.47.
8. M. Crosby and J. Stelovsky, “How do we read algorithms? a case study”. *Computer* **23**(1), pp. 25–35, Jan 1990, DOI: 10.1109/2.48797.
9. D. G. Feitelson, “Using students as experimental subjects in software engineering research – a review and discussion of the evidence”, Dec 2015. ArXiv:1512.08409 [cs.SE].
10. N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. Course Technology, 2nd ed., 1998.
11. T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger, “Using psycho-physiological measures to assess task difficulty in software development”. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 402–413, ACM, New York, NY, USA, 2014, ISBN 978-1-4503-2756-5, DOI: 10.1145/2568225.2568266.
12. I. D. Gilchrist and M. Harvey, “Refixation frequency and memory mechanisms in visual search”. *Current Biology* **10**(19), pp. 1209 – 1212, 2000, DOI: [http://dx.doi.org/10.1016/S0960-9822\(00\)00729-6](http://dx.doi.org/10.1016/S0960-9822(00)00729-6).
13. J. H. Goldberg and X. P. Kotval, “Computer interface evaluation using eye movements: methods and constructs”. *International Journal of Industrial Ergonomics* **24**(6), pp. 631 – 645, 1999, DOI: [http://dx.doi.org/10.1016/S0169-8141\(98\)00068-7](http://dx.doi.org/10.1016/S0169-8141(98)00068-7).
14. M. Halstead, *Elements of Software Science*. Elsevier Science Inc., 1977.
15. K. Holmqvist, M. Nyström, R. Andersson, R. Dewhurst, H. Jarodzka, and J. Van de Weijer, *Eye tracking: A comprehensive guide to methods and measures*. Oxford University Press, 2011.
16. A. Imazato, Y. Sasaki, Y. Higo, and S. Kusumoto, “Improving process of source code modification focusing on repeated code”. In *Product-Focused Software Process Improvement*, J. Heidrich, M. Oivo, A. Jedlitschka, and M. Baldassarre (eds.), *Lecture Notes in Computer Science*, vol. 7983, pp. 298–312, Springer Berlin Heidelberg, 2013, ISBN 978-3-642-39258-0, DOI: 10.1007/978-3-642-39259-7_24.
17. A. Jbara and D. G. Feitelson, “Quantification of code regularity using preprocessing and compression”. Manuscript, Jan 2014.
18. A. Jbara and D. G. Feitelson, “On the effect of code regularity on comprehension”. In *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 189–200, ACM, New York, NY, USA, 2014, ISBN 978-1-4503-2879-1, DOI: 10.1145/2597008.2597140.
19. A. Jbara and D. G. Feitelson, “JCS: Visual support for understanding code control structure”. In *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 300–303, ACM, New York, NY, USA, 2014, ISBN 978-1-4503-2879-1, DOI: 10.1145/2597008.2597801.

20. A. Jbara, A. Matan, and D. Feitelson, "High-MCC functions in the Linux kernel". *Empirical Software Engineering* **19**(5), pp. 1261–1298, 2014, DOI: 10.1007/s10664-013-9275-7.
21. M. Just and P. Carpenter, "A theory of reading: From eye fixations to comprehension". *Psychological Review* **87**, pp. 329–354, 1980.
22. J. L. Krein, L. Pratt, A. Swenson, A. MacLean, C. D. Knutson, and D. Eggett, "Design patterns in software maintenance: An experiment replication at Brigham Young University". In *2nd Intl. Workshop Replication in Empirical Software Engineering Research*, pp. 25–34, 2011, DOI: 10.1109/RESER.2011.10.
23. T. McCabe, "A complexity measure". *IEEE Trans. Softw. Eng.* **2**(4), pp. 308–320, Dec 1976, DOI: 10.1109/TSE.1976.233837.
24. N. Mennie, M. Hayhoe, and B. Sullivan, "Look-ahead fixations: Anticipatory eye movements in natural tasks". *Experimental Brain Research* **179**, pp. 427–442, 2007.
25. W. S. Murray and A. Kennedy, "Spatial coding in the processing of anaphor by good and poor readers: evidence from eye movement analyses". *Quarterly Journal of Experimental Psychology: Human Experimental Psychology* **40**, pp. 693–718+, 1988.
26. N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures". In *28th Intl. Conf. Softw. Eng.*, pp. 452–461, May 2006, DOI: 10.1145/1134285.1134349.
27. P. Oman and J. Hagemeister, "Construction and testing of polynomials predicting software maintainability". *J. Syst. & Softw.* **24**(3), pp. 251–266, Mar 1994, DOI: 10.1016/0164-1212(94)90067-1.
28. J. B. Pelz, R. Canosa, J. Babcock, and J. Barber, "Visual perception in familiar, complex tasks". In *In Proceedings of the 2001 International Conference on Image Processing*, pp. 12–15, 2001.
29. K. Rayner and M. H. Fischer, "Mindless reading revisited: Eye movements during reading and scanning are different". *Perception & Psychophysics* **58**(5), pp. 734–747, 1996.
URL <http://dx.doi.org/10.3758/BF03213106>
30. J. A. Renshaw, J. E. Finlay, D. Tyfa, and R. D. Ward, "Regressions re-visited: A new definition for the visual display paradigm". In *CHI '04 Extended Abstracts on Human Factors in Computing Systems*, pp. 1437–1440, ACM, New York, NY, USA, 2004, ISBN 1-58113-703-6, DOI: 10.1145/985921.986084.
31. P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers". In *Proceedings of the 36th International Conference on Software Engineering*, pp. 390–401, ACM, New York, NY, USA, 2014, ISBN 978-1-4503-2756-5, DOI: 10.1145/2568225.2568247.
32. Y. Sasaki, T. Ishihara, K. Hotta, H. Hata, Y. Higo, H. Igaki, and S. Kusumoto, "Pre-processing of metrics measurement based on simplifying program structures". In *19th Asia-Pacific Software Engineering Conference (APSEC)*, vol. 2, pp. 120–127, 2012, DOI: 10.1109/APSEC.2012.59.
33. T. D. Schneider and R. M. Stephens, "Sequence logos: a new way to display consensus sequences". *Nucleic Acids Res.* **18**, 1990.
34. J. Shao and Y. Wang, "A new measure of software complexity based on cognitive weights". *Canadian J. Electrical and Comput. Eng.* **28**(2), pp. 69–74, april 2003, DOI: 10.1109/CJECE.2003.1532511.
35. B. Sharif, M. Falcone, and J. I. Maletic, "An eye-tracking study on the role of scan time in finding source code defects". In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pp. 381–384, ACM, New York, NY, USA, 2012, ISBN 978-1-4503-1221-9, DOI: 10.1145/2168556.2168642.
36. B. Sharif and J. Maletic, "An eye tracking study on camelCase and under_score identifier styles". In *IEEE 18th International Conference on Program Comprehension (ICPC)*, pp. 196–205, June 2010, DOI: 10.1109/ICPC.2010.41.
37. B. Sharif and J. Maletic, "An eye tracking study on the effects of layout in understanding the role of design patterns". In *IEEE International Conference on Software Maintenance (ICSM)*, pp. 1–10, Sept 2010, DOI: 10.1109/ICSM.2010.5609582.
38. B. Shneiderman, "Measuring computer program quality and comprehension". *Intl. J. Man-Machine Studies* **9**(4), July 1977.
39. J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "Understanding understanding source code with functional magnetic resonance imaging". In *Proceedings of the 36th International Conference on Software Engineering*, pp. 378–389, ACM, New York, NY, USA, 2014, ISBN 978-1-4503-2756-5, DOI: 10.1145/2568225.2568252.

40. E. Soloway and K. Ehrlich, “Empirical studies of programming knowledge”. *IEEE Trans. Softw. Eng.* **SE-10(5)**, pp. 595–609, Sep 1984, DOI: 10.1109/TSE.1984.5010283.
41. H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto, “Analyzing individual performance of source code review using reviewers’ eye movement”. In *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications*, pp. 133–140, ACM, New York, NY, USA, 2006, ISBN 1-59593-305-0, DOI: 10.1145/1117309.1117357.
42. J. J. Vinju and M. W. Godfrey, “What does control flow really look like? Eyeballing the cyclomatic complexity metric”. In *12th Working Conf. Source Code Analysis and Manipulation*, Sep 2012.
43. H. Wang, M. Chignell, and M. Ishizuka, “Empathic tutoring software agents using real-time eye tracking”. In *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications*, pp. 73–78, ACM, New York, NY, USA, 2006, ISBN 1-59593-305-0, DOI: 10.1145/1117309.1117346.
44. K. D. Welker, P. W. Oman, and G. G. Atkinson, “Development and application of an automated source code maintainability index”. *J. Softw. Maintenance* **9(3)**, pp. 127–159, May 1997, DOI: 10.1002/(SICI)1096-908X(199705)9:3<127::AID-SMR149>3.0.CO;2-S.
45. E. J. Weyuker, “Evaluating software complexity measures”. *IEEE Trans. Softw. Eng.* **14(9)**, pp. 1357–1365, Sep 1988, DOI: 10.1109/32.6178.
46. S. Yusuf, H. Kagdi, and J. Maletic, “Assessing the comprehension of UML class diagrams via eye tracking”. In *15th IEEE International Conference on Program Comprehension*, pp. 113–122, June 2007, DOI: 10.1109/ICPC.2007.10.