

**Regularity of Code: A New Structural
Property and its Effect on Code
Complexity and Comprehension**

Thesis submitted for the degree of
Doctor of Philosophy

By

Ahmad Jbara

Submitted to the Senate of the Hebrew University of Jerusalem

December 2015

This work was carried out under the supervision of Prof. Dror Feitelson.

Acknowledgments

Five years of hard work, a lot of interest, and impressive innovations are over. This study would not have happened without the help of many people who escorted me over the course of this study each in his/her way. They deserve a great appreciation and I would like to seize the opportunity to thank each of them.

Firstly, I would like to express my sincere gratitude to my advisor Prof. Dror G. Feitelson for his continuous support, endless patience, and encouragement in failure moments. His support starts from editing issues and ends at pure scientific research discussions. Alongside his great support he knew when to say “I do not know, do it yourself”. I am convinced that this approach largely contributed in shaping my point of view towards research independency.

Besides my advisor, my sincere thank also goes to Prof. Amos Israeli (our dean at the computer science department of the Netanya Academic College), for his interest, encouragement, and attention.

Things do not work without the support of your family. I want to thank my parents, my two brothers and three sisters, the parents of my wife and her brothers for their close attention and for being in the background all the time to back me. A special thank goes to my great father, Hussein Jbara, for driving me to this study and for his major moral support.

Last but not the least, I want to say I am sorry to those who went to their beds without saying good night and woke up without saying good morning. My two little sons and my great wife underwent a very hard period. I was not there for them to be a true father and husband. I promise to compensate and hug you more for ever.

Abstract

Program comprehension is the process of building a mental model of a given source code. It lies at the basis of any software maintenance activity such as fixing bugs and adding new features.

Maintenance is important because it consumes a large part of the resources allocated along a software lifecycle. Therefore, program comprehension is also important not only for the fact that it is a preliminary vital step in maintenance, but also for its criticality for the success of this activity. In particular, the better the code is understood the more likely the maintenance will be successful.

However, comprehension is directly affected by complexity; the less complex is the code the easier a programmer can understand it. For example, nesting and non-linear flow are factors that affect complexity and probably make code harder to comprehend. Therefore, we wish to enhance comprehension by making programs less complex. This would be feasible if we could measure the complexity of programs. As summarized by H.James Harrington, “measurement is the first step that leads to control and eventually to improvement. If you cannot measure something, you cannot understand it. If you cannot understand it, you cannot control it. If you cannot control it, you cannot improve it.”

Over the years very many complexity metrics have been suggested. The large number of complexity metrics is probably an indication of a real difficulty in defining an ideal metric that is capable of reflecting complexity by providing one simple number.

The key problem of many existing metrics is that they estimate complexity by analyzing code syntax. For example, the McCabe’s cyclomatic complexity (MCC) is based on the number of independent execution paths in the code, which is equivalent to the number of conditions plus one. This metric is the most widely used since its introduction in 1976. Despite its popularity, it has been criticized over the years especially for the fact that it discards data flow

and focuses on just counting program elements without paying attention to their context in the code.

The dissatisfaction with the state of the practice leads to the research question of why MCC and other metrics are not good enough to reflect effective complexity? what do these metrics miss and what is needed to define better metrics?

In this thesis, we introduce regularity as an additional factor that affects complexity. Specifically, regular code has many repetitions of a pattern, and successive instances become easier to comprehend given the experience with previous ones.

The innovation of regularity is introducing context awareness and sensitivity: a piece of code can have different complexity depending on neighboring code. Interchangeably, code complexity is no longer absolute but depends on the context. In particular, in regularity, the initial instances of some patterns have higher complexity than those that appear later in the code as the effective complexity of the repeated instances is reduced due to leveraging the understanding gained in the initial instances.

Conversely, the current metrics, including MCC, unconsciously neglect code context. Specifically, they simply count code elements. For example, the lines of code (LOC) metric count lines and MCC counts conditions.

It is important however to adopt an empirical approach to explore what effects exist in practice. In particular, we conducted a family of diverse experiments that encompass a wide range of experiments starting with a very basic subjective ranking up to very sophisticated eyetracking based experiments. The results show that subjects sometimes estimated functions with very high cyclomatic complexity as not complex. Moreover, they performed comprehension tasks and achieved better results in regular code (despite having higher cyclomatic complexity) when compared with its non-regular counterpart.

Beyond the immediate effects we already presented, our empirical investigation revealed more insights about reading in general and reading regular code in particular. Specifically, the results show that code reading is very non-linear as opposed to reading in natural language text. As for regular code, it seems that it is not read completely where parts of it are only scanned.

Having considered the way programmers read regular code, this led to an event-based framework for analyzing code reading. In particular, we extended the set of events (e.g, reading, scanning) that have been suggested in previous studies and suggested a way of coding these events for further analysis.

It is true that in this work we introduce a new property that affects complexity and challenges existing metrics, but regularity is just an example of such a factor and does not solve everything. It introduces new and wider considerations that can serve as good guidelines for investigating new factors.

This thesis is structured as a collection of papers that introduced and analyzed the above ideas. The first paper examines the state of the practice of real functions with very high cyclomatic complexity and it serves as a motivation paper for further work in the same direction [1]. This work was extended to a journal paper where the same idea was examined in more real systems trying to draw a general picture [2]. As high cyclomatic functions are also very long we suggested a visualization tool that helps among others in regularity identification [3].

The next step was to investigate the effect of regularity on comprehension by experimentally comparing regular and non-regular implementations of the same real problem. The results show that regular code is easier to comprehend despite being longer and more complex (according to its cyclomatic complexity) [4].

Another significant aspect we have examined is the way programmers read regular code. This yielded a model that reflects the decreasing invested effort in regular code [5]. Moreover, in its extended journal version we argue that code reading is largely different from natural language reading [6].

Finally, we suggested a way to measure regularity by means of compression. It is a report that was not published yet [7].

Letter of Contribution

Five years of intensive work yielded 3 research papers (two were extended to journal version), 1 unpublished paper, and 1 short paper. In addition, during the course of this study two more publications were produced but they are not part of the final thesis as they are not related directly to the research field of it.

The first paper of this work is “High-MCC Functions in the Linux kernel”. The very initial ideas in its conference version were contributed by Adam Matan under the supervision of Prof. Dror Feitelson. These ideas were primarily based on manual investigations and were focused on one version of the Linux kernel.

This is the point where I started, automated the whole process (this enabled the study of more than 1000 versions of the Linux kernel), confirmed the initial ideas, extended them, and continued the whole work under the supervision of Prof. Dror Feitelson.

Contents

Acknowledgments	v
Abstract	vii
Letter of Contribution	xi
1 Introduction	1
2 High-MCC Functions in the Linux Kernel	17
3 JCSD: Visual Support for Understanding Code Control Structure	57
4 Quantification of Code Regularity Using Preprocessing and Compression	63
5 On the Effect of Code Regularity on Comprehension	75
6 How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking	89
7 Conclusions	123
Abstract in Hebrew	V

Chapter 1

Introduction

1 Introduction

Program comprehension is the process of building a mental model of a given source code. It lies at the basis of any software maintenance activity such as fixing bugs and adding new features.

Maintenance is important because it consumes a large part, up to 70% as reported by [1], of the resources allocated along a software lifecycle. Therefore, program comprehension is also important not only for the fact that it is a preliminary vital step in maintenance, but also for its criticality for the success of this activity. In particular, the better the code is understood the more likely the maintenance will be successful.

However, comprehension is closely related to complexity and directly affected by it [2]; the less complex is the code the easier a programmer can understand it. For example, nesting and non-linear flow are factors that affect complexity and probably make code harder to comprehend. Therefore, we wish to enhance comprehension by making programs less complex. This improvement would be feasible if we could measure the complexity of programs. As summarized by H. James Harrington, “measurement is the first step that leads to control and eventually to improvement. If you cannot measure something, you cannot understand it. If you cannot understand it, you cannot control it. If you cannot control it, you cannot improve it.”

Over the years very many complexity metrics have been suggested. These metrics capture varied aspects in the code. The lines of code (LOC) is the simplest and it reflects size. The McCabe’s cyclomatic complexity (MCC) counts the number of decision points in the code and by this it is a control flow metric [3]. Similarly *Npath* counts the number of acyclic execution paths in the code [4]. The software science metrics of Halstead measure programming effort by a formula based on counting *operators* and *operands* [5]. Data-flow metrics include *Dep-Degree* [6] and *Lifespan* [7] where both are based on program element counting. There have been attempts to define cognitive metrics, such as *CFS* [8], that went beyond simple counting and

provided different weights to different constructs. However, these weights were simplistic and did not reflect real complexity as perceived by programmers.

The large number of complexity metrics is probably an indication of a real difficulty in defining an ideal metric that is capable of reflecting complexity by providing one simple number. This primarily stems from the inherent problem of defining the term complexity and quantifying it [9, 10]. This leads to a lack of consensus in the community on a precise definition of the term comprehension [11].

The key problem of many existing metrics is that they estimate complexity by analyzing code syntax. For example, the McCabe’s cyclomatic complexity (MCC) is based on the number of independent execution paths in the code, which is equivalent to the number of conditions plus one. This metric is the most widely used since its introduction in 1976. Despite its popularity, it has been criticized over the years especially for the fact that it discards data flow and focuses on just counting program elements without paying attention to their context in the code [12, 13].

The dissatisfaction with the state of the practice leads to the research question of why MCC and other metrics are not good enough to reflect effective complexity? what do these metrics miss and what is needed to define better metrics?

In this thesis, we introduce regularity as an additional factor that affects complexity [14]. Specifically, regular code has many repetitions of a pattern, and successive instances become easier to comprehend given the experience acquired with previous ones. Listing 1 is an example of a C language function which is highly regular. In this function regularity occurs in lines 5–8, 9–12, 13–16, 17–20, 21–24, 25–28, 29–32, and 33–36. Similar and even more regular functions exist in real systems such as Linux but we do not provide an example here due to space limitations as such functions are generally very long [15].

The innovation of regularity is introducing context awareness and sensitivity: a piece of code can have different complexity depending on neighboring code. Interchangeably, code complexity is no longer absolute but depends on the context. In particular, with regularity, the initial instances of some patterns have higher complexity than those that appear later in the code as the effective complexity of the repeated instances is reduced due to leveraging the understanding gained in the initial instances to ease understanding in the later successive ones [14, 16].

Conversely, the current metrics, including MCC, unconsciously neglect code context. Specifically, they simply count code elements. For example, the lines of code (LOC) metric count lines and MCC just counts conditions.

This simple counting approach has led to complexity thresholds that do not reflect real complexity and as a result they are not good discriminators of complex and non-complex programs. Evidently, real software systems have many functions that are classified as very complex (according to these traditional metrics because their complexities are much higher than the suggested thresholds) while in practice they are not [15].

It is important however to adopt an empirical approach to explore what effects exist in practice. In particular, we conducted a family of diverse experiments starting with a very basic subjective ranking up to very sophisticated eyetracking based experiments. The results show that subjects estimated some functions with very high cyclomatic complexity as not complex. Moreover, they performed comprehension tasks and achieved better results in regular code (despite having higher cyclomatic complexity) when compared with its non-regular counterpart.

Beyond the immediate effects we already presented, our empirical investigation revealed more insights about reading code in general and reading regular code in particular. Specifically, the results show that code reading is very non-linear as opposed to reading natural language text. As for regular code, it seems that it is not read completely where parts of it are only

```

1 void func6(int **mat, int rs, int cs) {
2   int i, j, ii, jj, x, msk[9];
3   for (i = 0; i < rs; i++) {
4     for (j = 0; j < cs; j++) {
5       if (i >= 1 && j >= 1)
6         msk[0] = mat[i - 1][j - 1];
7       else
8         msk[0] = 0;
9       if (i >= 1)
10        msk[1] = mat[i - 1][j];
11      else
12        msk[1] = 0;
13      if (i >= 1 && j < cs - 1)
14        msk[2] = mat[i - 1][j + 1];
15      else
16        msk[2] = 0;
17      if (j >= 1)
18        msk[3] = mat[i][j - 1];
19      else
20        msk[3] = 0;
21      if (j < cs - 1)
22        msk[4] = mat[i][j + 1];
23      else
24        msk[4] = 0;
25      if (i < rs - 1 && j >= 1)
26        msk[5] = mat[i + 1][j - 1];
27      else
28        msk[5] = 0;
29      if (i < rs - 1)
30        msk[6] = mat[i + 1][j];
31      else
32        msk[6] = 0;
33      if (i < rs - 1 && j < cs - 1)
34        msk[7] = mat[i + 1][j + 1];
35      else
36        msk[7] = 0;
38      msk[8] = mat[i][j];
40      for (ii = 0; ii < 5; ii++)
41        for (jj = 0; jj < 9 - ii - 1; jj++)
42          if (msk[jj] > msk[jj + 1]) {
43            x = msk[jj];
44            msk[jj] = msk[jj + 1];
45            msk[jj + 1] = x;
46          }
47      printf("%d", msk[4]);
48    }
49    printf("\n");
50  }
51 }

```

Listing 1: An example of a highly regular function in C language. Taken from [14].

scanned [16].

Having considered the way programmers read regular code, this led to an event-based framework for analyzing code reading. In particular, we extended the set of events (e.g, reading, scanning) that have been suggested in previous studies and suggested a way of coding these events for further analysis [16].

It is true that in this work we introduce a new property that affects complexity and challenges existing metrics, but regularity is just an example of such a factor and does not solve everything. It introduces new and wider considerations that can serve as good guidelines for investigating new factors.

This thesis is structured as a collection of the papers that introduced and analyzed the above ideas. The papers are described in the following sections.

1.1 High-MCC functions in the Linux kernel

1.1.1 Authors

Ahmad Jbara - Adam Matan - Dror G. Feitelson

1.1.2 Status

- Conference - published at ICPC 2012.
- Journal - extended invited version was published in Empirical Software Engineering.

1.1.3 Full Citation

- Ahmad Jbara, Adam Matan, and Dror G. Feitelson. High-MCC functions in the Linux kernel. In *Proceedings of the 20th IEEE International Conference on Program Comprehension, ICPC 2012.*, Jun 2012

- Ahmad Jbara, Adam Matan, and Dror G. Feitelson. High-MCC functions in the Linux kernel. *Empirical Software Engineering*, 19(5):1261–1298, 2014

1.1.4 Summary

In spite of the large number of metrics that have, over the years, been proposed to measure program complexity, there is still no metric, including the most popular one of McCabe, that effectively measures complexity. Moreover, even a combination of these metrics is not sufficient. This motivates us to examine the state of the practice of real functions complexity and compare it to the state of the art of complexity metrics.

In particular, we applied the most widely used complexity metric of McCabe on functions of the Linux kernel. The results show that the practice as reflected in the very long and complex functions diverges from the common wisdom as reflected by the thresholds suggested and used for measuring their cyclomatic complexity. Especially, we found functions with very high MCC (up to 620 which is many times the highest threshold ever suggested). However, some of these functions seem to be well structured and are not as complex as their MCC value suggests. A close examination of their structure reveals quiet a flat and sometimes regular structure.

We focused on regularity as one possible structural property that enabled an extensive evolution of the supposed high complexity of these functions. In addition, a very initial subjective ranking experiment showed that subjects tend to rank such regular functions as not complex.

To draw a wider picture we investigated, in an extended journal paper, the same ideas in more software systems from varied domains. The results confirmed the insights and in some cases they were even stronger. For example, the highest MCC value we found in Linux was ten times the highest threshold ever defined, and in the new systems we found functions more than twenty times this value.

We conclude that high MCC does not necessarily mean high complexity. Specifically, simple syntactic metrics cannot capture all program aspects to reflect effective complexity. For example, regularity is one factor that needs a context-aware metric to be captured.

1.2 JCSD: Visual support for understanding code control structure

1.2.1 Authors

Ahmad Jbara - Dror G. Feitelson

1.2.2 Status

Conference - published at ICPC 2014.

1.2.3 Full Citation

Ahmad Jbara and Dror G. Feitelson. JCSD: Visual support for understanding code control structure. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, pages 300–303, New York, NY, USA, 2014. ACM

1.2.4 Summary

As the functions we investigate have high cyclomatic complexity values and many works have shown that this metric positively correlates with lines of code (LOC), it turns out that these functions are generally very long.

One way to see the whole structure of a given long function is to visualize it. In a previous work we proposed CSD (control structure diagram). In this diagram each construct type receives a different geometric shape. The size of the shape reflects the code block controlled by a specific construct. A level in the diagram represents nesting in the code.

This diagram is good, among other things, to clearly identify structural properties in the code such as regularity.

In this short paper we implemented CSD (control structure diagram) as a tool in a development environment to help developers capture the whole function's structure at a glance.

1.3 On the effect of code regularity on comprehension

1.3.1 Authors

Ahmad Jbara - Dror G. Feitelson

1.3.2 Status

Conference - published at ICPC 2014.

1.3.3 Full Citation

Ahmad Jbara and Dror G. Feitelson. On the effect of code regularity on comprehension. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, pages 189–200, New York, NY, USA, 2014. ACM

1.3.4 Summary

The real difficulty in defining the term complexity is a barrier that makes the measurement of code complexity problematic. However, due to its high importance very many attempts have proposed complexity metrics such as MCC and LOC.

As we have shown, these metrics fail in reflecting effective complexity primarily for their syntactic nature. In particular these metrics have provided false positive answers where functions should have been ranked as not complex.

We have suggested regularity as an additional factor that affects comprehension and even reduces effective complexity. This factor could explain the high values that are falsely given by MCC.

To show the effect of regularity on complexity we conducted controlled experiments where participants performed different comprehension tasks on two implementations of the same program. One version is regular and the other is not. As expected the regular versions are longer and ranked as more complex than their non-regular counterparts, according to the syntactic metrics.

The results show that subjects achieved better scores in the regular versions despite being longer and more complex. This shows that regularity reduces complexity as opposed to the syntactic metrics.

The explanation for this effect is the fact that in regular code the code segments repeat themselves and therefore understanding the initial segments helps in understanding the other ones. This explanation was subjectively verified by a post-experiment question.

A wider aspect that captures regularity but is ignored by syntactic metrics is context. In particular, conventional syntactic metrics treat all instances of the same construct or block as if they have the same complexity. However, as regularity suggests the complexity is reduced as we progress to new instances of a block we already read and understood.

1.4 How programmers read regular code: A controlled experiment using eye tracking

1.4.1 Authors

Ahmad Jbara - Dror G. Feitelson

1.4.2 Status

- Conference - published at ICPC 2015.

- Journal - extended invited version at Empirical Software Engineering. under review.

1.4.3 Full Citation

- Ahmad Jbara and Dror G. Feitelson. How programmers read regular code: A controlled experiment using eye tracking. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*, pages 244–254, May 2015
- Ahmad Jbara and Dror G. Feitelson. How programmers read regular code: A controlled experiment using eye tracking. *Empirical Software Engineering*, 2015. under review

1.4.4 Summary

We have shown that regular code has an effect on code comprehension. In particular, we showed that a regular implementation of the same problem is as easy to understand as its non-regular version, and in some cases even absolutely easier. This is achieved despite the fact that regular code is longer, and supposed to be more complex than its non-regular counterpart.

Our speculation was that subjects leverage their understanding of the initial instances of the the repeated pattern to make it easier to comprehend the other later instances.

To verify this conjecture we conducted an eyetracking based experiment to learn about the way subjects read and comprehend regular code. The results show that subjects indeed invest more time and effort in the initial instances as opposed to the later ones. In particular their efforts decay as they progress towards later segments and this behavior is governed by an exponential/cubic model.

This result again emphasizes the importance of context in complexity metrics. Specifically, two similar segments are treated as having the same

complexity by the syntactic metrics, where regularity reveals that complexity in such cases is not additive.

The study of the way programmers read regular code led to the investigation of their *scanpaths*. In particular we developed an event-based framework for analyzing code reading. In this framework we extended the set of events (e.g, reading, scanning) that have been suggested in previous studies and proposed a way for coding these events for further analysis [16].

In addition we concluded that code and in particular regular code are far from being linearly read as in natural language text.

1.5 Quantification of code regularity using preprocessing and compression

1.5.1 Authors

Ahmad Jbara - Dror G. Feitelson

1.5.2 Status

Manuscript, 2014.

1.5.3 Full Citation

Ahmad Jbara and Dror G. Feitelson. Quantification of code regularity using preprocessing and compression. Manuscript, Jan 2014

1.5.4 Summary

We have suggested regularity as an additional factor that affects comprehension by reducing complexity of functions that otherwise would be ranked as very complex.

As regularity is based on repetitions it is reasonable to think that the more repetitions a function has the more regular it is. To measure the extent

of repetition (regularity) we use compression. The higher the compression ratio the more regular is the code.

In this context we had to cope with two questions. Which compression scheme should be used? and what parts of the code should be compressed? To answer these questions we examined different well known compression schemes and applied them on different preprocessing levels of the code. This yielded many combinations of compression scheme and preprocessing level.

To select the best combination out of the 20 we had we required that a good combination should meet three design criteria: good discrimination, successfully handling small functions, and correlation with complexity as perceived by humans.

The results showed that the recommended combinations are *gzip* or *bicom* schemes applied on a code skeleton that contains keywords and formatting.

References

- [1] B. P. Leintz and E. F. Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.
- [2] H. Zuse. Criteria for program comprehension derived from software complexity metrics. In *IEEE Workshop Program Comprehension*, number 2, pages 8–16, 1993.
- [3] T.J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, Dec 1976.
- [4] Brian A. Nejmeh. Npath: a measure of execution path complexity and its applications. *Comm. ACM*, 31(2):188–200, Feb 1988.
- [5] M. Halstead. *Elements of Software Science*. Elsevier Science Inc., 1977.
- [6] D. Beyer and A. Fararoyy. A simple and effective measure for complex

- low-level dependencies. In *IEEE Intl. Conf. Program Comprehension*, number 18, pages 80–83, 2010.
- [7] J.L. Elshoff. An analysis of some commercial PL/I programs. *IEEE Trans. Softw. Eng.*, SE-2(2):113–120, 1976.
- [8] Jingqiu Shao and Yingxu Wang. A new measure of software complexity based on cognitive weights. *Canadian J. Electrical and Comput. Eng.*, 28(2):69–74, april 2003.
- [9] G. J Myers. *Software Reliability- principles and practices*. 1976.
- [10] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1979.
- [11] Jurgen J. Vinju and Michael W. Godfrey. What does control flow really look like? Eyeballing the cyclomatic complexity metric. In *Working Conf. Source Code Analysis and Manipulation*, number 12, Sep. 2012.
- [12] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering J.*, 3(2):30–36, Mar 1988.
- [13] M. Shepperd and D. C. Ince. A critique of three metrics. *J. Syst. & Softw.*, 26(3):197–210, Sep 1994.
- [14] Ahmad Jbara and Dror G. Feitelson. On the effect of code regularity on comprehension. In *Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014*, pages 189–200, New York, NY, USA, 2014. ACM.
- [15] Ahmad Jbara, Adam Matan, and Dror G. Feitelson. High-MCC functions in the Linux kernel. *Empirical Software Engineering*, 19(5):1261–1298, 2014.

- [16] Ahmad Jbara and Dror G. Feitelson. How programmers read regular code: A controlled experiment using eye tracking. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*, pages 244–254, May 2015.
- [17] Ahmad Jbara, Adam Matan, and Dror G. Feitelson. High-MCC functions in the Linux kernel. In *Proceedings of the 20th IEEE International Conference on Program Comprehension, ICPC 2012.*, Jun 2012.
- [18] Ahmad Jbara and Dror G. Feitelson. JCSD: Visual support for understanding code control structure. In *Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014*, pages 300–303, New York, NY, USA, 2014. ACM.
- [19] Ahmad Jbara and Dror G. Feitelson. How programmers read regular code: A controlled experiment using eye tracking. *Empirical Software Engineering*, 2015. under review.
- [20] Ahmad Jbara and Dror G. Feitelson. Quantification of code regularity using preprocessing and compression. Manuscript, Jan 2014.

Chapter 2

High-MCC Functions in the Linux Kernel

Ahmad Jbara, Adam Matan, Dror G. Feitelson.

Status:

Published.

Full citation:

Ahmad Jbara, Adam Matan, and Dror G. Feitelson. High-MCC functions in the Linux kernel. *Empirical Software Engineering*, 19(5):1261–1298, 2014

Note:

Invited extended journal version of “Ahmad Jbara, Adam Matan, and Dror G. Feitelson. High-MCC functions in the Linux kernel. In *Proceedings of the 20th IEEE International Conference on Program Comprehension, ICPC 2012.*, Jun 2012”

High-MCC Functions in the Linux Kernel

Ahmad Jbara · Adam Matan · Dror G. Feitelson

Published online: 12 September 2013
© Springer Science+Business Media New York 2013

Abstract McCabe’s Cyclomatic Complexity (MCC) is a widely used metric for the complexity of control flow. Common usage decrees that functions should not have an MCC above 50, and preferably much less. However, the Linux kernel includes more than 800 functions with MCC values above 50, and over the years 369 functions have had an MCC of 100 or more. Moreover, some of these functions undergo extensive evolution, indicating that developers are successful in coping with the supposed high complexity. Functions with similarly high MCC values also occur in other operating systems and domains, including Windows. For example, the highest MCC value in FreeBSD is 1316, double the highest MCC in Linux. We attempt to explain all this by analyzing the structure of high-MCC functions in Linux and showing that in many cases they are in fact well-structured (albeit we observe some cases where developers indeed refactor the code in order to reduce complexity). Moreover, human opinions do not correlate with the MCC values of these functions. A survey of perceived complexity shows that there are cases where high MCC functions were ranked as having a low complexity. We characterize these cases and identify specific code attributes such as the diversity of constructs (not only a `switch` but also `ifs`) and nesting that correlate with discrete increases in perceived complexity. These observations indicate that a high MCC is not necessarily an impediment to code comprehension, and support the notion that complexity cannot be fully captured

Communicated by: Michael Godfrey and Arie van Deursen

A. Jbara (✉) · A. Matan · D. G. Feitelson
School of Computer Science and Engineering, Hebrew University, 91904 Jerusalem, Israel
e-mail: ahmadjbara@cs.huji.ac.il

D. G. Feitelson
e-mail: feit@cs.huji.ac.il

A. Jbara
School of Mathematics and Computer Science, Netanya Academic College,
42100 Netanya, Israel

using simple syntactic code metrics. In particular, we show that regularity in the code (meaning repetitions of the same pattern of control structures) correlates with low perceived complexity.

Keywords Software complexity · McCabe cyclomatic complexity · Linux kernel · Perceived complexity · Code regularity

1 Introduction

Mitigating complexity is of pivotal importance in writing computer programs. Complex code is hard to write correctly and hard to maintain, leading to more faults (Lanning and Khoshgoftaar 1994; Binkley and Schach 1998). As a result, significant research effort has been expended on defining code complexity metrics and on methods to combine them into effective predictors of code quality (Ohlsson and Alberg 1996; Denaro and Pezzè 2002; Olague et al. 2007). Industrial testimony indicates that using complexity metrics provides real benefits over simple practices such as just counting lines of code (e.g. Jones 1994; Curtis et al. 1979; Koziolok et al. 2010; Schneidewind and Hinchey 2009).

One early metric that has been used in many studies is McCabe’s Cyclomatic Complexity (MCC; McCabe 1976). This metric essentially counts the number of linear paths through the code (the precise definition is given below in Section 2). In the original paper, McCabe suggests that procedures with an MCC value higher than 10 should be rewritten or split in order to reduce their complexity, and other somewhat higher thresholds have been suggested by others (e.g. MSDN 2008; Foreman et al. 1997; Stamelos et al. 2002; VerifySoft Technology 2005; Curtis et al. 2011). In general, proposed thresholds are typically well below 50, and there appears to be some agreement that procedures with much higher values are extremely undesirable.

Nevertheless, in the context of a study of Linux evolution, we have found functions with MCC values in the hundreds (Israeli and Feitelson 2010). This chance discovery led to a set of research questions:

1. What are the basic characteristics of high-MCC functions? Specifically,
 - 1.1 How common are such high-MCC functions? In other words, are they just a fluke or a real phenomenon reflecting the work practices of many developers?
 - 1.2 What causes the high MCC counts? One may speculate that they are the result of large flat switch statements, that do not reflect real complexity. But if other more complex and less regular constructs are found this raises the question of how developers cope with them.
 - 1.3 Does MCC correlate with other metrics, as has been shown in the past? Or does it provide independent complexity information?
2. Do high-MCC functions evolve with time? If these functions are “write once” functions that serve some fixed need and are never changed, then nobody except the original author really needs to understand them. But if they are modified many times as Linux continues to evolve, it intensifies the question of how do the maintainers cope with the supposedly high complexity.

3. What influences the perception of complexity? Specifically,
 - 3.1 Does a high MCC correlate with perceived complexity? In other words, does MCC indeed capture the essence of complexity?
 - 3.2 Can we find discrete elements of complexity? In other words, can we point out specific code attributes that, if present, make a function appear more complex? This is an extremely important question with respect to complexity metrics, as an affirmative answer may indicate that complexity is an additive property of code attributes.
 - 3.3 Is a visual representation of high-MCC functions better than code listings?
4. What other ingredients of complexity may be missing from MCC? In particular, in our work we found that some high-MCC functions have a very regular structure. This raised the question whether regularity may counteract the supposed complexity reflected by the high MCC.
5. Are all the high-MCC functions we found really required, or can some of this code be replaced or refactored? This issue reflects the tradeoff done by developers, where sometimes allowing additional code with high complexity metrics is nevertheless considered better than trying to minimize it.
6. Are the high-MCC functions unique to Linux, or do they also appear in other operating systems and domains?
7. Altogether, do the high-MCC functions indicate code quality problems with the Linux kernel?

To gain insight into these issues we analyzed the functions in Linux kernel version 2.6.37.5 that have $MCC \geq 100$, which turn out to have MCC values ranging up to 587—way above the scale that is considered reasonable. We also analyzed the evolution of all 369 functions that had $MCC \geq 100$ in any of the Linux kernel versions released since the initial release of version 1.0 in 1994 (more than a thousand versions). In addition we examined three other operating systems and three systems from different domains—Windows Research Kernel, OpenSolaris, FreeBSD, GCC, Firefox, and OpenSSL—and found that they also contain similar high-MCC functions. The highest MCC values were 246, 506, 1316, 1301, 699, and 371 respectively.

In a nutshell, we found that (in Linux) the most common source of high MCC counts is large trees of if statements, although several cases are indeed attributed to large switches. 33 % of the functions do not change, but the others may change considerably. About 5 % of the functions exhibit extreme changes in MCC values that reflect explicit modifications to their design, indicating active work to reduce complexity. We speculate that the ability to work with these functions stems from the fact that switches and large trees of ifs embody a separation of concerns, where each call to the function only selects a small part of the code for execution. This is especially true if they are nested in each other, rather than coming one after the other, so this explanation is especially relevant for the deeply-nested functions. On the other hand we also observed some cases of spaghetti-style *gotos*, which are not directly measured by MCC. Such observations motivate studying alternative ways in which code structure may be analyzed when assessing the resulting complexity. In particular, we suggest code regularity as an important attribute that may compensate for complexity.

The remainder of the paper is structured as follows. In the next section we define MCC and review its use. We characterize high-MCC functions in the Linux kernel in Section 3, and their evolution in Section 4. Results of the survey of perceived complexity are presented in Section 5, and the relationship with regularity in Section 6. Section 7 discusses the possibility of reducing high-MCC code. High-MCC functions in other operating systems and domains are examined in Section 8. Discussion, significance of our findings, and further research directions are presented in Section 9. This paper is an extended version of a previous conference paper (Jbara et al. 2012). The main additions are added experimentation (more subjects and additional experiments), the definition of a metric for code regularity and its effect, an examination of evidence for cloning, and showing that high-MCC functions exist also in other operating systems and domains.

2 McCabe's Cyclomatic Complexity

McCabe's cyclomatic complexity (MCC) is based on the graph theoretic concept of cyclomatic number, applied to a program's control-flow graph. The nodes of such a graph are basic blocks of code, and the edges denote possible control flow. For example, a block with an if statement will have two successors, representing the "then" option and the "else" option. The cyclomatic number of a graph g 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. (In a computer program, each procedure would be a separate connected component, and the end result is the same as adding the cyclomatic numbers of all of them.) McCabe suggested that the cyclomatic number of a control-flow graph represents the complexity of the code (McCabe 1976). He also showed that it corresponds to the number of linearly independent code paths, and can therefore be used to set the minimal number of tests that should be performed.

Another way to characterize the cyclomatic number of a graph is related to the notions of structured programming, where all constructs have single entry and exit points. The control-flow graph is then planar, and the cyclomatic number is equal to the number of faces of the graph, including the "outside" area. McCabe also demonstrated a straight-forward intuitive meaning of the metric: it is equal to the number of condition statements in the program plus 1 (if, while, etc.). If conditions are composed of multiple atomic predicates, we could also count them individually; this is sometimes called the "extended" MCC (Myers 1977). Note that MCC counts points of divergence, but not joins. It is thus insensitive to unconditional jumps such as those induced by `goto`, `break`, or `return`.

2.1 Thresholds on MCC

In principle MCC is unbounded, and intuition suggests that high values reflect potentially problematic code. It is therefore natural to try and define a threshold beyond which code should be checked and maybe modified. McCabe himself, in the original paper which introduced MCC, suggests a threshold of 10 (McCabe 1976), and this is also the value used by the code analysis tool sold by his company today (McCabe Software 2009). The Eclipse Metrics plugin also uses a threshold

of 10 by default, and suggests that the method be split if it is exceeded (Sauer 2005). VerifySoft Technology suggest a threshold of 15 per function, and 100 per file (VerifySoft Technology 2005). Logiscope also uses a threshold of 15 (Stamelos et al. 2002). The STAN static analysis tool gives a warning at 15, and considers values above 20 an error (Mens et al. 2008). The complexity metrics module of Microsoft Visual Studio 2008 reports a violation of the cyclomatic complexity metric for values of more than 25 (MSDN 2008). The Carnegie Mellon Software Engineering Institute defined a four-level scale as part of their (now legacy) Software Technology Roadmap (Foreman et al. 1997). High risk was associated with values of MCC above 20, and very high risk with values larger than 50. Heitlager et al. used these risk levels and suggested a complexity rating scheme based on the percentage of LOC falling within each risk level (Heitlager et al. 2007).

All the above thresholds consider functions in isolation. VerifySoft also suggests a threshold on the sum of all functions in the same file. An alternative approach is to consider the distribution of MCC values. The Gini coefficient, used to measure inequality in economics, was used by Vasa et al. to characterize the distribution of different metrics including MCC (Vasa et al. 2009); he found that the distribution was highly skewed, as we do too. Stark et al. propose a decision chart that plots the cumulative distribution function (CDF) of MCC values on a logarithmic scale, and if the CDF falls below a certain diagonal line then the project as a whole should be reviewed (Stark et al. 1994); in brief, this line requires 20 % of the functions to have an MCC of 1, allows about 60 % to be above 10, and dictates an upper bound of 90. However, it seems that this was not picked up by others, and using simple thresholds remains the prevailing approach.

2.2 Critique of MCC and Correlation with LOC

It should be noted that MCC is not universally accepted as a good complexity metric, and it has been challenged on both theoretical and experimental grounds.

Perhaps the most common objection to using MCC as a complexity metric is its strong correlation with lines of code (LOC) (Shepperd 1988; Shepperd and Ince 1994; Herraiz and Hassan 2011). This correlation has been demonstrated many times, and indeed, we find that also in the Linux kernel the correlation coefficient of MCC and LOC is a relatively high 0.88. But if we focus on only the high-MCC functions, the correlation is much lower. We revisit this issue in Section 3.4.

Ball and Larus note that with n predicates there can be between $n + 1$ and 2^n paths in the code, so the number of paths is a better measure of complexity than the number of predicates (Ball and Larus 2000). Others show that MCC only measures control flow complexity but not data flow complexity and has additional deficiencies (Shepperd 1988; Shepperd and Ince 1994). In particular, MCC is intrinsic to code, so it does not admit the possibility that code fragments interact with each other to either increase or decrease the overall complexity (Weyuker 1988). Finally, Nagappan et al. have shown that while MCC is a good defect predictor for some projects, there is no single metric (including MCC) that is good for all projects (Nagappan et al. 2006).

There is, however, no other complexity metric that enjoys wider acceptance and is free of such criticisms, so MCC remains widely used to this day. Oman's 'maintainability index' includes MCC as one of its components (Oman and Hagemeister 1994), and Baggen et al. recently used thresholds on MCC in the context of creating

a certification mechanism for maintainability (Baggen et al. 2012). Curtis et al. use a criterion of MCC above 30 to identify ‘highly complex components’, and find that MCC is one of the four most frequent violations of good architectural or coding practice over different languages (Curtis et al. 2011). The ‘weighted method count’ metric for object-oriented software is usually interpreted as the sum of the MCC over all methods in a class. Recently, Capiluppi et al. used MCC to evaluate the change in complexity of successive revisions of the same file in the Linux kernel (Capiluppi and Izquierdo-Cortázar 2013), and Soetens et al. used it to check the assumption that refactoring reduces complexity (as it turns out, most refactoring does not affect MCC) (Soetens and Demeyer 2010). Thus, given its wide use and availability in software development and testing environments, MCC merits an effort to understand it better.

2.3 Distribution of MCC in Linux

Our research question 1.1 concerned the prevalence of high-MCC functions. In a previous study of the Linux kernel we found that the distribution of MCC is very skewed, with many thousands of functions with extremely low MCC and few functions with extremely high MCC (the highest value observed was 620) (Israeli and Feitelson 2010). In addition, we found that the distribution has a heavy tail, namely one that decays according to a power law.

It is especially interesting to observe how this distribution has changed with time. Such a study reveals two seemingly contradictory findings (Israeli and Feitelson 2010). First, it was found that the absolute number of high-MCC functions is growing with time: in version 1.0 in 1994 there were only 15 functions with MCC of 50 or more, and in 2008 there were more than 400 such functions. At the same time it was also found that the distribution as a whole is shifting towards lower MCC values: In 1994 the median MCC was 4 and the 95th percentile was 20, but by 2008 the median was 2 and the 95th percentile was down to 13. This means that the number of low MCC functions is growing at a higher pace than the number of high-MCC functions.

In this paper we focus on the tail of the distribution, namely the functions with the highest MCC values. This is the interesting part of the distribution, because functions with such high MCC values are thought to be too complex and should not exist.

3 Analysis of High-MCC Functions in Linux

When studying the evolution of the Linux kernel, and in particular how various code metrics change with time, we found that some Linux kernel functions have MCC values in the hundreds (Israeli and Feitelson 2010). Here we focus on high-MCC functions in version 2.6.37.5, released on 23 March 2011, as well as on the evolution of high-MCC functions across more than a thousand versions released from 1994 to 2011.

3.1 Data Collection

To calculate the MCC we use the `pmccabe` tool (Bame 2011). This tool also calculates the extended MCC, i.e. it also counts instances of logical operators in predicates

(&& and ||). We use the extended version, in order to avoid the confounding effect of coding style (where a programmer uses either nested conditionals or a logical operator to achieve the same effect).

Our scripts parse all the implementation files of each Linux kernel, and collect various code metrics for functions with MCC above 100. However, in some cases the parsing is problematic. In particular, the Linux kernel is littered with `#ifdef` preprocessor directives, that allow for alternative compilations based on various configuration options (Liebig et al. 2010). As we want to analyze the full code base and not just a specific configuration, we ignore such directives and attempt to analyze all the code. As the resulting code may not be syntactically valid, the `pmccabe` tool may not always handle such cases correctly. Consequently a small part (around 1 %) of the source code is not included in the analysis. (Jbara and Feitelson (2013), As a side note, the conditional compilation itself may also add to the complexity of the code, but we discuss this issue in another paper).

3.2 Description of High-MCC Functions

The functions with MCC values of 100 or more in Linux kernel 2.6.37.5 have values ranging up to 587. 104 of these functions come from the `drivers` subdirectory, with others coming from `arch` (12 functions), `fs` (12 functions), `sound` (5 functions), `net` (3 functions), `lib` (1 function) and `crypto` (1 function). The sources of all 369 functions with $MCC \geq 100$ that ever appeared in Linux are tabulated in Table 1. We manually examined a few of the top functions in the `drivers` subdirectory and found them dominated by `switch` statements of symbolic constants. These constants essentially represent `ioctl` codes for devices, different modes for emulations, and usage tables of different human interface devices.

Our research question 1.2 concerns the origin of high MCC counts. A high MCC can be the result of any type of branching statements: cases in a `switch`, `if` statements, or the loop constructs `while`, `for`, and `do`. But in the high-MCC functions of Linux the origin is usually multiple `if` statements or cases in a `switch` statement, as shown in Fig. 1. These can be nested in various ways. Somewhat common structures are a large `switch` with small trees of `ifs` in many of its cases, or large trees of `ifs` and `elses`. Logical operators, which can also be considered as branch points due to short-circuit evaluation, also make some contribution. Loops are quite rare.

Apart from the highest-MCC function, which is an obvious outlier, the rest of the distribution shown in Fig. 1 is seen to decline rather slowly. Indeed, in this version of Linux there were 138 functions with $MCC \geq 100$, and 802 with $MCC \geq 50$. Thus high-MCC functions are not uncommon (albeit they are a very small fraction of the total functions in Linux—those with MCC of 50 or more constitute just 0.3 %).

3.3 Visualization of Constructs and Nesting Structure

High-MCC functions are naturally quite long, and include very many programming constructs. As a result, it is hard to grasp their structural properties. To overcome this problem and provide better insights into research question 1.2, we introduce control structure diagrams (CSD) to visualize the control structure and nesting. These are

Table 1 Classification of the 369 high-MCC functions according to the directories that contain them

Directory	Subdirectory	# high-MCC functions	Comments	
Drivers	Staging	65	New drivers being staged into the system	
	Media	35		
	Video	25		
	Sound	25		
	scsi	24		
	isdn	21		
	net	15		
	usb	14		
	char	14		Character device drivers e.g. ttys and mice
	gpu	11		
	Block	9		Block device drivers like IDE disks
	Others	27		
	Total	285		
Arch	m68k	6		
	sparc64	5		
	sparc	4		
	powerpc	4		
	parisc	4		
	x86	3		
	ia64	2		
	cris	1		
	mn10300	1		
	Total	30		
Sound	oss	18	Cross platform Open Sound System	
	pci	2		
	isa	1		
	Total	21		
fs	xf	4		
	ext4	2		
	ncpfs	2		
	Others	9		
	Total	17		
Net	ipv6	2		
	ipv4	2		
	core	2		
	802	1		
	atm	1		
	ieee80211	1		
	inet	1		
	Total	10		
Others	–	6		

somewhat similar to the diagrams used by Adams et al. (2009) to visualize patterns of using the C preprocessor.

In these diagrams (for example Fig. 2) the bar across the top represents the length of the function, which starts at the left and ends at the right. Below this the nesting of different constructs is shown, with deeper nesting indicated by a lower level. Each

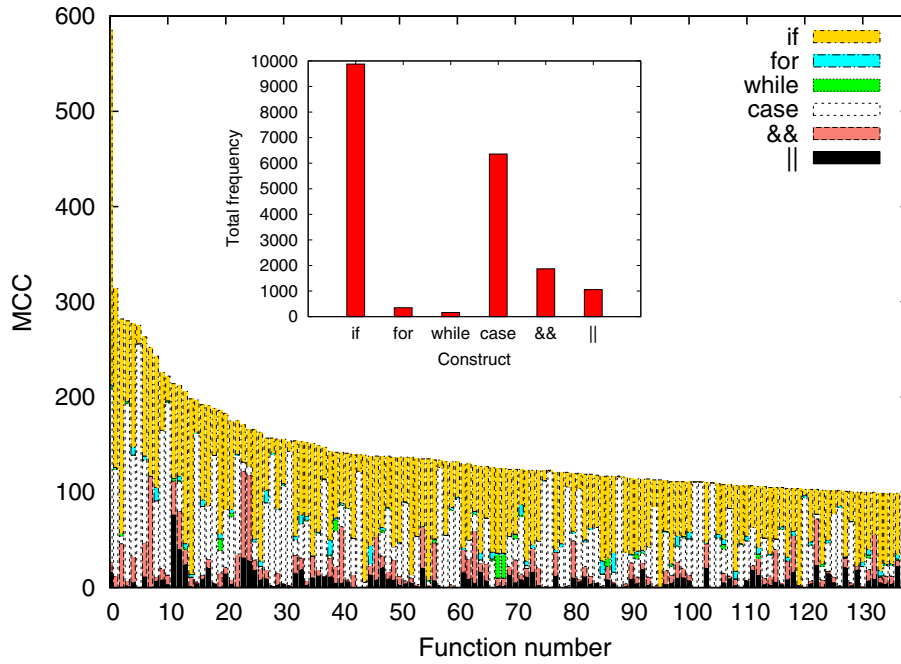


Fig. 1 Distribution of constructs in high-MCC functions

control type is represented by a different shape and color. Each construct (except large loops) is scaled so as to span the correct range of lines in the function. This helps to easily identify the dominant control structures, which are possible candidates for refactoring.

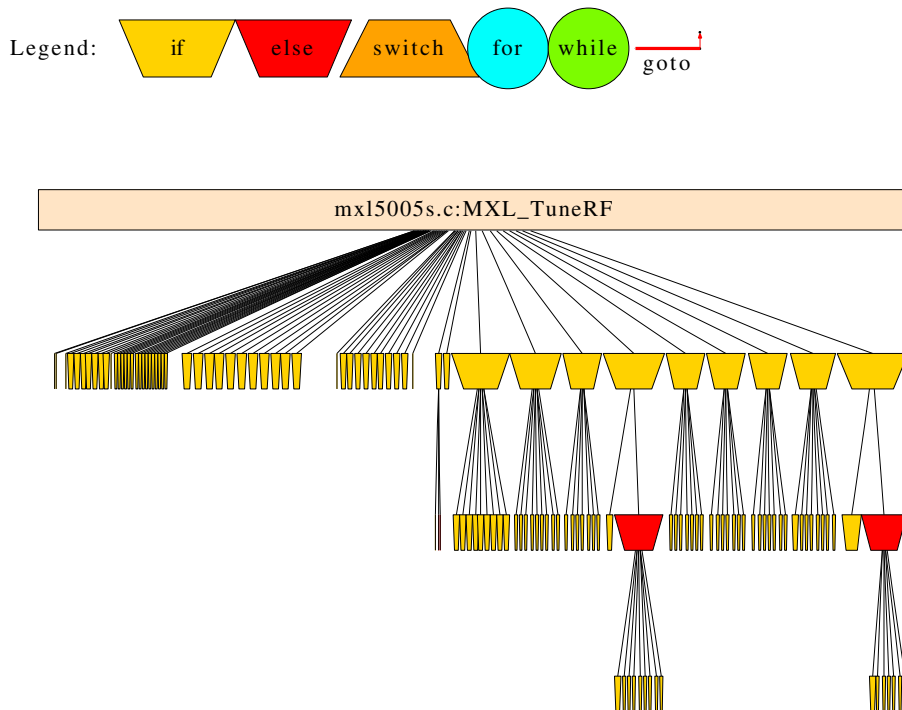
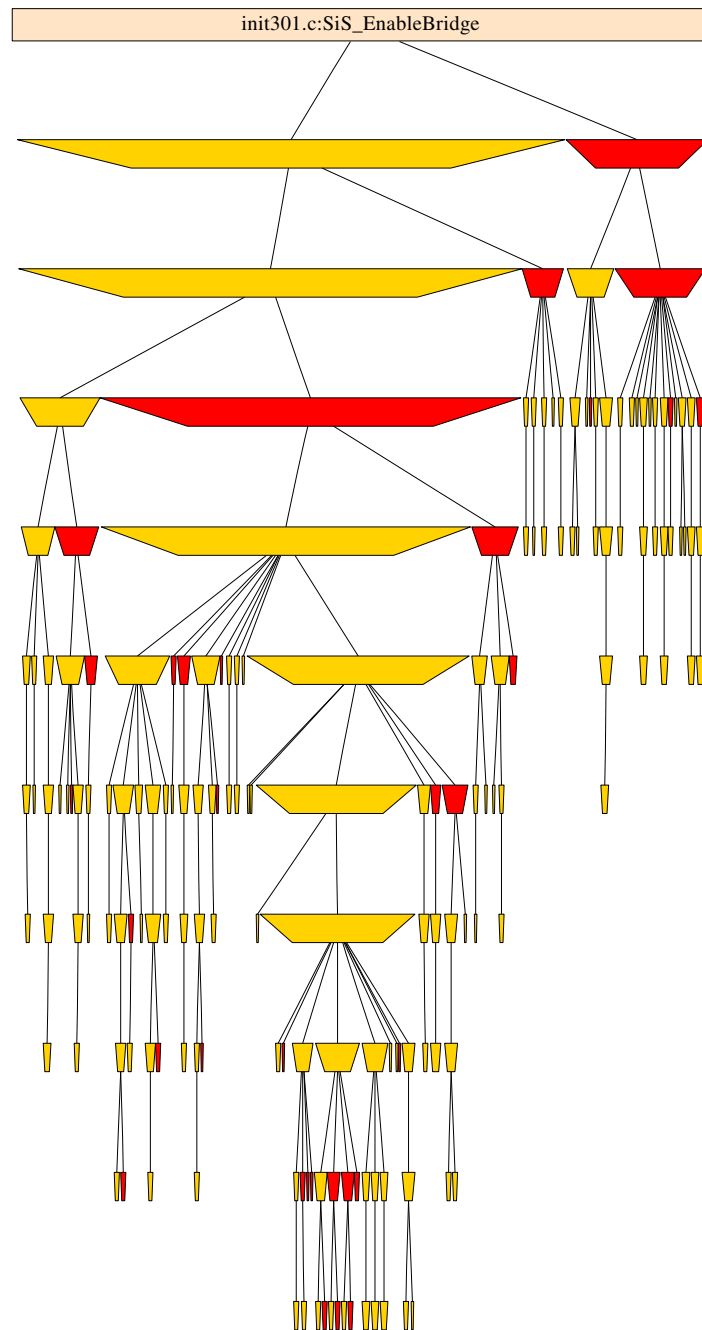


Fig. 2 A function that is a largely flat sequence of ifs

Fig. 3 A function with irregular ifs and relatively deep nesting



Using the CSDs we easily observe each function's nesting structure and regularity, which may affect the perceived complexity of the code.¹ Some of the high-MCC functions are relatively flat and regular. An example is shown in Fig. 2. This function starts with many small ifs in sequence, and then has 9 large ifs with nested small ifs, two of which have large else blocks with yet another level of nested small ifs. Despite the large number of ifs this function is shallow and regular and does not appear complicated. Other functions, like that shown in Fig. 3, include deep nesting and

¹Graphs for all functions analyzed are available at www.cs.huji.ac.il/~ahmadjbara/hiMCC.htm

appear to be more complicated. Regularity and its effect on perceived complexity are discussed in Section 6.

Recall that the high MCCs observed are predominantly due to if statements and cases in switch statements. This means that the flow is largely linear, with branching used to select the few pieces of code that should actually be executed in each invocation of the function. Only a relatively small fraction of the functions include loops, and in most cases these are small loops. Figure 4 shows an example of a function that had relatively many loops, and even in this case they can be seen to be greatly outnumbered by ifs and cases.

While most practitioners typically limit themselves to using nested structured programming constructs, some also use goto. The goto instruction is one that breaks the function's structure and decreases code readability, in particular when backwards

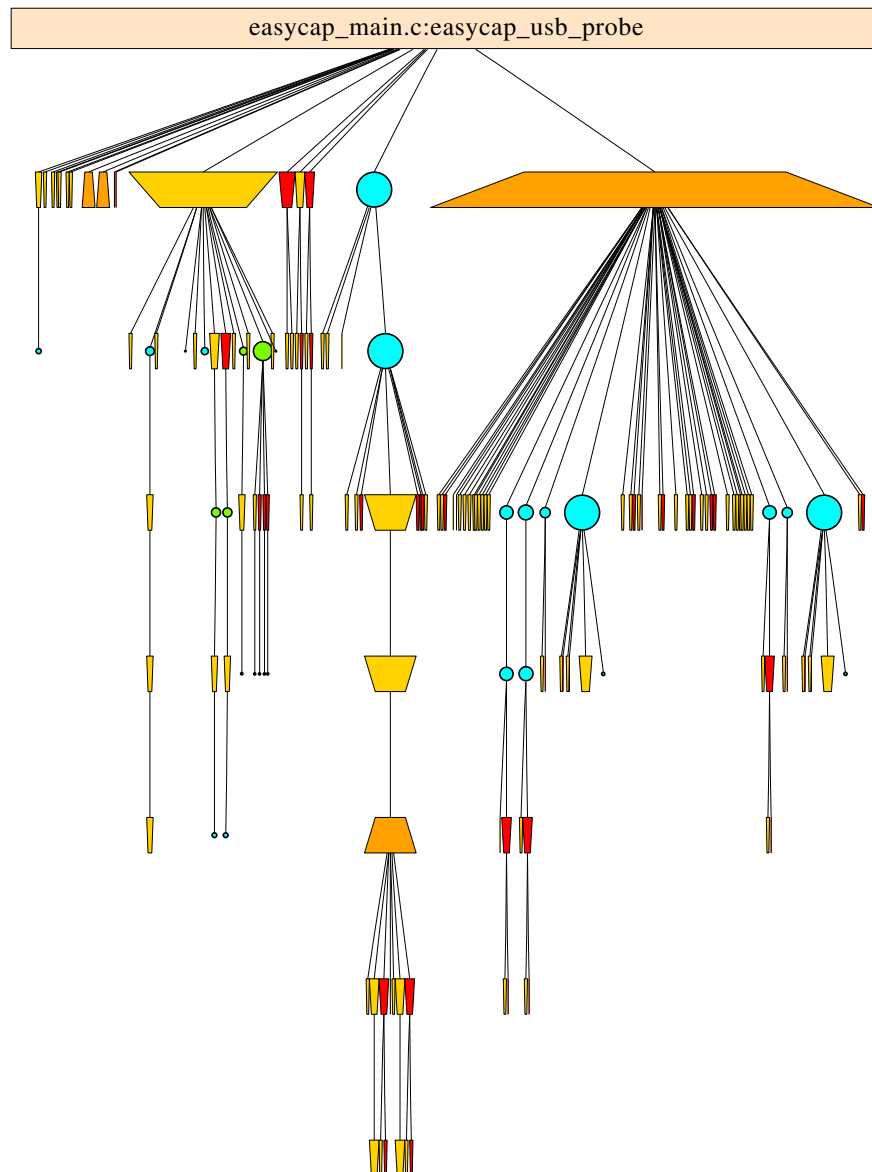


Fig. 4 A function with relatively many loops

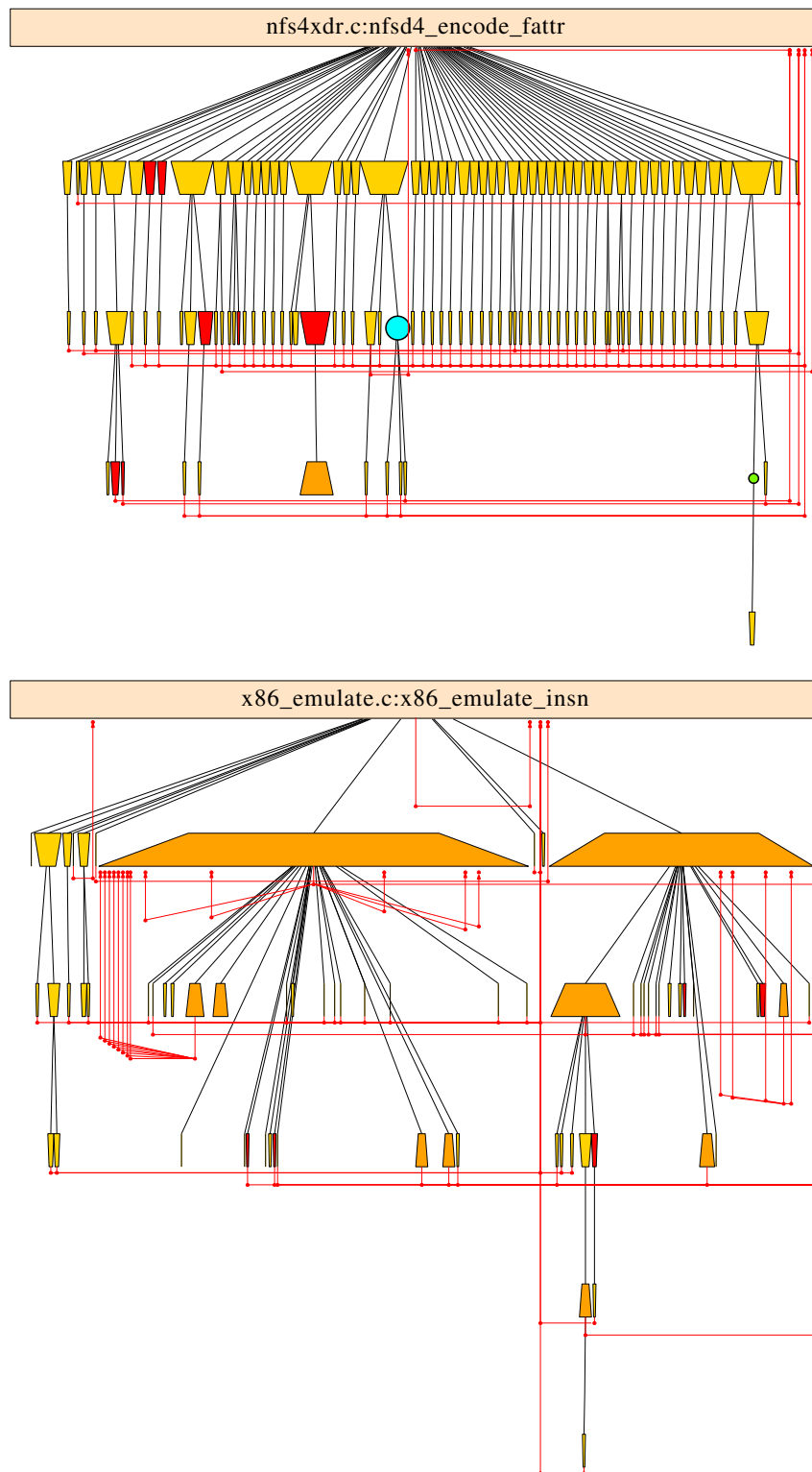
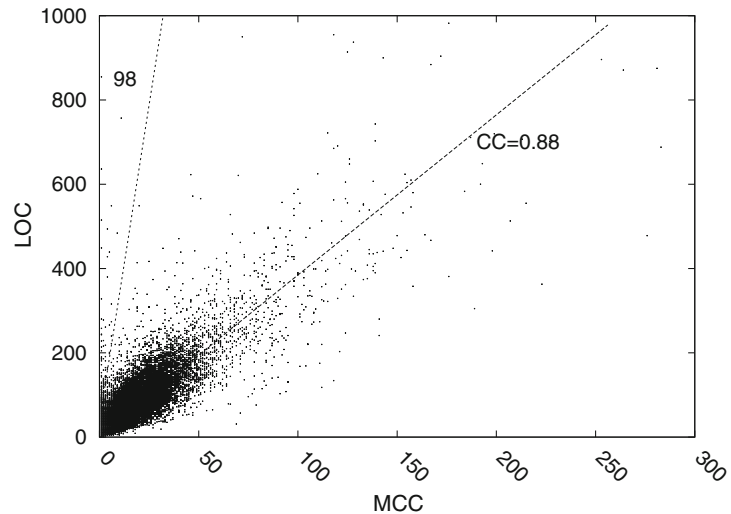


Fig. 5 Examples of functions using goto

jumps occur between successive constructs (Dijkstra 1968). The CSD visualizes the source and destination points of each goto and their relative locations within the code. Figure 5 shows examples of two functions that use goto. In the first gotos

Fig. 6 Correlation of MCC with LOC for all functions in Linux kernel 2.6.37.5



are used only to break out of nested constructs in case of error, and go directly to cleanup code at the end of the function. This is usually considered acceptable. But the second uses `gotos` to create a very complicated flow of control, which is much more problematic.

3.4 Correlation of MCC with Other Metrics

Research question 1.3 deals with the correlation of MCC with other metrics. Indeed, one of the criticisms of MCC is that it does not provide any significant information beyond that provided by other code metrics, notably LOC (lines of code). The claim is that longer code naturally has more branch points, and thus LOC and MCC are correlated. Indeed, when comparing the MCC and LOC of all the functions in the Linux kernel, a significant correlation is observed (Fig. 6). The correlation coefficient is 0.88, and the regression line indicates that on average there are 3.8 lines of code for every branch (unit of MCC). However, there is some variability, with a few functions

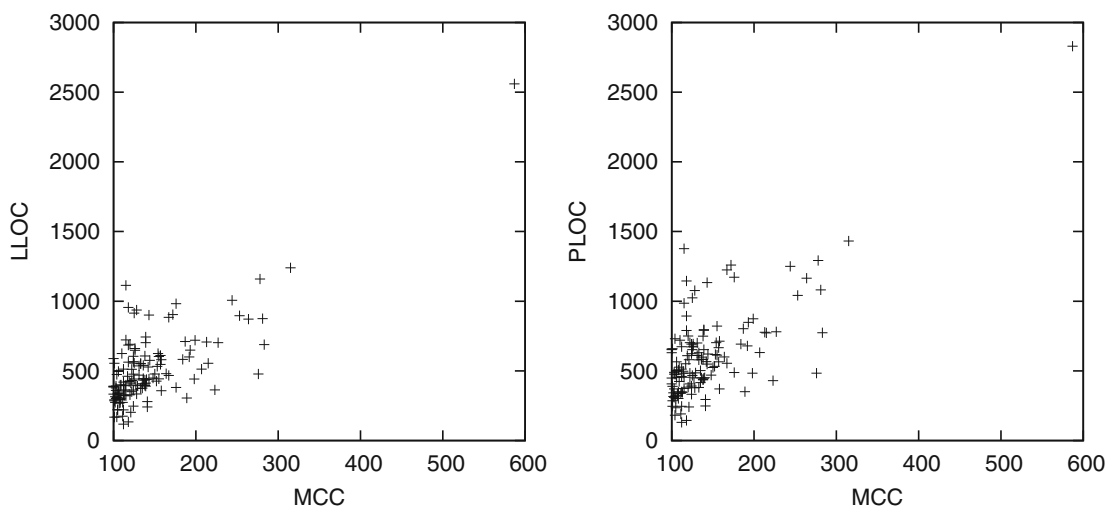


Fig. 7 Correlation of MCC with LLOC and PLOC

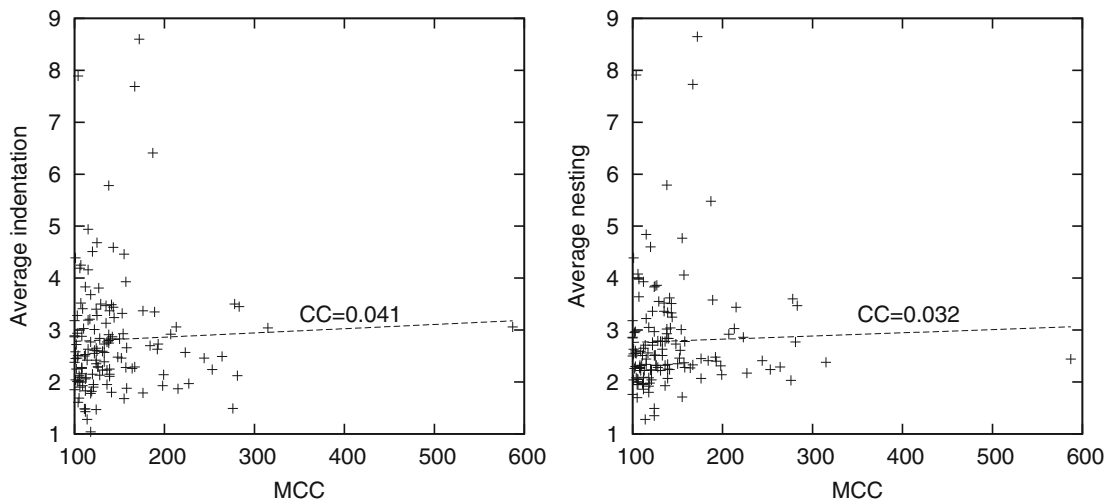


Fig. 8 Correlation of MCC with indentation and nesting

where the LOC outstrips the MCC by a factor of 30 or more (to the left of the top line in the figure).

But if we focus on the high-MCC functions, the picture is somewhat different. The results are shown in Fig. 7, with a distinction between LLOC, the non-comment non-blank lines of code, and PLOC, the total number of lines. The Spearman's rank correlation coefficients are 0.586 and 0.507, respectively, indicating a moderate degree of correlation; and indeed some functions have a relatively low MCC but high LOC, or vice versa. We used Spearman's coefficient rather than Pearson's because it is more sensitive to correlations when the relationships are not linear.

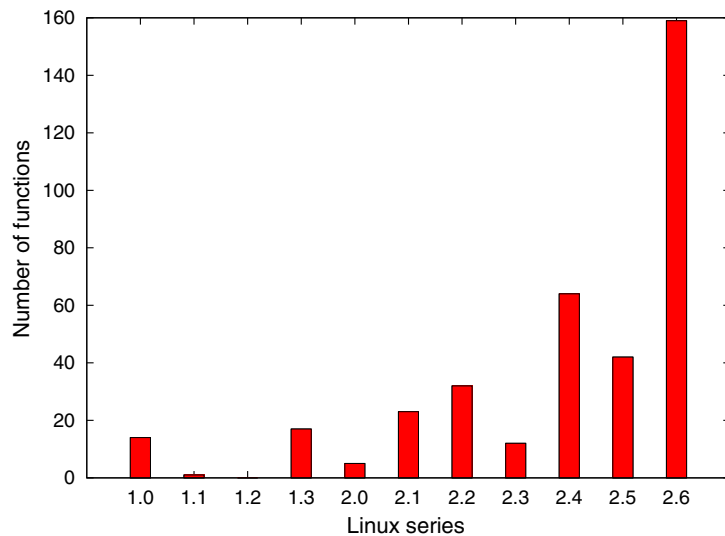
Another question is whether MCC is correlated with other complexity metrics. As an example, we checked the correlation of MCC with levels of indentation and nesting, based on the premise that indentation reflects levels of nesting and higher complexity (Hindle et al. 2008). Note that this has to be done carefully so as to avoid artifacts resulting from continuation lines where indentation does not reflect the structure of the code.

The results are shown in Fig. 8. Obviously there is almost no correlation of MCC with the average level of indentation or nesting in each function (verified by calculating the correlation coefficient). This reflects our findings that high-MCC functions could be either flat switches and sequences of ifs, or else deep trees of nested ifs, so a high MCC can come with either high or low nesting.

4 Maintenance and Evolution of High-MCC Functions

Linux is an evolving system (Israeli and Feitelson 2010). It has shown phenomenal growth during the 17 years till the time the kernel we studied was released in 2011: version 1.0 had 122,442 lines of actual code, and version 2.6.37.5 had 9,185,179 lines, an average annual growth rate of 29 %. This testifies to Lehman's law of "continuing growth" of evolving software systems (Lehman and Ramil 2003). Obviously, most of the functions in the current release didn't exist in the first release—they were added

Fig. 9 The distribution of new high-MCC functions (defined as those with $MCC > 100$) in Linux series. Note that the duration of the 2.6 series is much longer than the previous ones



at some point along the way. And there were also functions that were part of the kernel for some time and were later removed.

A function can achieve high MCC by incremental additions, or else a new function may already have a high MCC when it is added. In fact, this happened in all versions as shown in Fig. 9. (The relatively large number of new functions with MCC above 100 introduced during the 2.6 series is due to the length of this series, which was started in December 2003.) Regarding incremental growth, note that high-MCC functions are expected to be hard to maintain. It is therefore interesting to investigate their trajectory and check how often they are changed, and this was our research question 2. We did this for all Linux functions that achieved an MCC of 100 or more in any version of the kernel. There were 369 such functions.

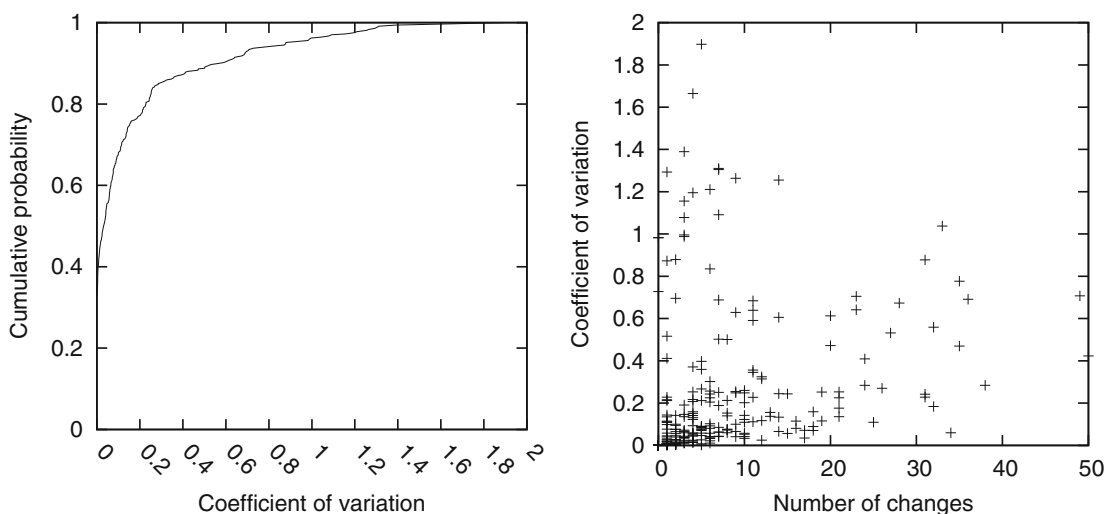


Fig. 10 *Left:* the distribution of the coefficient of variation of the MCC of 369 high-MCC functions. *Right:* scatter plot showing relationship between number of times the MCC changed and the degree of change as measured by the coefficient of variation (if the coefficient of variation equals 0 it means the MCC of this function did not change)

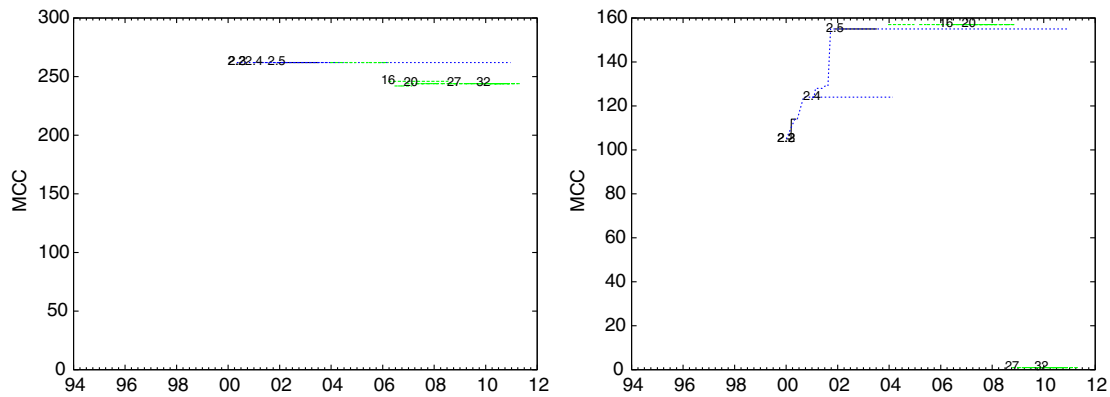


Fig. 11 Examples of functions whose MCC changed somewhat over time: `riocontrol`, and `ixj_ioctl`

To get an initial insight about the evolution of high-MCC functions, we calculate the coefficient of variation (CV) of the MCC of each function in different versions of Linux. The coefficient of variation is the standard deviation normalized by the average. Thus if a function never changes it will always have the same MCC, and the CV will be 0. If its MCC changes significantly with time, its CV can reach a value of 1 or even more. Figure 10 shows the distribution of the calculated CVs. About 33 % of the functions exhibit absolutely no change in the MCC across different versions of the kernel. Note that this does not necessarily mean that the functions were not modified at all, as we are only using data about the MCC. However it does indicate that in all likelihood the control structure did not change. Another large group of functions exhibit small to medium changes in MCC over time. Examples are shown in Fig. 11.² Finally, some functions exhibited significant changes in their MCC. Examples are shown in Fig. 12.

The degree to which the MCC changes is only one side of the story. In principle a very large change may occur all at once, or as a sequence of smaller changes. Therefore it is also interesting to check the number of times that the MCC was changed relative to the previous version. This has to be done carefully, because the Linux release scheme of using production and development versions (described below) implies that several versions may be current at the same time. Thus when a new branch is started, its previous version is typically near the start of the previous branch, not at its end.

Figure 10 shows a scatter plot that compares the degree of change with the number of changes. The correlation between these two metrics turns out to be relatively strong, with a Spearman's rank correlation coefficient of 0.83. This shows that additional changes tend to accumulate. However, despite the rapid rate in which new releases of the Linux kernel are made, the high-MCC functions do not change often. The highest number we saw was a function whose MCC changed 50 times.

An especially interesting phenomenon is that sometimes very large changes occur in production versions. The Linux kernel, up to the 2.6 series, employed a release

²In this and subsequent figures, we distinguish between development versions of Linux (1.1, 1.3, 2.1, 2.3, and 2.5), production versions (1.0, 1.2, 2.0, 2.2, and 2.4, shown as dashed lines), and the 2.6 series, which combined both types. These are identified only by their minor (third) number. The X axis is calendar years starting with the release of Linux in 1994.

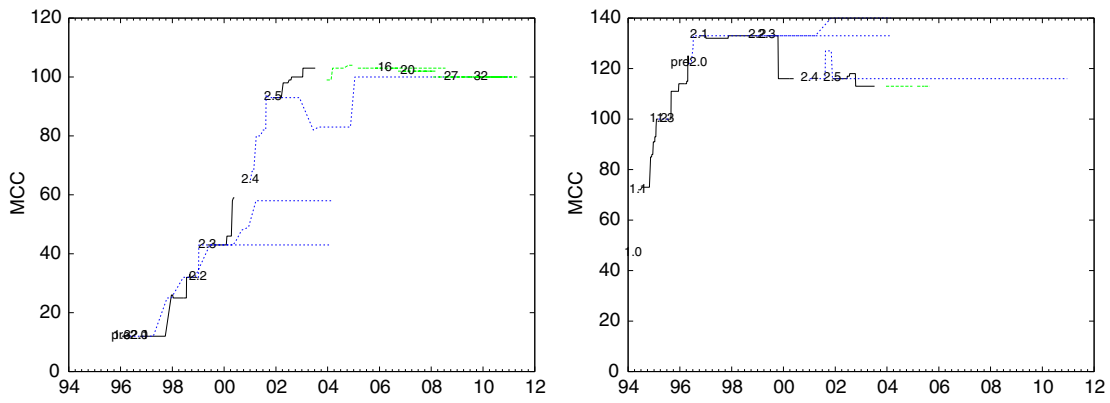


Fig. 12 Examples of functions that exhibit significant changes over time: `vortex_probe1`, and `st_int_ioctl`

scheme that differentiated between development and production. Development versions had an odd major number and their minor releases were made in rapid succession. Production versions, with even major numbers, were released at a much slower rate, and these releases were only supposed to contain bug fixed and security patches. However, our data shows several instances of large changes in the MCC of a function that occur in the middle of a production version (Figs. 13 and `vortex_probe1` from Fig. 12). Such behavior contradicts the “official” semantics of development vs. production versions. But at least in some of these cases the change was done in a production version during the interval between two successive development versions.

In most functions that saw a significant change in MCC the MCC grew. But there were also cases where the MCC dropped as shown in Fig. 14. The largest drop is in function `sys32_ioctl`. This is the function with the highest MCC ever, peaking at 620 in the later parts of kernel version 2.2. At an earlier time, in version 2.3.46, it had reached an MCC value of 563, but then in version 2.3.47 this dropped to 8. The reason was a design change, where a large switch was replaced by a table lookup (Israeli and Feitelson 2010). A similar change occurred in function `usb_stor_show_sense`, where a large switch statement was replaced by a call to a new function implementing a lookup table.

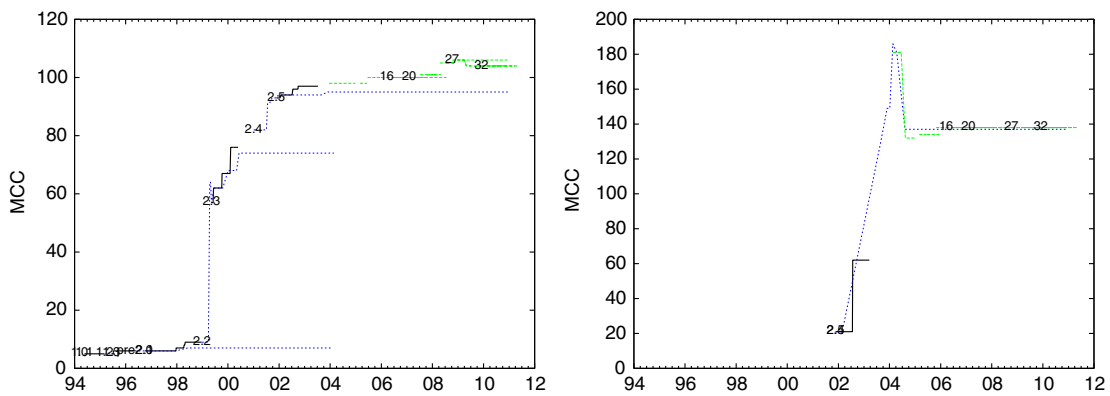


Fig. 13 Examples of functions that exhibit large changes in production versions: `sg_ioctl` and `SiS_EnableBridge`

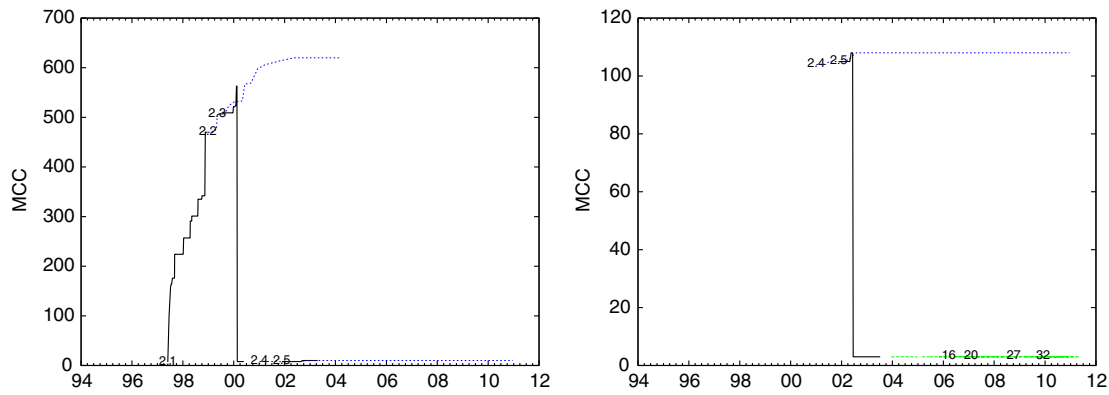


Fig. 14 Examples of functions that exhibit a sharp drop in MCC resulting from a design change: `sys32_ioctl` and `usb_stor_show_sense`

However, a sharp drop in MCC value does not necessarily mean a design change which yields reduced complexity. For example, the function `isdn_tty_cmd_PLUSF_FAX` had MCC 154 in version 2.2.14. In version 2.2.15 it dropped to 3 and the original code was replaced by conditional calls to two other new functions. One of these functions has MCC 154 exactly as the original function, and the other has MCC 15. Thus the high-MCC code just moved elsewhere. Likewise, in version 2.3.9 the function `l2o_proc_read_lan_media_operation` had MCC 102, which dropped to 12 in version 2.3.10. The original function had two large switches which were cloned later in the same function. In version 2.3.10 the two switches were replaced by two new functions. Each of the new functions contained one of the original switch statements and a new lookup table. The odd thing was that the lookup tables did not replace the switch statements and were not exploited to reduce complexity. Another example of an artificial reduction in MCC is function `fd_ioctl_trans`. The original function had many long compound if statements with heavy use of the or operator. In its reduced MCC version the logical or operator was replaced by the bitwise or which is not counted by the MCC metric.

The above examples may leave the impression that design changes to reduce MCC are purely technical. However, we also observed cases where the reduction resulted from a design change requiring a good understanding of the logic of the function,

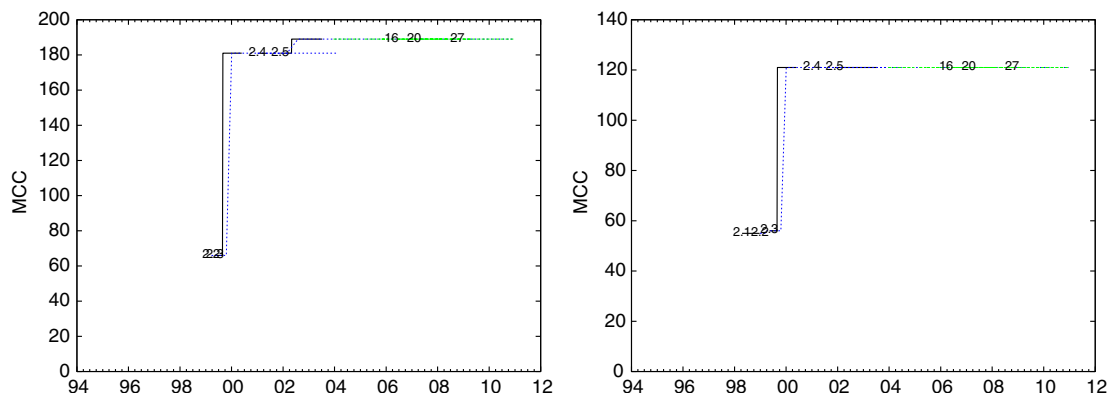


Fig. 15 Co-evolution of two related functions

as the changes are small and deeply interwoven within the code. An example of such a function is `main` in versions 2.4.25 and 2.4.26. The chief change in MCC resulted from defining 13 new secondary functions ranging from 1 to 50 lines of code. While in the old version negative numbers were used to indicate an error code when returning from a secondary function, in the new version these numbers were replaced by positive ones. In addition, in the old version all exceptional cases were handled locally, whereas in the new version the `goto` mechanism was used; upon exception execution jumps to a label which is located at the end of the function. All these changes require intimate understanding of the function.

Other interesting phenomena that occurred during maintenance were co-evolution and migration. Figure 15 shows the co-evolution of two related functions. These functions are `do_mathemu` in `/arch/sparc64/math-emu/math.c`, and `do_one_mathemu` in `/arch/sparc/math-emu/math.c`. This occurs when two related functions evolve according to a similar pattern. In many cases this happens because one of the functions was originally cloned from the other. In the above example, these are analogous functions in 32-bit and 64-bit architectures; when a large change was implemented, it was done in both in parallel. Also, in both cases the change that was initially done in a development version was soon after propagated to the contemporaneous production version.

An example of migration is shown in Fig. 16: the `vt_ioctl` function, which moved from `/drivers/char/vt.c` (MCC of 159 in kernel 2.5.35) to `/drivers/char/vt_ioctl.c` (same MCC of 159 in kernel 2.5.36). In fact, these two functions are indeed identical. As another example, `cpia_write_proc` from `/drivers/char/cpia.c`, with an MCC of 226 in kernel 2.2.26, morphed into `cpia_write_proc` in `/drivers/media/video/cpia.c`, with an MCC of 211 in kernel 2.4.0 (via the 2.3.99-prex series, where it already was 211). The change in MCC reflects some changes in the structure of the function. Much larger changes occurred when `x86_emulate_memop` from `/drivers/kvm/x86_emulate.c`, with an MCC of 285 in 2.6.24.7, morphed into `x86_emulate_insn` in `/arch/x86/kvm/x86_emulate.c` with an MCC of 174 in 2.6.25. While the second function is partly based on the first, significant changes were made, and the MCC changed considerably as well.

To summarize, high-MCC functions in the Linux kernel evolve in a variety of ways. This includes cases where a function changes significantly over time in a series of individual changes, and cases where functions are split or completely restructured.

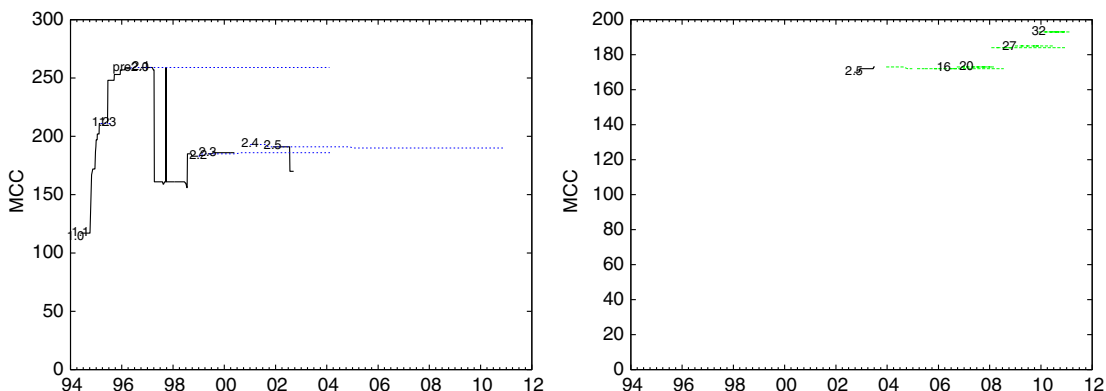


Fig. 16 Evolution of the `vt_ioctl` function, which migrated from one file to another

Taken together, these observations provide evidence for the capability of developers to handle these seemingly complex functions. In the next two sections we investigate whether they are indeed so complex.

5 Survey of Perceived Complexity

The *raison d'être* of the definition of MCC is the desire to be able to identify complex code, with the further goal of avoiding or restructuring it. This is also the reason for specifying threshold values, and requiring functions that surpass these thresholds to have proper justification. But the question remains whether MCC indeed captures complexity as perceived by human programmers.

5.1 Correlation of MCC and Perceived Complexity

To gain some insight into this question, which is our research question 3.1, we conducted a survey of the perceived complexity of high-MCC functions. The survey included 92 high-MCC functions that had been identified at the time. It was based on 14 participants, 8 from a summer Linux kernel workshop (advanced undergraduates, some with industrial experience, but with no prior kernel experience), and 6 that were recruited later (all were good students after an advanced course in C programming). The goal was to identify notions of perceived complexity, not to quantify the effect of complexity on developer performance. Thus the survey was conducted in two hour-long sessions, in which participants were required to page through each function for one minute and then give it a grade based on how complex (hard to understand) it looked to them. Grades were given on a personal relative scale.³ These individual scales were then linearly normalized to the range 0 to 10, and the average and standard deviation of the grades for each function were computed. The order in which the functions were presented was not related to MCC or any other attribute, but all participants received the list in the same order. At the end of the survey, participants were given an opportunity to comment in writing and some indeed provided notes with their insights.

The results, shown in Fig. 17, indicate little correlation between MCC and perceived complexity for high-MCC functions. In particular, some functions with relatively low MCC (within this select set of high-MCC functions) were graded as having either very high or very low perceived complexity. In the following we focus on these functions that were perceived as very different but this was not reflected by their MCC.

The functions that had very low average scores (and to a lesser degree also those with very high scores) also had relatively low standard deviations, as indicated by the short error bars. This is partly a result of scores having a limited range of 0–10; an average of say 1 then implies that it is highly improbable to have any high scores. But some of the functions with a moderate average complexity actually had both low and high scores of perceived complexity, leading to a high standard deviation.

³This was chosen to enable them to respond to surprises. Thus if they see a function they think is “very complex” and give it a high mark, and later another that is even much more complex, they can still express this using a value beyond their previously used range.

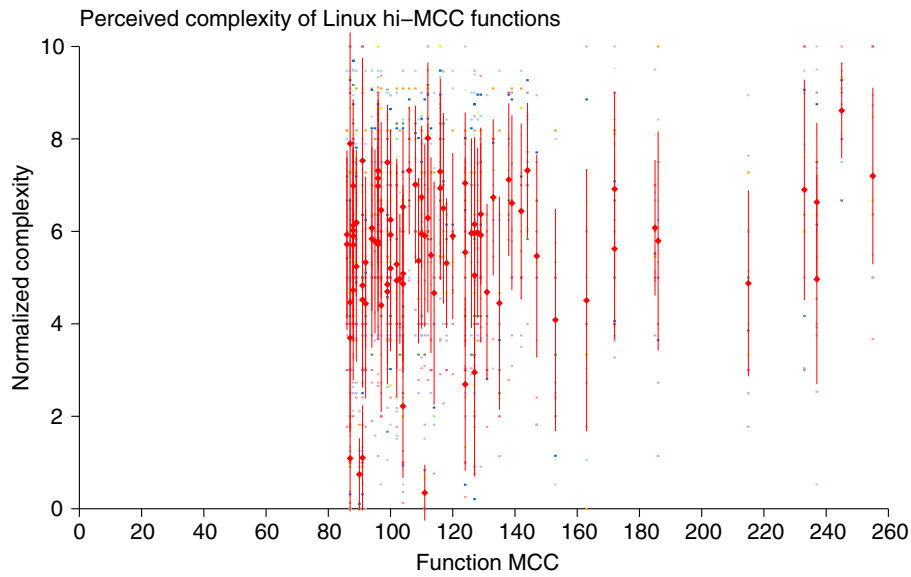


Fig. 17 Scatter plot showing relationship between measured MCC and perceived complexity. The small markings are individual grades. The average grade for each function is marked by a larger diamond, and the error bars denote standard deviations

This is partly due to the fact that complexity is not well defined and the grading was subjective. For example, one very long function with high MCC value suffered a strong disagreement among survey participants. This could be because this function is composed of a mix of simple as well as messy segments.

5.2 Aspects of Complexity Missed by MCC

The functions that were ranked as low complexity are relatively easy to characterize. These are generally functions dominated by a very regular switch construct, where the cases are very small and straightforward. For example, the switch may be used to assign error or status message strings to numerical codes, leading to a single instruction in each case as illustrated in Fig. 18.

In addition to these single-instruction cases, survey participants noted that long sequences of empty cases should not be counted as adding complexity; indeed, these are equivalent to predicates in which many options are connected by logical or (and of the tools we surveyed, VerifySoft indeed does not count empty cases). Furthermore,

Fig. 18 Example of simple switch structure from log_audio_status

```
switch (mod_det_stat0) {
case 0x00: p = "mono"; break;
case 0x01: p = "stereo"; break;
case 0x02: p = "dual"; break;
case 0x04: p = "tri"; break;
case 0x10: p = "mono with SAP"; break;
case 0x11: p = "stereo with SAP"; break;
case 0x12: p = "dual with SAP"; break;
case 0x14: p = "tri with SAP"; break;
case 0xfe: p = "forced mode"; break;
default: p = "not defined";
}
```

Fig. 19 Example of a sequence of independent ifs with the same structure, from `ixj_daa_write`. The full function includes 113 such ifs

```

bytes.high = 0x14;
bytes.low = j->m_DAAShadowRegs.SOP_REGS.SOP.cr4.reg;
if (!daa_load(&bytes, j))
return 0;

bytes.high = j->m_DAAShadowRegs.SOP_REGS.SOP.cr3.reg;
bytes.low = j->m_DAAShadowRegs.SOP_REGS.SOP.cr2.reg;
if (!daa_load(&bytes, j))
return 0;

bytes.high = j->m_DAAShadowRegs.SOP_REGS.SOP.cr1.reg;
bytes.low = j->m_DAAShadowRegs.SOP_REGS.SOP.cr0.reg;
if (!daa_load(&bytes, j))
return 0;

if (!SCI_Prepare(j))
return 0;

bytes.high = 0x1F;
bytes.low = j->m_DAAShadowRegs.XOP_REGS.XOP.xr7.reg;
if (!daa_load(&bytes, j))
return 0;

bytes.high = j->m_DAAShadowRegs.XOP_xr6_W.reg;
bytes.low = j->m_DAAShadowRegs.XOP_REGS.XOP.xr5.reg;
if (!daa_load(&bytes, j))
return 0;

bytes.high = j->m_DAAShadowRegs.XOP_REGS.XOP.xr4.reg;
bytes.low = j->m_DAAShadowRegs.XOP_REGS.XOP.xr3.reg;
if (!daa_load(&bytes, j))
return 0;

bytes.high = j->m_DAAShadowRegs.XOP_REGS.XOP.xr2.reg;
bytes.low = j->m_DAAShadowRegs.XOP_REGS.XOP.xr1.reg;
if (!daa_load(&bytes, j))
return 0;

bytes.high = j->m_DAAShadowRegs.XOP_xr0_W.reg;
bytes.low = 0x00;
if (!daa_load(&bytes, j))
return 0;

if (!SCI_Prepare(j))
return 0;

```

repeated use of the same code template (easily identified using a CSD), e.g. in a long sequence of small ifs that all have exactly the same structure, also reduces the perceived complexity considerably. An example is shown in Fig. 19.

At the other end of the spectrum, functions that received very high grades for perceived complexity tended to exhibit either of two features. One was the use of `gotos` to create spaghetti-style code, in which target labels are interspersed within the function's code in different locations. An example was shown in Fig. 5. Note that such a `goto` is deterministic, and therefore not counted by the MCC metric as a branch point. This should be contrasted with forward `gotos` that are used to break out of a

Fig. 20 Example of excessive line breaks that seem to make the code harder rather than easier to understand, from `balance_leaf`

```

if (ret_val
    && !item_pos) {
    pasted =
        B_N_PITEM_HEAD
        (tb->L[0],
         B_NR_ITEMS
         (tb->
          L[0]) -
          1);
    l_pos_in_item +=
        I_ENTRY_COUNT
        (pasted) -
        (tb->
         lbytes -
         1);
}

```

complex control structure in case of an error condition. Such `gotos` were tolerated by survey participants and even considered as improving structure.

The second feature that added to perceived complexity was unusual formatting. One manifestation of such formatting was using only 2 characters as the basic unit of indentation (instead of the common 8-character wide tab). This led to the code looking more dense and made it harder to decipher the control structure. Another manifestation was the use of excessive line breaks, even within expressions, as illustrated in Fig. 20. These observations hark back to the work of Soloway and Ehrlich (1984), who show that even expert programmers have difficulty comprehending code that does not conform to structural conventions. Obviously the problem could be avoided by using a pretty-printing routine to reformat the code, but evidently this was not done.

5.3 Comparing Functions to Identify Elements of Complexity

The functions that were found to have the lowest perceived complexity provide an especially interesting case study. These functions are generally based on large `switch` statements, and most if not all of their MCC score is derived from cases in these `switches`. We start by ranking these functions according to their perceived complexity. By comparing neighboring functions in this ranking we can then identify code characteristics that led to discrete increases in perceived complexity (thus answering research question 3.2). This could be done in the first 7 functions; beyond that, it was not possible to identify individual discrete changes any more.

The function with the lowest perceived complexity score is indeed very simple. This function has one parameter, and its body comprises a single `switch` statement with a long sequence of cases that are compared against the function's parameter. The values of the cases are numeric constants and their bodies are single-line blocks that return a string value. Moreover, the cases are grouped into sets of logically related cases. These sets are paragraphed (separated by blank lines) and headed by a single-line comment.

The next function, which was graded as twice more complex than the first one, accepts one non-scalar parameter, and again contains one `switch` statement with a

```

char *capi_info2str(u16 reason)
{
    switch (reason) {

/*--- informative values (corresponding message was processed) ---*/
    case 0x0001:
        return "NCPI_not_supported_by_current_protocol , _NCPI_ignored";
    case 0x0002:
        return "Flags_not_supported_by_current_protocol , _flags_ignored";
    case 0x0003:
        return "Alert_already_sent_by_another_application";

/*--- error information concerning CAPI_REGISTER ---*/
    case 0x1001:
        return "Too_many_applications";
    case 0x1002:
        return "Logical_block_size_too_small , _must_be_at_least_128_Bytes";
    case 0x1003:
        return "Buffer_exceeds_64_kByte";
    case 0x1004:
        return "Message_buffer_size_too_small , _must_be_at_least_1024_Bytes";
    case 0x1005:
        return "Max_number_of_logical_connections_not_supported";
    case 0x1006:
        return "Reserved";
    case 0x1007:
        return "The_message_could_not_be_accepted_because_of_an_internal_
            busy_condition";
    case 0x1008:
        return "OS_resource_error_(no_memory?)";
    case 0x1009:
        return "CAPI_not_installed";
    case 0x100A:
        return "Controller_does_not_support_external_equipment";
    case 0x100B:
        return "Controller_does_only_support_external_equipment";

/*--- error information concerning message exchange functions ---*/
    case 0x1101:
        return "Illegal_application_number";
    case 0x1102:
        return "Illegal_command_or_subcommand_or_message_length_less_than_12_
            _bytes";
    case 0x1103:
        return "The_message_could_not_be_accepted_because_of_a_queue_full_
            condition_!!_The_error_code_does_not_imply_that_CAPI_cannot_
            receive_messages_directed_to_another_controller , _PLCI_or_NCCI";
    case 0x1104:
        return "Queue_is_empty";
    case 0x1105:
        return "Queue_overflow , _a_message_was_lost_!!_This_indicates_a_
            configuration_error_._The_only_recovery_from_this_error_is_to_
            perform_a_CAPI_RELEASE";
    case 0x1106:
        return "Unknown_notification_parameter";
    case 0x1107:
        return "The_Message_could_not_be_accepted_because_of_an_internal_
            busy_condition";
    case 0x1108:
        return "OS_Resource_error_(no_memory?)";
    case 0x1109:
        return "CAPI_not_installed";
    case 0x110A:
        return "Controller_does_not_support_external_equipment";
    case 0x110B:
        return "Controller_does_only_support_external_equipment";

        ... // 4 paragraphs of cases removed to save space

    default: return "No_additional_information";
    }
}

```

Listing 1 Listing of the function with the lowest perceived complexity

```

void usb_stor_show_command(struct scsi_cmnd *srb)
{
    char *what = NULL;
    int i;

    switch (srb->cmnd[0]) {
    case TEST_UNIT_READY: what = "TEST_UNIT_READY"; break;
    case REZERO_UNIT: what = "REZERO_UNIT"; break;
    case REQUEST_SENSE: what = "REQUEST_SENSE"; break;
    case FORMAT_UNIT: what = "FORMAT_UNIT"; break;
    case READ_BLOCK_LIMITS: what = "READ_BLOCK_LIMITS"; break;
    case REASSIGN_BLOCKS: what = "REASSIGN_BLOCKS"; break;
    case READ_6: what = "READ_6"; break;
    case WRITE_6: what = "WRITE_6"; break;
    case SEEK_6: what = "SEEK_6"; break;
    case READ_REVERSE: what = "READ_REVERSE"; break;
    case WRITE_FILEMARKS: what = "WRITE_FILEMARKS"; break;
    case SPACE: what = "SPACE"; break;
    case INQUIRY: what = "INQUIRY"; break;
    case RECOVER_BUFFERED_DATA: what = "RECOVER_BUFFERED_DATA"; break;
    case MODE_SELECT: what = "MODE_SELECT"; break;
    case RESERVE: what = "RESERVE"; break;
    case RELEASE: what = "RELEASE"; break;
    case COPY: what = "COPY"; break;
    case ERASE: what = "ERASE"; break;
    case MODE_SENSE: what = "MODE_SENSE"; break;
    case START_STOP: what = "START_STOP"; break;
    case RECEIVE_DIAGNOSTIC: what = "RECEIVE_DIAGNOSTIC"; break;
    case SEND_DIAGNOSTIC: what = "SEND_DIAGNOSTIC"; break;
    case ALLOW_MEDIUM_REMOVAL: what = "ALLOW_MEDIUM_REMOVAL"; break;
    case SET_WINDOW: what = "SET_WINDOW"; break;
    case READ_CAPACITY: what = "READ_CAPACITY"; break;
    case READ_10: what = "READ_10"; break;
    case WRITE_10: what = "WRITE_10"; break;
    case SEEK_10: what = "SEEK_10"; break;
    case WRITE_VERIFY: what = "WRITE_VERIFY"; break;
    case VERIFY: what = "VERIFY"; break;
    case SEARCH_HIGH: what = "SEARCH_HIGH"; break;
    case SEARCH_EQUAL: what = "SEARCH_EQUAL"; break;
    case SEARCH_LOW: what = "SEARCH_LOW"; break;
    case SET_LIMITS: what = "SET_LIMITS"; break;
    case READ_POSITION: what = "READ_POSITION"; break;
    case SYNCHRONIZE_CACHE: what = "SYNCHRONIZE_CACHE"; break;
    case LOCK_UNLOCK_CACHE: what = "LOCK_UNLOCK_CACHE"; break;
    case READ_DEFECT_DATA: what = "READ_DEFECT_DATA"; break;
    case MEDIUM_SCAN: what = "MEDIUM_SCAN"; break;
    case COMPARE: what = "COMPARE"; break;
    case COPY_VERIFY: what = "COPY_VERIFY"; break;
    case WRITE_BUFFER: what = "WRITE_BUFFER"; break;
    case READ_BUFFER: what = "READ_BUFFER"; break;
    case UPDATE_BLOCK: what = "UPDATE_BLOCK"; break;
    case READ_LONG: what = "READ_LONG"; break;
    case WRITE_LONG: what = "WRITE_LONG"; break;
    case CHANGE_DEFINITION: what = "CHANGE_DEFINITION"; break;

        ... // cases removed to save space

    default: what = "(unknown_command)"; break;
    }
    US_DEBUGP("Command_%s_(%d bytes)\n", what, srb->cmd_len);
    US_DEBUGP("");
    for (i = 0; i < srb->cmd_len && i < 16; i++)
        US_DEBUGPX("_%02x", srb->cmnd[i]);
    US_DEBUGPX("\n");
}

```

Listing 2 Listing of the second function with low perceived complexity

long sequence of cases. The values of the cases are symbolic constants (except a few cases of numeric constants) and their bodies assign string values to a shared variable and then break. There is no paragraphing nor comments. After the switch statement there are a very simple loop and a call to a macro. The loop references (for the first time) a variable which was defined before the switch statement, and the macro uses the variable which was previously assigned within the switch statement. Listings 1 and 2 represent the first and second functions.

The third and fourth functions were very similar to each other and received very close perceived complexity grades. They accept one non-scalar parameter and are composed of many separate switch statements with paragraphing but no comments. The values of the cases are numeric constants and the bodies are assignments to a shared variables, followed by break. A few of these switch statements are governed by a very simple if and else, so we see some nesting. Nevertheless, the structure of these functions is still quite flat and regular.

The fifth function introduces several new elements for the first time in this series, and its average grade is again double that of the previous one. Its header is much more complex than previous functions, and contains an additional modifier besides the traditional structure. Moreover, it contains more parameters than before where some are simple and scalar and others are aggregate. These parameters are listed over multiple lines, and in one case the type of a parameter was defined in one line and its name in the next line. This function is still dominated by a large switch statement with mostly (80 %) consecutive empty cases. The rest of the cases contain one if statement or a for loop with a nested if statement. In both cases the blocks of statements are very simple, but the conditions in the ifs span multiple lines.

The sixth function is composed of one large switch statement where each of its cases is composed of another large switch statement with one simple line for each of its cases. Moreover, the first case of the outer switch actually contains an if/else construct with two switch statements in them.

The last function, which was graded as a bit more complex than the previous one, is composed of two large switch statements that are controlled by if and else. The cases of these switch statements are composed of nested ifs and elsels with simple conditions. Roughly, the blocks within the different cases create five categories of regular blocks. Despite the deep nesting in the different cases, the impression is that this nesting is used to break up ifs with very complicated conditions. This is obvious because each of these blocks performs a single statement in its innermost level.

The above allows us to identify the following elements of complexity, which are generally not acknowledged by metrics like MCC:

- Ending a case with a break, followed by some additional processing after the switch, is more complex than having a return directly in the case.
- Several small switches (probably switching on different variables) are more complex than one large switch.
- Using constructs of different types, e.g. ifs in addition to a switch, increases complexity.
- Adding parameters to a function increases complexity.
- Increasing the nesting of constructs in each other increases complexity.
- Embedding switch statements within ifs and elsels is more complex than having the switch at the top level.

In some of these cases the more complex version cannot be avoided due to the logic of the program. But still we can suggest the following *Dos* and *Don'ts* lessons:

- Don't separate processing, localize whenever it is possible.
- If possible, prefer one large switch rather than splitting across many smaller ones.
- Try to avoid mixing constructs of different types.
- Use paragraphing (empty lines separating blocks of code) and comments.

6 Regularity and Perceived Complexity

As we have already stated, High-MCC functions are quite long. Therefore, a visual representation such as that provided by CSDs may ease capturing their code as a whole, and may help in grasping structural properties and regularities. This raises the empirical question of whether a visual view of high-MCC functions has an advantage over a simple listing of the code, from a human point of view. This is research question 3.3.

As noted above, using CSDs exposed some functions as being very regular while others appear to have irregular code structure. This reflects a combination of the sequence of constructs used, their nesting pattern, and formatting aspects such as indentation and paragraphing. It seems likely that these factors contribute to the perception of complexity, even though they are not taken into account by the MCC metric. A second question is therefore whether regularity correlates with perceived complexity. If it does, this would answer our research question 4 in the affirmative.

To answer these questions we conducted an experiment with 15 experienced programmers. We required that the subjects must have experience in the C language. All subjects were males except two, with an average age of 31, and an average of 4.8 years experience with C.

The experiment consisted of 30 high-MCC functions, presented in two different formats. In one phase the code listing of the functions was presented, and in the other phase the CSD diagrams of the functions were presented. The two phases were performed separately with a break of at least one day between them. Which phase (code or CSD) was done first was randomized across subjects. The task was to assign each function with a perceived complexity score, as in the previous experiment. Before starting, participants were presented with a short description of CSDs and an example showing the code and CSD of the same function side by side.

Somewhat surprisingly, the results show that the CSD view had no advantage over the code view. In fact, in two-thirds of the cases there was no significant difference in the average scores of the two types of view. Thus it seems that experienced programmers can achieve a good impression of code by paging through it, and seeing a graphical representation of the code structure did not provide much additional information.

To answer the question of whether perceived complexity correlates with regularity, we used the Lempel–Ziv compression algorithm (Ziv and Lempel 1978) to compress each of the 30 functions and computed the percentage of reduction in size for each one. As this algorithm is based on identifying recurring substrings in the input, we expect that functions with high regularity will be compressed better than irregular functions. But it is still unclear what parts of the code should be compressed

These results suggest that low compressibility correlates with high perceived complexity, and by implication, that irregularity correlates with high perceived complexity.

7 Possibility of Replacing or Refactoring High-MCC Functions

To answer research question 5 (is all the high-MCC code really necessary) we surveyed all 369 such functions that were collected from more than a thousand versions of the Linux kernel, using the version with the highest MCC value for each one. This complements the quantitative metrics discussed above with a qualitative discussion aimed to gain some insights about their nature. We checked cloning, replacement of code by a lookup table, and the option of factoring out some functionality to a subroutine.

To find possible instances of cloning we compared the source code of each pair of high-MCC functions. For doing this we used the `diff` Linux command, with parameters to disregard differences in spaces and blank lines. We also allowed up to 10 % of the total lines of the compared functions in each pair to be different. We repeated this process twice: Once for the full source code, and again based on the skeleton of the functions (only the keywords and braces, as explained in Section 6) while preserving formatting and nesting. Using the code structure comparison, we found 56 sets of clones, where 34 are pairs of functions, 13 involve 3 functions, 5 have 4 functions, and 4 include no less than 5 clones. For the full code, we found 51 sets of clones. These results indicate that nearly a quarter of the high-MCC functions are clones of other high-MCC functions. The existence of so many clones indicates that developers found it better to create clones with small changes rather than to abstract away the common functionality and adjust it for different uses by parameterization.

As we stated earlier, some of the high-MCC functions are written in a way that enables replacing them by a lookup table. We manually examined the 369 functions and counted those that are likely candidates for replacement by a lookup table. We found 19 such functions. In addition, we observed some functions that can be partially replaced (meaning that they contain a few code segments that can be replaced with a lookup table). There were 23 such functions. These include two sets of size 2 and 3 which also appeared in the clone list. As demonstrated in Section 4, replacement of a high-MCC function by a simpler function based on table lookup is a transformation that indeed occurs in practice.

An especially interesting question is whether well-known refactoring techniques may be applied to high-MCC functions. As high-MCC functions are long, there is a good chance for applying refactoring techniques such as *function extraction*. As an initial check, we tried to identify clone code segments within a given function. We were assisted by the CSD diagram of each function to get initial impression about cloned segments. We reviewed all 369 CSDs manually in a single-evaluator style, and subjectively extracted 61 functions that have what appear to be large cloned segments. Two examples are shown in Fig. 23. These 61 functions have no overlap with the lookup table functions, but may overlap with the clone list. This indicates that about 1 in 6 high-MCC functions may be amenable to function-extraction refactoring, but at the same time, that developers prefer to replicate code rather than doing so.

To answer this, we analyzed the source code of three additional operating systems, and three open source systems from other domains. In the operating systems domain we chose Windows, FreeBSD, and OpenSolaris. From non-OS domains we chose GCC (compilers), Firefox (browsers), and the OpenSSL toolkit. The Windows Research Kernel (WRK) contains the source code for the NT-based kernel which is compatible with Windows Server 2003. Its source code includes core sources for object management, processes, threads, virtual memory, and the I/O system. It does not include Plug-and-Play, power management, virtual DOS machine, and the kernel debugger engine. The other five are the full codebases of FreeBSD, OpenSolaris, GCC, Firefox, and OpenSSL respectively.

Table 2 summarizes initial results of our analysis. We see that all these systems contain high-MCC functions with extreme values at their upper bound. For example, in the FreeBSD system the highest MCC value was 1316 which is 26 times higher than the highest threshold that was ever defined. It is true that the absolute number of these functions in each system is small, but they represent a non-negligible fraction of the control flow constructs in the respective systems. This is listed in the table under ‘percent of MCC values’, meaning what fraction of the total MCC summed over all the functions in the system is contained in the high-MCC functions. For example, in the Windows system functions with an MCC above 50 account for more than 18 % of the total MCC in the system.

According to Table 2 there are relatively few high-MCC functions and a large number of low-MCC functions. This observation indicates that the distribution of MCC values is skewed in all of the systems. An important class of skewed distributions are distributions with heavy tails. The common definition of heavy-tailed distributions is that their tail is governed by a power law, so $\Pr(X > x) \propto x^{-\alpha}$. To test the existence of a power-law tail one can use the log-log complementary distribution plot. This plot should produce a straight line for a perfect power law tail where its slope corresponds to the tail index α . Such plots for the MCC values of functions in our four operating systems and three other applications are shown in Fig. 24. In all cases the lines look straight, and do not plunge downwards as they would for short-tail distributions. Because we are interested in the tail of the distribution we focused on the top 1 % of the values and performed a linear regression. The results show that the tail indices for all the systems are a bit higher than 2, so these distributions have a bounded variance. The common definition of heavy tailed distributions requires a tail index in the range between 0 and 2, which leads to unbounded variance. Thus

Table 2 High-MCC function characteristics of four operating systems and three open source projects from other domains.

Name	Version	Total funcs	Max MCC	# high-MCC funcs		% of MCC values	
				≥ 100	> 50	≥ 100	> 50
Windows	WRK-v1.2	4074	246	18	84	7.0	18.6
FreeBSD	9 (stable)	67528	1316	103	490	5.3	11.8
OpenSolaris	8	21259	506	34	202	4.2	11.6
Linux	2.6.37.5	259137	587	138	765	1.6	5.1
Firefox	9 (stable)	26444	699	27	181	3.3	9.9
GCC	4.8.0	72542	1301	248	938	10.2	21.3
OpenSSL	1.0.0k	6560	371	22	78	9.0	18.2

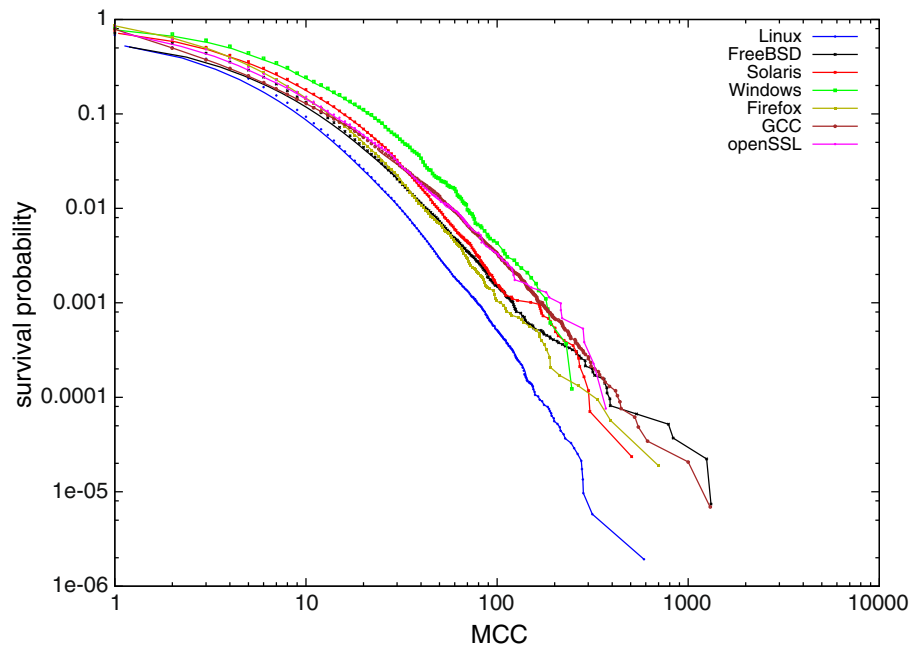


Fig. 24 Log-log complementary distribution plot of MCC values in four different systems

these distributions are a border case: they have a power law tail, but with a tail index of slightly more than 2.

9 Discussion and Conclusions

We have shown that the practice as reflected in the Linux kernel regarding large and complex functions diverges from common wisdom as reflected by thresholds used in various automatic tools for measuring MCC. This is not surprising, as a simplistic threshold cannot of course capture all the considerations involved in structuring the code. However, it does serve to point out an issue that deserves more thorough empirical research. We now turn to the implications of our findings.

9.1 MCC and Linux Quality

The basic underlying question we faced was whether the high-MCC functions in the Linux kernel constitute a code quality problem, or maybe such functions are actually acceptable and the warnings against them are exaggerated. This was our final research question, 7, and we can now discuss it based on all our findings.

Linux provides several examples where long and sometimes complex functions with a high MCC seem to be justified. It is of course possible to split such functions into a sequence of smaller functions, but this will be an artificial measure that only improves the MCC metric, and does not really improve the code. On the contrary, it may even be claimed that such artificial dissections degrade the code, by fragmenting pieces of code that logically belong together.

For example, one class of functions that tend to have very high MCC values are those that parse the options of some operation, in many cases the flag values of an

`ioctl` (I/O control) system call for some device. There can be very many such flags, and the input parameter has to be compared to all of them. Once a match is found, the appropriate action is taken. Splitting the list of options into numerous shorter lists will just add clutter to the code.

Another class of functions that tend to have high MCC values are functions concerned with the emulation of hardware devices, typically belonging to unavailable (possibly legacy) architectures. The device may have many operations that each needs to be emulated, and furthermore this needs to take into account many different attributes of the device. Thus there are very many combinations that need to be handled, but partitioning them into meaningful subgroups may not be possible.

Despite the inherent size (and high MCC) of these functions, in many cases it may be claimed that they do not in fact cause a maintenance burden. This can happen either because they need not be maintained, or because they are actually not really complex.

As we saw in Section 4, more than a third of our functions exhibited no or negligible changes during the period of observation. In some of the other functions, which had larger changes, there was only a single large-change event. Thus most functions actually displayed strong stability the vast majority of the time. On average these functions do not require much effort to maintain.

Alternatively, functions with a high MCC may not really be so difficult to comprehend and maintain. MCC counts branch points in the code. If the cumulative effect of many branch points is to describe a complex combination of concerns, it may be hard for developers and maintainers to keep track of what is going on. But if the branching is used to separate concerns, as in the example of handling different flag values in an `ioctl`, this actually makes the code readable.

Our conclusion is therefore that for the most part the high-MCC functions found in Linux do not constitute a serious problem. On the contrary, they can serve as examples of situations where prevailing dogmas regarding code structure may need to be lifted.

9.2 Refinements to the MCC Metric

The observation that the MCC value of a function may not reflect “real” complexity as it is perceived by developers has been made before. Based on this, there have been suggestions to modify the metric to better reflect perceived complexity. Two previously suggested refinements are the following:

- Do not count cases in a large switch statement. This was mentioned already in McCabe’s original paper (McCabe 1976), and is re-iterated in the MSDN documentation (MSDN 2008).
- Also do not count successive if statements, as successive decisions are not as complex as nested ones (Harrison et al. 1982).

Both of these modifications together define McCabe’s “essential” complexity metric, leading to a reduced value that assigns complexity only to more convoluted structures. But at the same time McCabe suggests a lower threshold of only 4 for this metric (McCabe Software 2009).

Generalizing the above, we suggest that one should not penalize “divide and conquer” constructs where the point is to distinguish between multiple *independent*

actions. This may include nested decision trees in addition to switch statements and sequences of if statements. Note, however, that this refines the simple syntactic definition, as it is crucial to ensure that the individual conditions are indeed independent. For example, a switch statement in which a *non-empty* case falls through to the next case violates this independence, and thus adds complexity to the code.

The above suggestions are straightforward consequences of applying the principle of independence to basic blocks of code. However, this does not yet imply that they lead to any improvements in terms of measuring complexity. This would require a detailed study of code comprehension by human developers, which we leave for future work.

One more aspect that should be considered in MCC refinement is regularity. It is reasonable to think that regular functions need less effort to comprehend than irregular ones. As we have already seen compression algorithms tend to reflect the regularity extent in functions. This can be used to help in counterbalancing the exaggerated values of the MCC metric. In addition, we note based on our experience with Linux scheduling (e.g. Etsion et al. 2006) that at least in some cases complexity is much more a result of how the logic of the code is expressed than a result of its syntactical structure. For example, even knowing the scheduling algorithm, it was hard to understand how the code implements this algorithm, despite the fact that its MCC was reasonably low. Thus syntactic metrics like MCC cannot be expected to give the full picture.

9.3 Threats to Validity

Our results are subject to several threats to validity.

Linux uses `#ifdefs` to enable configuration to different circumstances. Analyzing code that contains such directives may be problematic due to unbalanced braces. We are aware of this and dropped files that were tagged as syntactically incorrect by the `pmccabe` tool. In spite of their low percentage, these files may contain interesting functions with high MCC values that we would have missed.

While `pmccabe` is a well known tool for calculating MCC values, we found a bug in it: it counted the caret symbol (bitwise xor) as adding to the MCC value. We wrapped `pmccabe` with code that fixed this bug, and manually confirmed the results for selected functions. However, other bugs may exist in this and other tools.

In assessing the evolution of high-MCC functions, we actually rely on the MCC values. This is not necessarily right because a function may change without affecting the control constructs, or it may be that one construct was deleted but another was added. Thus our counts of changes may err on the conservative side. Our survey on perceived complexity also suffers from a few threats. For example, grading 92 functions within 2 hours is difficult and causes fatigue, which may affect the grading of the last functions. Moreover, a learning effect may also occur.

The survey of perceived complexity suffers from being subjective. It would have been good to also include some low-MCC functions in this survey, to see whether subjects distinguish between them and the high-MCC functions. In subsequent work we are also complementing this work by using a controlled experiment involving tasks related to code comprehension, specifically understanding, fixing bugs, and adding features (Jbara and Feitelson 2013, in preparation).

Regarding external validity, we have verified that high-MCC functions exist also in other operating systems and in some specific systems from other domains, and are not unique to Linux. However, these are only preliminary results as we only examined one specific system from each domain. Also, our results are limited to systems coded in C, and do not necessarily generalize to systems written in an object-oriented style.

9.4 Future Work

One avenue for additional work is to assess the prevalence of high-MCC functions. It is plausible that an operating system kernel is more complex than most applications, due to the need to handle low-level operations. Although our results have shown that such functions also exist in other domains it would be interesting to repeat this study for more systems in these domains and even move to new domains.

Another important direction of additional research is empirical work on comprehension and how it correlates with MCC. This is especially needed in order to justify or refute suggested modifications to the metric, and indeed alternative metrics and considerations, and improve the ability to identify complex code. For example, our perceived complexity survey identified formatting and backwards gotos as factors that should most probably be taken into account. An interesting challenge is to try and see whether the functions with spaghetti gotos could have been written concisely in a more structured manner.

Regarding the correlation between perceived complexity and regularity as reflected by the Lempel-Ziv algorithm we think that retaining formatting attributes (such as indentation and linebreaks) besides the control structure is a reasonable direction as these attributes may affect regularity and perceived complexity.

Finally, in the context of studying Linux, the main drawback of our work is its focus on a purely syntactic complexity measure. It would be interesting to follow this up with semantic analysis, for example what happens to the functionality of high-MCC functions that seem to disappear into thin air. Thus this study may be useful in pointing out instances of interesting development activity in Linux.

References

- Adams B, De Meuter W, Tromp H, Hassan AE (2009) Can we refactor conditional compilation into aspects? In: 8th Intl. conf. aspect-oriented softw. dev., pp 243–254. doi:[10.1145/1509239.1509274](https://doi.org/10.1145/1509239.1509274)
- Baggen R, Correia JP, Schill K, Visser J (2012) Standardized code quality benchmarking for improving software maintainability. *Software Quality J* 20(2):287–307. doi:[10.1007/s11219-011-9144-9](https://doi.org/10.1007/s11219-011-9144-9)
- Ball T, Larus JR (2000) Using paths to measure, explain, and enhance program behavior. *Computer* 33(7):57–65. doi:[10.1109/2.869371](https://doi.org/10.1109/2.869371)
- Bame P (2011) pmccabe. <http://parisc-linux.org/~bame/pmccabe/overview.html>. Accessed 18 Sept 2011
- Binkley AB, Schach SR (1998) Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In: 20th Intl. conf. softw. eng., pp 452–455. doi:[10.1109/ICSE.1998.671604](https://doi.org/10.1109/ICSE.1998.671604)
- Capiluppi A, Izquierdo-Cortázar D (2013) Effort estimation of FLOSS projects: a study of the Linux kernel. *Empir Softw Eng* 18(1):60–88. doi:[10.1007/s10664-011-9191-7](https://doi.org/10.1007/s10664-011-9191-7)
- Curtis B, Sappidi J, Subramanyam J (2011) An evaluation of the internal quality of business applications: does size matter? In: 33rd Intl. conf. softw. eng., pp 711–715. doi:[10.1145/1985793.1985893](https://doi.org/10.1145/1985793.1985893)
- Curtis B, Sheppard SB, Milliman P (1979) Third time charm: stronger prediction of programmer performance by software complexity metrics. In: 4th Intl. conf. softw. eng., pp 356–360

- Denaro G, Pezzè M (2002) An empirical evaluation of fault-proneness models. In: 24th Intl. conf. softw. eng., pp 241–251. doi:[10.1145/581339.581371](https://doi.org/10.1145/581339.581371)
- Dijkstra EW (1968) GoTo statement considered harmful. *Commun ACM* 11(3):147–148. doi:[10.1145/362929.362947](https://doi.org/10.1145/362929.362947)
- Etsion Y, Tsafirir D, Feitelson DG (2006) Process prioritization using output production: scheduling for multimedia. *ACM Trans Multimed Comput Commun Appl* 2(4):318–342. doi:[10.1145/1201730.1201734](https://doi.org/10.1145/1201730.1201734)
- Foreman J, Gross J, Rosenstein R, Fisher D, Brune K (1997) C4 software technology reference guide: a prototype (CMU/SEI-97-HB-001). Retrieved from the Software Engineering Institute, Carnegie Mellon University. <http://www.sei.cmu.edu/library/abstracts/reports/97hb001.cfm>. Accessed 10 Aug 2013
- Harrison W, Magel K, Kluczny R, DeKock A (1982) Applying software complexity metrics to program maintenance. *Computer* 15(9):65–79. doi:[10.1109/MC.1982.1654138](https://doi.org/10.1109/MC.1982.1654138)
- Heitlager I, Kuipers T, Visser J (2007) A practical model for measuring maintainability. In: 6th Intl. conf. quality inf. & comm. tech., pp 30–39. doi:[10.1109/QUATIC.2007.8](https://doi.org/10.1109/QUATIC.2007.8)
- Herraiz I, Hassan AE (2011) Beyond lines of code: do we need more complexity metrics? In: Oram A, Wilson G (eds) Making software: what really works, and why we believe it. O'Reilly Media Inc., pp 125–141
- Hindle A, Godfrey MW, Holt RC (2008) Reading beside the lines: indentation as a proxy for complexity metrics. In: 16th IEEE Intl. conf. program comprehension, pp 133–142. doi:[10.1109/ICPC.2008.13](https://doi.org/10.1109/ICPC.2008.13)
- Israeli A, Feitelson DG (2010) The Linux kernel as a case study in software evolution. *J Syst Softw* 83(3):485–501. doi:[10.1016/j.jss.2009.09.042](https://doi.org/10.1016/j.jss.2009.09.042)
- Jbara A, Feitelson DG (2013) Characterization and assessment of the Linux configuration complexity. In: 13th IEEE Intl. working conf source code analysis & manipulation
- Jbara A, Matan A, Feitelson DG (2012) High-MCC functions in the Linux kernel. In: 20th IEEE Intl. conf. program comprehension, pp 83–92. doi:[10.1109/ICPC.2012.6240512](https://doi.org/10.1109/ICPC.2012.6240512)
- Jones C (1994) Software metrics: good, bad, and missing. *Computer* 27(9):98–100. doi:[10.1109/2.312055](https://doi.org/10.1109/2.312055)
- Koziulek H, Schlich B, Bilich C (2010) A large-scale industrial case study on architecture-based software reliability analysis. In: 21st Intl. symp. software reliability eng., pp 279–288. doi:[10.1109/ISSRE.2010.15](https://doi.org/10.1109/ISSRE.2010.15)
- Lanning DL, Khoshgoftaar TM (1994) Modeling the relationship between source code complexity and maintenance difficulty. *Computer* 27(9):35–40. doi:[10.1109/2.312036](https://doi.org/10.1109/2.312036)
- Lehman MM, Ramil JF (2003) Software evolution—background, theory, practice. *Inf Process Lett* 88(1–2):33–44. doi:[10.1016/S0020-0190\(03\)00382-X](https://doi.org/10.1016/S0020-0190(03)00382-X)
- Liebig J, Apel S, Lengauer C, Kästner C, Schulze M (2010) An analysis of the variability in forty preprocessor-based software product lines. In: 32nd Intl. conf. softw. eng., vol 1, pp 105–114. doi:[10.1145/1806799.1806819](https://doi.org/10.1145/1806799.1806819)
- McCabe T (1976) A complexity measure. *IEEE Trans Softw Eng* 2(4):308–320. doi:[10.1109/TSE.1976.233837](https://doi.org/10.1109/TSE.1976.233837)
- McCabe Software (2009) Metrics & thresholds in McCabe IQ. URL: www.mccabe.com/pdf/McCabe%20IQ%20Metrics.pdf. Visited 23 Dec 2009
- Mens T, Fernández-Ramil J, Degrandart S (2008) The evolution of Eclipse. In: Intl. conf. softw. maintenance, pp 386–395. doi:[10.1109/ICSM.2008.4658087](https://doi.org/10.1109/ICSM.2008.4658087)
- MSDN Visual Studio Team System 2008 Development Developer Center (2008) Avoid excessive complexity. URL: msdn.microsoft.com/en-us/library/ms182212.aspx. Visited 23 Dec 2009
- Myers GJ (1977) An extension to the cyclomatic measure of program complexity. *SIGPLAN Not* 12(10):61–64. doi:[10.1145/954627.954633](https://doi.org/10.1145/954627.954633)
- Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: 28th Intl. conf. softw. eng., pp 452–461. doi:[10.1145/1134285.1134349](https://doi.org/10.1145/1134285.1134349)
- Ohlsson N, Alberg H (1996) Predicting fault-prone software modules in telephone switches. *IEEE Trans Softw Eng* 22(12):886–894. doi:[10.1109/32.553637](https://doi.org/10.1109/32.553637)
- Olague HM, Etkorn LH, Gholston S, Quattlebaum S (2007) Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Trans Softw Eng* 33(6):402–419. doi:[10.1109/TSE.2007.1015](https://doi.org/10.1109/TSE.2007.1015)
- Oman P, Hagemeister J (1994) Construction and testing of polynomials predicting software maintainability. *J Syst Softw* 24(3):251–266. doi:[10.1016/0164-1212\(94\)90067-1](https://doi.org/10.1016/0164-1212(94)90067-1)
- Sauer F (2005) Eclipse metrics plugin 1.3.6. URL metrics.sourceforge.net/. Visited 23 Dec 2009

- Schneidewind N, Hinchey M (2009) A complexity reliability model. In: 20th Intl. symp. software reliability eng., pp 1–10. doi:[10.1109/ISSRE.2009.10](https://doi.org/10.1109/ISSRE.2009.10)
- Shepperd M (1988) A critique of cyclomatic complexity as a software metric. *Softw Eng J* 3(2):30–36. doi:[10.1049/sej.1988.0003](https://doi.org/10.1049/sej.1988.0003)
- Shepperd M, Ince DC (1994) A critique of three metrics. *J Syst Softw* 26(3):197–210. doi:[10.1016/0164-1212\(94\)90011-6](https://doi.org/10.1016/0164-1212(94)90011-6)
- Soetens QD, Demeyer S (2010) Studying the effect of refactorings: a complexity metrics perspective. In: 7th Intl. conf. quality inf. & comm. tech., pp 313–318. doi:[10.1109/QUATIC.2010.58](https://doi.org/10.1109/QUATIC.2010.58)
- Soloway E, Ehrlich K (1984) Empirical studies of programming knowledge. *IEEE Trans Softw Eng* SE-10(5):595–609. doi:[10.1109/TSE.1984.5010283](https://doi.org/10.1109/TSE.1984.5010283)
- Stamelos I, Angelis L, Oikonomou A, Bleris GL (2002) Code quality analysis in open source software development. *Inf Syst J* 12(1):43–60. doi:[10.1046/j.1365-2575.2002.00117.x](https://doi.org/10.1046/j.1365-2575.2002.00117.x)
- Stark G, Durst RC, Vowell CW (1994) Using metrics in management decision making. *Computer* 27(9):42–48. doi:[10.1109/2.312037](https://doi.org/10.1109/2.312037)
- Vasa R, Lumpe M, Branch P, Nierstrasz O (2009) Comparative analysis of evolving software systems using the Gini coefficient. In: 25th Intl. conf. softw. maintenance, pp 179–188. doi:[10.1109/ICSM.2009.5306322](https://doi.org/10.1109/ICSM.2009.5306322)
- VerifySoft Technology (2005) McCabe metrics. http://www.verifysoft.com/en_mccabe_metrics.html. Accessed 23 Dec 2009
- Weyuker EJ (1988) Evaluating software complexity measures. *IEEE Trans Softw Eng* 14(9):1357–1365. doi:[10.1109/32.6178](https://doi.org/10.1109/32.6178)
- Ziv J, Lempel A (1978) Compression of individual sequences via variable-rate coding. *IEEE Trans Inf Theory* IT-24(5):530–536. doi:[10.1109/TIT.1978.1055934](https://doi.org/10.1109/TIT.1978.1055934)



Ahmad Jbara is a PhD student at the School of Computer Science and Engineering of the Hebrew University. He received his Master's degree in computer science from Bar-Ilan University in 2007. His research focuses on program comprehension and code complexity metrics. Ahmad is also a lecturer at the computer science department of the Netanya Academic College.



Adam Matan is a software developer at the Creative Innovation Center of Yahoo! Tel Aviv. His Alma Mater is the Hebrew University of Jerusalem, where he studied computer science and humanities. His current interests include search algorithms, domain specific languages and Scala.



Dror G. Feitelson is an associate professor of computer science at the Hebrew University of Jerusalem, Israel. His research interests include software evolution and program comprehension, especially what makes software hard to understand. In addition he has worked on parallel job scheduling and experimental performance evaluation, and maintains the parallel workloads archive.

Chapter 3

JCSD: Visual Support for Understanding Code Control Structure

Ahmad Jbara, Dror G. Feitelson.

Status:

Published.

Full citation:

Ahmad Jbara and Dror G. Feitelson. JCSD: Visual support for understanding code control structure. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, pages 300–303, New York, NY, USA, 2014. ACM

JCSD: Visual Support for Understanding Code Control Structure

Ahmad Jbara^{1,2} Dror G. Feitelson²

¹School of Mathematics and Computer Science
Netanya Academic College, 42100 Netanya, Israel

²School of Computer Science and Engineering
The Hebrew University of Jerusalem, 91904 Jerusalem, Israel

ABSTRACT

Program comprehension is a vital mental process in any maintenance activity. It becomes decisive as functions get larger. Such functions are burdened with very many programming constructs as lines of code (LOC) strongly correlate with the McCabe's cyclomatic complexity (MCC). This makes it hard to capture the whole code of such functions and as a result hinders grasping their structural properties that might be essential for maintenance.

Program visualization is known as a key solution that assists in comprehending complex systems. As a matter of fact we have shown, in a recent work, that control structure diagrams (CSD) could be useful to better understand and discover structural properties of such functions. For example, we found that the code regularity property, and even cloning, can be easily identified by CSDs.

This paper presents JCSD, which is an Eclipse plug-in that implements CSD diagrams for Java methods. In particular it visualizes the control structure and nesting of a Java method, and by this it easily conveys structural characteristics of the code to the programmer and helps him to better understand and refactor.

Categories and Subject Descriptors

D.2.6 [Software Engineering]: Programming Environments—*Graphical environments*

General Terms

Design, Human Factors

Keywords

Visualization, code regularity, code complexity, MCC, LOC

1. INTRODUCTION

Program visualization (PV) is one aspect of Software visualization (SV). It has been defined as "the program is specified in a conventional, textual manner, and the graphics is used to illustrate some aspect of the program or its run-time execution" [10].

Software visualization has been around for a long time to facilitate both the human understanding and effective use of computer

software [12]. In particular, by providing visual representation of the code (program visualization) programmers can handle and understand complex software more easily.

The comprehension of a given program becomes crucial as programs get larger. One of the key solution to this is better visualization [7]. This insight is consistent with our results about the effectiveness of using visualization to explore interesting properties of very long functions mainly in the Linux kernel and in many other systems [6, 5]. Our criteria for selecting long functions was those with high (higher than twice the highest threshold ever suggested) McCabe's cyclomatic complexity (MCC) as it is well known that MCC has a strong correlation with lines of code (LOC) [4]. Therefore, functions with high-MCC values are not only very long but also burdened with very many control constructs.

This makes it hard to capture the whole code of such functions as it spans over many pages and studying their structural properties is also not easy, in particular when such properties are scattered across the long code yet are related.

To cope with this we suggested to visualize such functions focusing on their structure and therefore we presented the control structure diagram (CSD) [6, 5]. In particular it was applied on high-MCC functions taken from the Linux kernel.

We showed that the high MCC does not really reflect effective complexity as there are functions with much higher MCC values but they are still well structured and easy to understand and handle.

This diagram has proved to be very useful as we have used it to examine the structure of these long functions. We identified regular code segments within them. Regularity in code means that code segments are repeated many times in the same function usually in a consecutive manner, but there are cases where instances appear separately. Such structural property is easily identified when the functions are visualized, especially when the block sizes are reflected in the diagram as it is the case in CSD.

Moreover, we conducted an experiment to check correlation between regularity and perceived complexity. In this experiment we got better correlation when the functions were presented to the subjects using CSD rather than the code listing.

To summarize, using CSD diagrams we realized that long functions that are supposed to be hard (according to traditional metrics) are actually well structured and easy to handle.

These promising results lead to the thought of integrating CSDs in a development environment. Our efforts produced an Eclipse plug-in called JCSD which is used to create CSD diagram for Java methods.

The JCSD tool is available for downloading and more details at URL <http://www.cs.huji.ac.il/%7eahmadjbara/jcsd/jcsd.html>.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICPC'14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06...\$15.00
<http://dx.doi.org/10.1145/2597008.2597801>

2. THE JCSD

2.1 Design and Implementation

JCSD is a plug-in that has been developed for the Eclipse IDE for visualizing the control structure and nesting of Java methods. It is an implementation of our earlier study about CSD diagrams [6, 5]. In its current version the plug-in has one simple use case: *when ever a programmer clicks a method's name in the project explorer window of Eclipse it produces a control structure diagram (CSD) in a separate resizable window.* Figure 1 shows an example of a method and its CSD diagram.

The view window of the diagram is a typical Eclipse inner window. It is movable and resizable. It enables the programmer to move it and position it vertically or horizontally. Moreover, in cases the function is extremely long there is a possibility to resize the window which automatically scales the diagram while resizing.

The general process of creating the CSD diagram is composed of three basic stages. Initially, the source code of the method is pre-processed: comments are removed, braces are added as required, and line breaks are added, if needed, after opening and closing braces and semicolon.

After preprocessing, a tree is built where nodes represent constructs. Each node contains the construct type it represents and its block size.

On the basis of the tree built, the CSD is drawn and the tool listens to the *resize* event to enable future scaling of the diagram. Each time the *resize* event is raised the diagram is redrawn on the basis of the same tree.

2.2 Control Structure Diagram - CSD

As we stated earlier, CSD is a visualization of the control structure and nesting of a Java method. Three design criteria were considered when we designed this diagram:

- Capturing the whole code at once as much as possible.
- Reflecting the size of the different blocks in the code.
- Identifying the different types of control flow constructs.

To meet the first criterion we visualize the structure only by considering constructs and nesting while discarding simple statements. For the second criterion we allocate more space on the diagram for larger blocks. In particular we measure the size of each block and scale its representation in the diagram accordingly. For the last criterion we provide different geometric shapes and colors for different types of constructs. For example, the *if* statement is represented by a yellow trapezoid, and the wider the trapezoid the larger the block it controls in the code. Figure 1 shows an example of a Java method along with its CSD produced by the JCSD tool.

The CSD diagram is structured as follows. At the top appears the legend that maps between the different constructs and their geometric shapes. The whole function is denoted by a bar, which contains its name, underneath the legend. The content of the function is represented in the diagram from left to right while nesting is reflected by deeper levels. Moreover, empty space between the geometric shapes represents uncontrolled lines of code between controlled blocks in the code.

For example, Figure 1 visualizes a method called *nextGen*. This method receives a board representing a generation of *Conway's Game of Life* and it creates the next generation. This method, at its top level, is composed of three blocks. The first block encompasses the uncontrolled lines 3 and 4. The second block starts at line 6 and ends at 11, while the third one spans the lines 13 to 29.

These three blocks are represented in the CSD diagram at the first level right underneath the function bar. The first block is represented by a proportional empty space at the left. Next to it is the second block which is represented by a small ellipse, and the third block is represented by a larger ellipse as it contains more lines in the original code than the previous block.

Levels in the diagram reflect nesting in the code. For example, the leftmost yellow trapezoid in level 3 in the diagram represents the *if* statement in line 6 in the listing which is nested in two *for* loops.

2.3 Analysis of Example Usage

To illustrate our tool we used the *jEdit* open source project which is a programmer's text editor written in the Java language. We applied our tool on the *getSize* method taken from the *ExtendedGridLayout* class of *jEdit*.

Figure 2 shows the CSD of this method while its listing is shown at URL <http://www.cs.huji.ac.il/~7eahmadjbara/jcsd/getSize.java> as it is quite long. Moreover, Table 1 shows its common metrics values.

According to the traditional metrics used the *getSize* method is supposed to be hard for understanding. It has 348 lines of code spanning about 6 pages. This is what a programmer really sees and it is not easy to capture the whole structure of methods with such size. However, it is easy by using a CSD.

When examining its control flow complexity (measured by MCC) we see that it is on the highest threshold ever given. Moreover, when considering its lines of code metric together with its MCC we recognize that the average size of its control blocks is about 6 lines, therefore the method is not purposelessly long but very many control constructs are interwoven within its lines. This could be easily observed by the diagram without considering complexity metrics.

The nesting metrics also indicate that this method is not easy to handle. The average nesting of all its lines is 1.3 which means that on average every line is nested. However, one might think that there is one control construct that encompasses all others and this metric does not really reflect nesting. Again, the CSD helps to identify that. In our example, this is not the case.

Based on the traditional metrics and the method's listing this method is portrayed as not easy to understand. However, examining its CSD diagram reveals a different picture. The regularity property seems to be relatively dominant as there are two large code segments (framed in Figure 2) that are likely similar. Moreover, within these two framed segments there are four repeated sub trees. The first instance appears at the left of the first frame while the second instance appears at the end of the first frame. The third and fourth instances appear in the second framed segment. These two framed segments appear at lines 60 and 173 of their method's listing at the URL presented earlier.

The fact that a code segment is totally repeated helps in investing less efforts in understanding its instances and may be an indicator for the need of refactoring.

3. RELATED WORK

One of the key solutions to code comprehension is visualization, in particular when programs get larger [7]. Nassi and Shneiderman suggested a visual model by proposing a flowchart language which prevents unrestricted transfers of control and supports structured programming [11]. In their model the details of the code are reflected as the constructs and their conditions appear within the flowchart. This might make the control over large programs difficult. Trying to grasp structural properties does not necessarily need the examination of the details. Moreover, the size of the differ-

Metric name	Description	Value
LLOC	Number of logical lines of code excluding blank lines and comments.	307
PLOC	Number of physical lines of code.	348
MCC	McCabe's cyclomatic complexity.	50
Max nesting	Max nesting within function.	3
Average nesting	Average nesting of all logical lines of code.	1.3

Table 1: Typical metrics values of the *getSize* method.

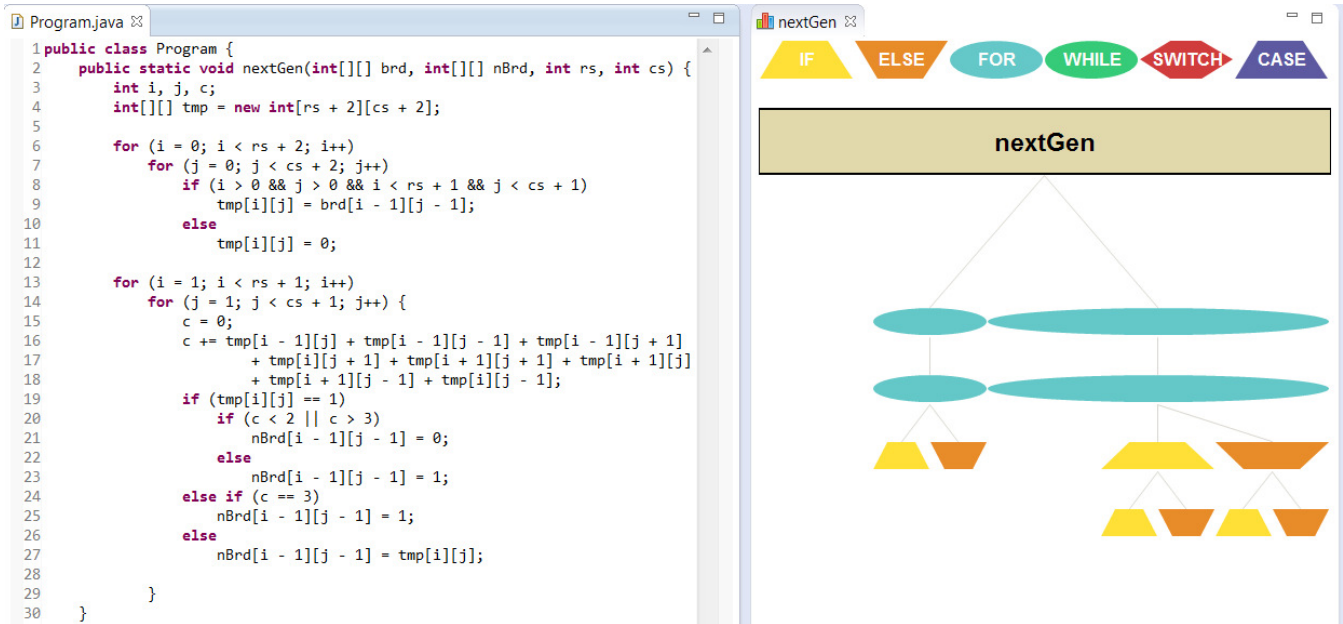


Figure 1: The control structure diagram (CSD) for the *nextGen* method of the Conway's game of life.

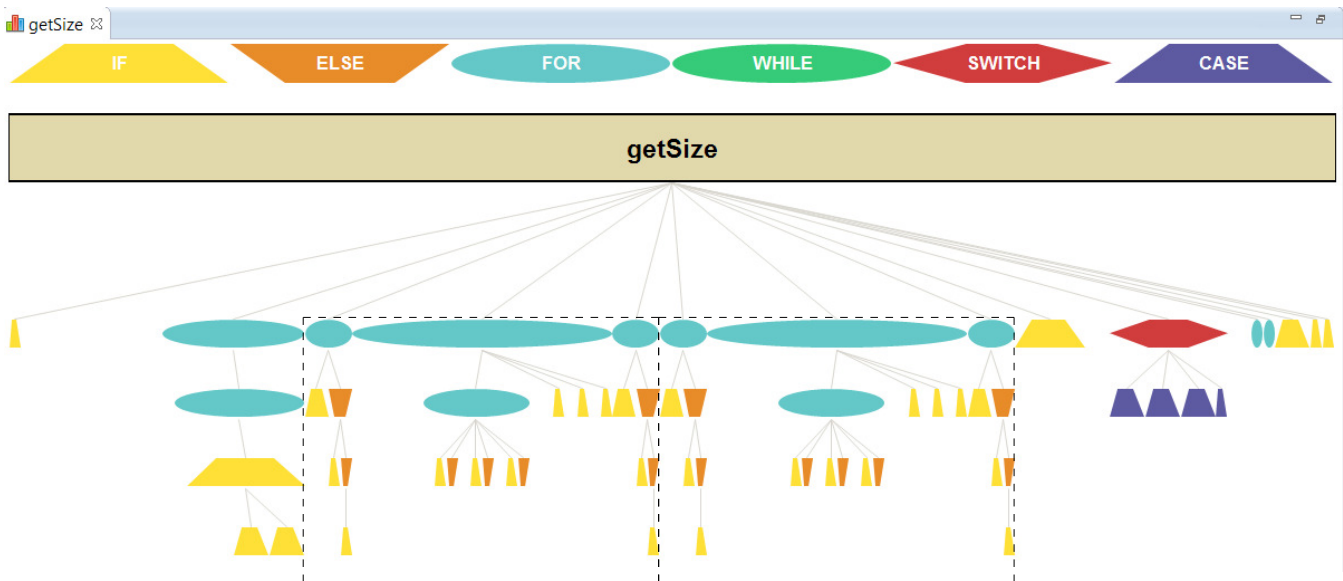


Figure 2: The control structure diagram (CSD) for the *getSize* method of the *ExtendedGridLayout* class from the *jEdit* project. Note the similarity between the structure of the framed sub trees.

ent blocks that are represented is not reflected, therefore makes it harder to capture the structure of the program.

Another work that investigates control structure is a work by Cross et al. where they present a Control Structure Diagram (CSD) to clearly depict the control constructs and the control flow at all levels of program abstraction [2]. Except of its name, it is totally different from our diagram. This diagram is superimposed upon the source code which means that the programmer is obliged to see all the code, and therefore misses the power of abstraction as he sees the details. Moreover, a diagram that reflects every line in the code would experience some difficulty in visualizing long methods which are the real challenge. In addition, their diagram is not aware to the size of the blocks that are controlled by the different constructs.

Eick et al. suggested *SeeSoft* [3, 1] where a line of code is represented by a colored line (each construct type is colored differently) with a height of one pixel and its length reflects the length of the code line. The structure of the code is reflected by preserving indentation.

Marcus et al. presented *sv3D* [9, 8]. This visualization technique is based on *SeeSoft* and added a third dimension where each code line is mapped to a rectangular cuboid.

4. FUTURE WORK

One avenue is to empirically evaluate our tool to provide an evidence of its viability in typical comprehension tasks.

Another direction is enhancing it by adding more features to help discovering more interesting structural properties. In particular, making JCSD an interactive tool. For example, it would be helpful to show the code of a specific block when the programmer's mouse hovers on its shape in the diagram, or adding a feature that enables examining simultaneously the code segments of two sub trees so the programmer could compare between them for cloning or regularity detection.

Moreover, implementing CSD for other languages and IDEs would expose it for more communities and bring more feedback. In particular, migration to other languages would be easy as the control constructs and the means for formatting and layout building are shared between different languages.

Last interesting direction is implementation of this idea as a web-based tool. The purpose is to enable programmers who search for code segments in the Internet to be able to examine its structure before copying it to his development environment.

Acknowledgments

Thanks to Elinor Alpay, Daniel Shragai, and Chen Shabo from the Computer Science Department of the Netanya Academic College for their active involvement in the implementation phase of this tool as a part of their final project of their Bachelor studies. This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13).

5. REFERENCES

[1] T. Ball and S. G. Eick, "Software visualization in the large".

- Computer* **29(4)**, pp. 33–43, Apr 1996, doi:10.1109/2.488299. URL <http://dx.doi.org/10.1109/2.488299>
- [2] I. Cross, J.H. and S. Sheppard, "The control structure diagram". In *Computers and Communications, 1988. Conference Proceedings., Seventh Annual International Phoenix Conference on*, pp. 274–278, Mar 1988, doi:10.1109/PCCC.1988.10084.
- [3] S. G. Eick, J. L. Steffen, and E. E. Sumner, Jr., "Seesoft-a tool for visualizing line oriented software statistics". *IEEE Trans. Softw. Eng.* **18(11)**, pp. 957–968, Nov 1992, doi:10.1109/32.177365. URL <http://dx.doi.org/10.1109/32.177365>
- [4] I. Herraiz and A. E. Hassan, "Beyond lines of code: Do we need more complexity metrics?" In *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson (eds.), pp. 125–141, O'Reilly Media Inc., 2011.
- [5] A. Jbara, A. Matan, and D. G. Feitelson, "High-MCC functions in the Linux kernel". In *Proceedings of the 20th IEEE International Conference on Program Comprehension, ICPC 2012.*, Jun 2012.
- [6] 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.
- [7] F. Lemieux and M. Salois, "Visualization techniques for program comprehension literature review". In *Proceedings of the 2006 Conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the Fifth SoMeT_06*, pp. 22–47, IOS Press, Amsterdam, The Netherlands, The Netherlands, 2006, ISBN 1-58603-673-4.
- [8] A. Marcus, L. Feng, and J. Maletic, "Comprehension of software analysis data using 3d visualization". In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pp. 105–114, May 2003, doi:10.1109/WPC.2003.1199194.
- [9] A. Marcus, L. Feng, and J. I. Maletic, "3d representations for software visualization". In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pp. 27–ff, ACM, New York, NY, USA, 2003, ISBN 1-58113-642-0, doi:10.1145/774833.774837. URL <http://doi.acm.org/10.1145/774833.774837>
- [10] B. A. Myers, "Taxonomies of visual programming and program visualization". *J. Vis. Lang. Comput.* **1(1)**, pp. 97–123, Mar 1990, doi:10.1016/S1045-926X(05)80036-9.
- [11] I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming". *SIGPLAN Not.* **8(8)**, pp. 12–26, Aug 1973, doi:10.1145/953349.953350. URL <http://doi.acm.org/10.1145/953349.953350>
- [12] B. A. Price, R. M. Baecker, and I. S. Small, "A principled taxonomy of software visualization". *Journal of Visual Languages & Computing* **4(3)**, pp. 211 – 266, 1993, doi:<http://dx.doi.org/10.1006/jvlc.1993.1015>.

Chapter 4

Quantification of Code Regularity Using Preprocessing and Compression

Ahmad Jbara, Dror G. Feitelson.

Status:

Manuscript.

Full citation:

Ahmad Jbara and Dror G. Feitelson. Quantification of code regularity using preprocessing and compression. Manuscript, Jan 2014

Quantification of Code Regularity Using Preprocessing and Compression

Ahmad Jbara^{1,2} Dror G. Feitelson²

¹School of Mathematics and Computer Science
Netanya Academic College, 42100 Netanya, Israel

²School of Computer Science and Engineering
The Hebrew University of Jerusalem, 91904 Jerusalem, Israel

Abstract—Complexity metrics are useful to identify potentially problematic code. But there is little agreement regarding which complexity metrics should be used and exactly what should be measured, partly because many different factors influence code complexity and comprehension. Code regularity has recently been identified as another such factor, which may compensate for other complexity factors, especially in long functions with high cyclomatic complexity. Given that regularity consists of repeated code structures, it has been suggested to measure regularity by compressing the code with standard text compression tools. But such compression can be done in many ways. We compare five widely available compression tools (*LZ77*, *gzip*, *LZMA*, *bzip2*, and *bicom*) and four levels of preprocessing the code (using the code as is, or reducing it to a skeleton of keywords and possibly some formatting). The comparison is done in terms of how the different combinations discriminate between different functions, how they correlate with human perceptions of complexity, and how well they handle relatively short functions. The results show that different combinations of compression tool and code preprocessing lead to significantly different levels of discrimination and correlation with human perceptions, and in addition some combinations are extremely bad in handling many functions and should be avoided. Our recommendation is to use *gzip* or *bicom* on a code skeleton containing keywords and formatting.

Index Terms—Software complexity metrics, Code regularity, Compression.

I. INTRODUCTION

Program comprehension is a vital preliminary step of software maintenance. The ability to comprehend a given program naturally depends on the programmer’s experience, his or her knowledge of the problem domain, and the complexity of the program itself [21]. Our focus is on the measurement of program complexity, and in particular of one specific factor that has an influence on this complexity, namely the code’s regularity.

Measuring code complexity is difficult because there are so many different factors that have an effect on developers who are trying to comprehend, correct, or modify the code. As a result there is no single metric of complexity, and in fact, any given metric will fail to match human perceptions of complexity in some cases [3], [14], [12]. Myriad metrics are therefore used to measure distinct aspects of complexity: McCabe’s cyclomatic complexity (MCC) and nesting measure control flow complexity [17], [7], Halstead’s metrics measure vocabulary and operator use [6], fan-in and fan-out measure data flow [8], and other metrics measure elements of style

and formatting [9], [5]. Regularity was recently introduced as yet another code attribute that may affect comprehension [12], [13], [10], [22]. Specifically, it was demonstrated that developers faced with long regular functions perceive them as less complex than the conventional metrics (e.g. LoC and MCC) suggest, and also perform cognitive tasks better than when faced with shorter non-regular versions of the same functions. An example of such a regular function from the Linux kernel is shown in Figure 1.

In order to further investigate the importance and effects of code regularity we need an objective metric that can quantify the degree to which given code is regular. Intuitively, regularity means that the same structures in the code repeat themselves over and over again. It has therefore been suggested that regularity may be quantified by compressing the code, and noting the compression ratio [13]. This indirect methodology is based on the mechanisms used in compression algorithms, where repeated segments are replaced with pointers to earlier instances in order to derive a shorter representation.

Still, this basic idea may be implemented in many different ways. First, there is the question of which compression scheme to use. In the following we compare common tools such as *gzip* and *bzip2* and more exotic ones like *LZMA* and *bicom*. Then there is the question of possible preprocessing of the code, to better express the regularities in the control structure. We therefore compare compression of the raw code with compression of a skeleton containing only the keywords, possibly with some of the formatting.

Our goal in the present work is to find the best combination of compression scheme and preprocessing, within the framework of using compression to quantify regularity. Naturally, this does not go to say that there are no other ways to quantify regularity. However, finding the best parameters is enough to support continued work on code regularity, and is also important for future comparisons with competing approaches.

In order to identify the best combination, we use all of the available combinations to compress 18755 functions taken from seven systems in different domains. These functions have an MCC of 20 or more to exclude short and simple functions where regularity is not expected to play a part. Our results show that different combinations indeed lead to very different results, so it is important to select the compression methodology carefully. In particular, some of the combinations

```

static int amd811le_calc_coalesce(struct net_device *dev)
{
    struct amd811le_priv *lp = netdev_priv(dev);
    struct amd811le_coalesce_conf * coal_conf = &lp->coal_conf;
    int tx_pkt_rate;
    int rx_pkt_rate;
    int tx_data_rate;
    int rx_data_rate;
    int tx_pkt_size;
    int rx_pkt_size;

    tx_pkt_rate = coal_conf->tx_packets - coal_conf->tx_prev_packets;
    coal_conf->tx_prev_packets = coal_conf->tx_packets;

    tx_data_rate = coal_conf->tx_bytes - coal_conf->tx_prev_bytes;
    coal_conf->tx_prev_bytes = coal_conf->tx_bytes;

    rx_pkt_rate = coal_conf->rx_packets - coal_conf->rx_prev_packets;
    coal_conf->rx_prev_packets = coal_conf->rx_packets;

    rx_data_rate = coal_conf->rx_bytes - coal_conf->rx_prev_bytes;
    coal_conf->rx_prev_bytes = coal_conf->rx_bytes;

    if (rx_pkt_rate < 800) {
        if (coal_conf->rx_coal_type != NO_COALESCE) {
            coal_conf->rx_timeout = 0x0;
            coal_conf->rx_event_count = 0;
            amd811le_set_coalesce(dev, RX_INTR_COAL);
            coal_conf->rx_coal_type = NO_COALESCE;
        }
    }
    else {
        rx_pkt_size = rx_data_rate / rx_pkt_rate;
        if (rx_pkt_size < 128) {
            if (coal_conf->rx_coal_type != NO_COALESCE) {
                coal_conf->rx_timeout = 0;
                coal_conf->rx_event_count = 0;
                amd811le_set_coalesce(dev, RX_INTR_COAL);
                coal_conf->rx_coal_type = NO_COALESCE;
            }
        }
        else if ((rx_pkt_size >= 128) && (rx_pkt_size < 512)) {
            if (coal_conf->rx_coal_type != LOW_COALESCE) {
                coal_conf->rx_timeout = 1;
                coal_conf->rx_event_count = 4;
                amd811le_set_coalesce(dev, RX_INTR_COAL);
                coal_conf->rx_coal_type = LOW_COALESCE;
            }
        }
        else if ((rx_pkt_size >= 512) && (rx_pkt_size < 1024)) {
            if (coal_conf->rx_coal_type != MEDIUM_COALESCE) {
                coal_conf->rx_timeout = 2;
                coal_conf->rx_event_count = 4;
                amd811le_set_coalesce(dev, RX_INTR_COAL);
                coal_conf->rx_coal_type = MEDIUM_COALESCE;
            }
        }
        else if (rx_pkt_size >= 1024) {
            if (coal_conf->rx_coal_type != HIGH_COALESCE) {
                coal_conf->rx_timeout = 2;
                coal_conf->rx_event_count = 3;
                amd811le_set_coalesce(dev, RX_INTR_COAL);
                coal_conf->rx_coal_type = HIGH_COALESCE;
            }
        }
    }
}
/* NOW FOR TX INTR COALESC */
if (tx_pkt_rate < 800) {
    if (coal_conf->tx_coal_type != NO_COALESCE) {
        coal_conf->tx_timeout = 0x0;
        coal_conf->tx_event_count = 0;
        amd811le_set_coalesce(dev, TX_INTR_COAL);
        coal_conf->tx_coal_type = NO_COALESCE;
    }
}
else {
    tx_pkt_size = tx_data_rate / tx_pkt_rate;
    if (tx_pkt_size < 128) {
        if (coal_conf->tx_coal_type != NO_COALESCE) {
            coal_conf->tx_timeout = 0;
            coal_conf->tx_event_count = 0;
            amd811le_set_coalesce(dev, TX_INTR_COAL);
            coal_conf->tx_coal_type = NO_COALESCE;
        }
    }
    else if ((tx_pkt_size >= 128) && (tx_pkt_size < 512)) {
        if (coal_conf->tx_coal_type != LOW_COALESCE) {
            coal_conf->tx_timeout = 1;
            coal_conf->tx_event_count = 2;
            amd811le_set_coalesce(dev, TX_INTR_COAL);
            coal_conf->tx_coal_type = LOW_COALESCE;
        }
    }
    else if ((tx_pkt_size >= 512) && (tx_pkt_size < 1024)) {
        if (coal_conf->tx_coal_type != MEDIUM_COALESCE) {
            coal_conf->tx_timeout = 2;
            coal_conf->tx_event_count = 5;
            amd811le_set_coalesce(dev, TX_INTR_COAL);
            coal_conf->tx_coal_type = MEDIUM_COALESCE;
        }
    }
    else if (tx_pkt_size >= 1024) {
        if (tx_pkt_size >= 1024) {
            if (coal_conf->tx_coal_type != HIGH_COALESCE) {
                coal_conf->tx_timeout = 4;
                coal_conf->tx_event_count = 8;
                amd811le_set_coalesce(dev, TX_INTR_COAL);
                coal_conf->tx_coal_type = HIGH_COALESCE;
            }
        }
    }
}
return 0;
}

```

Fig. 1. Example of a regular function from the Linux kernel.

fail to effectively compress thousands of functions, because they (or some of their preprocessed versions) are too short. As being able to handle functions of modest length is important, these combinations should be avoided.

The remainder of this paper is structured as follows. In the next section we motivate our work and present its research questions. Our methodological approach, including a description of the compression schemes and preprocessing levels, is presented in section III. We present the results and analyze them in section IV, also showing the correlation of regularity with perceived complexity and documentation of the code. Finally, we discuss the results and conclude in section VI.

II. MOTIVATION AND RESEARCH QUESTIONS

In view of the large number of metrics that have been defined for measuring code complexity, it is now accepted that there is no one metric or factor that fully reflects the complexity of source code [4], [18].

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 [13], [10]. Specifically, we conducted several experiments where developers with different levels of experience were required to understand functions and to perform maintenance tasks on functions, where different subjects were actually working on different versions of the same function. Thus we could evaluate the dependency between performance and the style in which the function was coded. Additional experiments required subjects to evaluate and grade a set of functions. The produced rankings provide us with “ground truth” regarding how human developers perceive code complexity. Using this information we can now ensure that our metrics reflect human perception, a quality that is missing in many metrics that were proposed on theoretical grounds.

The preliminary operational definition of regularity we used in that study was based on compression. We applied the *gzip* tool to compress 30 functions and used the compression ratio as a metric for regularity. This was actually done twice: first with the full function code and then using only the control structure while removing formatting, layout, and expressions.

Comparing the compression ratios with the human grading, we found a weak correlation between the grades and the compression ratios achieved on the whole function. We found a moderate correlation between the grades and the compression ratios of the control structure, and even better correlation when using the grades given based on the visual representation of the functions.

These results show that the methodology of calculating the compression has an effect on the results. Consequently, a systematic investigation of the methodology is needed.

Our ultimate goal is to define an objective metric for code regularity that reflects perceived complexity. Based on the framework of using compression ratios to quantify regularity, this may be itemized into the following research questions:

- 1) Does it matter what compression scheme is used?

- 2) What elements of the source code should be compressed?
- 3) What is the best combination of compression scheme and code preprocessing level that would reliably reflect code regularity?

To answer these questions, we need a way to evaluate the available compression schemes and combinations. We use the following three criteria:

- 1) Good discrimination. We expect functions with different levels of regularity to exhibit different compression ratios. A good compression algorithm that compresses all the functions to the same degree would be useless for us, even if the compression ratios are all very high. To check this we use thousands of functions from multiple sources, and look at the distributions of compression ratios produced by the different compression-preprocessing combinations.
- 2) The compression ratio should negatively correlate with perceived complexity: a higher compression ratio means higher regularity which should yield better comprehension. To verify this we use the same 30 functions we used in the previous work [13], and check the correlation of complexity scores we have with the compression ratios achieved by the different compression-preprocessing combinations.
- 3) Success on as many functions as possible. Some compression schemes fail to compress some functions, especially when only a minimal skeleton is used, probably because they are too short. We obviously prefer metrics that can work on any function.

III. METHODOLOGICAL APPROACH

A. Compression Schemes

As explained above our operational quantification of regularity is based on compression. However, there are many compressing schemes available. The compression schemes that we examine in this work are three that are based on the Lempel-Ziv algorithm: *LZ77*, *gzip*, and *LZMA*. Furthermore, we also examine *bzip2* and *bicom*. Table I summarizes these schemes and indicates the versions used.

Text compression usually works on complete files: an input file is compressed to create an output file. To work on functions, we create temporary files that include only the function of interest.

The most basic compression scheme we use is the original Lempel-Ziv algorithm *LZ77* [25]. This is a dictionary-based compression scheme, where repeated occurrences of a string are replaced with a pointer to the original occurrence. The key point is that the pointer consumes less space than the string itself assuming the matched string is long enough. Literals that are not matched are output verbatim. The dictionary need not be stored, as it can be reconstructed during the decompression.

A very popular version of the Lempel-Ziv algorithm is implemented in the *gzip* tool, which is part of the common GNU software distribution. It combines *LZ77* with Huffman

TABLE I
COMPRESSION SCHEMES USED IN THIS STUDY.

<i>Tool</i>	<i>Version</i>	<i>Description</i>
<i>LZ77</i>	N/A	The basic Lempel-Ziv dictionary-based compression algorithm as implemented by Marcus Geelnard.
<i>gzip</i>	1.4	a variant of the Lempel-Ziv algorithm as included in the GNU project.
<i>LZMA</i>	4.32.0.beta3	Lempel-Ziv Markov-chain Algorithm, another improved version of the Lempel-Ziv algorithm.
<i>bzip2</i>	1.0.5	Compression algorithm using the Burrows-Wheeler block sorting transformation and Huffman coding.
<i>bicom</i>	1.01	Bijjective compression based on prediction by partial matching.

coding. The output file format includes some static overhead (magic number, version number, timestamp, original file name, and CRC check) so in some cases, especially with small input files, the output may be larger than the input.

Another compression scheme based on the Lempel-Ziv algorithm that we use is *LZMA*. This is based on *LZ77* followed by a range encoder. The dictionary size is huge relative to previous implementations, with special support for repeatedly used match distances. The encoding is done using context-based prediction.

Another compression scheme we use is *bzip2*. This uses, at its core, the *Burrows-Wheeler* block sorting transformation, which treats blocks of input to create sequences of repetitions of the same symbol. This is then put through run-length encoding and Huffman coding. Similar to the *gzip* tool, *bzip2* always performs the compression even if the compressed file is larger than the input.

The last compression scheme we use is *bicom*. This is a bijjective compressor from the PPM family. Bijjective means that it can always operate both ways: any file can be both compressed and decompressed. In other words, it does not produce any specified file format. PPM means prediction by partial matching. This is an adaptive statistical compression scheme, where the last n symbols are used to predict what will come next. Arithmetic coding is used to represent the output. This compressor is efficient even for very short input sequences. It was developed for a Windows platform, but we easily compiled and used it on a Linux system.

B. Code Preprocessing Levels

Regularity in code may occur in different forms, such as repeated block structures, formatting, identifier names, and operators usage. We believe that regularity in structure, which is dominated by the control-flow constructs, has a large effect on the overall understanding of the code. When using compression to quantify regularity, the question is then what parts of the code should be compressed to best reflect regularity and provide a good correlation with humans' opinions.

We define four levels of code preprocessing. The first level is the *raw code*, where we take the source code as is (including

TABLE II
KEYWORDS IN THE C LANGUAGE AND THEIR LETTER-CODE MAPPINGS.

Keyword	Mapping
if	A
else	B
while	C
for	D
switch	E
case	F
do	G
?	H

TABLE III
THE SYSTEMS FROM WHICH FUNCTIONS WERE TAKEN.

Name	Version	Domain	# functions
Windows	WRK-v1.2	Op. syst.	420
FreeBSD	9 (stable)	Op. syst.	1413
OpenSolaris	8	Op. syst.	864
Linux	2.6.37.5	Op. syst.	2819
Firefox	9 (stable)	Browser	681
GCC	4.8.0	Compiler	1391
OpenSSL	1.0.0k	Library	194

comments and blank lines) and compress it. The other extreme is the *unformatted control flow skeleton*, where we remove all expressions, layout, and comments. Thus we are left with just the sequence of control flow constructs (keywords) and braces (to preserve the nesting), with no linebreaks. In between are two levels where we retain the formatting (linebreaks and indentation), in order to better reflect the block structure. The difference between them is that one contains only the *formatting*, while the other also indicates the existence of individual *statements* (by retaining each statement’s semicolon).

A potential problem with preserving keywords from a given programming languages is that keywords have different lengths, and there may be common substrings that cause keywords to overlap. These characteristics may be expected to affect the compression without reflecting any regularity. For example, multiple repetitions of `switch` may be compressed more than a similar sequence of the shorter `if`. To prevent such bias in the compression process we replace all the occurrences of each keyword with a single letter. For example, the keyword `if` is replaced by the letter code A. Table II shows the mapping between keywords and letter codes. (It should be noted that such a replacement was not used in our previous study [13].) The results of applying the different transformations are exemplified in Figure 2.

C. Data Collection

We have 5 compression schemes combined with 4 code preprocessing levels yielding 20 different combinations. We examine these combinations on 18755 C functions from different systems taken from different domains. The different systems, their domains, and the number of functions extracted (filtered) from each system are summarized in Table III.

Initially, our scripts extracted all functions of all systems. However, there is an intrinsic problem in C source code that is caused by the C preprocessor (CPP) conditional compilation directives. This interweaving causes problems in particular due

<p>1 Raw</p> <pre>is_prime(int n) { int i, flag=0; for (i=2; i<=n/2; ++i) { if (n%i==0) { flag=1; break; } } if (flag==0) printf("%d is prime",n); else printf("%d not prime",n); }</pre>	<p>2 Skeleton</p> <pre>{ ; D; ; { A { ; } } A ; B ; }</pre>
<p>3 Format</p> <pre>{ D { A { } } A B }</pre>	<p>4 Keywords</p> <pre>{D{A{}}AB}</pre>

Fig. 2. Example of the four levels of preprocessing the code.

to unbalanced braces. To avoid this we dropped each source file that has such problems.

Functions that passed the first step were filtered by their cyclomatic complexity value. We took functions with MCC 20 or higher to ensure a minimum size of the function’s structure, as regularity is especially meaningful for long functions and compression may fail on very small functions. We use the *pmccabe* [1] tool to calculate the MCC values of the different functions.

After these two filtering steps we had 18755 different functions. However, for many of these functions some of the different compression schemes yielded negative reduction percentages. Removing all these problematic cases led to a reduced set of 7744 functions.

Looking at the 11011 “bad” functions we found that almost all of them (about 95%) have MCC lower than 40, which means that they are relatively small functions. As compression schemes perform better on large inputs, this might explain the negative values they received. To support this conjecture we looked at all combinations to see where the negative values come from, and found them all in the most extreme preprocessing level (keywords and braces only) compressed by the *bzip2* scheme and in some cases also the *LZMA* scheme. In this level each function is in its shortest form, as we remove all its content including formatting and layout and preserve only control flow keywords which are then replaced with a single letter. Thus the functions may be reduced to a few dozen characters. The problems with *bzip2* and *LZMA* do not mean that such behavior does not occur in other schemes, but it is not as prevalent. Table VII shows the different combinations

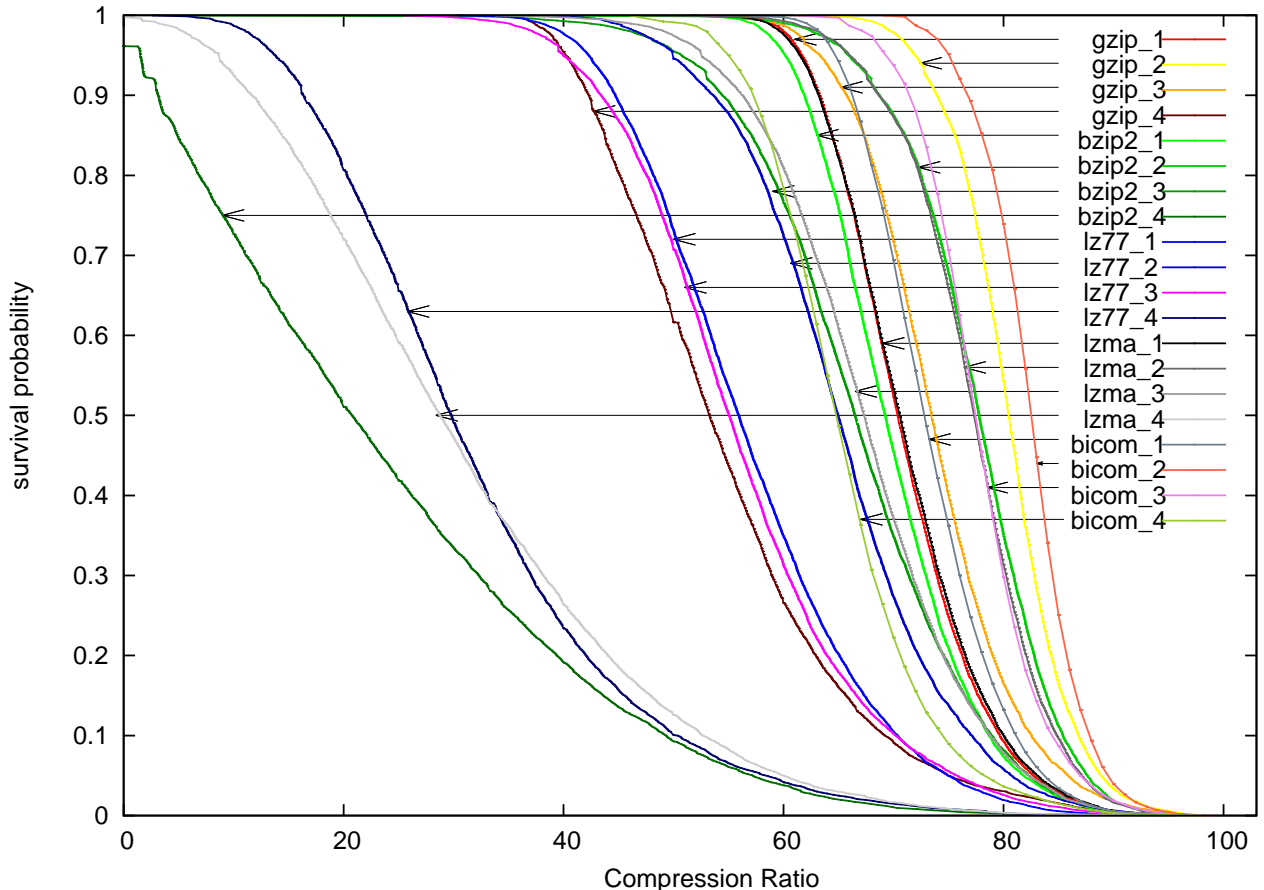


Fig. 3. Complementary cumulative distribution plots of compression ratios for 7744 functions using 20 combinations of preprocessing level and compression scheme.

and the number of functions out of the total 18755 that each combination failed to compress. Apparently the other schemes handle small inputs better. We show later that, for schemes that do not fail often, using the full 18775 functions or the reduced set of 7744 functions does not lead to significant changes in the results.

IV. RESULTS AND ANALYSIS

A. Discrimination

As described above we have 20 candidate combinations of compression scheme and preprocessing level. We applied these to 7744 functions taken from 7 different systems that belong to 4 domains. Each function was preprocessed at 4 different levels, and each of the results was then compressed by 5 different compression schemes.

Figure 3 shows the distribution of the results using the complementary cumulative distribution function (survival function) of the obtained compression ratios. This distribution function shows the probability to observe a sample that is bigger than a given value.

According to this figure both the compression scheme and the preprocessing level have a significant effect on the

achieved compression. The different schemes and the different preprocessing levels lead to different distributions. This means that selecting the best compression scheme and preprocessing level is indeed important. Arbitrarily selecting a popular compression scheme with some or no preprocessing is inappropriate.

When comparing compression schemes, the figure shows that some compression schemes consistently compress better than others, relatively independently of the preprocessing level. For example, the *gzip*, and *bicom* schemes compress very well at all preprocessing levels, so all their distributions are concentrated between moderate and high compression ratios. There is even a slight advantage for the *bicom* scheme (compresses better than *gzip*).

The *bzip2* and *LZMA* schemes exhibit similar behavior for three of the preprocessing levels. But with the keywords only preprocessing level (level 4) the distributions also include low compression ratios. Interestingly, the *LZ77* scheme has more diverse distributions: one is relatively high, two distribute over moderate up to high values, and one concentrates at rather low values.

When using the results to compare preprocessing levels, we observe that the *raw code* preprocessing level (level 1)

TABLE IV
DISCRIMINATION ABILITY OF THE DIFFERENT COMBINATIONS, AS MEASURED BY THE DIFFERENCE IN COMPRESSION RATIOS AT THE 15TH AND 85TH PERCENTILES OF THE DISTRIBUTIONS OF FIGURE 3.

Combination	7744 functions		18215 functions	
	Width	per func.	Width	per func.
<i>lz77_1</i>	20.7	0.0038	21.2	0.0016
<i>lz77_2</i>	18.0	0.0033	21.3	0.0016
<i>lz77_3</i>	20.7	0.0038	24.7	0.0019
<i>lz77_4</i>	26.9	0.0049	29.9	0.0023
<i>gzip_1</i>	13.2	0.0024	13.7	0.0010
<i>gzip_2</i>	9.7	0.0017	12.9	0.0010
<i>gzip_3</i>	13.1	0.0024	17.8	0.0013
<i>gzip_4</i>	21.8	0.0040	31.3	0.0024
<i>lzma_1</i>	13.6	0.0025		
<i>lzma_2</i>	12.1	0.0022		
<i>lzma_3</i>	17.9	0.0032		
<i>lzma_4</i>	33.3	0.0061		
<i>bzip2_1</i>	13.6	0.0025		
<i>bzip2_2</i>	12.8	0.0023		
<i>bzip2_3</i>	19.3	0.0035		
<i>bzip2_4</i>	38.6	0.0071		
<i>bicom_1</i>	11.0	0.0020	11.0	0.0009
<i>bicom_2</i>	7.0	0.0013	10.0	0.0007
<i>bicom_3</i>	9.0	0.0016	12.0	0.0009
<i>bicom_4</i>	13.0	0.0023	18.0	0.0013

compresses relatively highly across the different schemes. One explanation is that the code is the longest at this level when compared with others, and therefore has more potential for compression. Moreover, the input content at this level is real source code and English text, which are what most compression schemes are optimized to handle (as stated explicitly in the *gzip* manuals for example).

High compression ratios are obviously a desirable trait for compression schemes. But in the context of using compression to measure regularity, uniformly high compression ratios may be counterproductive. Instead, what we want is a good discrimination between input functions that have different degrees of regularity. (In the next section we add to this the requirement that this discrimination also corresponds to complexity as perceived by human developers.)

To assess the discrimination provided by the different combinations of compression and preprocessing, we focus on the central 70% of each distribution. This is the steepest part of the graph, excluding the bottom 15% which are always considerably lower and those above the 85th percentile which are always considerably higher. A large difference between the 15th and 85th percentiles of the compression ratio distribution indicate that good discrimination is possible. A small difference runs the risk that small changes in the code may lead to large and inappropriate changes in the placement in the distribution. In addition, we also divide this span of compression ratios (*width* column in Table IV) by the number of functions, to see the average difference per function (*per function* column in Table IV).

Table IV shows the results for the different combinations, both for the common set of 7744 functions and for the larger set of 18215 functions (out of a total of 18755) that are handled successfully by *LZ77*, *gzip*, and *bicom*.

According to this table combinations at level 4 (keywords and braces only, with no formatting) exhibit the best discrimination, sometimes by a wide margin. One explanation for this is that because it is the shortest representation of the code, compression ratios are necessarily lower, and every little difference in length or regularity has an effect.

Levels 1 (raw) or 3 (keywords with formatting) vie for second place. With *bicom* raw code provides a bit more discrimination, whereas with *LZMA* and *bzip2* the formatted skeleton appears a bit better. With *LZ77* and *gzip* they are essentially the same. Level 2 (including also semicolons for statements) is nearly always the least discriminative.

The results are not changed when looking at different sets of functions. Obviously, in order to achieve a fair comparison, all the combinations should be evaluated on the same set of functions. However, some of the combinations turn out to mishandle a large fraction of the original 18755 functions (this is discussed further below). The problem is that maybe limiting the evaluation to the subset of 7744 functions that all combinations can handle may distort the results regarding the better schemes, which can actually handle many more functions. We therefore also checked the distributions for a much wider set of functions, which are well handled by only three compression schemes. The results for this set are also presented in Table IV. They are consistent with those discussed above for the smaller set of functions.

B. Correlation of Regularity with Human Perception

In this section we examine the different combinations of compression schemes and preprocessing levels against other factors that are supposed to be related to regularity: perceived complexity and documentation in the source code.

1) *Regularity and Perceived Complexity*: In one of the experiments conducted in our previous work 30 diverse functions with high MCC (McCabe’s cyclomatic complexity) values were presented to 15 experienced programmers [13]. The experiment was conducted in two phases, on different days. In one phase each subject was presented with listings of the functions and in the other phase he was presented with code structure diagrams (CSD: a visual representation of the code structure [12], [11]). Which representation came first was randomized across subjects. The subjects were asked to assign a perceived complexity score to each function in each representation. The result was a moderate negative correlation between perceived complexity and regularity, where regularity was measured by compression using *gzip* of a keywords plus braces representation of the code.

We can now compute the correlations between the rankings, from our previous work [13], and all 20 combinations of compression and preprocessing, to see which combination best matches the rankings of the human programmers. The two correlation methods (Spearman and Pearson) yielded very close results but we preferred the Spearman rank correlation because the data tends to be non linear. The results are presented in Table V and Figure 4, for both modes of presenting the functions (visual and listing).

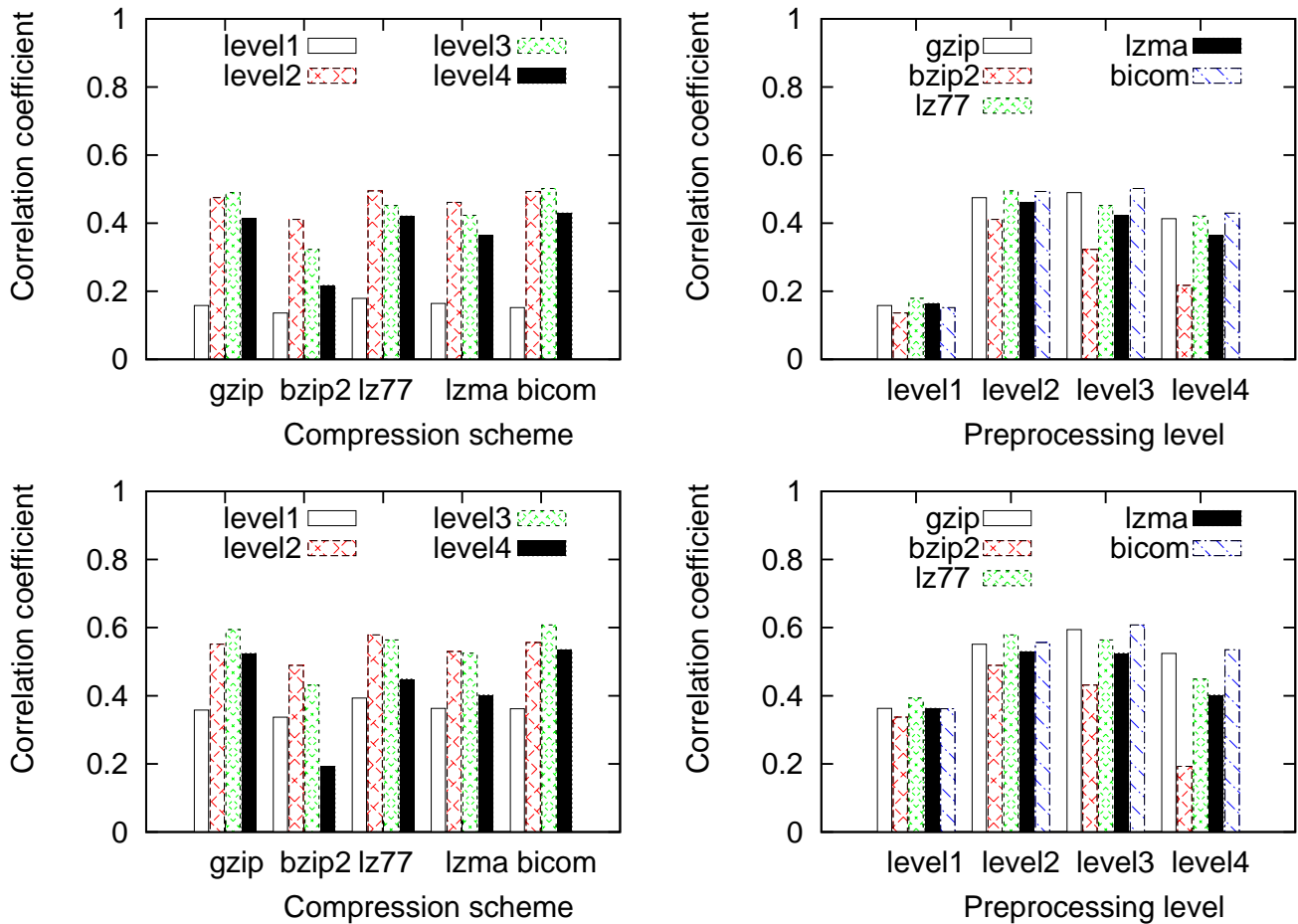


Fig. 4. Correlation between perceived complexity and compression ratios. Top row shows results when using code listings and second row when using CSD visualizations.

TABLE V
CORRELATIONS BETWEEN PERCEIVED COMPLEXITY AND COMPRESSION RATIO FOR DIFFERENT COMBINATIONS OF COMPRESSION SCHEMES AND PREPROCESSING LEVELS.

Combination	Code	CSD
lz77_1	-0.179	-0.393
lz77_2	-0.495	-0.579
lz77_3	-0.452	-0.564
lz77_4	-0.421	-0.449
gzip_1	-0.158	-0.363
gzip_2	-0.475	-0.551
gzip_3	-0.490	-0.594
gzip_4	-0.413	-0.524
lzma_1	-0.164	-0.363
lzma_2	-0.461	-0.530
lzma_3	-0.423	-0.525
lzma_4	-0.364	-0.402
bzip2_1	-0.136	-0.337
bzip2_2	-0.411	-0.489
bzip2_3	-0.323	-0.431
bzip2_4	-0.217	-0.193
bicom_1	-0.152	-0.362
bicom_2	-0.493	-0.556
bicom_3	-0.502	-0.608
bicom_4	-0.430	-0.535

According to these results, using the raw code (preprocessing level 1) has very low correlation across all schemes. The other three preprocessing levels achieve reasonable correlations for all compression schemes except *bzip2*. When using the *code listing representation*, levels 2 and 3 achieve the best correlation for all schemes and level 4 achieves slightly lower results. Similar results are achieved when using *CSD representation*. As for the highest correlation overall, it occurs at level 3 of *bicom* for both the visual mode and code listing representations. *gzip* also achieved high correlations which are not far from those of *bicom*.

2) *Regularity and Comments*: Regularity is characterized by repeated code; similar code segments may consecutively occur within a function. It seems reasonable to assume that programmers document such regular functions less than other non-regular ones. The rationale is that once the first instance of some repeated code segment is documented the programmer fairly believes that following instances of that pattern are understandable by implication, so he would provide less comments for these instances or even not provide comments at all.

TABLE VI
CORRELATION BETWEEN COMMENTS AND COMPRESSION RATIOS FOR
DIFFERENT COMBINATIONS OF COMPRESSION SCHEMES AND
PREPROCESSING LEVELS.

<i>Combination</i>	<i>Correlation coefficient</i>
<i>lz77_1</i>	-0.471
<i>lz77_2</i>	-0.298
<i>lz77_3</i>	-0.240
<i>lz77_4</i>	-0.224
<i>gzip_1</i>	-0.428
<i>gzip_2</i>	-0.253
<i>gzip_3</i>	-0.177
<i>gzip_4</i>	-0.201
<i>lzma_1</i>	-0.358
<i>lzma_2</i>	-0.209
<i>lzma_3</i>	-0.122
<i>lzma_4</i>	-0.142
<i>bzip2_1</i>	-0.298
<i>bzip2_2</i>	-0.118
<i>bzip2_3</i>	-0.034
<i>bzip2_4</i>	-0.034
<i>bicom_1</i>	-0.392
<i>bicom_2</i>	-0.284
<i>bicom_3</i>	-0.217
<i>bicom_4</i>	-0.197

Thus we conjecture that the more the function is regular the less likely it is to be documented. To examine this idea we measure the comments of each function of this study and check if there is any correlation between this measure and the regularity as quantified by the 20 different combinations of compression scheme and preprocessing.

There are many ways to measure comments: character-based length, word-based length, non-stop-word length, or just the number of comments. However, no matter which metric one chooses, it should be normalized relative to the function length (LOC). In other words, we are interested more in the density of commenting than in their absolute number. This point of view is required to reliably reflect situations where we have equal regularity measures for functions with different lengths. In this work we calculated the character-based length of all comments divided by the logical lines of code of that function.

We applied the Spearman nonparametric rank correlation coefficient between the comments ratio and the compression ratio for all 20 combinations of compression schemes and preprocessing levels of the 7744 set of functions. The results are shown in Table VI. Generally, a moderate correlation is achieved when the raw version of the functions is used. The best correlation is achieved in level 1 for the *gzip* and *LZ77* schemes, with a slight advantage of the latter one. Many other combinations also showed some correlation but it was weaker. *bzip2* showed essentially no correlation in its non-raw versions.

These results show that this direction is promising and apparently programmers indeed document regular code less than non-regular code. However, more work should be done in the direction of the best way of measuring comments and the ways developers document regular code.

C. Handling Small Functions

Generally, compression schemes are good with large inputs. However, most of the functions in any system are not con-

TABLE VII
NUMBER OF FUNCTIONS EACH COMBINATION FAILS TO COMPRESS,
PRODUCING NEGATIVE REDUCTION.

<i>Combination</i>	<i># Bad functions</i>
<i>lz77_1</i>	0
<i>lz77_2</i>	4
<i>lz77_3</i>	15
<i>lz77_4</i>	534
<i>gzip_1</i>	0
<i>gzip_2</i>	4
<i>gzip_3</i>	15
<i>gzip_4</i>	59
<i>lzma_1</i>	0
<i>lzma_2</i>	8
<i>lzma_3</i>	96
<i>lzma_4</i>	7994
<i>bzip2_1</i>	0
<i>bzip2_2</i>	45
<i>bzip2_3</i>	287
<i>bzip2_4</i>	10942
<i>bicom_1</i>	0
<i>bicom_2</i>	0
<i>bicom_3</i>	0
<i>bicom_4</i>	0

sidered large. For example, in this work we collected 18755 different functions from different systems where more than half of them have a cyclomatic complexity below 40. Many more functions have MCC below 20 and were not included in our sample to begin with. (The cyclomatic complexity is a relevant threshold criterion as we look at the control structure of each function).

Functions with relatively low MCC values lead to small input files that might cause the compression algorithms to create compressed files that are larger than the original ones. In particular, one should remember that most of the compression schemes have some headers that enlarge the output files without reflecting real compression.

The problem is critical in levels 2, 3, and especially 4, as in these levels much of the functions' contents are removed and the resulting input files are very small. This greatly reduces the effectiveness of the compression schemes in identifying and quantifying regular code as they fail to compress these files by shortening them.

We have already seen that more than half of the functions checked failed to be compressed in level 4 of *bzip2*, and more than 40% failed in level 4 of *LZMA*. It is important to mention that other schemes and levels fail also, but not as massively as *bzip2* and *LZMA*. Table VII shows the numbers of the "bad" functions under the different combinations. Note that in this work we considered only functions with MCC above 20. We expect that functions in the range between 10 and 20 would cause many more failures for the different compression schemes.

While *gzip* fails for a relatively small number of functions, *bicom* stands out for its ability to compress small files — it did not fail for a single function, regardless of preprocessing level. We believe that these differences and the inability of the current compression schemes to deal with small files indicate that there is room for considering other compression schemes,

especially ones that are good at small files and maybe even irreversible (lossy) compression schemes. This is permissible in our context because we are not concerned with storing the information, just with measuring the regularity.

At the same time, we note that the shorter the function, the smaller the scope it has for regularity. Moreover, short functions are typically considered easier to understand, so the question of regularity is less pressing for short functions.

V. RELATED WORK

To the best of our knowledge we are the first to study regularity in the context of code comprehension. In [12] we examined more than 1000 versions of the Linux kernel where we identified functions with very high cyclomatic complexity values. We found that these functions are not really as complex as their MCC complexity metric suggests. In particular, many turned out to be well structured and very regular. We then suggested the use of compression (using *gzip*) as an operational metric to enable the quantification of regularity. A survey we conducted provided empirical evidence for correlation between the measured regularity scores and perceived complexity by developers. In [13] we extended this work to encompass more systems and more domains, finding that regular functions also occur in systems and domains other than Linux.

Based on these results, we set out to verify the conjecture that regularity is one of the factors that allows developers to handle long high-MCC functions successfully. In a controlled experiment we compared the performance of subjects in terms of time and correctness when working on different implementations of the same specification, where one of the implementations adopted a regular style [10]. We found that the subjects working with the regular version achieved better results than others. The tasks that were used to assess comprehension in this experiment were *feature adding*, *bug fixing*, and *functionality description*.

Similar observations and results were reported in [22]. They introduced the idea of Control Flow Pattern (CFP) and Compressed Control Flow Pattern (CCFP). They used CCFPs to eliminate some repetitive structure from flow graphs. They concluded that methods with high cyclomatic complexity have very low entropy and are easy to understand.

Sasaki et al. also ascribed the large cyclomatic values that some modules exhibit to the presence of repeated structures such as consecutive if-else structures [20]. They claimed that it would not be so difficult to understand such source code. They proposed to preprocess the code to make complexity measurement more efficient.

Regularity has been noticed before, but not quantified. Chaudhary et al. conducted an experiment to study the effect of control and execution structures on program comprehension [2]. One result that contradicted their intuitive expectation was the positive correlation between the subjects' score and the control structure complexity. They attributed this result to the existence of syntactic and semantic regularities in the code. They claimed that these regularities reduced the efforts in the learning process and yielded a higher score.

Regularity has also been considered in other areas. Lipson has defined structural regularity as the compressibility of the description of the structure [15]. In addition to the regularity definition, a metric for quantifying the amount of regularity was suggested. It was defined by the inverse of the description length or Kolmogorov complexity.

Recently, Zhao et al. have shown that regularity leads to spontaneous attention [24]. This may be part of the explanation of why regular code is easier to understand.

There are also works that have used the term "regularity" with different meanings. For example, Lozano et al. use regularity in the context of naming conventions, complementary methods, and interface definitions [16]. Zhang suggested a revised version of Halstead's length equation. He based it on the fact that the distribution of lexical tokens in the studied systems follow Zipf's law [23]. Similar results, regarding the distribution of lexical tokens, were presented by [19].

VI. DISCUSSION AND CONCLUSIONS

We have already shown in a previous work that *regularity* is yet another factor that may have a substantial effect on code comprehension. We also suggested to measure it using compression. In this study we have performed a methodological investigation of this idea, and considered 20 different combinations of compression scheme and code preprocessing level. We used 5 compression schemes, namely *LZ77*, *LZMA*, *gzip*, *bzip2*, and *bicom*. We used 4 levels of preprocessing which are based on control-flow structure, formatting, and statement awareness. The effectiveness of the 20 combinations was evaluated by how well they discriminate between functions, how well their compression ratios correlate with perceived complexity, and how well they handle small functions.

The results show that *bzip2* and *LZMA* are problematic even with not-so-small functions, so they are less useful and should not be used. *bicom* is best on small files, but has somewhat lower discrimination than *gzip*. *gzip* is also very good, except with the most extreme preprocessing.

The *bicom* scheme achieved the highest correlations with perceived complexity so it best reflects effect on humans. *gzip*'s performance was very close to that of *bicom*. Similar results were also achieved by *LZ77*, with an advantage of being more discriminative.

As for preprocessing levels, level 1 (using the raw code) leads to very low correlations, so this should not be considered and preprocessing should definitely be used. Several interactions occur between the correlations and other attributes. The most extreme preprocessing, level 4, led to the best discrimination, but had somewhat lower correlations than levels 2 and 3. Preprocessing level 2 gives the highest correlations for *LZ77*, *LZMA*, and *bzip2*, but level 3 was better for *gzip* and *bicom*.

Our conclusion is that *gzip* or *bicom* combined with preprocessing level 3 (retain keywords, braces, and formatting, but not statements) are the best combination. *bicom* may be better at handling small functions, and has the advantage of not adding a header that distorts the compression ratio (due to its bijective nature). But *gzip* is more widely available. Luckily,

the combination we used in previous work turns out to be near optimal, and therefore the results are valid.

These results indicate that compression is a promising way to measure regularity, but it is important to choose an adequate scheme. Not all schemes correlate with perceived complexity and not all of them have the same discrimination ability. Furthermore, the whole code of the functions is not representative and a preprocessing step on the code should be performed prior to compressing.

VII. THREATS TO VALIDITY

Our work suffers from several threats to validity. We preprocessed the code in various levels by removing different things and retaining others. It might be that some of the removed stuff has an effect on regularity and we missed that. For example, Green et al. present a set of coding guidelines that are partially based on formatting. They suggest considering a program as a table by using vertical alignments, and considering the use of white space to show structure [5]. These guidelines represent factors that are part of regularity. Our work considers indentation and structure but not all these factors.

Another threat is that most compression schemes add headers to the compressed output, which distorts the compression ratio. This can be avoided by careful parsing of the output files to better reflect the true representation of the compressed data.

A third threat is that we evaluated the effectiveness of different combinations based on their correlation with a previous study in which complexity grades were given to 30 functions. Comparisons with larger sets of functions, and with multiple methods of assessing their complexity, would increase confidence in the results and allow for more general conclusions.

For future work, an interesting issue is measuring regularity of small functions. Indeed, we have shown a scheme that is capable of compressing small functions, but it has somewhat lower discrimination ability. Moreover, in this work we examined only functions with cyclomatic complexity of 20 or more, but a significant fraction of functions is below that.

Another avenue is to examine other families of compression schemes which were not examined in this study. For example, lossy compression scheme may be adequate as humans do not really read repeated code line by line and allow themselves to skip predictable parts.

Acknowledgments

this research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13).

REFERENCES

- [1] P. Bame, "pmccabe". URL <http://parisc-linux.org/~bame/pmccabe/overview.html>. (Visited 18 Sep 2011).
- [2] B. Chaudhary and H. Sahasrabudhe, "Two dimensions of program comprehension". *Intl. J. Man-Machine Studies* **18**(5), pp. 505–511, 1983.
- [3] 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.
- [4] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. Course Technology, 2nd ed., 1998.
- [5] R. Green and H. Ledgard, "Coding guidelines: Finding the art in science". *Comm. ACM* **54**(12), pp. 57–63, Dec 2011, doi:10.1145/2043174.2043191.
- [6] M. Halstead, *Elements of Software Science*. Elsevier Science Inc., 1977.
- [7] W. Harrison, K. Magel, R. Kluczny, and A. DeKock, "Applying software complexity metrics to program maintenance". *Computer* **15**(9), pp. 65–79, Sep 1982.
- [8] S. Henry and D. Kafura, "Software structure metrics based on information flow". *IEEE Trans. Softw. Eng.* **SE-7**(5), pp. 510–518, Sep 1981, doi:10.1109/TSE.1981.231113.
- [9] A. Hindle, M. W. Godfrey, and R. C. Holt, "Reading beside the lines: Indentation as a proxy for complexity metrics". In *16th IEEE Intl. Conf. Program Comprehension*, Jun 2008.
- [10] 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.
- [11] 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.
- [12] A. Jbara, A. Matan, and D. G. Feitelson, "High-MCC functions in the Linux kernel". In *Proceedings of the 20th IEEE International Conference on Program Comprehension, ICPC 2012.*, Jun 2012.
- [13] 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.
- [14] B. Katzmarski and R. Koschke, "Program complexity metrics and programmer opinions". In *20th IEEE Intl. Conf. Program Comprehension*, Jun 2012.
- [15] H. Lipson, "Principles of modularity, regularity, and hierarchy for scalable systems". *Journal of Biological Physics and Chemistry* **7**(4), pp. 125–128, 2007.
- [16] 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.
- [17] T. McCabe, "A complexity measure". *IEEE Trans. Softw. Eng.* **2**(4), pp. 308–320, Dec 1976, doi:10.1109/TSE.1976.233837.
- [18] 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.
- [19] 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.
- [20] Y. Sasaki, T. Ishihara, K. Hotta, H. Hata, Y. Higo, H. Igaki, and S. Kusumoto, "Preprocessing of metrics measurement based on simplifying program structures". In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, vol. 2, pp. 120–127, 2012, doi:10.1109/APSEC.2012.59.
- [21] S. Tilley, S. Paul, and D. Smith, "Towards a framework for program understanding". In *Program Comprehension, 1996, Proceedings., Fourth Workshop on*, pp. 19–28, Mar 1996, doi:10.1109/WPC.1996.501117.
- [22] 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.
- [23] 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.
- [24] 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.
- [25] 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.

Chapter 5

On the Effect of Code Regularity on Comprehension

Ahmad Jbara, Dror G. Feitelson.

Status:

Published.

Full citation:

Ahmad Jbara and Dror G. Feitelson. On the effect of code regularity on comprehension. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, pages 189–200, New York, NY, USA, 2014. ACM

On the Effect of Code Regularity on Comprehension

Ahmad Jbara^{1,2} Dror G. Feitelson²

¹School of Mathematics and Computer Science
Netanya Academic College, 42100 Netanya, Israel

²School 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICPC’14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06...\$15.00
<http://dx.doi.org/10.1145/2597008.2597140>

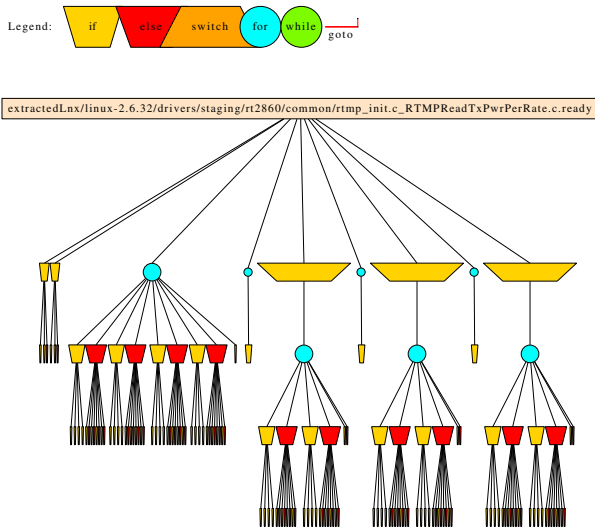


Figure 1: Code structure diagram (CSD) of a regular function from the Linux kernel.

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

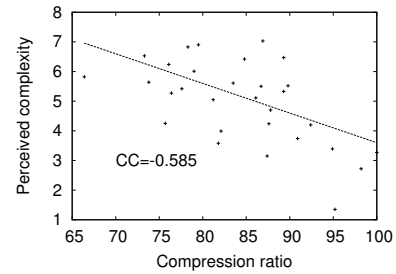


Figure 2: Correlation of perceived complexity with regularity for 30 functions from the Linux kernel, from [18].

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 pre-processing 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.

Program	Regular version			Non-regular version			Description
	LOC	MCC	Reg.	LOC	MCC	Reg.	
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 *phase1*). 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 ex-

perience, 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:

- H_{10} : different values of MCC or LOC do not impact the correctness of the solutions given.
- H_1 : 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 \pm 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.

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

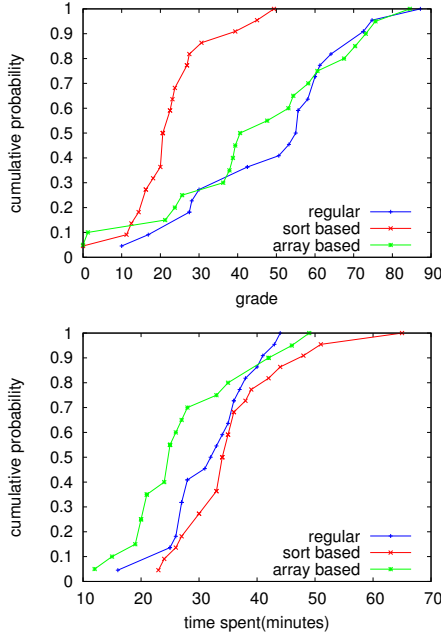


Figure 3: Distributions of average grade for all phases and total time taken for experiment 1. Cumulative probability is the probability that a specific sample is smaller than or equal to a given value.

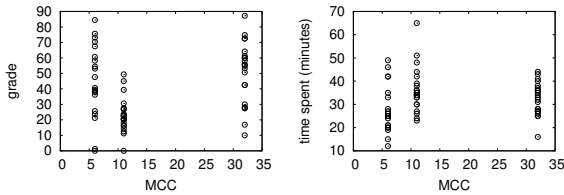


Figure 4: Distributions of grades and time vs. MCC. The MCC values 6, 11, and 32 are for the array, sort, and regular styles, respectively (per Table 1).

($\rho = .764$). 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:

- H_{20} : different values of MCC or LOC do not impact the time spent when performing a comprehension tasks on different solution styles.
- H_2 : 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

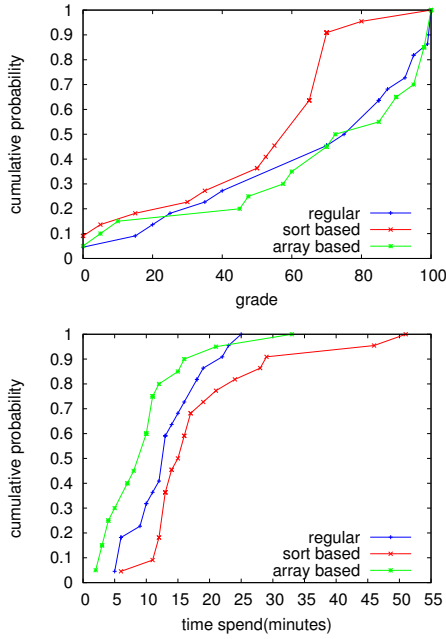


Figure 5: Phase 1 results.

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

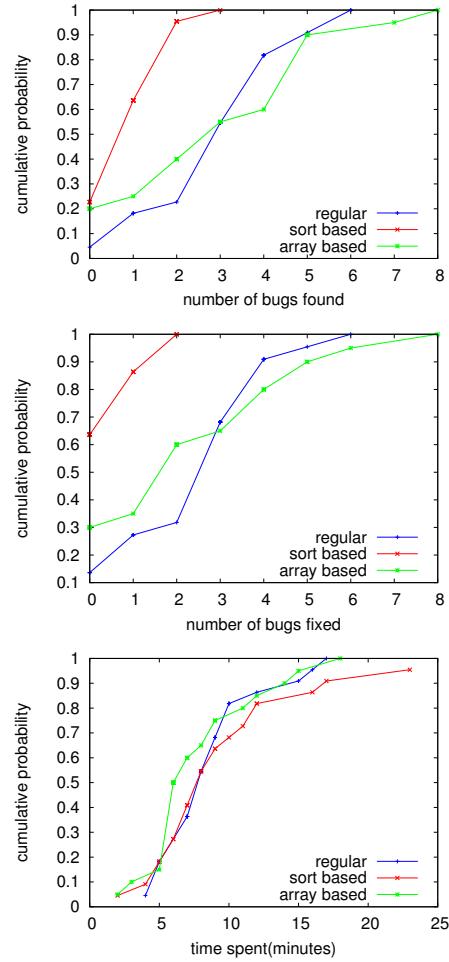


Figure 6: Phase 2 results.

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

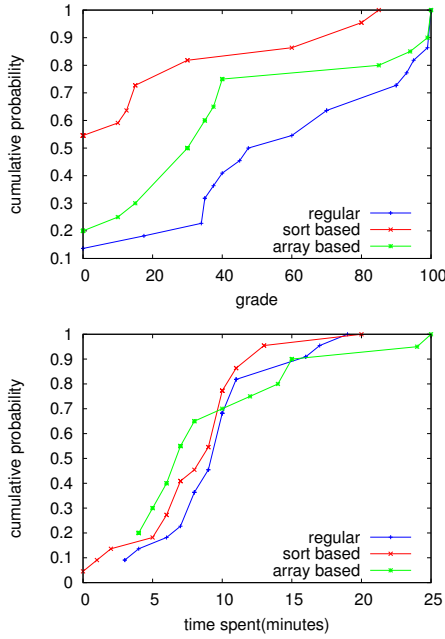


Figure 7: Phase 3 results.

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

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

order	Correctness		Time	
	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

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.

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-

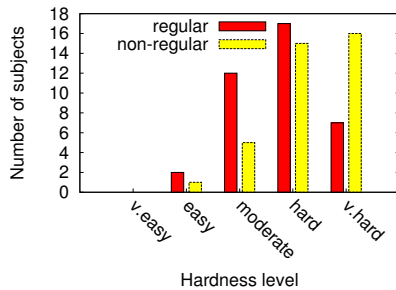


Figure 8: Distribution of perceived hardness ratings

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 et 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		Experiment 2	
Version	Time/LOC	Version	Time/LOC
Regular	0.25±0.05	Median reg	0.068±0.023
Sort	0.75±0.19	Median nonreg	0.125±0.050
Array	0.95±0.35	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, regu-

larity 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. Sahasrabudhe, "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. Hagemester, "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.

Chapter 6

How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking

Ahmad Jbara, Dror G. Feitelson.

Status:

Under review.

Full citation:

Ahmad Jbara and Dror G. Feitelson. How programmers read regular code: A controlled experiment using eye tracking. *Empirical Software Engineering*, 2015. under review

Note:

Invited extended journal version of “Ahmad Jbara and Dror G. Feitelson. How programmers read regular code: A controlled experiment using eye tracking. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*, pages 244–254, May 2015”

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 an irregular one with the same functionality, despite being 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 or cubic 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 are read more slowly and in addition they are looked at more times. Further, a few recurring 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.

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

A. Jbara
School of Mathematics and Computer Science, Netanya Academic College, 42100, Netanya, Israel.

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

1 Introduction

Although there is a general agreement on the importance of code complexity metrics, there is little agreement on specific metrics and in particular their accuracy [33]. Syntactic metrics like lines of code (LOC) and McCabe’s cyclomatic complexity (MCC) are 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. In particular, they lead to inflated measurements of well-structured long functions that are actually reasonably simple to comprehend [16].

In previous work [16, 15] we introduced *regularity* as a new factor that questions the additivity of the classical syntactic metrics. Regularity is the repetition of code segments (patterns), where instances of these patterns are usually successive. Figures 4 and 5 show examples of regular code (the repeated instances are indicated by rectangles).

Regular code is generally longer than its non-regular alternative, 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 non-regular alternative which is shorter normal code. The speculation was that regularity helps because repeated instances are easier once the initial ones are understood [15].

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 of such a 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 or a cubic function.

The consequence is that 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 [36], 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

Table 1: Attributes of the two versions of the programs used in the experiment. The Reg. column represents compression ratio.

Program	Regular version			Non-regular version			Description
	LOC	MCC	Reg.	LOC	MCC	Reg.	
Median	53	18	79.3%	34	13	60.7%	Find 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

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. 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

While a large number of metrics for measuring code complexity have been proposed, no one of them is capable of fully reflecting the complexity of source code [7, 21]. In previous work we have suggested *regularity* as an additional factor that affects code comprehension, especially in long functions, and provided experimental evidences for its significance [16, 15]. 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 performance and the style in which the function was coded.

To quantify the level of regularity of the different versions, we use an operational definition that is based on compression. We have systematically investigated different compression schemes and code preprocessing levels [14, 16], and found that different combinations yield different results. The combination that gave the best correlation with perceived complexity was to strip the code down to a skeleton of keywords and formatting, and use the *gzip* compression routine. Regularity is quantified by the compression ratio.

Regular functions by definition contain repetitive code segments that, at least to some extent, come one directly after another. This suggests that understanding one of these segments would help understanding subsequent ones. Based on this we argue 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.

The purpose of this work is twofold. First is to replicate our previous study regarding where we compared the performance of subjects when maintaining regular programs versus their non-regular counterpart. The second purpose is to study the way developers investigate regular code, and whether their efforts in repetitive segments are equal. If the efforts are not the same 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. If developers need less effort to understand repeated segments that they already encountered then our model would be a good

context-dependent weighting factor for metrics that consider all segments using the same mechanism and hence yield exaggerated measures. An example of very long functions that some metrics classify as very hard while humans classify as simple and well structured was presented in [16].

The specific research questions this paper addresses are:

- Do developers follow any pattern when they are required to comprehend regular code? In particular, are their efforts equally divided among regular segments?
- Assuming there is a pattern that governs the investment of effort, which model might fit and describe it?
- 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?
- In terms of correctness and completion time, are the results consistent with those of our previous work [15]?

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: *finding the medians of all 3×3 neighborhoods* and *finding the maximal Manhattan-radius around a point with all same value*. The programs were taken from our previous work [15], and they meet the following design 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.

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.

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 [15] 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 [34].

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 [19,5,29]. 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. 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.

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 averages of the 4 groups. According to this table we see that in terms of groups and style the averages are quite similar.

Table 2: Average grades of the participants in the different groups.

Style	All courses	Programming courses
Regular (median)	84.0±7.9	86.5±11.1
Regular (diamond)	86.6±9.0	87.0±9.8
	85.1±8.1	86.7±10.0
Non-regular (median)	82.2±11.0	83.2±9.9
Non-regular (diamond)	85.6±8.1	86.2±7.7
	84.1±9.0	84.8±8.3

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

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 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.

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 also 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 a 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 [17].

The observable variables that are measured to represent visual effort are *fixation count*, *total fixation time*, and *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 Average Pupil Dilation

It has been shown that there is a positive correlation between cognitive effort and pupil size. Hess showed that the pupil size increases with the increase of arithmetic complexity [11]. In a different domain, Just et al. found a correlation between pupil size and sentence complexity during a comprehension task [18]. Similar findings were presented by Ganhholm et al. in the context of working memory load [9].

On the basis of the above works we use pupil dilation as a measure of complexity and apply it to compare regular code with non regular code. In addition, we use pupil size to examine our claim about decreasing complexity of repeated instances of code segments.

We define the metric as the average of pupil size in all fixations in a given area of interest (AOI). We discard gazes between fixations (saccades) as pupil size

reflects complexity and complexity is experienced during processing which happens in fixations.

3.7.4 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

The main purpose of this work is exploring the way developers approach regular code rather than comparing its performance to non regular code. We have investigated this in a previous work [15]. Nevertheless, we replicate that work to confirm the results. 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 [31] 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

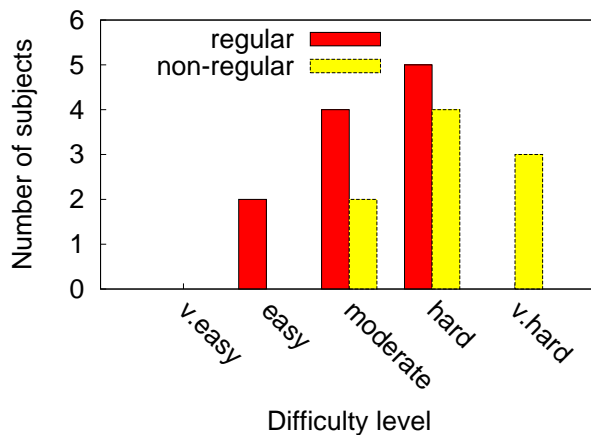


Fig. 1: Distribution of perceived difficulty ratings.

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 [15].

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 1 shows the distribution of the subjects' answers. In particular it shows that a third of the

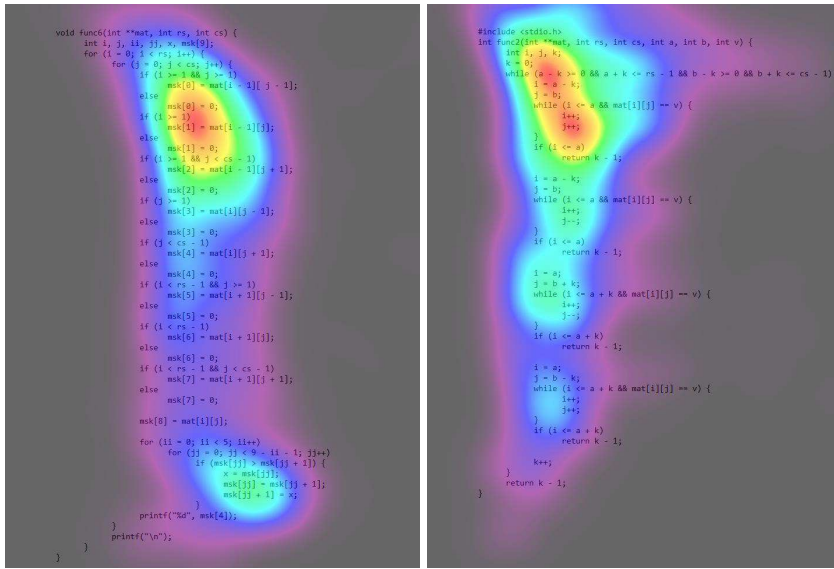


Fig. 2: Left: heat map of the regular implementation of the *median* program based on 6 subjects. Right: heat map of the regular implementation of the *diamond* program based on 4 subjects (we excluded the fifth subject due to a contradiction between his heat map and think-aloud results).

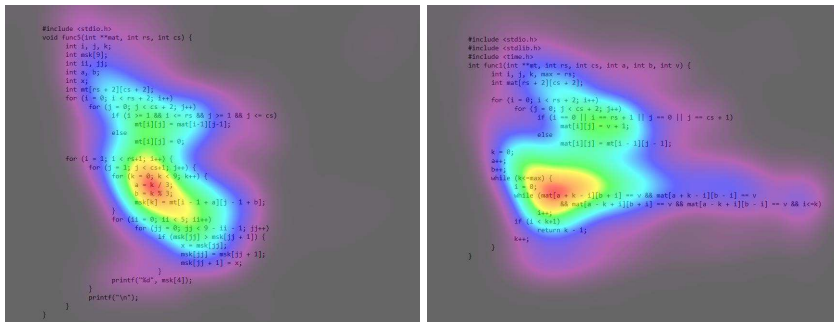


Fig. 3: Left: heat map of the non-regular implementation of the *median* program based on 3 subjects (we excluded the fourth subject as he failed the calibration process). Right: heat map of the non-regular implementation of the *diamond* program based on 5 subjects.

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 have 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 [15].

```

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];
            else
                msk[0] = 0;
            if (i >= 1)
                msk[1] = mat[i - 1][j];
            else
                msk[1] = 0;
            if (i >= 1 && j < cs - 1)
                msk[2] = mat[i - 1][j + 1];
            else
                msk[2] = 0;
            if (j >= 1)
                msk[3] = mat[i][j - 1];
            else
                msk[3] = 0;
            if (j < cs - 1)
                msk[4] = mat[i][j + 1];
            else
                msk[4] = 0;
            if (i < rs - 1 && j >= 1)
                msk[5] = mat[i + 1][j - 1];
            else
                msk[5] = 0;
            if (i < rs - 1)
                msk[6] = mat[i + 1][j];
            else
                msk[6] = 0;
            if (i < rs - 1 && j < cs - 1)
                msk[7] = mat[i + 1][j + 1];
            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. 4: The areas of interest (AOIs) of the *median* regular implementation.

```

#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 +
        i = a - k;
        j = b;
        while (i <= a && mat[i][j] == v) {
            i++;
            j++; AOI1
        }
        if (i <= a)
            return k - 1;
        i = a - k;
        j = b;
        while (i <= a && mat[i][j] == v) {
            i++;
            j--; AOI2
        }
        if (i <= a)
            return k - 1;
        i = a;
        j = b + k;
        while (i <= a + k && mat[i][j] == v) {
            i++;
            j--; AOI3
        }
        if (i <= a + k)
            return k - 1;
        i = a;
        j = b - k;
        while (i <= a + k && mat[i][j] == v) {
            i++;
            j++; AOI4
        }
        if (i <= a + k)
            return k - 1;
        k++;
    }
    return k - 1;
}

```

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

4.2 Visual Effort

4.2.1 Heat Map

One way to identify regions which garner special attention is using *heat maps*. They 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 investigate whether subjects follow an obvious pattern in terms of effort allocation, and by this we answer our first research question.

Figure 2 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

Table 5: Measures (averages) of the AOIs of the *median* regular implementation.

AOI	#fixations	Complete fixation time	Pupil size
AOI1	311.6	133714.0	19.98
AOI2	393.8	182459.3	19.85
AOI3	287.6	144096.0	19.54
AOI4	173.0	82537.0	19.55
AOI5	130.0	66345.5	19.38
AOI6	116.0	49318.6	19.19
AOI7	97.6	43115.3	19.11
AOI8	87.0	31101.8	19.18

Table 6: Measures (averages) of the AOIs of the *diamond* regular implementation.

AOI	#fixations	Complete fixation time	Pupil size
AOI1	496.7	235035.8	22.57
AOI2	239.0	92919.0	22.19
AOI3	179.0	80597.5	22.21
AOI4	129.2	46648.0	22.61

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 3. 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. The *median* version was divided into 8 areas of interest as shown in Figure 4 (one AOI for each instance), and the *diamond* version was likewise divided into 4 areas of interest (Figure 5). For each AOI we compute *fixation count*, *total fixations time*, and *average size of subjects' pupil*.

Tables 5 and 6 show the average measures of all areas of interest for the regular versions of both programs. 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. This behavior is preserved in terms of all measures with slight digression in few cases, but the general pattern is pretty evident.

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

Table 7: Transitions frequencies between AOIs of the *median* regular implementation.

	AOI1	AOI2	AOI3	AOI4	AOI5	AOI6	AOI7	AOI8
AOI1	<i>980</i>	<u>254</u>	23	9	9	8	3	4
AOI2	254	<i>1085</i>	<u>214</u>	21	6	5	8	2
AOI3	26	211	<i>761</i>	<u>138</u>	11	4	6	9
AOI4	14	23	112	<i>432</i>	<u>113</u>	13	6	10
AOI5	5	8	24	97	<i>408</i>	<u>85</u>	8	5
AOI6	6	5	14	14	70	<i>319</i>	<u>83</u>	15
AOI7	4	7	5	10	13	71	<i>280</i>	<u>53</u>
AOI8	4	5	10	7	7	15	48	<i>187</i>

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

	AOI1	AOI2	AOI3	AOI4
AOI1	<i>1339</i>	<u>128</u>	12	6
AOI2	123	<i>541</i>	<u>77</u>	15
AOI3	12	60	<i>383</i>	<u>52</u>
AOI4	5	13	35	<i>279</i>

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 reading pattern and whether regularity affects the supposed story order in natural languages and semi-linear order in source code [3].

According to [12] a transition is a saccade from one AOI to another one. A transition matrix contains the frequencies of direct transitions between all pairs of AOIs. Tables 7 and 8 show the 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* [12]. 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 number of within-AOI saccades decreases as we progress to higher AOIs. This again indicates that subjects face more difficulty in initial AOIs and comprehension becomes easier in the repeated instances later.

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 there are 254 transitions from AOI2 to AOI1 and 214 to

AOI3. This property is preserved for all AOIs in both tables except in two cases in Table 7 where in one case the values are almost equal. 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 one case). Also, the ratio of transitions back and forth becomes more skewed: in Table 7 there are 254 transitions from AOI1 to AOI2 and also from AOI2 to AOI1, but only 70 transitions from AOI6 to AOI5 compared with 85 from AOI5 to AOI6. This can be interpreted as meaning that the need for the previous segment is reduced over time and the inference process becomes easier.

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 1217. This follows our observation from the heat map above regarding the renewed focus on non-regular segments that follow regular ones.

4.2.4 Pupil Dilation

It is well known that there is a strong correlation between pupil size and mental effort, and therefore also a correlation between pupil size and the complexity of the task. Table 5 shows the average size of the subjects' pupil in all areas of interest in the regular version of the *median* program. According to this table the average size in the first AOI is 19.98, in the second AOI it is 19.85, in the third 19.54, and this behavior is roughly preserved as we progress to the next areas. Similar behavior occurs in the regular version of the *diamond* program (Table 6). Thus the pupil size data too indicates that successive repeated code segments become easier to comprehend.

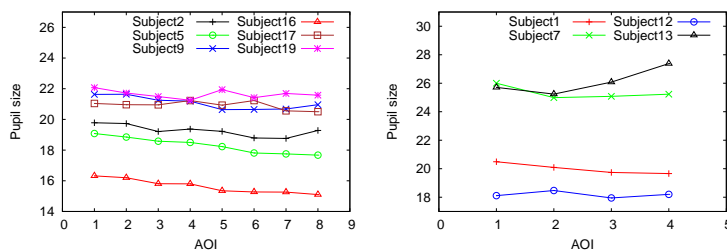


Fig. 6: Left: Changes to the pupil size of 6 subjects over 8 AOIs of the regular version of the *median* program. Right: Changes to the pupil size of 4 subjects over 4 AOIs of the regular version of the *diamond* program.

However, the differences are smaller than for the fixation data. To investigate this more deeply, we consider the individual distributions for the different subjects. As shown in Figure 6, some have a clear downward trend, e.g. *Subject5* and *Subject1*. For others there is a mainly downward trend, but it is not monotonous — as for *Subject2* and *Subject9*. Finally there are those where there is no clear trend, and even one with an upward trend. In general, the subjects who worked

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 other 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.

on the *median* program exhibited stronger trends, maybe because the *diamond* program had only 4 repeated segments.

4.2.5 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 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 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. So the added value of scanpaths over heatmaps and transition matrices is temporality.

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 noisy data. Second, we identify recurring patterns which representing scanpath events, and analyze the scanpaths according to these events.

4.3.1 Traditional Approach

In this study the average number of fixations is relatively high, therefore showing them directly superimposed over the code does not make sense. To learn about the temporality 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 7 and 8). Moreover, these figures include more AOIs than we depicted in 4 and 5. 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 in the end.

Figure 7 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 7 (right) and Figure 8. In particular, going back and forth while progressing towards lower AOIs occurs more than once within a subject's complete scanpath.

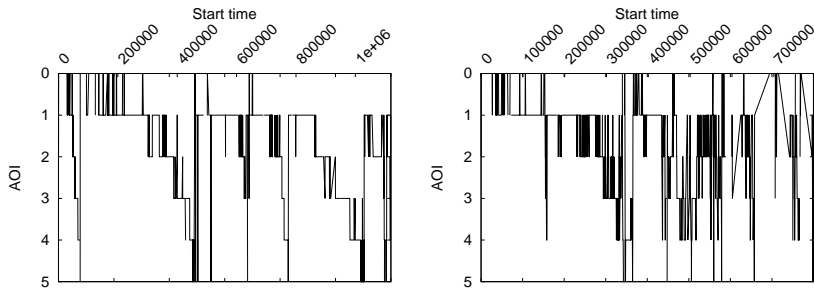


Fig. 7: Left: The fixations of *subject1* on AOIs of the *diamond* program over time. Right: The fixations of *subject12* on AOIs of the *diamond* program over time.

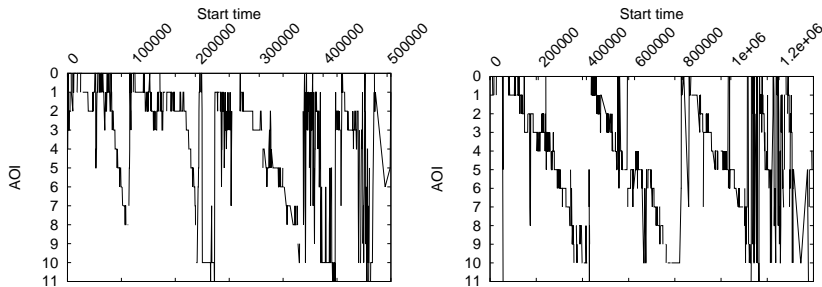


Fig. 8: Left: The fixations of *subject9* on AOIs of the *median* program over time. Right: The fixations of *subject16* on AOIs of the *median* program over time.

4.3.2 Scanpath Smoothing

Figures 7 and 8 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.

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.

Smoothing is a technique primarily used in the signal processing domain to reduce noise in the signal amplitudes (y-axis values). 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.

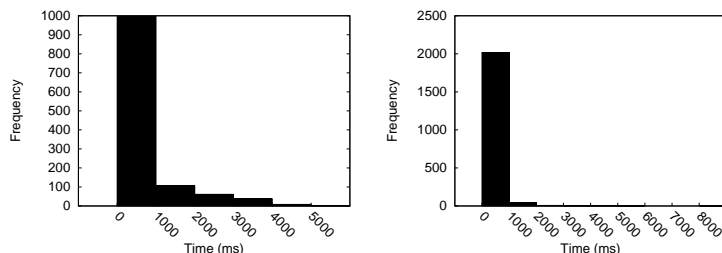


Fig. 9: Left: The distribution of dwell time of *subject1* on the *diamond* program. Right: The distribution of dwell time of *subject9* on the *median* program.

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 (discarded dwells with one point). The most frequent dwell time can be an indicator for the appropriate smooth width. Figure 9 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 resides around the area of 1000. 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 (except 1000) are not so large therefore we took the highest value we could.

Figures 10 and 11 are smoothed versions of Figures 7 and 8 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 10a *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 10c *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 10d) 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

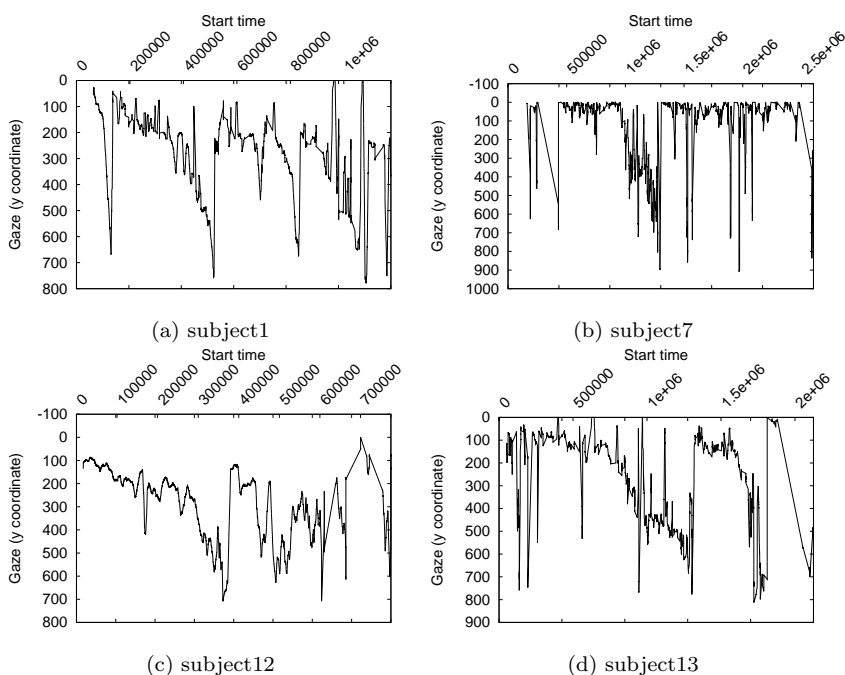


Fig. 10: Smoothed scanpaths of the *diamond* program subjects.

moves. After this a new scan starts that again covers all the code but is shorter than the previous one.

Figure 10b 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 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 scans to the end. This might be a clue of comprehension difficulty as reflected by the grade this subject achieved (76.5, average=90.4) and the very high spent time (41 min, average=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

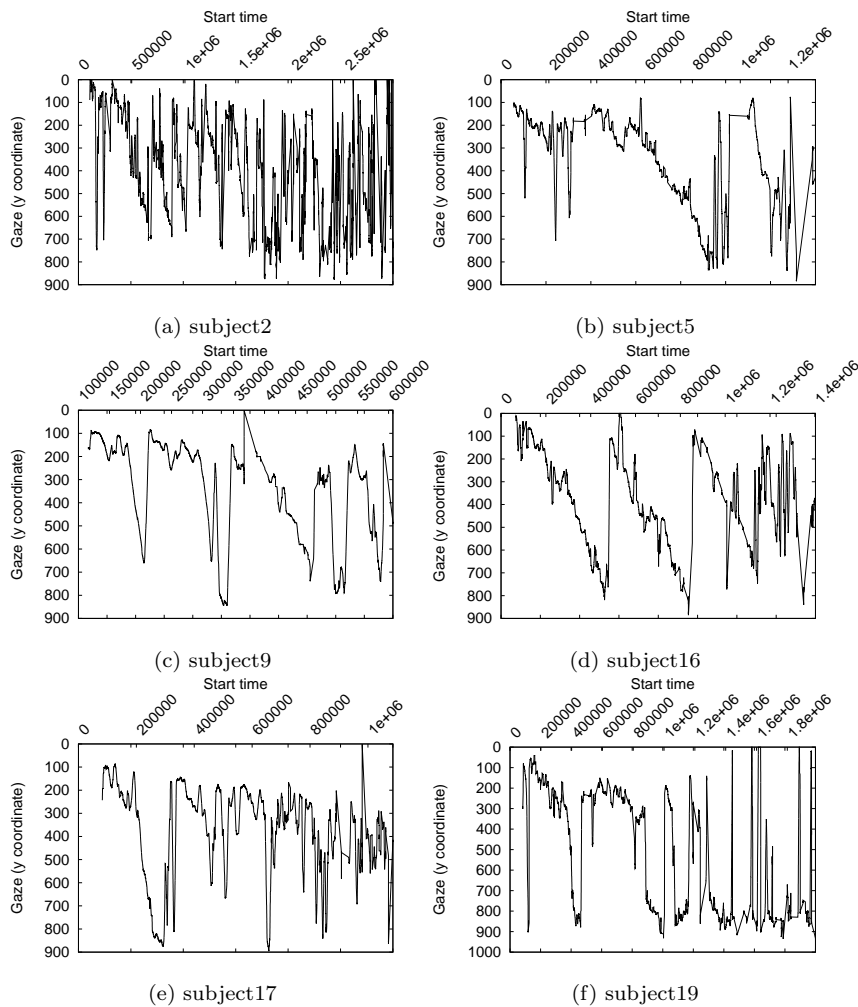


Fig. 11: Smoothed scanpaths of the *median* program subjects.

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 [32]. 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. This property is unavoidable when the regular structure of the code is considered.

4.3.3 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 [12]. In this section we analyze the scanpaths according to a set of events that have been published and reviewed in [12] as well as a few new events we introduce. Before delving in the analysis we introduce the events and describe them in Table 10. Some of the events were tagged as new which means that they were not listed in [12] and we are not aware of studies that define them as such, except for the *prescan* event that has received some attention [32] but not necessarily under this naming and context. We believe that the need for these new events reflects the fact that reading code is different from conventional reading [4, 6].

Interestingly, some events interact with their neighborhood, which means that the behavior before the event and after it is expected. For example, 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 10, 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 [24]). 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 dividing the scanpath into equal units of time which may produce segments that are composed of different events.

Table 11 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* event 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 but in 50% of the subjects it is divided nearly equally. In one subject it has 42% of its total count which means that the *look back* event is rare but thanks to an extreme value for one subject it was ranked high.

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

Table 10: Scanpath events that occur in eye-movement sequences. Letters in bold represent the events' names for symbol representation of scanpaths.

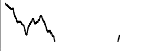



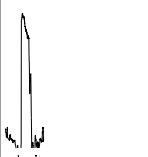
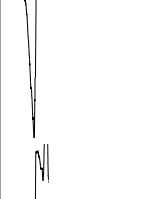
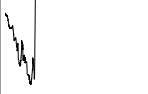
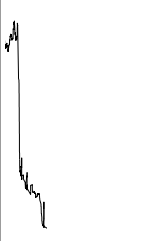
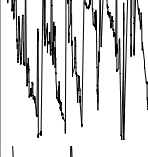
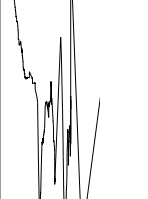
Event	Code	Description	Suggested Meaning	Illustration
Reading	R	A slow progress along the y axis (moderate slope).	Comprehension	
Fixation	F	Reading in one place.	Comprehension	
Scan	S	A fast progress along the y axis (steep slope).	Hypothesis testing	
Look ahead	A	Jump ahead along the y axis then back.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

Table 12: 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	$y = -20422.9x + 183489.2$	0	0.340
		Logarithmic	$y = -65923.5 * \ln(x) + 178972.5$	0	0.292
		Quadratic	$y = 325.1x^2 - 23349.5x + 188366.8$	0	0.340
		Cubic	$y = 1743.5x^3 - 23212.9^2 + 66443.9x + 102060.5$	0	0.375
		Exponential	$\ln(y) = -0.254x + 12.2$	0	0.465
		Power	$\ln(y) = -0.819 * \ln(x) + 12.1$	0	0.400
	Inverse	$y = 48715.4 + \frac{126189.4}{x}$	0.002	0.187	
	#fixations	Linear	$y = -43x + 393.1$	0	0.384
		Logarithmic	$y = -144.3 * \ln(x) + 390.9$	0	0.356
		Quadratic	$y = 3.2x^2 - 72.6x + 442.6$	0	0.393
		Cubic	$y = 3.1x^3 - 38.9x^2 + 88.6x + 287.5$	0	0.422
		Exponential	$\ln(y) = -0.228x + 6.0$	0	0.489
		Power	$\ln(y) = -0.756 * \ln(x) + 6.0$	0	0.443
		Inverse	$y = 101.0 + \frac{290.3}{x}$	0	0.252
diamond	complete fixation time	Linear	$y = -57748.2x + 258171$	0.007	0.414
		Logarithmic	$y = -132855.2 * \ln(x) + 219355.4$	0.003	0.475
		Quadratic	$y = 27041x^2 - 192958x + 393380.6$	0.013	0.487
		Cubic	$y = -25237x^3 + 216320.3x^2 - 614418x + 658370.5$	0.029	0.515
		Exponential	$\ln(y) = -0.405x + 12.2$	0.066	0.222
		Power	$\ln(y) = -0.858 * \ln(x) + 11.9$	0.07	0.216
	Inverse	$y = -13799 + \frac{244990.5}{x}$	0.002	0.505	
	#fixations	Linear	$y = -116.2x + 551.6$	0.006	0.432
		Logarithmic	$y = -265.6 * \ln(x) + 472.0$	0.003	0.489
		Quadratic	$y = 52x^2 - 376.2x + 811.6$	0.011	0.501
		Cubic	$y = -31.2x^3 + 286.3x^2 - 898.1x + 1139.7$	0.03	0.512
		Exponential	$\ln(y) = -0.375x + 6.2$	0.026	0.308
		Power	$\ln(y) = -0.806 * \ln(x) + 5.9$	0.026	0.308
		Inverse	$y = -7.95 + \frac{485.8}{x}$	0.002	0.511

instances are actually not read, but only scanned, as evident for example in the scanpaths of subjects 5, 9, 17, and 19 in Figure 11.

4.4 Modeling Effort in Repeated Instances

We claim that not all code segments in a program should have equal weight, especially if they have the same structure or they are clones. The rationale is that once the developer understands one instance it is easier for him to understand the other instances and therefore he needs less effort.

Based on this claim we observe that many widely used complexity metrics unfairly present inflated measurements of a given code. For example, the McCabe cyclomatic 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 predicts the effort needed to understand a repeated instance on the basis of its ordinal number. To do so we use the fixation data for all the subjects and check the fit of candidate functions to this

data. The natural candidates are various decreasing functions. Table 12 shows the models found by the 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 except one (the exponential model of the *complete fixation time* measure for the *diamond* program).

The best model turns out to depend on the program. For the *median* program the best model could be the *exponential* one, which explains about 46–49% of the observed variation. Not far behind it is the *power* model which explains about 40–44% of the observed data. Other models are far behind them, therefore both models are good candidates. As for the *diamond* program the best models are the *cubic* and the *inverse*. They succeed in explaining more than 50% of the observed variation in both measures. An additional good candidates are the *quadratic* and the *logarithmic* models which are relatively close to the best models. The worst models for this program are the *exponential* and the *power* models, which in one case are not statistically significant and in the other explain only 30% of the variation which is relatively low.

The reason for the relatively low values of the R^2 of the different models is the way the observed values distribute. For example, for the *median* version we have 8 AOIs. For each AOI we have a column of the measurements for each subject. Due to the natural variability between subjects, 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 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 the *median* program data are that all model equations are pretty much similar (up to fractional digits) to those of Table 12 and 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 92% of the variation while the worst model explained a bit more than 50%. Similarly, in the complete fixation time measure, the *exponential* model explained 92% of the variation while the worst model explained 45%.

As for the *diamond* program the results show that the *linear* and *quadratic* models are not significant, so we are left with the other five. Of these, the *cubic* model shows a perfect fit, while the other four 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. These results are true for the both the complete fixation time as well as the number of fixations measures.

But when selecting a model one should consider the characteristics of the function and not only the R^2 of the fit. For example, the seven model functions for the number of fixations on the *diamond* program from Table 12 are shown in Figure 12 (right). This shows that if we extrapolate to larger x s, the quadratic model grows to infinity, while the logarithmic, linear, inverse, and cubic models attain negative values. The exponential and power models have the more appropriate attribute of tending asymptotically to zero.

As for the *median* version (Figure 12 left) the linear, logarithmic, and quadratic models behave as in the *diamond* program although not as steeply, and the *inverse*

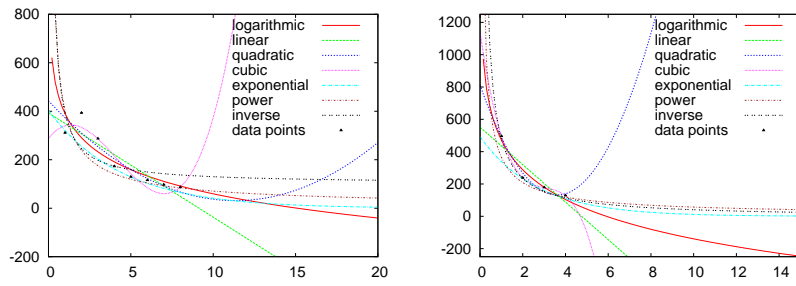


Fig. 12: Extrapolation of the model functions for the #fixation measure from Table 12. Data points are from Tables 5 and 6. Left: *median* version. Right: *diamond* version.

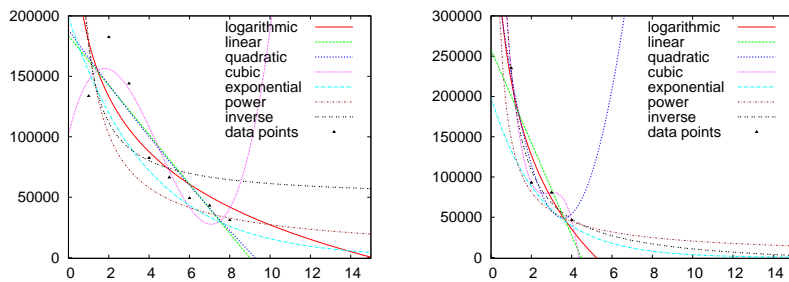


Fig. 13: Extrapolation of the model functions for the complete fixation time measure from Table 12. Data points are from Tables 5 and 6. Left: *median* version. Right: *diamond* version.

model converges to a positive value. However, the cubic model changed its direction to positive infinity to compensate for the non-monotonicity of the first point.

When considering the *complete fixation time measure* models which are shown in Figure 13, we see that the *logarithmic*, *linear*, *cubic*, *inverse*, *exponential* and *power* models behave as in Figure 12 for both programs. However, the *quadratic* model attains negative values at its minimum.

As we stated earlier, the best model depends on the program. However, it seems that it does not depend on the measure nor on the interaction between the program and the measure. On the basis of the above observations one may claim that all models, except the exponential and the power models, are bad for extrapolation as some of them grow to infinity and others to negative values for larger xs . 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.

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.

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 tacking [1,27,28,37].

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.

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 [10]. 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 [20].

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 [25]. Oman et al. defined the maintainability index on the basis of three other syntactic metrics [22,35].

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 [36].

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 [33]. 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 [23]. 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 [13]. 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.

Eye tracking has recently been used in several code comprehension studies. Sharif et al. have used eye tracking in multiple works. In [27] 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 [2]. Likewise, in [28] 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.

As for events (patterns) in eye tracking, Uwano et al. identified the *scan* pattern in subjects’ eye movements and defined it as a preliminary scan of the source code. They showed that there is an inverse correlation between time spent scanning the code and the time for finding defects [32]. Additional patterns that have been identified are reviewed in [12]. Sharif et al. replicated the Uwano study but with more participants and additional eye-tracking measures [26]. This work also investigates how programmers find defects in source code and concludes 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. Yusuf et al. used eye tracking to identify the most effective characteristics of UML class diagrams that support software tasks. 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 complexity model parameters.

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 [30,8]. Such additional measurement could be interesting also in the context of dealing with code regularity.

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 of this segment compared with its 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 [36].

However, the current experiments are not extensive enough to enable a full model to be formulated. Of the two programs we used, one produced results which favor an exponential model, while the other's results do not. 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.

Our observations and methods serve to illuminate some individual aspects of code reading, naturally focusing on regular code. We are convinced that there is a need for much further study to be able to draw the big picture.

Acknowledgments

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13).

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.
URL <http://doi.acm.org/10.1145/1117309.1117356>
2. 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.
3. 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>
4. 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.
URL <http://doi.acm.org/10.1145/2094131.2094133>
5. 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.
6. 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.
7. N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. Course Technology, 2nd ed., 1998.
8. 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.
URL <http://doi.acm.org/10.1145/2568225.2568266>
9. E. Granholm, S. K. Morris, A. J. Sarkin, R. F. Asarnow, and D. V. Jeste, “Pupillary responses index overload of working memory resources in schizophrenia”. *Journal of Abnormal Psychology* **106**, 1997.
10. M. Halstead, *Elements of Software Science*. Elsevier Science Inc., 1977.
11. E. H. Hess, “Attitude and pupil size”. *Scientific American* **212**, 1965.
12. 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.
13. 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.
URL http://dx.doi.org/10.1007/978-3-642-39259-7_24
14. A. Jbara and D. G. Feitelson, “Quantification of code regularity using preprocessing and compression”. Manuscript, Jan 2014.
15. 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.

16. 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. URL <http://dx.doi.org/10.1007/s10664-013-9275-7>
17. M. Just and P. Carpenter, "A theory of reading: From eye fixations to comprehension". *Psychological Review* **87**, pp. 329–354, 1980.
18. M. A. Just and P. A. Carpenter, "The intensity dimension of thought: Pupillometric indices of sentence processing". *Canadian Journal of Experimental Psychology* **47**, 1993.
19. 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.
20. T. McCabe, "A complexity measure". *IEEE Trans. Softw. Eng.* **2**(4), pp. 308–320, Dec 1976, doi:10.1109/TSE.1976.233837.
21. 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.
22. P. Oman and J. Hagemester, "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.
23. 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.
24. T. D. Schneider and R. M. Stephens, "Sequence logos: a new way to display consensus sequences". *Nucleic Acids Res.* **18**, 1990.
25. 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.
26. 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. URL <http://doi.acm.org/10.1145/2168556.2168642>
27. 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.
28. 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.
29. B. Shneiderman, "Measuring computer program quality and comprehension". *Intl. J. Man-Machine Studies* **9**(4), July 1977.
30. 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. URL <http://doi.acm.org/10.1145/2568225.2568252>
31. 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.
32. 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. URL <http://doi.acm.org/10.1145/1117309.1117357>
33. 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.
34. 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.

URL <http://doi.acm.org/10.1145/1117309.1117346>

35. 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. URL [http://dx.doi.org/10.1002/\(SICI\)1096-908X\(199705\)9:3<127::AID-SMR149>3.0.CO;2-S](http://dx.doi.org/10.1002/(SICI)1096-908X(199705)9:3<127::AID-SMR149>3.0.CO;2-S)
36. E. J. Weyuker, "Evaluating software complexity measures". *IEEE Trans. Softw. Eng.* **14(9)**, pp. 1357–1365, Sep 1988, doi:10.1109/32.6178.
37. 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.

Chapter 7

Conclusions

7 Conclusions

Bill Curtis has stated that there must be “as many complexity measures as there are computer scientists” [1]. This points to the inherent difficulty in defining one simple number that is capable of measuring complexity. Prather even clearly asserted that: “It is fairly certain that no one magic number can serve as a measurement for all the characteristics of software” [2]. He continued “One expects that different metrics will be needed in estimating a program’s inherent psychological complexity, its readability ...”.

With this in mind, following is a list of the main points that conclude this thesis. Their order is not according to their importance but rather a story order.

7.1 The Practice Does not Match the Common Wisdom

McCabe’s cyclomatic complexity has been one of the most popular complexity metric since its introduction. McCabe suggested 10 as the threshold above which functions are considered complex and refactoring should be applied to reduce complexity. Since then other thresholds have been suggested with values up to 50.

We investigated a few large scale software systems for the MCC of their functions and revealed that there are very many functions that have very high MCC values (up to twenty times the highest threshold ever suggested).

The immediate question asked was: are these functions very complex as their MCC values suggest? A few practical investigations showed that these numbers are far from reflecting real complexity and they place the common wisdom thresholds in a big doubt regarding their reliability. Here are the actions we performed:

- Some of these functions (of the Linux system) evolve over time. In particular, developers succeeded to maintain them which means that

their very high MCC values are not a real barrier that prevents comprehension.

- Subjective ranking of some of these functions ranked them as very easy to comprehend.
- Manual close examination of some of them showed that they are actually very long but well structured which explains why subjects tended to think they are not complex.
- Due to their length we visualized them and this clearly emphasized their flat and regular structure.

In conclusion, it seems that a simple number threshold (for MCC but fairly for others) is hard to set due to the large variability of the factors that might affect it. For example, the human factor is one that has a large effect on complexity and at the same time hard to capture.

7.2 Visualization of Long Functions

One of the best ways to capture the whole structure of a very long function is visualization. To this end we proposed and implemented CSD; a control structure diagram that helped us to easily identify regularity in code.

This diagram reflects many structural aspects such as construct type, nesting level, and code block size. we believe that this diagram has a potential for wider usage than just identifying regularity. Its comprehensive aspects could help in detecting new properties that might have an effect on complexity.

7.3 Regularity: A New Structural Property that Affects Comprehension

The most widely used complexity metric was introduced in 1976 by McCabe. Since then it has been largely criticized, mainly for its failure to capture all considerations involved in structuring code. Admittedly, other complexity metrics also suffer from this drawback but they are less common. Moreover, there is no metric or a combination of metrics that is capable to measure complexity seemingly due to the fact that they are not aware of all consideration that might have an effect on comprehension. These considerations include program aspects and humans variability.

In this thesis we introduce *regularity* as an additional factor that affects comprehension. Regularity characterizes code that has many successive repetitions of a pattern, where the effort invested in the initial instances is much larger than the one invested later instances because programmers leverage their experience in the first segments to make it easier to understand the other ones.

To show to what extent regularity affects comprehension we conducted experiments where subjects performed various comprehension tasks on regular and non-regular implementations of the same problem. The results showed that despite the supposed complexity of the regular implementations (they are longer and have much higher cyclomatic complexity) subjects achieved better results. Therefore, we conclude that regularity of code may have a large impact on comprehension by humans and may compensate for high values of cyclomatic complexity and lines of code.

To help integrating regularity in future or existing metrics we conducted an eyetracking-based experiment to explore the way programmers read regular code for the sake of understanding. We concluded that the best model would be the exponential one. This model confirmed our initial argument regarding the fact that programmers leverage their understanding of the initial segments to help themselves understanding the later ones.

7.4 Context Awareness

As has been mentioned, regularity is a new factor we have come up with that affects comprehension. This new factor leads to *context awareness* which is an aspect that has a wider implications and might help in revealing new properties, such as regularity, that might have an effect on comprehension.

Context awareness points to the ability of being sensitive to some factors that somehow are related to the measured element. This means that complexity is no longer absolute and it is relative to other factors such as location and neighbors.

Therefore, we believe that future metrics as well as existing metrics should consider this type of awareness in order to better reflect complexity.

Admittedly, *context awareness* has been around for a long time but has not received appropriate attention. Martin Shepperd in a critique paper on the cyclomatic complexity stated that “the complexity of a decision cannot be considered in isolation, but must take into account other decisions within its scope” [3].

7.5 Challenging Existing Metrics

A very large number of metrics has been developed over the years. This broad arsenal fails in providing an effective way to measure complexity. Nevertheless they are still in use.

The new property (regularity) we presented in this thesis challenges the reliability of the existing metrics in reflecting effective complexity. Following this new property and its wider aspect (context awareness) an introspection of the existing metrics is probably necessary and perhaps a redefinition of some is required. For example, we have already shown the failure of MCC in predicting complexity in the presence of regular code and even suggested a way to model regularity for integration in MCC. In particular, we found that the effort invested in comprehending successive regular instances is gov-

erned by a decreasing functions, therefore, we suggest to give these segments different weights according to their order in the sequence.

This considerations should also be taken into account whenever a new metric is proposed.

7.6 Non Linearity of Code Reading

It is well known that natural language text is read quite linearly; from left-to-right, top-to-bottom. We have shown that code is read in a non-linear manner and in particular it seems that some regular code is not read at all but rather just scanned.

This examination has been largely based on an event-based framework we have developed for analyzing scanpaths built when code is read by programmers. This framework can be used in any eyetracking-based experiment for better analysis of subject's reading path.

7.7 Experimentally Grounded Insights

Our results were based on a family of experiments we conducted in every stage of this thesis. It starts by subjectively ranking High-MCC functions, comparing the effect of regular and non-regular implementations on comprehension, and finally studying the way programmers read regular code using eyetracking means.

It is very important and even necessary to experimentally verify conjectures in general because it has been shown that there is a large gap between the measurements of existing metrics and complexity as perceived by humans. Beyond the verification of our conjectures, the experimental observations led to unknown and surprising insights that would not otherwise be achieved.

References

- [1] B. Curtis. In search of software complexity. In *IEEE workshop on quantitative software models*, pages 95 –106, 1979.
- [2] R. E. Prather. An axiomatic theory of software complexity measure. *The Computer J.*, 27:340–347, Jan 1984.
- [3] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering J.*, 3(2):30–36, Mar 1988.

Bibliography

- [1] Ahmad Jbara, Adam Matan, and Dror G. Feitelson. High-MCC functions in the Linux kernel. In *Proceedings of the 20th IEEE International Conference on Program Comprehension, ICPC 2012.*, Jun 2012.
- [2] Ahmad Jbara, Adam Matan, and Dror G. Feitelson. High-MCC functions in the Linux kernel. *Empirical Software Engineering*, 19(5):1261–1298, 2014.
- [3] Ahmad Jbara and Dror G. Feitelson. JCSD: Visual support for understanding code control structure. In *Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014*, pages 300–303, New York, NY, USA, 2014. ACM.
- [4] Ahmad Jbara and Dror G. Feitelson. On the effect of code regularity on comprehension. In *Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014*, pages 189–200, New York, NY, USA, 2014. ACM.
- [5] Ahmad Jbara and Dror G. Feitelson. How programmers read regular code: A controlled experiment using eye tracking. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*, pages 244–254, May 2015.
- [6] Ahmad Jbara and Dror G. Feitelson. How programmers read regular code: A controlled experiment using eye tracking. *Empirical Software Engineering*, 2015. under review.
- [7] Ahmad Jbara and Dror G. Feitelson. Quantification of code regularity using preprocessing and compression. Manuscript, Jan 2014.

רגולריות של קוד: מאפיין מבני חדש והשפעתו על

מורכבות והבנת הקוד

חיבור לשם קבלת תואר דוקטור לפילוסופיה

מאת

אחמד גבארה

הוגש לסנט האוניברסיטה העברית

דצמבר 2015

עבודה זו נעשתה בהדרכתו של פרופ' דרור פייטלסון.

תקציר

הבנת קוד של תוכנית מחשב (Program Comprehension) היא תהליך שבו נבנה מודל מנטלי עבור הקוד הנתון. תהליך זה הוא הבסיס לכל פעילות התחזוקה (Maintenance) של התוכנה כדוגמת תיקון באגים והוספה של דרישות חדשות לתוכנה קיימת.

חשיבותה של תחזוקת תוכנה נובעת מהעובדה שחלק גדול מהמשאבים אשר מוקצים במהלך חי התוכנה מנוצלים בפעילות זו. לכן, נובע מכך שגם תהליך הבנת הקוד חשוב, לא רק בשל היותו שלב מקדים לכל פעילות תחזוקתית, אלא בשל העובדה שהצלחת תהליך התחזוקה תלויה במידה רבה בהבנת הקוד תחילה. בפרט, ככל שהקוד מובן יותר סבירות גבוהה שתהליך התחזוקה יצליח יותר.

מנגד, הבנת קוד מושפעת בצורה ישירה מסיבוכיות הקוד (Code Complexity) שזה אומר שככל שהקוד מסובך פחות יהיה קל יותר למתכנת להבין אותו. למשל, קינון (Nesting) וזרימה לא לינארית של קוד הם גורמים אשר משפיעים על סיבוכיות הקוד וככל הנראה הופכים אותו לקשה להבנה. לכן, אנחנו רוצים לשפר את הבנת הקוד על ידי כתיבת תוכניות קשות פחות. כמובן, הישימות של מטרה זו תלויה ביכולת שלנו למדוד סיבוכיות של קוד. כפי שג'יימס הרינגטון סיכם, "מדידה הינה השלב ההתחלתי שמוביל לשליטה שמובילה לשיפור. אם אין יכולת למדוד משהו, אין אפשרות להבין אותו. אם אין אפשרות להבין משהו, אינך יכול לשלוט בו. אם אינך יכול לשלוט במשהו אינך יכול לשפר אותו."

במשך השנים הוגדרו הרבה מדדים לסיבוכיות של קוד. הכמות הגדולה של מדדים אלה מעידה אולי יותר מכל דבר אחר על הקושי האמיתי בהגדרת מדד אידיאלי שיש לו יכולת לשקף סיבוכיות של קוד על ידי הצגת מספר אחד פשוט. הבעיה העיקרית של מדדי הסיבוכיות הקיימים היא שמדדים אלה מעריכים סיבוכיות על ידי ניתוח תחבירי של הקוד הנתון. למשל, המדד הציקלומטי של McCabe (McCabe's Cyclomatic Complexity - MCC) מתבסס על מספר מסלולי הביצוע הבלתי תלויים בקוד שהינו שקול למספר התנאים ועוד אחד. מדד זה נחשב לשימושי ביותר מאז הצגתו ב-1976. למרות הפופולריות שלו, מדד זה קבל הרבה ביקורות במשך השנים בעיקר בשל העובדה שהוא מתעלם מזרימת הנתונים (Data Flow) בקוד ומסתפק בספירת אלמנטים של התוכנית ללא התייחסות להקשר שלהם. חוסר שביעות הרצון מהיכולות המעשיות של המדדים הקיימים מובילה לשאלת המחקר של למה מדד ה-MCC ואחרים אינם מספיק טובים על מנת לשקף סיבוכיות אפקטיבית? מה חסר במדדים אלה ומה צריך כדי להגדיר מדדים טובים יותר?

בעבודת גמר זו אנחנו מציגים רגולריות (Regularity) כגורם נוסף שיש לו השפעה על סיבוכיות הקוד. בפרט, בקוד רגולרי קיימות הרבה חזרות של תבניות מסוימות כאשר הבנת מופעים רצופים של תבניות אלה הופכת לקלה יותר בהינתן הניסיון שהמתכנת צובר בהבנת המופעים הראשונים של כל תבנית.

החידוש העיקרי ברגולריות הוא המודעות להקשר (Context Awareness) והרגישות אליו: קטע קוד מסוים ניתן לאבחנה ברמות שונות של סיבוכיות וזאת כתלות בקוד שכן. לחלופין, סיבוכיות של קוד אינה מוגדרת יותר כערך מוחלט אלא תלויה הקשר. בפרט, ברגולריות המופעים ההתחלתיים נמדדים כמסובכים יותר מאשר המופעים בהמשך וזאת בשל העובדה שהסיבוכיות האפקטיבית של המופעים החוזרים קטנה בגלל שמתכנתים ממנפים את הבנתם לקטעים הראשונים כדי להפוך את ההתמודדות בקטעים החוזרים הבאים לקלה יותר.

מנגד, מדדי הסיבוכיות הנוכחיים, כולל מדד ה-MCC, באופן בלתי מודע מזניחים את היבט ההקשר בקוד (Code Context). בפרט, הם פשוט סופרים אלמנטים בקוד. למשל, מדד ה-LOC (lines of code) סופר שורות קוד ומדד ה-MCC סופר תנאים.

קיימת חשיבות גדולה לאמץ גישה אמפירית על מנת לחקור את ההשפעות שקיימות בפרקטיקה. בפרט, הרצנו סדרה של ניסויים מגוונים שהתחילו בניסוי פשוט של דירוג סובייקטיבי (Subjective Ranking) וכלו בניסוי שהתבסס על עקיבה אחר תנועות עיניים (Eyetracking-based Experiment). התוצאות מראות שנסיינים מעריכים חלק מהפונקציות עם סיבוכיות ציקלומטית גבוהה כלא מסובכות. יתרה מזו, הנסיינים ביצעו משימות הבנה והשיגו תוצאות טובות יותר בקוד רגולרי (למרות היותו מסובך יותר על פי מדד ה-MCC) בהשוואה לקוד שאינו רגולרי.

מעבר לאפקטים המידיים שהצגנו, המחקר האמפירי שלנו חשף תובנות אודות קריאת קוד באופן כללי וקריאה של קוד רגולרי באופן פרטי. בפרט, התוצאות מראות שקריאה של קוד היא לא לינארית בהשוואה לקריאת טקסט של שפה טבעית. באשר לקוד רגולרי, ככל הנראה שקוד זה לא נקרא בכללותו אלא, חלקים ממנו רק נסקרים.

המחקר על הדרך שבה מתכנתים קוראים קוד רגולרי הוביל לתשתית מבוססת אירועים אשר משמשת כבסיס לניתוח קריאה של קוד באופן כללי. בפרט, הרחבנו את אוסף האירועים אשר מתרחשים בזמן תהליך קריאה של קוד (קריאה, סריקה וכו') ואשר הוצגו בעבודות קודמות. בנוסף להגדרה של אירועים חדשים הצענו דרך לקידוד נתוני קריאה לניתוחים עתידיים על ידי שימוש בטכניקות מתחום הביואינפורמטיקה.

אמנם בעבודה זו אנחנו מציגים מאפיין חדש שיש לו השפעה על סיבוכיות הקוד ועל כן הוא מציב, בין היתר, אתגר בפני המדדים הקיימים, אבל רגולריות הינה רק דוגמה של גורם כזה והיא אינה פותרת הכל. רגולריות בעצם חושפת היבטים רחבים יותר שיכולים לשמש כקווים מנחים בגילוי והצגה של גורמים חדשים.

עבודת גמר זו מורכבת מאסופה של מאמרים שמציגים ומנתחים את הרעיונות שהוצגו לעיל. המאמר הראשון בוחן את תמונת המצב הפרקטית על ידי הסתכלות בפונקציות אמיתיות בעלות ערכי MCC גבוהים במיוחד. מאמר זה משמש כמאמר מוטיבציה למחקר המשך באותו כיוון [1]. מאמר זה הורחב לגרסת ג'ורנל כאשר אותם רעיונות נבחנו אך

הפעם במערכות תוכנה נוספות כדי לנסות לתאר תמונת מצב כללית יותר [2]. בשל העובדה שפונקציות עם סיבוכיות ציקלומטית גבוהה הן גם ארוכות מאוד הצענו כלי ויזואליזציה שעוזר בין יתר הדברים בזיהוי רגולריות בקוד [3]. הצעד הבא היה לחקור את השפעת הרגולריות על הבנה של קוד וזאת על ידי השוואה ניסיונית של קוד רגולרי עם קוד שהוא לא רגולרי של אותה תוכנית. התוצאות מראות שקוד רגולרי קל יותר להבנה למרות היותו ארוך יותר ומסובך יותר (על פי MCC) [4].

היבט חשוב נוסף שבחנו אותו היה הדרך שבה מתכנתים קוראים קוד רגולרי. מחקר זה הניב מודל אשר משקף את המאמצים אשר מושקעים בהבנה של קוד רגולרי כפונקציה יורדת [5]. בגרסת הג'ורנל המורחבת של מאמר זה הראינו שקריאת קוד שונה בהרבה מאשר קריאה של טקסט של שפה טבעית [6]. לבסוף, הצענו דרך כדי למדוד רגולריות באמצעים של דחיסה. עבודה זו עדיין לא התפרסמה והיא מופיעה כדוח שמסכם את המסקנות בתחום זה [7].