Code Complexity Management for Practitioners

Vard Antinyan
University of Gothenburg
P.O. Box 40530
Sweden
vard@chalmers.se

Miroslaw Staron
University of Gothenburg
P.O. Box 40530
Sweden
miroslaww.staron@cse.gu.se

Anna Sandberg
University of Gothenburg
P.O. Box 40530
Sweden
anna.sandberg@volvocars.com

ABSTRACT

Code complexity has massive influence on defect-proneness and maintainability of software. Therefore, many complexity measures were created in the past for managing it. These measures, however, were created based on theoretical frameworks, which require that a given measure should fulfill certain a priory properties of complexity. But fulfilling the properties does not guarantee that the measure can indicate complex code as experienced by practitioners. For this reason the existing complexity measures evidently have been of little help in supporting such decisions as refactoring or proactive code improvements. The goal of this study is to provide empirical basis for more accurate assessment and thus management of code complexity. To reach the goal, 24 distinct code characteristics are identified, which influence the increase of code complexity. The influence of each of the characteristics is evaluated by a survey with 393 developers worldwide. Every characteristic is examined in the context of results to understand its essentiality in coding. The results show that there are nine code characteristics that have substantial influence on complexity increase but are poorly captured in the current complexity measures. Crucially, these characteristics are mostly avoidable in practice.

KEYWORDS

Code complexity, metric, measure, maintainability, defects, software quality, technical debt

1 INTRODUCTION

Code complexity influences the ability of software practitioners to progress software development. Complexity directly affects the maintainability and defect proneness of code. Therefore, research interest on the topic of complexity has been high over the past 40 years. This has resulted in designing code complexity measures [1-3] to guide complexity management. Complexity measurement permits quantification of complexity, and estimation of its influence on maintainability and defect proneness. The concept of complexity, however, is not an atomic concept, so it has been difficult to design a single measure that quantifies complexity thoroughly. Instead, several complementary measures were designed to measure different aspects of complexity.

Designing complexity measures often has followed theoretically established frameworks, according to which a complexity measure should either fulfill a predetermined set of properties or comply with a set of rules [4-7]. Theoretical frameworks for creating measures are necessary because they propose a common foundation upon which complexity measures should be designed. Nonetheless, the practice has shown that theoretical frameworks alone are dissatisfactory for designing useful complexity measures, because detailed knowledge supporting the design of measures can be discerned from empirical data. Specifically, to design a complexity measure one must:

1. Identify the characteristics of code that have distinct influence on complexity
2. Understand whether these characteristics are measurable in practice
3. Evaluate the influence of these characteristics on complexity increase

Because these factors are not addressed fully in the design of complexity measures, existing measures are usually perceived as being simplistic and only moderately accurate in maintainability assessment. A typical example of this is when two source code functions have the same cyclomatic complexity value. The cyclomatic complexity is the same, but the experience shows that one of the functions is more complex because, for example, it has more nested blocks [8]. These kinds of issues are apparent in many well-recognized complexity measures and have been discussed previously [9-12]. In practice, certain areas of code are perceived to be intrinsically more complex and, therefore, more difficult to maintain despite their relatively small size [13, 14]. This paper is an endeavor to provide empirical support for accurate complexity assessment and management. These results can also serve for designing more sophisticated complexity measures.

To conduct this research we assumed that the aforementioned three points can be partially clarified if we consider the collective experience of software practitioners. The aim of this study, therefore, was to acquire such knowledge using the following research question:

> *What code characteristics are the main triggers of complexity in software practitioners' experience?*

This study presents the evaluation results of code characteristics as complexity triggers based on a survey of 393 software practitioners worldwide. In summary, the results showed that:

1. Of the 24 proposed code characteristics, nine substantially influence the complexity growth.
2. The nine characteristics are not essential for code and are mostly avoidable.
3. Seven characteristics, which are the essential constituents of code, have only insignificant effect on complexity increase.

These results and their discussions provide valuable insights on how to develop simple code. These results can further be used alongside theoretical frameworks for developing sophisticated complexity measures.

2 The Basis of the Current Complexity Measures

The current complexity measures are primarily based on or evaluated by theoretical frameworks. Theoretical frameworks help with providing a common ground for complexity measures. That is the *properties* that a complexity measure should fulfill and the *rules* that it should comply with. Theoretical frameworks help to evaluate whether a complexity measure actually measures complexity. The properties and rules, in their turn, are derived from the essential understanding of software complexity. One of the early works dedicated to complexity understanding is reported by Weyuker [4]. In this work she formulates a set of properties which shall be necessary for newly defined complexity measures. Even though the work was criticized for its simplicity, it still provided a fresh ground for measurement validation. This was called property-based complexity measurement [15], the essence of which was to understand what basic properties a particular complexity measure shall fulfill in order to be a valid complexity measure. Another framework was provided by Schneidewind [5]. The methodology relies on six validity criteria for a measure: association, consistency, discriminative power, tracking, predictability, and repeatability. Schneidewind claims that fulfilling these criteria provides a good rationale for creating a useful measure. Briand, et al. [15] presented property based measurement for facilitating the selection and validation of measures. Among other code attributes, such as *size*, *length*, and *coupling*, they also formulated properties that a complexity measure should hold in order to be a "true" complexity measure. These properties are based on the available intuitive definitions of complexity. For example, it is intuitive to assume that each of the complexities of two pieces of code should be smaller than the combined complexity of theirs. Nonetheless, there are more difficult aspects of complexity that cannot be captured by simple intuition. For example, what characteristics make code more complex and what characteristics make it less complex. Therefore, to understand the essence of complexity a deeper scrutiny is required. Kaner [12] explored the use of software measures in the field of software engineering and found that there are too many simplistic measures of complexity (and other simplistic measures generally). He noticed that the use of such measures is not rare, so it is important to put more effort in the design of measures to get more insightful data.

Generally there has been a lot of effort in understanding how applicable the measurement theory is in practice. A large portion of these efforts is relevant for complexity measures too. For example Briand, et al. [16] found that the application of software measurement theory sometimes can be questionable due to several factors, such as undefined scale types of several measures, endless discussions of what exact properties complexity measures should fulfill, what kind of statistical model should be used for measures' validation, etc. In a mapping study Kitchenham [17] concludes that there is a large body of empirical validation of measures. It seems in measurement research we do not quite know how much a measure should be validated, nonetheless. Therefore, even some old and well-established measures can still be validated by researchers. A recent study reported by Mair and Shepperd [18] concludes that software engineers' participation should be considered when designing prediction measures in companies. McGarry [19] (p. 128) emphasizes the importance of users' feedback for designing adequate measures. Sellami and Abran [20] found that the existing measurement validation frameworks rely on validation criteria of different philosophies. As a way forward they suggested building consolidated framework based on multiple validation types. There are also international standards of software measurement, such as ISO/IEC 15939 [21] and ISO/IEC 25000 [22]. These standards aim to facilitate the design and evaluation of measurement and also provide a common vocabulary to the community. They, however, need consolidation for providing explicit guidance for assessing the usefulness of measures.

3 A Scrutiny of Code Complexity Sources

The term *complexity* has been used widely in many disciplines, usually to describe an intrinsic quality of systems that strongly influences human understandability of these systems. Unfortunately, there is no generally accepted definition of complexity that would facilitate its measurement. Therefore, every discipline has its own approximate understanding of how to quantify complexity.

In the IEEE standard computer dictionary, code complexity is defined as "the degree to which a system or component has a design or implementation that is difficult to understand and verify" [23]. According to Zuse [3], the true meaning of code complexity is the difficulty in understanding, changing and maintaining code. Fenton and Bieman [1] view code complexity as the resources spent on maintaining a solution for a given task. Similarly, Basili [24] defines code complexity as a measure of the resources allocated by a system or human while interacting with a piece of code to perform a given task. These definitions do not facilitate the measurement and reduction of complexity because they focus on the effects of complexity rather than the very fabric of complexity. Briand, et al. [15] have suggested that complexity should be defined as an intrinsic attribute of code and not its perceived difficulty by an external observer, which would indeed aid the understanding of the essence of complexity.

To outline a landscape of the source of code complexity that would facilitate the design of the survey questions and the interpretation of the results, we adopted a general definition of system complexity that considers it to be an intrinsic attribute of a system. An example of such a definition is provided by Moses [25], who defines complexity as "an emergent property of a system due to its many elements and interconnections". This is very similar to the definition of Rechtin and Maier [26], stating that "a complex system has a set of different elements so connected or related as to perform a unique function not performable by the elements alone". These two definitions are suitable for understanding and measuring code complexity because they indicate the origin of complexity, namely different elements and their interconnections in the code. Elements and interconnections appear to be the direct sources of code complexity, i.e., those sources that should be used in complexity measurement and management. Based on these two definitions, we can imply that:

1. The more elements and interconnections the code contains, the more complex the code
2. Because the elements and interconnections always have some kind of representation (for reading, understanding, and interpreting), the complexity depends on this representational clarity
3. If we consider that any system usually evolves over time, the evolution of elements and interconnections also determines a change in complexity.

Considering these three points, we postulate that there are three direct sources of code complexity:

1. Elements and their connections in a unit of code
2. Representational clarity of the elements and interconnections in a unit of code
3. Evolution of a unit of code over development time.

Elements and their connections: Complexity emerges from existing elements and their interconnections in a unit of code. For a unit of code, the elements are different types of source code statements (e.g., constants, global and local variables, invocations, etc.). The interconnections of elements can be expressed both by mathematical operators (e.g., addition, division, multiplication, etc.) and control statements, Boolean operators, pointers, nesting level of code, etc. Each type of element and each type of connection increases the magnitude of code complexity to a different extent.

Representational clarity: Edmonds [27] noticed that ascribing complexity to a system is only adequate if the system is representable in terms of a communicable language. Otherwise, complexity could not be experienced by people and thus relevant. Therefore, there is an aspect of complexity that emerges from unclear representation of the code. This means that there could be a difference between what a given element does and what its representation implies that it does. A typical example is using misleading names for functions and variables in source code.

Intensity of evolution: Code evolution can be characterized by the frequency and magnitude of changes of that code. Evolution of the code is also regarded as a source of complexity because this changes the information about how a given piece of code operates in order to complete a given task. If a software engineer already has knowledge on how the code operates, then the evolution of the code will partly or completely destroy that knowledge because changes will introduce a new set of elements and interconnections into the code. *This does not imply that changing the code always makes the code more complex, it only implies that the level of complexity, solely driven by changes in the code, increases.* At the same time, the level of complexity that emerges from elements and their connections might decrease and thus potentially reducing overall complexity. This occurs often in practice when the code is refactored successfully.

We used these three direct sources of complexity to correctly identify those code characteristics that belong to any of these sources as direct complexity triggers. Subsequently, we developed the survey questions to evaluate these characteristics.

4 Research Methodology

To address the research question, we conducted an online survey [28] with software engineers worldwide. There were 29 questions in total, 28 of which were structured, using a six-point Likert scale. An even number for the scale values avoided a scale midpoint, thereby ensuring that respondents could choose a higher or lower estimate than average. The four initial questions were demographics-related. The next 24 questions were concerned with code characteristics as complexity triggers. The respondents were asked to evaluate the influence of a given characteristic on complexity increase. The last question was open-ended, requesting to share the respondent's views on what else characteristics there are that can have influence on complexity.

Survey participants were software engineers that were a part of 30 open source product development worldwide. As they indicated in the survey, most of them were also involved in developing commercial products.

We shared the online address of the survey with the software engineers of the products through their mailing lists. Only one request was sent to prompt a response from the participants. In total 393 responses were received. The response rate was estimated by counting the number of potential respondents in the mailing lists. The response rate was 28%.

To minimize any misunderstanding of words or concepts in the survey questions, two pilot studies were conducted prior to the survey launch. Feedback from a group of nine software engineers from companies was also used to improve the survey and the choice of assessment scales. The nine software engineers were software architects and developers that we had collaborated with previously. This test group was also asked to interpret their understanding of the survey questions in order to identify any misinterpretations. The survey was only launched once all nine engineers understood the survey questions as they were intended to be understood. The results of the pilot studies are not included in the results of this study.

4.1 Demographics and the Related Questions

The four questions of the survey investigated the participant demographics. These questions provide general information on the respondents and the products in order understand the general relevance of the collected answers. Four fields were given for information related to demographics, as presented with the specified options in Table 1. Data for the four fields were collected using the following four statements:

1. Select the most appropriate title for you
2. Select the years of experience that you have in software development
3. Select the types of product you have developed
4. Select the size of products that you developed

Table 1 Demographic information

Job Title	Experience	Product type	Size of product
Developer (238)	< 1 years (12)	Industrial (255)	Hundreds of LOC (96)
Tester (5)	1-2 years (32)	Academic (194)	Thousands of LOC (182)
Architect (16)	3–5 years (81)	Open source (273)	Tens of thousands of LOC (224)
Team leader, (37)	6–10 years (93)	Amateur (182)	Millions of LOC (71)
Product owner (2)	11–15 years (53)	Other (28)	
Project manager (11)	> 15 years (126)		
Researcher (52)			
Student (27)			
Other (1 unemployed)			

In the cases of "Job Title" and "Experience", options were given by radio buttons with a "one-choice-only" option. Checkboxes

were specified for the "product type" and "product size" options. In Table 1, the number of responses obtained per demographical category is shown in brackets.

4.2 Selected Code Characteristics

The main part of the survey concerned code characteristics with the objective of understanding the extent to which each code characteristic increases code complexity. In a previous work of our, we were designing code complexity measurement systems for two companies where approximately 20 software engineers were involved. Based on biweekly reference group discussions over nine months, on such topics as the origin of complexity and which code characteristics are usually considered in complexity measurement, we determined 24 common code characteristics that were used in this study. These characteristics belong to one of the three main sources of the complexity landscape presented in Section 2. The three main sources, complexity characteristics and their descriptions are shown in Table 2. Twenty-three of the characteristics are directly observable in the code. Only one of them – *many developers* – is difficult to observe in the code but it still has direct influence on complexity. This is because every developer, who makes changes on the same piece of code, has her/his own contribution to code change. The information needed to learn about the change in this case comes from multiple developers.

To investigate the effect of these 24 characteristics on code complexity, one statement (question) per characteristic was formulated for being answered using the specified Likert scale. For example, the statement for function calls is shown Figure 1. The three dots at the end of the statement were to be completed by one of the options given in the Likert scale. The second line explained in more detail what was meant by the given characteristic to ensure no uncertainty on the part of the respondent.

The rest of the statements about code characteristics were organized the same way as that shown Figure 1. In most of the statements, we intentionally emphasized that "*many* of something" makes code complex, i.e., *many* operators, *many* variables, *many* control statements, etc. Wherever it was not possible to use the "*many*" keyword, we used equivalent intensifiers, for example, the *deep* nesting, the *frequent* changes, etc. If we decided to use a specific number instead of many, it would not be clear why a particular threshold number is chosen for the question. Moreover, for different characteristics this number might need to be different. Oppositely, the use of non-specific intensifier, such as "*many*" is usually interpreted as: *how drastically complexity increases with the increasing number of a given characteristic.* Therefore, it is easier to answer the questions.

Table 2 Code characteristics and descriptions

Three sources of complexity	24 Code Characteristics	Description of the Characteristic
Elements and interconnections	Many operators	All mathematical operators (e.g., =, +, -, /, mod, sqrt)
	Many local variables	Locally declared variables in the code
	Many global variables	Globally declared variables in the code
	Many non-nested branching statements	Branching statements are 'if', 'else', 'continue', 'switch', 'break', etc. Non-nested use of them is considered.
	Many non-nested looping statements	Looping statements are 'while', 'do while', 'for', 'foreach', etc. Non-nested use of them is considered.
	Deep nesting	The code is nested if there are many code-blocks inside one another
	Many pointers	Variables that show addresses of other variables
	Many calls	All unique invocations of methods or functions in the code
	Many arrays	All arrays or alike data structures that represent collections of variables
	Many logically unrelated tasks	Relatively unrelated pieces of tasks solved in the same unit of code
	Many preprocessors	All preprocessors inside functions or methods: an example in C language is: '#if', '#else', '#elif', '#endif', etc.
	Many threads	Much concurrency in a unit of code
	Deep inheritance	Classes which have many ancestors
	Many objects	Objects are the instances of classes which are used in real functionality
	Many abstract definitions	Abstract methods, classes, interfaces
	Many polymorphic functions	Many functions with the same name
Representational clarity	Very long names	Names that were intended to provide thorough information about functions and variables but got out of proportion in length
	Too short names	Names that due to shortness cannot deliver meaningful information to describe the given variable or function
	Very long lines of code	Lengthy line of code that cannot be embraced at one glance
	Many incorrect indentations	Indentations that do not show the hierarchical order in the code
	Misleading comments	Comments that are not accurate or has become obsoleted over development time
	Misleading comments	Code that does not have any comments at all
Evolution	Frequent changes	This relates to code that changes frequently thus behaving differently over development time
	Many developers	This relates to code that is modified by many developers in parallel

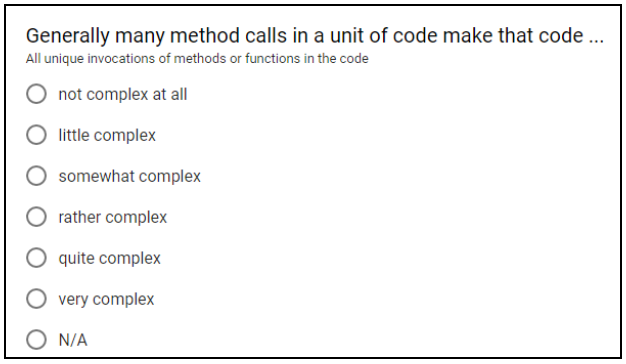

Figure 1 Example of a question regarding a given code characteristic

At the end of the survey, an open question was included to allow respondents to suggest other code characteristics that they believed could significantly increase complexity.

The six intensifier words used in Likert scale are chosen based on knowledge available in linguistics. Their order is checked in linguistic forums. For example, commonly, it is considered that the intensifier "quite" is stronger than the intensifier "rather". These choices were later tested and verified in the pilot study too, to make sure that the six intensifiers and their order are ambiguous for the respondents.

5 Results and Discussion

Figures 2 and 3 show the results of the evaluation. Every bar in the figures represents a code characteristic. The six Likert scale categories of evaluation are represented by colors. Dark green color represents the lowest level of influence of a characteristic on complexity increase. The rest of the colors are ordered by increasing influence through the vertical axis. The vertical axis shows the number of responses by percentage. The number of respondents per color is written right on the colors. Thus, the length of the area of a color is proportional to the number of respondents who selected that particular influence level for a given characteristic. For example, the first bar, which corresponds to the characteristic "*misleading comments*", has the largest dark red area, indicating that 137 of the respondents believed that many misleading comments make code *very complex.*

Generally *misleading comments* were regarded as the most influential characteristic of all. Several respondents argued that code comments have tendency to become obsolete over time, because they are not updated. Therefore, comments should be embedded in the names of functions and variables. Comments should be used as one-liners to describe a whole function or a class or otherwise a large portion of code. This finding goes against the convention that in average 10-15% of code should be the comments.

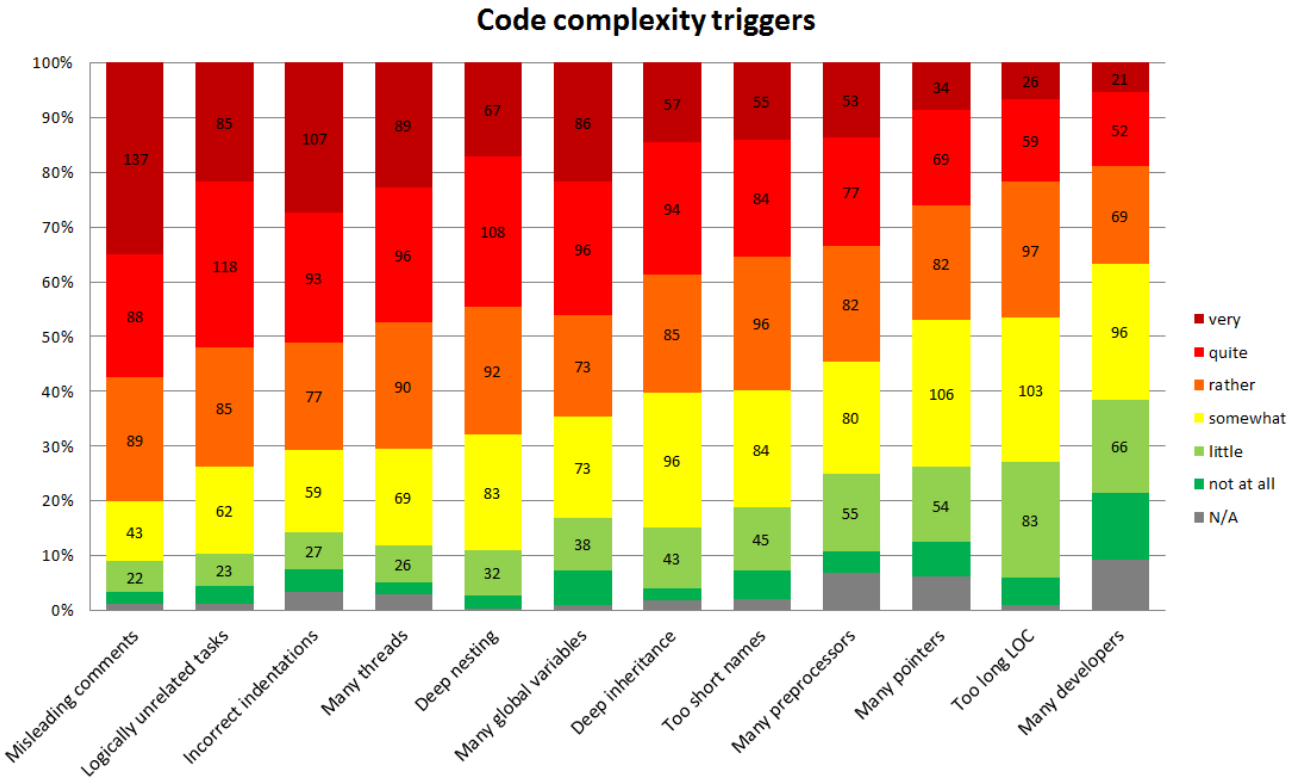

Figure 2 Evaluated influence of code characteristics on complexity increase (part 1)

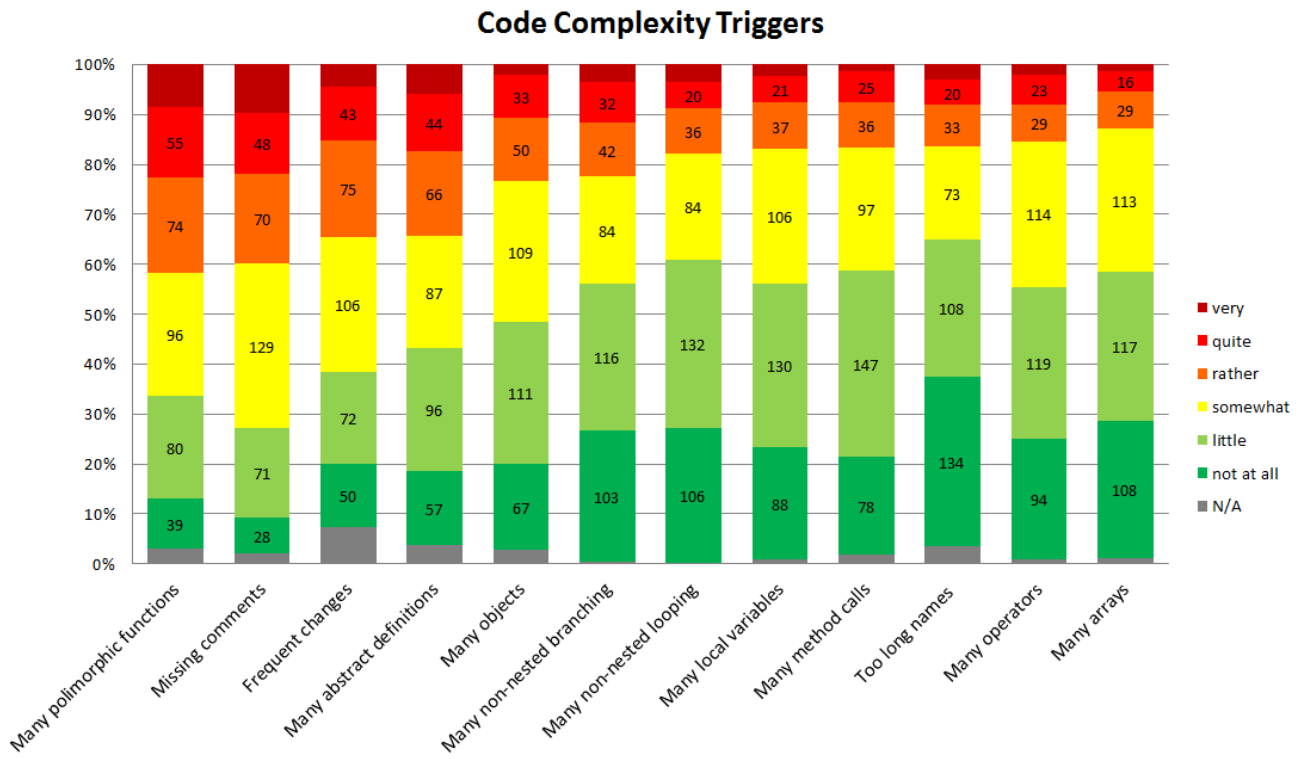

Figure 3 Evaluated influence of code characteristics on complexity increase (part 2)

It seems that in large products, which are in active maintenance over time, comments should be minimized, so that the risk of obsolation and misleading is reduced.

The second influential characteristic is *logically unrelated tasks*. This relates to the common malpractice when conceptually unrelated tasks are solved in one function. This may seem to be obvious problem that can be avoided. The conceptual separation of tasks, however, is not any easy in practice, because it greatly depends on the abstraction level of thinking. Therefore logically unrelated tasks are relative to the thinking. In many cases, however, it is obvious that some tasks are easily separable. Generally such functions are distinguished by large mass of code and several independent blocks in it, solving conceptually independent tasks.

The third influential characteristic is the *incorrect indentation*. This characteristic is pure representational and 100% evitable. Understandably, this characteristic has large influence on complexity because it determines the hierarchical belonging of different blocks of code with one-glance visual contact. If indentations are misplaced, the programmer needs to carefully examine the code by checking the structural characteristics (curly braces in many programming languages). It is remarkable that Python language enforces correct indentations therefore making Python code naturally more readable. It would be sensible if the syntax of other programming languages could oblige this rule too. To our view there are no obvious impediments in doing so.

The fourth influential characteristic is the multithreading. Using many threads in a unit of code can cause major problems with reading and writing values of variables. The main problem then becomes multiple threads modifying the same memory at the same time. One can imagine that linearly increasing number of threads in a unit of code can cause exponentially increasing difficulty of understanding how exactly several threads use variables and cooperate in working with the memory. Certainly multithreading can be an advantage in solving certain tasks, but it is very rare that many threads are absolutely needed at the same time in a piece of code. Usually there are alternative solutions making it possible to keep the number of threads limited, so that the code is clearly understandable. For example creating separate instances of variables for groups of threads can be one solution.

The fifth influential characteristic is deep nesting. Nesting emerges from using many conditional statements inside one another. A code statement that is in the deepest level of nesting requires human short-term memory to remember all of the conditional statements of the higher levels in order to understand how the current statement shall operate. If the nesting level of a code unit is more than human short term memory span (about seven elementary concepts to remember) [29], working memory has to go back and forward over the conditional statements intensively in order to make sense out of it. In this case elaborative rehearsal process takes place to activate long-term memory, which consumes much more time than the sheer use of short term memory. And of course this time is the very maintenance time that a developer spends on that piece of code. Deep nesting mostly can be

avoided by refactoring the block into separate functions, combining the conditional tests, or using early returns.

The sixth influential characteristic is *global variables*. Global variables are variables that are defined outside of functions/methods and usually are accessible for different parts of code. It is easy to underestimate the consequences of using many global variables. Over time and evolution of software there is a risk that completely unrelated methods in the code can use and modify the values of these variables. This can cause both severe errors and also significant amount of time to understand and fix the errors. The complexity that emerges due to many global variables is not local; it is not confined by a specific part of code, but rather permeates into other areas of code. It is always desirable to use local instead of global variables in code. If global variables are absolutely necessary to use, than labeling conventions should be used, so that the name indicates that it is a global variable.

The seventh influential characteristic is *deep inheritance*. Depth of inheritance indicates a *class* that has a deep hierarchy of ancestors. There is a popular opinion that the base class should be defined generic and the rest of the classes should be more specific and inherit generic features of their parent class. Then, more specific child classes can belong to the parent class, and so forth. Similar to nesting problem, it is a bad idea if the hierarchy level exceeds three, and utterly bad idea if the hierarchy level exceeds seven [29]. This can cause change-prone child classes, increased coupling, and hence exploding complexity. Similar to *nesting*, inheritance-caused problems can be substantially time consuming, because of elaborative rehearsal process that a human brain has to conduct in order to figure out what exactly features a child class possesses. Similar to the previous characteristics, high level of inheritance is not an essential part of coding. Classes always can be independent from one another.

The eighth influential characteristic is too short names of methods and variables. This characteristic indicates that the names of variables and functions most likely are not meaningful. Therefore, in software maintenance it is difficult to understand their purpose. This characteristic is purely representational and nearly always avoidable, because it is always possible to create a name that characterizes a variable to a good extent.

The ninth and last of the most influential characteristics is *preprocessors*. Preprocessors are constructs that allow modifying a given input data before it can be used by another program. The most common use is the modification of the source code before it will be compiled. This allows e.g. customizing the source code syntax or extending a language. The complexity that emerges due to using many preprocessors can be explained several ways: Syntax customization in many places in a piece of code can cause substantial difficulty of understanding which parts of the code will be executed and how these parts will operate together. Another problem is that a preprocessor does not recognize scope, so it can replace code in whichever area that it is placed. Preprocessors normally can be avoided, and many programming languages do not even provide this feature. However, if necessary, they can be used with very limited amount so that different instances of them do not intervene with each other creating confusion.

The discussed nine characteristics were evaluated to have substantial influence on complexity increase: The red-orange areas of their bars in figure 2 are greater than 50% of all estimates. Also, as we discussed, it seems that the use of these characteristics can be drastically limited, if not avoided at all. Particularly, it is remarkable that the first five characteristics have substantial influence on complexity increase, and they are mostly avoidable.

The next two characteristics, succeeding the first nine, *many pointers* and *too long names* are evaluated to have significant influence too. Pointers are mostly used in low level programming languages, and allow manipulating the memory addresses of other variables. Among other uses, pointers are efficient for dereferencing, that is, to access the data that the pointer points. Using many pointers increases complexity because the addresses of variables get manipulated and therefore vulnerable. It is hard to understand the convoluted operations with variables' addresses. It is not always that is easy to avoid pointers, because they can be efficient for specific tasks, but their use can be safeguarded thereby decreasing complexity. For example higher level languages are consciously designed to avoid raw memory pointers.

Too long lines of code also have significant influence on complexity increase. This can be explained by the limited human attention span. Generally the attention span is sharp in a limited range, and decreases drastically with increasing range (Anderson [30], chapter 3). Therefore, it is not a good idea to have too long lines of code. Conventionally, well-experienced programmers keep this length within the range of a fourth and a third of a computer screen. However, this length also depends on the logical decomposability of the line. Logically independent statements usually should be in different lines, unless they belong to one logical cluster of statements.

Next influential characteristics are *many developers*, *many polymorphic functions*, and *missing comments*. Particularly *many developers* indicate that a piece of code can be changed by many simultaneously, which can be a significant source of evolutional complexity. This means that the information about a piece of code is continuously changing over time, and it is difficult to learn what exactly it does. This can also increase complexity in the architecture by impeding the efficient communication between architectural units: an explanation that we got from practitioners in our previous research. "*Frequent changes*" is similar to this characteristic.

Seven out of the last eight characteristics in the figure are essential for coding. These are: *many objects*, *many non-nested branching*, *many looping statements*, *many local variables*, *many method calls*, and *many operators*. *Arrays* can actually be regarded as a subcategory of *local variables*. These seven characteristics are essential constituents of source code. They comprise the very fabric of code and cannot be avoided or reduced. It is remarkable that they were evaluated to have insignificant influence on complexity increase. Therefore, it is sensible to infer that code does not have to be as complex as we often see in practice. The most influential characteristics are avoidable and the essential characteristics have little influence on complexity increase. Therefore, it is desirable that practitioners stick with the essential code characteristics so the complexity of code can be drastically reduced. The essence of these findings concur with the first lesson of Kernighan's and Plauger's renowned book [31] – "write clearly – don't be too clever."

6 Implications for Complexity Measurement

Effective complexity management can be carried out if it is supported by measurement. Measurement permits automation and automation permits a standardized use of measurement systems in practice. Measurement systems, in their turn, ultimately permit

practitioners managing complexity continuously, thus building simplicity into the product during development.

As we discussed earlier, the current complexity measures are created based on the existing theoretical frameworks. Juxtaposing the basis of the current complexity measures and findings of this paper, we arrive to contradictory results. For example, one of the most popular complexity measures, cyclomatic complexity [32], is roughly based on counting the number of conditional statement in code, because, in theory it is shown that with linearly increasing conditional statements, the number of unit tests needed for full testing increases exponentially. According to Figure 3, however, non-nested conditional statements are insignificant source of complexity. This contradiction can be explained with the fact that "the number of unit tests" for fully tested piece of code is not the most relevant aspect of complexity for practitioners. This is because if the software is decomposed into too small pieces then the coupling between them increases, thus increasing the number of tests in higher abstraction levels. Therefore, in order for not going into those trade-off issues, practitioners simply want understandable code, which is then testable to a reasonable degree in all levels.

Similarly, the Halstead measures of software science [33] are roughly based on counting the number of variables and operators, which according to Figure 3, have insignificant influence on complexity increase. This contradiction is probably due to the assumption that only structural aspects of complexity matters – the complexity emerging from the structural elements. But this is clearly not true, because representational [34] and evolutional [35] complexity exist in code too.

The coupling measure of Henry and Kafura [36] is based on the number of functions calls and size of code. According to Figure 3, however, function calls are insignificant source of complexity in a unit of code. Of course the coupling measure is based on more sophisticated formula, that is the product of fan-in, fan-out, and size of code, but by scrutinizing this formula deeper it can be indicated that the size factor adds purely probabilistic factor to the complexity increase. That is, larger size indicates more code to work with, and therefore more time to spend. But on the other hand larger size also provides more functionality and therefore more profit. We must also notice, however, that the measure of Henry and Kafura, or at least the product of fan-in and fan-out, can be a good indicator for architectural complexity. Architectural complexity is not in the scope of this work, but it would be interesting to find out to what extent exactly fan-in and fan-out affect architectural complexity.

One of the measures Chidamber and Kemerer [37], DIT (depth of inheritance), was evaluated to have substantial influence on complexity increase. *Deep inheritance* is actually one of the nine influential characteristics in the Figure 2. Two measures of Chidamber and Kemerer, *coupling between objects* and *response for a class*, is very similar to coupling measure of Henry and Kafura, and therefore can be investigated for architectural complexity. The remaining two measures of Chidamber and Kemerer clearly are not measures of complexity (*weighted methods per class* is a measure of size and *lack of cohesion of methods* is a measure of cohesion) and so we do not discuss them here.

Curiously, the most influential characteristics are not captured in the existing code complexity measures. There were attempts to capture them previously, for example Buse and Weimer [34] measured the effect of several representational characters such as, *line length, identifier length, comments*, etc., on code readability, Harrison and Magel [38] defined a measure based on *nesting depth*, etc. But these works never had logical continuations so that rigorously defined and tested measures could come about.

An important implication of this paper is that measures are needed for capturing the nine influential characteristics of code. Well-defined measures can permit evaluating the influence of these characteristics on code quality more objectively. Practitioners than can use measurement systems for more structured complexity management. Until then, we hope that the results of this paper will provide good insights for better management of code complexity.

7 CONCLUDING REMARKS

This paper presented the evaluation results of 24 distinct code characteristics. Their influence on complexity increase was evaluated by an online survey with 393 software practitioners worldwide. According to the cumulative experience of practitioners nine most influential characteristics substantially increase complexity. Curiously, most of these characteristics are not captured in the popular complexity measures. We discussed these characteristics one by one, indicating that all of them are avoidable in code to a great extent. Instead, seven characteristics that are essential for coding, seems to have insignificant influence in complexity increase, therefore they can be largely exploited. These results can serve as a handy guide for better code complexity management. Furthermore, these results can be used alongside theoretical frameworks for developing sophisticated complexity measures.

Acknowledgement

A year ago we published a paper in a prestigious journal of software engineering. All of the three reviewers of the paper expressed their wish to extend the study for evaluating more exhaustive list of code characteristics. We didn't get to know the reviewers, but this paper is to comply with their wish.

REFERENCES

[1] N. Fenton and J. Bieman, Software metrics: a rigorous and practical approach: CRC Press, 2014.

[2] A. Abran, Software metrics and software metrology: John Wiley & Sons, 2010.

[3] H. Zuse, "Software complexity," NY, USA: Walter de Cruyter, 1991.

[4] E. J. Weyuker, "Evaluating software complexity measures", IEEE Transactions on Software Engineering, vol. 14, pp. 1357-1365, 1988.

[5] N. F. Schneidewind, "Methodology for validating software metrics," IEEE Transactions on Software Engineering, vol. 18, pp. 410-422, 1992.

[6] B. Kitchenham, S. L. Pfleeger, and N. Fenton, "Towards a framework for software measurement validation," IEEE Transactions on Software Engineering, vol. 21, pp. 929-944, 1995.

[7] L. Briand, K. El Emam, and S. Morasca, "Theoretical and empirical validation of software product measures," International Software Engineering Research Network, Technical Report ISERN-95-03, 1995.

[8] S. Sarwar, M. Muhammd, S. Shahzad, and I. Ahmad, "Cyclomatic complexity: The nesting problem," in Eighth International Conference on Digital Information Management (ICDIM), 2013, pp. 274-279.

[9] M. Shepperd and D. C. Ince, "A critique of three metrics," Journal of Systems and Software, vol. 26, pp. 197-210, 1994.

[10] N. E. Fenton and M. Neil, "A critique of software defect prediction models," IEEE Transactions on Software Engineering, vol. 25, pp. 675-689, 1999.
[11] J. Graylin, J. E. Hale, R. K. Smith, H. David, N. A. Kraft, and W. Charles, "Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship," Journal of Software Engineering and Applications, vol. 2, p. 137, 2009.
[12] C. Kaner, "Software engineering metrics: What do they measure and how do we know?," In METRICS 2004. IEEE CS, 2004.
[13] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," IEEE Transactions on Software Engineering, vol. 18, pp. 423-433, 1992.
[14] N. E. Fenton and M. Neil, "Software metrics: successes, failures and new directions," Journal of Systems and Software, vol. 47, pp. 149-157, 1999.
[15] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," IEEE Transactions on Software Engineering, vol. 22, pp. 68-86, 1996.
[16] L. Briand, K. El Emam, and S. Morasca, "On the application of measurement theory in software engineering," Empirical Software Engineering, vol. 1, pp. 61-88, 1996.
[17] B. Kitchenham, "What's up with software metrics?–A preliminary mapping study," Journal of systems and software, vol. 83, pp. 37-51, 2010.
[18] C. Mair and M. Shepperd, "Human judgement and software metrics: vision for the future," in Proceedings of the 2nd international workshop on emerging trends in software metrics, 2011, pp. 81-84.
[19] J. McGarry, Practical software measurement: objective information for decision makers: Addison-Wesley Professional, 2002.
[20] A. Sellami and A. Abran, "The contribution of metrology concepts to understanding and clarifying a proposed framework for software measurement validation," in Proceedings of the 13th International Workshop on Software Measurement (IWSM), Montreal, Canada, 2003, pp. 18-40.
[21] International Standard ISO/IEC 15939:2001, Information technology — Software engineering — Software measurement process," 2001.
[22] International Standard ISO/IEC 25020. Software and system engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Measurement reference model and guide," International Organization for Standarization, 2007.
[23] A. Geraci, F. Katki, L. McMonegal, B. Meyer, and H. Porteous, "IEEE Standard Computer Dictionary," A Compilation of IEEE Standard Computer Glossaries. IEEE Std, vol. 610, 1991.
[24] V. Basili, "Qualitative software complexity models: A summary," Tutorial on models and methods for software management and engineering, 1980.
[25] J. Moses, "Complexity and Flexibility. Professor of Computer Science and Engineering," ed: MIT/ESD, 2001.
[26] E. Rechtin and M. W. Maier, The art of systems architecting: CRC Press, 2010.
[27] B. Edmonds, "What is Complexity?-The philosophy of complexity per se with application to some examples in evolution," The evolution of complexity, 1995.
[28] L. M. Rea and R. A. Parker, Designing and conducting survey research: A comprehensive guide: John Wiley & Sons, 2014.
[29] G. A. Miller, "The magical number seven, plus or minus two: some limits on our capacity for processing information," Psychological review, vol. 63, p. 81, 1956.
[30] J. R. Anderson, Cognitive psychology and its implications: Macmillan, 2005.
[31] B. W. Kernighan and P. J. Plauger, "The elements of programming style," The elements of programming style, by Kernighan, Brian W.; Plauger, PJ New York: McGraw-Hill, c1978., 1978.
[32] T. J. McCabe, "A complexity measure," IEEE Transactions on software Engineering, pp. 308-320, 1976.
[33] M. H. Halstead, Elements of software science vol. 7: Elsevier New York, 1977.
[34] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," IEEE Transactions on Software Engineering, vol. 36, pp. 546-558, 2010.
[35] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in Proceedings of the 30th international conference on Software engineering, 2008, pp. 181-190.
[36] S. Henry and D. Kafura, "Software structure metrics based on information flow," IEEE transactions on Software Engineering, pp. 510-518, 1981.
[37] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on software engineering, vol. 20, pp. 476-493, 1994.
[38] W. A. Harrison and K. I. Magel, "A complexity measure based on nesting level," ACM Sigplan Notices, vol. 16, pp. 63-74, 1981.