On the Effect of Code Regularity on Comprehension

Ahmad Jbara[1,2] Dror G. Feitelson[2]
[1]School of Mathematics and Computer Science
Netanya Academic College, 42100 Netanya, Israel
[2]School of Computer Science and Engineering
The Hebrew University of Jerusalem, 91904 Jerusalem, Israel

ABSTRACT

It is naturally easier to comprehend simple code relative to complicated code. Regrettably, there is little agreement on how to effectively measure code complexity. As a result simple general-purpose metrics are often used, such as lines of code (LOC), McCabe's cyclomatic complexity (MCC), and Halstead's metrics. But such metrics just count syntactic features, and ignore details of the code's *global structure*, which may also have an effect on understandability. In particular, we suggest that code regularity—where the same structures are repeated time after time—may significantly reduce complexity, because once one figures out the basic repeated element it is easier to understand additional instances. We demonstrate this by controlled experiments where subjects perform cognitive tasks on different versions of the same basic function. The results indicate that versions with significant regularity lead to better comprehension, while taking similar time, despite being longer and having higher MCC. These results indicate that regularity is another attribute of code that should be taken into account in the context of studying the code's complexity and comprehension. Moreover, the fact that regularity may compensate for LOC and MCC demonstrates that complexity cannot be decomposed into independently addable contributions by individual attributes.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Complexity measures*

General Terms

Experimentation, Measurement, Human factors

Keywords

Code regularity, code complexity, MCC, LOC

1. INTRODUCTION

Some code is easy to understand, while other code may be difficult to understand. The attribute that makes code hard to understand is generally called "code complexity". It is important to be able to define and measure code complexity, because doing so may enable reliable predictions of defect density (complex code is harder to get right) and maintenance effort (complex code is harder to understand, correct, and modify), and enable identification of code that should be subjected to further scrutiny and possibly refactoring.

However, the concept of code complexity has proven to be elusive. Many complexity metrics have been proposed, but all have been attacked on various theoretical and practical grounds. Thus it seems that complexity cannot be captured by a single simple metric: different (combinations of) metrics may be needed for different projects, and interactions between the metrics should also be considered [12, 32, 29].

The McCabe cyclomatic complexity (MCC) metric is a widely used metric that measures one specific aspect of complexity, namely the cyclomatic complexity of the control flow of the code [27]. In previous work we studied MCC in the Linux kernel and some other large projects [18], and found a wide gap between the practice as reflected in these projects and the suggested thresholds on MCC in different works [27, 44, 43, 8] and tools [30, 45]. For example, we found many hundreds of functions with MCC higher than 100, whereas suggested thresholds for MCC range between 10 and 50, above which the code is considered "too complex". But some of these "high complexity" functions appeared to be well structured, and underwent extensive evolution [18]. The conclusion was that this metric does not necessarily reflect the effective complexity, especially in high-MCC functions.

Using a visualization of the structure of the code in terms of constructs and nesting, it was obvious that some of these long functions are very regular, with a certain pattern of nested constructs being repeated very many times (see Fig. 1 for an example). We speculated that this regularity is an important factor in making the functions manageable. Indeed, in a survey where participants subjectively ranked high MCC functions, we found a significant correlation between functions' subjective ratings and their regularity [18].

This last result spurred a larger research effort to better understand regularity and its implications, including

- Formally defining regularity and finding ways to measure it effectively. Our results in this area are outlined in the next section.
- Performing controlled experiments to precisely measure the impact of regularity and its relation to other metrics. The current paper is the initial part of this effort.
- Trying to understand why and how regularity affects developer performance, using experiments with eye-tracking and other means. This is ongoing work and results will be published separately.

ICPC '14, June 2-3, 2014, Hyderabad, India

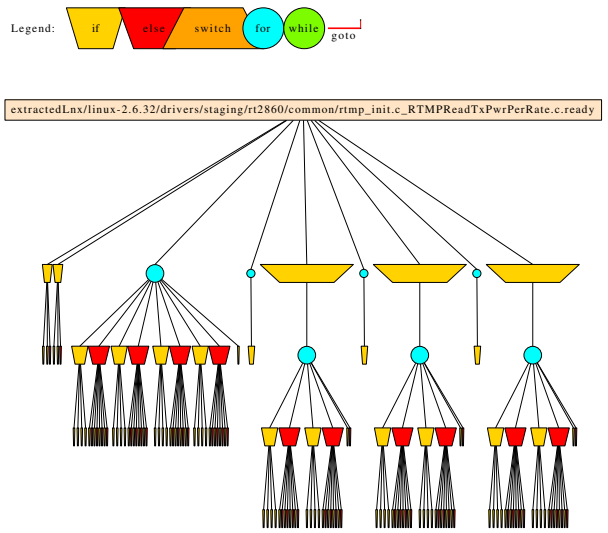

Figure 1: Code structure diagram (CSD) of a regular function from the Linux kernel.

Our focus is on establishing the effect of regularity on comprehension. In this we emphasize the interest in cognitive and human aspects of software development (as in e.g. [1, 38, 48]), as opposed to other studies which focus on direct predictions of project attributes while avoiding the human element (e.g. [34, 9, 31, 4, 28]). This complements recent works which have shown that complexity metrics, e.g. MCC, do not reflect complexity as it is perceived by humans [18, 21, 46, 11].

To enable this study we focus on trying to isolate the effect of regularity by controlling all other sources of variability. Thus we do not try to mine existing data from various projects. Instead we conduct controlled experiments using different solution styles for the same problem, where one is based on regular repeated structures and the others are not. Subjects are then asked to perform typical comprehension tasks on either of the versions, and we evaluate their performance when doing so.

The results show that, in terms of correctness, subjects working on regular code did better overall than those faced with non-regular code, while taking about the same amount of time. Since the regular versions are typically longer, this implies that the subjects spent less time on average on each line of code. We thus conclude that regularity may compensate for high MCC and LOC at least in some cases, and should therefore be taken into account alongside these commonly used metrics. Importantly, these experiments use functions of moderate length, so they also show that regularity is relevant for "normal" code and is not limited to extreme cases such as the high-MCC functions from Linux studied previously.

In the next section we review the work on quantifying regularity, followed by motivation and high-level research questions in Section 3. The methodological approach is presented in Section 4. Sections 5 and 6 presents top-level and detailed analyses of the results of our first experiment, and Section 7 analyzes the second experiment. Related work is reviewed in Section 8, and the discussion and conclusions are in Section 9.

2. MEASURING CODE REGULARITY

As mentioned above, we suggested code regularity as an explanation for the success in writing and maintaining extremely long functions in the Linux kernel [18]. Our observation was based on identifying repeated structures in the code, where the same block

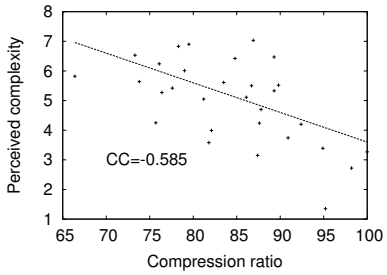

Figure 2: Correlation of perceived complexity with regularity for 30 functions from the Linux kernel, from [18].

of nested control structures (and in many cases even the same types of expressions) are repeated again and again. An example of such a function is given in Fig. 1. The challenge was then to come up with a metric that can quantify the prevalence of such structures.

Our metric for regularity is based on the observation that regularity lies at the basis of text compression. For example, the well-known Lempel-Ziv algorithm compresses text by maintaining a dictionary of observed strings. When some string is seen again, a pointer to the previous instance is used instead of the string itself [51]. Therefore *the compression ratio can be used as a metric for regularity*. To apply this insight to code regularity we conducted a systematic study of compression schemes and their effectiveness in this context. Quantifying regularity using compression has also been suggested in other domains [24].

Our study involved 5 compression schemes and 4 levels of preprocessing the code [17]. The 20 resulting combinations were evaluated based on their correlation with perceptions of the complexity of 30 functions as rated by human developers in the Linux study. The conclusion was that the most promising combination is the well-known *gzip* utility applied to a skeleton of the code obtained by removing all the statements, expressions, and comments, and leaving just the keywords, braces, and formatting (specifically indentation). The keywords are then mapped to single-letter codes to avoid effects that depend on keyword length.

Comparing this metric with the Linux perception results led to a correlation coefficient of -0.585, indicating that higher regularity as measured by the compression ratio indeed correlates with a perception of lower complexity by programmers (see Fig. 2). For comparison, the correlation coefficient of MCC and LOC with the survey results were -0.29 and 0.16, respectively.

For completeness, we also mention how we measure LOC and MCC. There are many versions of LOC (lines of code), e.g. with or without comments and blank lines. As long as one is consistent the differences are typically small, so we simply use the Linux utility *wc -l* which counts newline characters. The files did not contain comments or blank lines. MCC was defined by McCabe to be the cyclomatic number of a function's control flow graph [27]. For a graph g this is $V(g) = e - n + 2p$, where n is the number of nodes, e the number of edges, and p the number of connected components. We use the "extended" version of MCC, which also counts logical operators within predicates, as calculated by the *pmccabe* tool [2].

3. RESEARCH QUESTIONS

While the experiment cited above showed a correlation between regularity and perceived complexity, this was limited to a *perception* by human subjects. The experiment did not show that regularity actually affects programmer *performance*. Evaluating such an effect is the focus of the present paper. Specifically, we set out to

Table 1: Attributes of the three versions of the program used in experiment 1.

Version	LOC	MCC	Reg.	Description
Regular	125	32	89.7%	Loop with switch on digits. Count and track max within switch.
Sort	48	11	55.6%	Loop to find number of digits, double loop to sort them, and loop with complex if for processing.
Array	29	6	36.5%	Loop to collect digit frequencies in an array, followed by processing.

Table 2: Attributes of the two versions of the programs used in experiment 2.

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

investigate the following high-level research questions, from which we later derive more detailed ones. The questions are:

Q1. How does regularity affect programmer performance on tasks that require code comprehension in terms of correctness? Is it easier to handle regular code?

Q2. How does regularity affect programmer performance on tasks that require code comprehension in terms of the time needed to perform such tasks? Is it faster to handle regular code?

Q3. How large is the effect of regularity relative to the effect of commonly used metrics such as MCC and LOC? Can regularity compensate for high MCC and LOC?

The purpose of this paper is to report the evidence we found for the importance of code regularity as a factor that affects comprehension. In particular, we demonstrate that functions with high MCC and LOC may have enough regularity to actually be relatively simple, whereas functions that have lower MCC and LOC values can be much harder to understand. This leads us to renounce the direct use of MCC and LOC as major guidelines for software development. Instead, one should consider the *effective* MCC and LOC after taking regularity into account.

4. METHODOLOGICAL APPROACH

To study the effect of regularity on comprehension we conducted two controlled experiments with dozens of participants, different versions of 3 different programs, and 3 typical comprehension tasks.

4.1 Test Programs

It is problematic to compare different functions with different levels of regularity, because differences in the domain and functionality may confound the results without being identified. To best evaluate the effect of regularity one therefore needs programs that can be implemented in different ways while retaining precisely the same functionality. In our first experiment we use a set of three versions of one such program. The common specification for all three versions is *a function that receives a number and returns the most frequent digits of this number*. The rationale for this choice was that it is not trivial, and facilitates implementations using different approaches. Moreover, versions of this program need only simple constructs of the C language so they fit a wide range of subjects, and a minimal knowledge of the language is sufficient. The three versions are described in Table 1. To avoid the side effects of formatting on comprehension we formatted all of them using the default formatting mechanism of the *Eclipse* IDE. All test programs are available at URL http://www.cs.huji.ac.il/%7efeit/papers/RegExp/.

The main problem with these functions is that it may be claimed that their specification is just an unnatural exercise. However, we have in fact seen similar implementations in Linux [18]. Moreover, to the degree that these functions are indeed unnatural, using them leads to conservative results because they do not match programmer expectations [42]. Nevertheless, we later conducted a second experiment using two versions of each of 2 additional programs related to image processing. These programs generally operate on all the pixels of a 2D image. One version first copies the image into a larger matrix to create a boundary around it, and then does the processing in a very condensed manner. The other uses repeated structures to perform the processing while checking for different edge conditions, leading to a regular structure. These two approaches are both reasonable, and the functions are realistic. A description of these programs is given in Table 2.

In both cases, the relatively low number of functions is due to the desire to collect enough statistics about each version, while randomizing experimental aspects such as presentation order.

4.2 Task Design

The design of the tasks in experiment 1 was motivated by the comprehension framework from Pacione et al. [36], also adopted by [7]. Pacione et al. stated that *a set of typical software comprehension tasks should seek to encapsulate the principal activities typically performed during real world software comprehension*. They divided software comprehension activities into those that are performed to gain an overall understanding and those that carry out a specific task such as bug fixing. In particular, two of the list of comprehension activities they elicited from the literature were *Investigating the functionality of (a part of) the system* and *Adding to or changing the system's functionality*. Thus we define three tasks to be performed on each program version: understanding functionality, bug fixing, and adding a new feature.

In more detail, the experiment comprised three comprehension tasks which we call *phase1*, *phase2,* and *phase3*. In *phase1* the subject is presented with one program version and is asked to answer *what does the function do* (an open question). In *phase2* a buggy version of the program from *phase1* is presented and the subject is asked to find and fix the bugs in this program (without looking back at the version from *phase 1*). The subject does not know in advance the number of bugs. In *phase3* the program version from *phase1* is presented again and the subject is asked to add a feature to it. The new feature was *modify the program so that it prints an appropriate message if all digits of the original number also appear in the result*.

For bug fixing we introduced 8 bugs. The bugs types were motivated by two classification schemes identified in [3] and used in [19]. According to one scheme a bug can be classified as *omission* or *commission*. Bugs of omission are those where the programmer forgets to include some code, while bugs of commission are

Table 3: A list of the bugs that were applied in the different versions. Bugs 5 and 6 are implemented differently in different versions.

Bug no.	Type(scheme1)	Type(scheme2)	Correct	Buggy
1	Commission	Initialization	maxFreq=0	maxFreq=1
2	Omission	Computation	pValue=1	removed
3	Commission	Data	switch(number%10)	switch(number/10)
4	Commission	Data	number=number/10	number=number%10
5	Commission	Control	if (di==maxFreq)	if (di!=maxFreq)
6	Commission	Computation	case 0: maxDigits=maxDigits*10	case 0: maxDigits=maxDigits+0*pValue
7	Commission	Computation	pValue=pValue*10; maxDigits=maxDigits+2*pValue	maxDigits=maxDigits+2*pValue; pValue=pValue*10
8	Omission	?	unsigned long long int maxFreq	unsigned long int maxFreq

incorrect code which exists in the program. According to the other scheme a bug can belong to one of six types: *Initialization*, *Control*, *Computation*, *Interface*, *Data*, and *Cosmetic*. Table 3 shows the bugs and their classification according to the two schemes. We applied the same 8 bugs in all 3 versions.

In experiment 2 our emphasis was on obtaining data for more example programs. Therefore only the task of understanding what the program does was used. Each subject was asked to perform this task on two different programs, one being a regular version and the other a non-regular version. In addition, subjects were asked to provide their evaluation of the difficulty of the programs and what features made them difficult.

4.3 Grading Solutions for Correctness

In grading the solutions in experiment 1 we followed [22, 7, 41]. In [22], two graders worked together to grade a programming task in pair-programming style. Initially, they reviewed several of the solutions to determine how best to grade them and set a five-point scale. For a comprehension task each grader assessed half of the cases after agreeing on a binary rubric. A similar approach was adopted in [7]. In [41] three modification tasks were assigned a 10-point score and were graded by a TA who had extensive experience in grading student programs. A similar approach was adopted in grading the functional correctness of recalled programs.

Our approach was qualitatively similar. To evaluate the answers of the subjects in phase 1 we used a multi-pass style for 60% of the analyzed cases where two or three evaluators were involved. The grades were based on a scale of 0–100. Initially the first author performed the grading according to a personal rubric. In the second pass the first author together with a colleague made another evaluation based on a rubric that both agreed on. However, in some cases there was a substantial gap between the grades of the two passes. To resolve this and to verify the other cases where the differences were relatively small we performed a third pass. The first author selected the top 5 cases that have extreme differences and another random set of 5 normal cases. These 10 cases were evaluated by the second author based on the same evaluation rubric used in the second pass, without knowing which set of 5 they came from. The results of pass three were as follow: the 5 random normal cases were evaluated quite close to the first two passes. In the extreme cases the third evaluator was close to the grades in the first pass in two cases and to the second pass in the rest. We then used all the data from the three passes to set the final grades. In cases where the difference between the first pass and the second was relatively small we take the grade in the second pass. In moderate differences (up to 10 points) we average the grades of the first two passes. In extreme cases we average the two closest grades. Based on this experience, the other 40% of cases were evaluated by the first author alone.

A similar style was applied in the third phase. The first author made an initial evaluation. A second pass was done by the first author together with the same colleague from the first phase. The final grade was set by the average of the two passes. The second phase was evaluated in a single pass single evaluator style due to the objectivity of the answers. The grade assigned was simply the number (or percent) of bugs found.

In experiment 2 we exploited our experience from experiment 1. The first author initially graded the solutions of each group immediately after their session, as was done for experiment 1. Two weeks later he performed a second pass on all the results together in order to adjust them on a common scale.

4.4 Subjects

The subjects in experiment 1 were recruited in four sessions: 13 computer science students from the Hebrew University of Jerusalem, 27 third year and 15 second year computer science students from Netanya Academic College, and 11 computer science education students from the Technion institute of technology. Thus we have a total of 66 subjects, from which 2 were removed because they did not submit results.

All participants, except those from the Technion, were enrolled in courses taught by the authors. Participation in the experiment was anonymous and not compulsory. The analyzed group in experiment 1 is composed of 19 females and 43 males (2 did not state their gender). The average age is 27.6, the average industrial experience is 1.7 years, and the average year of study is 2.8.

Experiment 2 was similar and was done in two sessions: 24 computer science students from the Hebrew University, and 20 computer science students from Netanya Academic College, for a total of 44 subjects. Of these, 5 submitted nearly empty forms so they were removed from the analysis. The 39 remaining subjects had an average age of 24.9 and an average industrial experience of 0.6 years. There were 7 females and 32 males.

4.5 Procedure

The authors were the experimenters of all 6 sessions of the two experiments. Each participant received a booklet which contained a demographic form to be filled and the material for the different functions and tasks to be performed, including space for answers. In experiment 1 there were 3 variants of this booklet, one for each version of the program. In experiment 2 there were 4 variants, each including a single version of both programs. The variants differed in which program was represented by the regular version, and which came first. The variants were interleaved before distribution

to ensure an equal number of participants for each variant and no adjacent participants receiving the same variant. Which subject got which version was random based on seating order.

The experimenter initially gave a general overview. Participants were told that the experiment is about comprehension but were not told the specific goal. The participants were not limited in time. At the beginning and end of each phase the participants were required to write the time. A clock was projected on a screen to ensure reliability. In experiment 1 phase 2 we asked them to also write the time when they found each bug.

Participants were required not to go back to a previous phase once they finished it. This was included in the written instructions and was emphasized by the experimenter. In experiment 2, however, they were allowed to revise their evaluation of the first program after seeing the second program. To enable us to compare the first evaluation with the second (in case it was changed) we asked them to write the new evaluation at the end.

4.6 Variables and Analysis

The design of the experiments has the following independent variables: program, solution style, MCC, LOC, regularity, and demographic details. In experiment 1 there is only one program, but in experiment 2 there are two. The style is regular, array based, or sort in experiment 1, and regular or irregular in experiment 2. Our main interest is naturally in the effect of solution style and metric values. The demographic variables (gender, age, and industrial experience) are almost fairly distributed among the different groups of subjects so they should not have an effect. We believe that future work should examine the effect of experience on solution styles like those we are investigating here.

The dependent variables measure the performance of the participants in terms of time and correctness. We measure the time spent by requiring the subjects to fill in the start time and the end time for each phase. We subjectively evaluate their answers on functionality and feature adding as described above and compute the percentage of corrected bugs in the second phase.

We use Analysis of Variance (ANOVA) to test whether the means of the different groups of the solution style are identical. In this context, a generalization of the t-test is used when the number of groups to compare is larger than two. In experiment 1 we use ANOVA to investigate whether there is a significant difference between the three solution styles for correctness and completion time, while in experiment 2 we use a mixed repeated measure.

When using ANOVA there are three main assumptions that should be met: normality of the dependent variables, homogeneity of variances, and independence of cases. Regarding the first assumption, ANOVA is considered robust against the normality assumption when in each group there are at least 10 participants. For the second assumption we use Levene's test. If this fails, we can use Welch ANOVA instead of one-way ANOVA. Moreover, a post-hoc test is used to identify the statistically significant pairs. However, this test depends on the ANOVA test used: for the one-way ANOVA the Tukey test should be used, but for Welch ANOVA the Games-Howell test should be used instead. The third assumption is met as each subject is only involved in one case.

For experiment 2 we use mixed ANOVA as the tasks performed by each subject are consecutive (repeated). Mixed ANOVA compares the mean differences between groups that are split on the basis of two independent variables. One variable is the programming style which is a *within-subjects* factor. This factor specifies the conditions for each subject; in our case each subject performs two comprehension tasks one after the other. The second variable is the order factor which is a *between-subjects* factor. This factor helps splitting the subjects into two groups based on the order in which the subject receives the functions.

In this experiment our primary dependent variables are the scores the subjects achieved on each function and the time spent in understanding each function. We could also discard the order effect factor and run a repeated measure ANOVA as we counterbalanced the treatments for each subject. Counterbalancing is a technique used to minimize order effect. The primary purpose of the mixed ANOVA is to check whether there is an interaction between our within-subjects factor (regular vs. non-regular programming style) and between-subjects factor (the order of performing the tasks) in terms of effect on the dependent variable.

5. TOP-LEVEL RESULTS FROM EXPERIMENT 1

Table 4 summarizes the averages and standard deviations of the measured dependent variables. It shows for each solution style the score and time taken in the different phases. The overall column presents the average grade for the answers in all phases and the total time spent to give these answers.

According to this table the quality of answers was best for the *regular* version when considering the overall average of all phases. Next is the *array* version. Participants did the worst with the *sort* version. Regarding the average total time spent on all the phases, the participants of the *array* version did better than other versions, while those of the *sort* version were again the worst. But the differences were small.

5.1 Correctness Results

Testing the significance of the dependence of correctness scores on code metrics is complicated by the interaction between the metrics. In our test cases MCC and LOC are highly correlated with regularity. Therefore code with high MCC, which is expected to lead to worse performance, also has high regularity, which is expected to lead to good performance. And indeed we find such cases where the effects cancel out. We therefore use hypotheses which focus on one metric, and state that the commonly assumed effect need not occur:

- $H1_0$: different values of MCC or LOC do not impact the correctness of the solutions given.
- $H1$: high values of MCC or LOC do not necessarily decrease correctness and low values do not necessarily increase correctness.

due to the high correlation between them, in the analysis we treat MCC and LOC together, without going into the discussion of whether MCC adds complexity information beyond the size information that is contained in LOC [40]. We use ANOVA to compare the means of the groups of the levels of the solution style variable and determine if any of the means are statistically significantly different in the correctness dependent variable.

Since the assumption of homogeneity of variance failed, we used the Welch ANOVA. There was a statistically significant difference between the groups of the solution style levels as determined by Welch ANOVA ($F(2, 35.4) = 19.23, \rho = .000$)[1]. A Games-Howell post-hoc test showed that the *regular* ($\rho = .000$) and *array* ($\rho = .002$) subjects' groups did statistically significantly better when compared to the *sort* style. However, there was no statistically significant difference between the *regular* and *array* styles

[1] F ratio is the between-group variability divided by within-group variability. Parameters for F represent degrees of freedom. ρ is the significance of the F ratio. Significance level used is 0.05.

Table 4: Experiment 1 descriptive statistics (average ± standard deviation) of the measured dependent variables for each phase and solution style. Correctness is on a scale of 0-100 and time is in minutes.

		Phase 1		Phase 2		Phase 3		Overall	
Version	N	Correctness	Time	Correctness	Time	Correctness	Time	Correctness	Time
Regular	22	68.8±31.5	13.5±5.6	37.5±18.5	8.9±3.4	57.2±35.3	9.8±4.2	50.2±20.0	32.2±6.9
Sort	22	52.3±27.2	19.1±10.9	10.5±9.1	8.7±5.1	18.9±29.4	8.4±4.3	23.0±11.0	36.2±9.5
Array	20	69.5±33.1	10.1±7.2	35.9±28.5	8.1±4.0	40.5±35.5	9.5±6.2	39.4±23.5	27.7±10.2

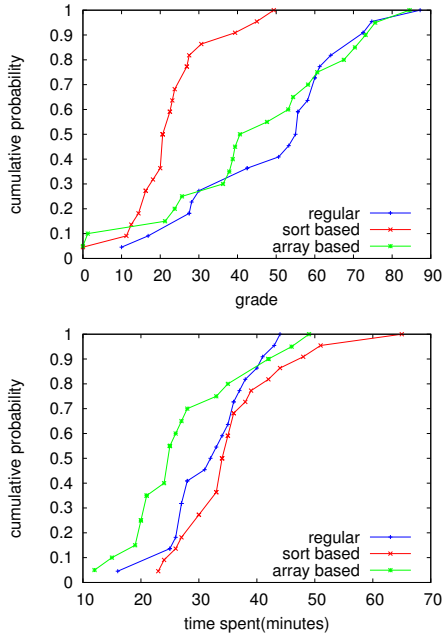

Figure 3: Distributions of average grade for all phases and total time taken for experiment 1. Cumulative probability is the probability that a specific sample be smaller than or equal to a given value.

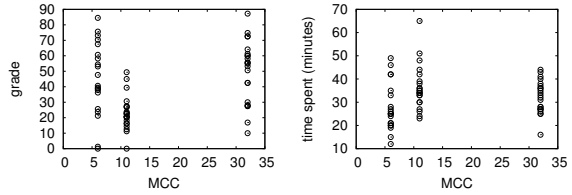

Figure 4: Distributions of grades and time vs. MCC. The MCC values 6, 11, and 32 are for the array, sort, and regular styles, respectively (per Table 1).

($\rho = .764$). This means that the *regular* style, despite its high MCC and LOC, is not necessarily worse than the *array* style which has very low MCC and LOC. In other words, there is a possibility that their means are identical. The lack of significant difference between the *array* and *regular* styles is illustrated in Fig. 3 top (which shows not only the average but the whole distribution) as their curves are pretty close and even cross each other, while both are far from the curve of the *sort* style. This figure shows for each grade on the X axis the percent of subjects who achieved this grade or less.

Fig. 4 shows a scatter plot of the different distributions, to emphasize the lack of correlation with MCC. Again, the distribution for *sort* is seen to be different from the other two. Specifically, the range from the first to the third quartile of the distribution is 15–25, as opposed to 30–65 for the other two. In addition to the large difference between them the subjects' variability in *sort* is much smaller — they all did badly. *Regular* and *array* are similar despite the wide difference in MCC.

5.2 Time Results

We now test the null hypothesis regarding the average of total time spent in all phases by the subjects of each solution style. The hypotheses are similar to the ones for correctness:

- $H2_0$: different values of MCC or LOC do not impact the time spent when performing a comprehension tasks on different solution styles.
- $H2$: high values of MCC or LOC do not necessarily increase the time spent and low values do not necessarily decrease time spent.

Again, we use ANOVA to compare the means of the groups of the levels of the solution style variable and determine whether any of the means are statistically significantly different in their time-spent dependent variable.

In this case the homogeneity assumption was met so the one-way ANOVA was used. There was statistically significant difference between the groups of the solution style level as determined by one-way ANOVA ($F(2, 61) = 4.65, \rho = .013$). A Tukey post-hoc test shows that the *array* style subjects' group did statistically significantly better when compared to the *sort* style. However, there was no statistically significant differences between the *array* and *regular* ($\rho = .249$) as well as between the *regular* and *sort* ($\rho = .311$). This lack of significant difference is illustrated in Fig. 3 where the curve of *regular* falls between the curves for *array* and *sort*. We speculate that the similarity between the timing for participants of the *sort* style and the others is a result of frustration and not spending sufficient time answering the questions, as reflected by their relatively low correctness scores.

In addition to looking at the total time, we can also consider the average time per line of code (this is discussed more below, see Table 6). In this case we get statistically significant differences between the groups of the different styles ($F(2, 61) = 2.75, \rho = .000$). The Tukey post-hoc test shows that the *regular* style is significantly better than the other styles while the *sort* style is better than the *array*. Again, this last result is probably explained by the two versions having relatively close LOC but the *sort* subjects gave up more quickly so they spent less time.

6. DETAILED ANALYSIS OF RESULTS FROM EXPERIMENT 1

In this section we analyze and test hypotheses regarding the specific phases of the experiment. For the different subjects' groups

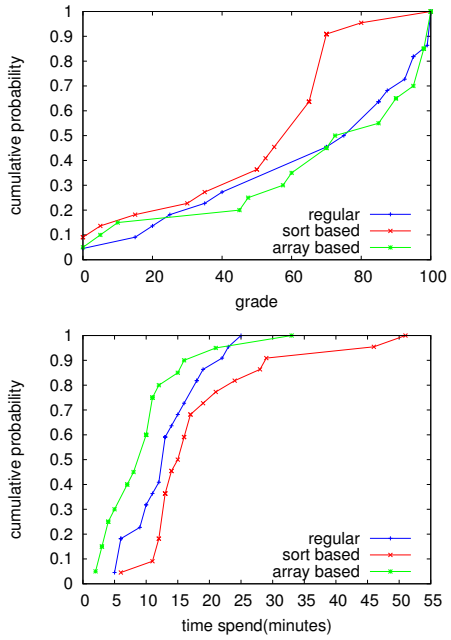

Figure 5: Phase 1 results.

and the three phases we compare the differences between the means and check whether these differences are statistically significant. Moreover, we investigate which phases impacted the overall results as presented in the previous section. We use null hypotheses like before, stating that MCC, LOC, and regularity do not affect the grades or the time needed to achieve them, and the corresponding alternative hypotheses.

6.1 Phase 1 - Understanding Functionality

There are two null hypotheses, concerning correctness grades and time, and derived from the general description of all hypotheses in this section that was presented above. Using ANOVA there was no statistically significant difference between these groups ($F(2, 61) = 2.18, \rho = 0.122$) which means that we cannot reject the null hypothesis and there is a possibility that the means are identical. This test was run after the homogeneity assumption was met.

This result indicates that despite the high MCC and LOC of the *regular* version and the low values of the other two, the subjects of the *regular* version did not do significantly worse as would be expected from functions with high MCC and LOC. Table 4 shows that the means of the *regular* version (which has the highest MCC) and the *array* version (which has the lowest MCC) are almost equal, and both are relatively far from the *sort* version. Fig. 5 can explain the large difference in the means but the lack of its significance. The curves of the three versions in this figure look the same for the lowest 35% of the cases which means that there were no differences between the groups for the subjects who achieved bad scores. However, for the remaining 65%, the figure shows that the *regular* and *array* are rather similar and both are quite different from the *sort* version. Specifically, the *sort* version subjects tend to achieve grades in the range 50–70, whereas with the other versions many subjects achieved grades above 80.

Regarding the time-spent-variable hypothesis, the ANOVA (homogeneity assumption was met) found a statistically significant difference between the three groups ($F(2, 61) = 6.27, \rho = 0.03$). A Tukey post-hoc test showed that there is a statistically significant difference between *sort* and *array* with ($\rho = 0.03$) while there are

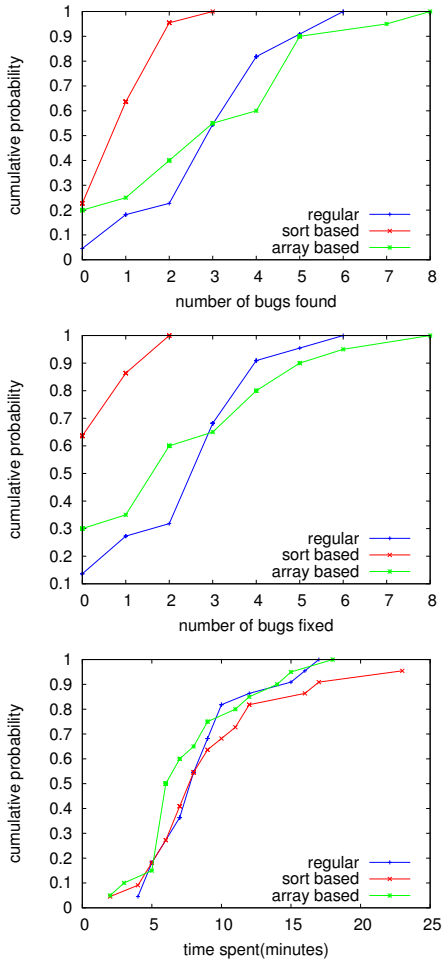

Figure 6: Phase 2 results.

no significant differences between the other pairs. However, when considering time as a function of LOC the differences are statistically significant between *regular* and the two other version, while there is no significant difference between *array* and *sort*.

When looking at the distributions of time spent (Fig. 5 bottom) we see that they have a significant overlap. This explains the fact that the averages are not statistically significantly different. However, when looking at each decile of the distribution, we find that consistently (except the last data point) *array* < *regular* < *sort* (a phenomenon called "stochastic dominance"). *sort* is also distinguished by having a much higher maximum (longer tail).

6.2 Phase 2 - Fixing Bugs

Regarding bug fixing, we had three measured variables: bugs revealed, bugs fixed, and time spent. The homogeneity assumption was not met for the two first variables, and was met for the time spent.

Welch ANOVA analysis shows that there is a statistically significant difference between the groups for the number of bugs revealed ($F(2, 34.09) = 18.69, \rho = .000$) and for the number of bugs fixed ($F(2, 32.63) = 20.97, \rho = .000$). A Games-Howell post-hoc test showed that there is a significant difference between *regular* and *sort* as well as between *array* and *sort*. This result is the same for the first two variables. No significant difference was found between *regular* and *array*. This can be seen in Fig. 6 where the curves of

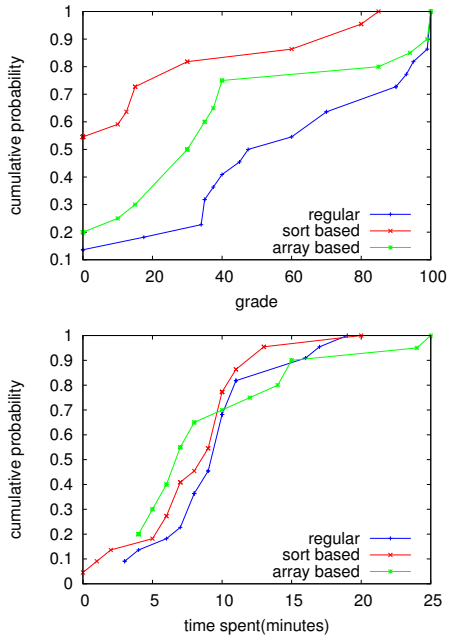

Figure 7: Phase 3 results.

the *regular* and *array* versions are quite close and the *sort* curve is far from both throughout.

Regarding time spent there were no significant differences between the groups. Again, when investigating the time as a function of LOC we get that the *regular* version has statistically significant differences when compared with the other two versions. However, the interesting result here is that the *sort* version had a statistically significant difference with regard to the *array* version (explained below in section 6.4).

6.3 Phase 3 - Adding a New Feature

In this phase the null hypotheses and their alternatives are also derived from the general description above. ANOVA was again used to compare between the means of the different groups. Regarding the correctness variable the ANOVA shows that there are statistically significant differences between the groups $F(2, 61) = 7.19, \rho = 0.002$). The homogeneity assumption also was met. A Tukey post-hoc test shows that there is a significant difference between the *regular* version and the *sort* version, while there is no significant difference between all other combinations.

Fig. 7 shows that grade distributions in phase 3 are the most distinct from each other throughout, but time is not. In particular, the *regular* distribution of the correctness variable is markedly higher than the other two throughout. This is the cause for the overall higher grades of *regular* relative to *array* in Table 4.

When investigating the time variable the ANOVA shows no significant differences. In other words, we cannot reject the null hypothesis regarding the time spent.

6.4 Fatigue Effects

It is also interesting to track the changes in subjects' behaviors from phase to phase.

One observation is that the difference between the time taken to perform phase 1 using the 3 different program styles is rather large, but it converges for the later two phases (see Table 4).

Another observation is that the biggest change is for subjects who were working with the *sort* version. For these subjects the time invested dropped to less than half going from phase 1 to 2, and stayed there for phase 3. For subjects working with the other two styles the differences were not so big, and phase 3 took more time than phase 2. Note that the low time for *sort* in phase 3 does not correspond to better results, and in fact their grades were substantially lower. We therefore suggest that a reasonable interpretation of this is that the "willingness to keep trying" of the subjects of the *sort* version decreases and they give up sooner. The interesting point is that the *sort* subjects gave up sooner, despite the fact that their average time was much shorter than an hour, while it is known that fatigue effects typically occur only in experiments that span more than an hour [13]. So maybe this reflects frustration more than fatigue.

Table 5: Results (average±standard dev.) of experiment 2.

	Correctness		Time	
order	Reg.	Non Reg.	Reg.	Non Reg
1st	80.6±25.2	47.8±29.8	15.6±7.5	12.9±4.4
2nd	64.2±33.8	53.9±28.9	13.4±9.6	15.4±6.6

7. ANALYSIS OF RESULTS FROM EXPERIMENT 2

The goal of the second experiment was to reproduce the differences between regular and non-regular code for additional functions, using the first task. The null hypothesis and alternative are again the same as above. The results are shown in Table 5.

7.1 Correctness Results

According to our analysis, the average score of the regular functions when presented first was 80.6 and when second it was 64.2. As for the non regular functions, subjects achieved much lower scores: when presented first the average score was 47.8, when presented second it was 53.9.

The very obvious conclusion is that in terms of correctness subjects did better in the regular style regardless of the order of presentation so it is most likely to be the easier style to comprehend. Indeed, according to the mixed ANOVA analysis, there was a significant main effect of the programming style being examined, $F(1, 34) = 14.68, \rho = 0.001$. This effect tells us that if we ignore the order by which the functions were given, the scores of the two styles are significantly different.

Given that each subject received two functions in this experiment, we need to consider the effect of order. According to ANOVA the main effect of the order between-subjects factor is not significant ($F(1, 34) = 0.40, \rho = 0.530$). The fact that the F ratio is less than 1 means that there was more error than variance created by the experiment, in effect negating the possibility of significance. Thus if we ignore the programming style it appears that the first and second functions would achieve similar scores.

It is also interesting to check whether there is an interaction between the presentation order and the programming style. In other words, are the scores achieved for the two styles affected by the order in which the styles are examined by subjects? According to ANOVA this effect is not significant ($F(1, 34) = 4.01, \rho = 0.053$). However, the significance level is very close to the cut-off point of 0.05, and the means of the different styles in the different groups show that an interaction seems to exist: subjects achieved much better scores with the regular style for the style presented first, whereas they achieved marginally better scores with non-regular for the style presented second. A possible interpretation is that working on a non-regular function is harder, and after this expe-

Figure 8: Distribution of perceived hardness ratings

rience subsequent performance is suppressed, whereas working on a regular function does not suppress subsequent work on another, non-regular function.

7.2 Time Results

The time spent by the subjects performing the tasks for the different styles given in different orders is quite similar. For example, the averages for the regular style function in the two groups were 15.6 (minutes) for the first group and 13.4 (minutes) for the second group. As for the non regular style subjects spent 12.9 in the first group and 15.4 in the second.

In terms of significance there were no significant differences at all. Moreover, it seems that there was no fatigue effect. Subjects spent quite similar times on the later functions as on the first functions despite the fact that they achieved much lower scores on them.

7.3 Difficulty of Programming Style

After answering the question regarding functionality, we asked the subjects to rank each function on an ordinal scale of difficulty (very easy, easy, moderate, hard, very hard). Fig. 8 shows the results. Nobody ranked the functions as very easy, and only a few as easy; together with the time spent this shows that the functions were reasonably challenging for our subjects. Many more ranked regular versions as moderate, and similarly many more ranked non-regular versions as very hard.

We also asked the subjects whether they want to change their mind regarding the ranking of the first function after seeing the second function. Out of 37 valid answers, 9 changed their mind. These 9 answers distribute as follows: 5 decreased their ranking of regular functions (made them easier), and 2 each increased and decreased their ranking of the non-regular functions. The data in Fig. 8 is from before this change, so in the final ranking the difference is even bigger.

Finally, we also asked subjects to indicate what caused them to rank the functions the way they did. We focus here on the answers given by those who ranked regular functions as easy or moderate, as opposed to those who ranked them as hard or very hard.

Subjects who ranked regular functions as easy or moderate justified this noting the discord between the initial impression and the actual complexity. For example, one wrote "*It seems a bit more daunting at first because of the length and the if statements, but they were not as complicated as they seemed to be initially.*" Several respondents even specifically identified the regularity, for example writing "*Consistency in the* if *dynasty. After understanding the first ifs, there is a consistency.*"

As for the subjects who ranked regular functions as hard, almost all of them justified this by complaining about too many ifs and loops. One also complained about bad variable names, and another suggested that refactoring was in order. These statements do not explain why it was hard to understand the functions, but rather comment on the quality of the solutions. Interestingly, when comparing their grades, the average score of all those who ranked regular functions as hard was 70.4, which is not far behind the average score (76.1) of those who ranked regular functions as easy or moderate. The impression is that subjects who ranked regular functions as hard did not experience real difficulty, but rather were dissatisfied with the solution style.

8. RELATED WORK

A large number of code complexity metrics has been defined, based on various aspects of the source code [12]. The LOC metric is the simplest one and reflects the code size. The MCC metric counts the number of decision points in the code, and as such it is considered a control-flow metric [27]. Likewise, the *Npath* metric counts the number of acyclic execution paths [33]. Halstead's software science metrics provide a measure for the programming effort [14]. Other metrics focus on the data-flow aspect of the code. The *Dep-Degree* metric counts the number of edges in the definition-use graph [5], and *Lifespan* is the average of all *spans* of all variables in a method where *span* is defined as the number of LOC between one occurrence of a variable and its next occurrence [10]. The *CFS* (cognitive functional size) belongs to the cognitive category of metrics. It is based on cognitive weights for the different control structures [39]. There are also composite metrics that combine several different aspects of the code rather than focusing on one. Oman et al. use LOC, MCC, and Halstead's metrics to define a maintainability index [35, 47].

None of the metrics that have been defined so far reflect all aspects of source code complexity, and it is hard to envision any that would. In particular, regularity in the code seems not to have been considered up to now. However, regularity has indeed been considered in areas unrelated to program code. Lipson has defined structural regularity as the compressibility of the description of the structure [24]. In this work different forms of regularity were described: repetitions, symmetries, and self similarities. In addition, this work suggested using the inverse of the description length or Kolmogorov complexity as a metric for quantifying the amount of regularity. Recently, Zhao at al. have shown that regularity leads to spontaneous attention [50]. This may be part of the explanation of why regular code is easier to understand.

It should be noted that the term "regular" is sometimes used with different meanings. For example Lozano et al. also look at regularities in the code, but they mean naming conventions, complementary methods, and interface definitions [25]. Others have considered statistical regularity, where certain aspects of the code follow a well-defined statistical distribution. For example, Zhang suggested a revised version for Halstead's length equation, based on the fact that the distribution of lexical tokens in the studied systems follow Zipf's law [49]. A similar result was introduced by [37]. These works have no connection to our notion of regularity.

Closer to our work, Chaudhary et al. conducted an experiment to study the effect of control and execution structures on program comprehension [6]. One result that contradicted their intuitive expectation was the positive correlation between the subjects' score and the control structure complexity. They attributed this to the existence of syntactic and semantic regularities in the code. They claimed that these regularities reduced the effort in the learning process and yielded higher score. Also, works on cloning and copy-paste (e.g. [26, 23, 20, 16]) are somewhat related to our work, as repeated code fragments may be a result of cloning and copy-paste. In particular, Harder et al. conducted the first controlled experiment to investigate the effect of clones on programmer performance in bug-fixing tasks [15].

Table 6: Time per line of code versions. Experiment 2 values are lower because it has only one task.

Experiment 1	
Version	Time/LOC
Regular	0.25±0.05
Sort	0.75±0.19
Array	0.95±0.35

Experiment 2	
Version	Time/LOC
Median reg	0.068±0.023
Median nonreg	0.125±0.050
Diamond reg	0.087±0.029
Diamond nonreg	0.154±0.037

To the best of our knowledge, regularity as we defined and quantified it in [18, 17] is a novel metric for software, and this is the first paper to systematically and empirically assess its effect.

9. DISCUSSION AND CONCLUSIONS

In this study we conducted controlled experiments to compare the performance of maintenance tasks when faced with a program implemented in different programming styles, where one is regular and others are not. We conclude that the regularity of code may have a large impact on comprehension by humans, and may compensate for high MCC and LOC. Thus we believe that regularity should be included among code complexity metrics alongside common metrics such as MCC and LOC, and that the interactions between these metrics should be taken into account. Importantly, these results hold for moderately long functions from common settings, extending the scope considered in our previous work which was confined to very long functions in the Linux kernel.

MCC and LOC are usually believed to be monotonically related to complexity. Thus high MCC and LOC levels supposedly lead to high levels of complexity. But in spite of the high MCC and LOC values of the *regular* version in experiment 1, which are about three times higher than the *sort* version and five times higher than the *array* version, subjects using the *regular* version almost always did significantly better than those of the *sort* version and never decidedly worse than the *array* version. These results contradict the expectations that functions with high MCC and LOC be hard to comprehend. Similar results were obtained in experiment 2.

Thus we have shown again that the MCC and LOC metrics do not fully reflect code complexity as experienced by humans. This in itself is not new, as other studies have shown various deficiencies of MCC and LOC. However, few if any have done so using controlled experiments in which MCC and LOC are the main independent variables, based on using different implementations of the same functionality. Thus our results contribute rigor to the discussion on MCC and LOC and their problems. At the same time, these results should not be interpreted as implying that striving for low MCC and LOC is inadvisable, but only that low MCC and LOC values are not necessarily good and high values are not necessarily bad.

More importantly, we suggest an explanation for *why* and *when* high MCC and LOC values are actually OK. High MCC and high LOC can result from code regularity, where the same structures are repeated many times. This led us to speculate that functions with high regularity would be comprehensible despite their high MCC and LOC. Moreover, the results also showed that functions with regular code do not take more time to comprehend, despite their length and supposed complexity. Thus regularity compensates for high MCC and LOC, and explains why they are not monotonically related to complexity. Such interactions also means that complexity cannot be decomposed into additive contributions by individual code attributes.

These results can be interpreted to mean that regularity affects the *effective* MCC and LOC of a function. In other words, regularity makes the individual lines easier to understand *on average*. Hence the effective MCC and LOC of regular code are lower than the measured MCC and LOC. Using the total time results from Tables 4 and 5 we can calculate the average time per line of code, and compare the different versions. For experiment 1, we indeed find that the time per line in the *regular* version is 3 times lower than in the *sort* version, and nearly 4 times lower than in the *array* version. Note that the real factor for *sort* may actually be even higher than indicated, because subjects faced with the *sort* version seem to have given up earlier than others.

The notion of effective MCC and LOC suggested here requires much more work to establish its validity in general. It is reasonable to assume that not all lines of code are alike. In particular, maybe repeated lines in regular code are indeed scanned much faster, while other lines are scanned at the same rate as non-regular code. This would enable an automated estimation of effective MCC and LOC based on identification of code repetition. We intend to use eye-tracking experiments to try and investigate this issue.

Another interesting point we observed is that the motivation of the subjects of the *sort* version in experiment 1 seems to decrease over the phases, as the time they spend decreases from phase to phase. Such a decrease does not occur with the other versions. Taken together with the low grades that the *sort* subjects received in terms of correctness, these observations may indicate that they become frustrated with the difficulty to cope with this version of the code. This was the reason for the post-test briefings used in experiment 2 to assess the subjective feelings of the different subjects, and complement the objective metrics that were collected.

Our work suffers from several threats to validity. We suggest that regularity is an additional attribute that affects complexity, but in this work we examined only several regular functions. While our results are also supported by previous work on perceived complexity of Linux functions [18], much additional work remains on quantifying the effect of regularity and on measuring regularity. In particular, we need to look into different styles of regularity. There is also the danger that the specific programs used induce some confounding effects. For example, one of the respondents of experiment 2 mentioned problematic variable naming. However, we applied the same level of naming in all versions, therefore minimizing the effect of naming on one version rather than on others.

Another threat is that the demographics of our subjects may not be representative, or may interact with solution styles to have an effect on comprehension. While using students as subjects is not optimal, this has often been done before. In our analysis we found no demographic-related effects, but the groups resulting from factorization are quite small to generalize.

For future work, an interesting issue is the possible relationship between regularity and bug proneness, especially in the long run. It is possible that long regular code will eventually lead to more bugs, because changes would most probably have to be replicated in the repeated constructs, and some may be missed. Harder et al., in a controlled experiment, did not succeed to achieve decisive results regarding the effect of clones on programmer performance in bug-fixing tasks [15]. Therefore comprehensibility may not be the same as code quality. This effect is hard to study, as it will require data about the long-term usage of regular functions.

10. ACKNOWLEDGMENTS

Thanks to Orit Hazzan, Gershon Kagan, Noa Regonis, Niv Reggev, and Ran Hassin for discussions of this work and help with the experiments. This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13).

11. REFERENCES

[1] V. Arunachalam and W. Sasso, "*Cognitive processes in program comprehension: An empirical analysis in the context of software reengineering*". *J. Syst. & Softw.* **34(3)**, pp. 177–189, Sep 1996, DOI:10.1016/0164-1212(95)00074-7.

[2] P. Bame, "*McCabe-style function complexity*". URL http://parisc-linux.org/ bame/pmccabe/overview.html.

[3] V. Basili and R. Selby, "*Comparing the effectiveness of software testing strategies*". *IEEE Trans. Softw. Eng.* **SE-13(12)**, pp. 1278–1296, 1987, DOI:10.1109/TSE.1987.232881.

[4] N. Bettenburg, M. Nagappan, and A. E. Hassan, "*Think locally, act globally: Improving defect and effort prediction models*". In 9th *Intl. Workshop Mining Softw. Repositories*, pp. 60–69, Jun 2012, DOI:10.1109/MSR.2012.6224300.

[5] D. Beyer and A. Fararooy, "*A simple and effective measure for complex low-level dependencies*". In 18th *IEEE Intl. Conf. Program Comprehension*, pp. 80–83, 2010, DOI:10.1109/ICPC.2010.49.

[6] B. Chaudhary and H. Sahasrabuddhe, "*Two dimensions of program comprehension*". *Intl. J. Man-Machine Studies* **18(5)**, pp. 505–511, 1983.

[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] B. Curtis, J. Sappidi, and J. Subramanyam, "*An evaluation of the internal quality of business applications: Does size matter?*" In 33rd *Intl. Conf. Softw. Eng.*, pp. 711–715, May 2011, DOI:10.1145/1985793.1985893.

[9] G. Denaro and M. Pezzè, "*An empirical evaluation of fault-proneness models*". In 24th *Intl. Conf. Softw. Eng.*, pp. 241–251, May 2002, DOI:10.1145/581339.581371.

[10] J. Elshoff, "*An analysis of some commercial PL/I programs*". *IEEE Trans. Softw. Eng.* **SE-2(2)**, pp. 113–120, 1976, DOI:10.1109/TSE.1976.233538.

[11] J. Feigenspan, S. Apel, J. Liebig, and C. Kastner, "*Exploring software measures to assess program comprehension*". In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pp. 127–136, Sept 2011, DOI:10.1109/ESEM.2011.21.

[12] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. Course Technology, 2nd ed., 1998.

[13] M. Fisher, A. Cox, and L. Zhao, "*Using sex differences to link spatial cognition and program comprehension*". In 22nd *Intl. Conf. Softw. Maintenance*, pp. 289–298, 2006, DOI:10.1109/ICSM.2006.72.

[14] M. Halstead, *Elements of Software Science*. Elsevier Science Inc., 1977.

[15] J. Harder and R. Tiarks, "*A controlled experiment on software clones*". In 20th *IEEE Intl. Conf. Program Comprehension*, pp. 219–228, 2012, DOI:10.1109/ICPC.2012.6240491.

[16] P. Jablonski and D. Hou, "*Aiding software maintenance with copy-and-paste clone-awareness*". In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pp. 170–179, June 2010, DOI:10.1109/ICPC.2010.22.

[17] A. Jbara and D. G. Feitelson, "*Quantification of code regularity using preprocessing and compression*". manuscript, Jan 2014.

[18] A. Jbara, A. Matan, and D. G. Feitelson, "*High-MCC functions in the Linux kernel*". *Empirical Softw. Eng.* 2013, DOI:10.1007/s10664-013-9275-7. Accepted for publication.

[19] N. Juristo, S. Vegas, M. Solari, S. Abrahao, and I. Ramos, "*Comparing the effectiveness of equivalence partitioning, branch testing and code reading by stepwise abstraction applied by subjects*". In 5th *IEEE Intl. Conf. Software Testing, Verification and Validation*, pp. 330–339, 2012, DOI:10.1109/ICST.2012.113.

[20] C. J. Kasper and M. W. Godfrey, "*"Cloning considered harmful" considered harmful: Patterns of cloning in software*". *Empirical Softw. Eng.* **13(6)**, pp. 645–692, Dec 2008, DOI:10.1007/s10664-008-9076-6.

[21] B. Katzmarski and R. Koschke, "*Program complexity metrics and programmer opinions*". In 20th *IEEE Intl. Conf. Program Comprehension*, Jun 2012.

[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] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "*CP-Miner: A tool for finding copy-paste and related bugs in operating system code*". In 6th *Symp. Operating Systems Design & Implementation*, pp. 289–302, Dec 2004.

[24] H. Lipson, "*Principles of modularity, regularity, and hierarchy for scalable systems*". *Journal of Biological Physics and Chemistry* **7(4)**, pp. 125–128, 2007.

[25] A. Lozano, A. Kellens, K. Mens, and G. Arevalo, "*Mining source code for structural regularities*". In 17th *Working Conf. Reverse Engineering*, pp. 22–31, Washington, DC, USA, 2010, DOI:10.1109/WCRE.2010.12.

[26] J. Mayrand, C. Leblanc, and E. M. Merlo, "*Experiment on the automatic detection of function clones in a software system using metrics*". In *Intl. Conf. Softw. Maintenance*, pp. 244–253, Nov 1996, DOI:10.1109/ICSM.1996.565012.

[27] T. McCabe, "*A complexity measure*". *IEEE Trans. Softw. Eng.* **2(4)**, pp. 308–320, Dec 1976, DOI:10.1109/TSE.1976.233837.

[28] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "*Local versus global lessons for defect prediction and effort estimation*". *IEEE Trans. Softw. Eng.* **39(6)**, pp. 822–834, Jun 2013, DOI:10.1109/TSE.2012.83.

[29] T. Menzies, J. Greenwald, and A. Frank, "*Data mining code attributes to learn defect predictors*". *IEEE Trans. Softw. Eng.* **33(1)**, pp. 2–13, Jan 2007, DOI:10.1109/TSE.2007.256941.

[30] MSDN Visual Studio Team System 2008 Development Developer Center, "*Avoid excessive complexity*". URL msdn.microsoft.com/en-us/library/ms182212.aspx, undated. (Visited 23 Dec 2009).

[31] N. Nagappan and T. Ball, "*Static analysis tools as early indicators of pre-release defect density*". In 27th *Intl. Conf. Softw. Eng.*, pp. 580–586, May 2005, DOI:10.1145/1062455.1062558.

[32] 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.

[33] B. A. Nejmeh, "*Npath: a measure of execution path complexity and its applications*". *Comm. ACM* **31(2)**, pp. 188–200, Feb 1988.

[34] N. Ohlsson and H. Alberg, "*Predicting fault-prone software modules in telephone switches*". *IEEE Trans. Softw. Eng.* **22(12)**, pp. 886–894, Dec 1996, DOI:10.1109/32.553637.

[35] 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.

[36] M. Pacione, M. Roper, and M. Wood, "*A novel software visualisation model to support software comprehension*". In 11th *Working Conf. Reverse Engineering*, pp. 70–79, 2004, DOI:10.1109/WCRE.2004.7.

[37] D. Pierret and D. Poshyvanyk, "*An empirical exploration of regularities in open-source software lexicons*". In 17th *IEEE Intl. Conf. Program Comprehension*, pp. 228–232, 2009, DOI:10.1109/ICPC.2009.5090047.

[38] J. Rilling and T. Klemola, "*Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics*". In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pp. 115 – 124, may 2003, DOI:10.1109/WPC.2003.1199195.

[39] 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.

[40] M. Shepperd, "*A critique of cyclomatic complexity as a software metric*". *Software Engineering J.* **3(2)**, pp. 30–36, Mar 1988.

[41] B. Shneiderman, "*Measuring computer program quality and comprehension*". *Intl. J. Man-Machine Studies* **9(4)**, July 1977.

[42] 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.

[43] SRI, "*Software technology roadmap: Cyclomatic complexity*". In URL www.sei.cmu.edu/str/str.pdf, 1997. (Visited 28 Dec 2008).

[44] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "*Code quality analysis in open source software development*". *Inf. Syst. J.* **12(1)**, pp. 43–60, Jan 2002, DOI:10.1046/j.1365-2575.2002.00117.x.

[45] VerifySoft Technology, "*McCabe metrics*". URL www.verifysoft.com/en_mccabe_metrics.html, Jan 2005. (Visited 23 Dec 2009).

[46] 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.

[47] 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.

[48] S. Xu, "*A cognitive model for program comprehension*". In 3rd *ACIS Intl. Conf. Softw. Eng. Research, Management, & Apps.*, pp. 392–398, Aug 2005, DOI:10.1109/SERA.2005.2.

[49] H. Zhang, "*Exploring regularity in source code: Software science and Zipf's law*". In 15th *Working Conf. Reverse Engineering*, pp. 101–110, 2008, DOI:10.1109/WCRE.2008.37.

[50] J. Zhao, N. Al-Aidroos, and N. B. Turk-Browne, "*Attention is spontaneously biased toward regularities*". *Psychological Sci.* **24(5)**, pp. 667–677, May 2013, DOI:10.1177/0956797612460407.

[51] J. Ziv and A. Lempel, "*A universal algorithm for sequential data compression*". *IEEE Trans. Information Theory* **IT-23(3)**, pp. 337–343, May 1977, DOI:10.1109/TIT.1977.1055714.