A Pragmatic View on Code Complexity Management

Vard Antinyan, University of Gothenburg, Sweden
Anna B. Sandberg, Ericsson AB, Sweden
Miroslaw Staron, University of Gothenburg, Sweden

Source code, unlike other artifacts, is structurally sophisticated, representationally abstract, and progressively versatile through time, therefore, it is considered to be among the most complex artifacts created by human. Today, many alarm about the incessant growth of complexity of the contemporary software systems [1] but there are no well-accepted strategies to assess complexity. What are the complexity assessment techniques that software engineers could employ in order to manage complexity? Complexity measures that are created so far are not much of help in making code simpler. The best they are used for is to predict maintainability and defects [2]. Hence, there is a need for complexity measures which could facilitate complexity reduction. To envision such measures one needs a meticulous understanding of how complexity is experienced by developers. This paper is an endeavor to underpin complexity understanding by scrutinizing how complexity is experienced by developers and how certain code characteristics impact complexity. We summarize by introducing new facts that should be used for designing better complexity assessment methods.

Complexity

The concept of complexity emerges from the systems theory to elucidate the difficulty of system understanding due to its elements and interconnections. Maier [3] (pp. 5-6) noticed that the definitions of complexity and system are remarkably similar.

System is a set of different elements so connected or related as to perform a unique function not performable by the elements alone

Complexity is the quality of being composed of interconnected or interwoven parts

These definitions entail that complexity is an intrinsic quality of a system. More elements and more interconnections in a system create more difficulty of understanding of how the system works. Because the difficulty of understanding emerges from increasing number of elements and interconnections, by which complexity is defined, we can state that difficulty of understanding emerges from complexity. Here we should notice that even though understanding is a purely cognitive quality, and complexity is a purely system quality, they are unshakably connected in their essence. In fact, it would be hard to describe what complexity is without considering that it is sensed and interpreted by brain. Hence, to scrutinize the impact of complexity on system development one needs a basic understanding of such functions of brain that sense information, manipulate it, and make decisions accordingly when developing the system.
Complexity in the Brain

Any information that can be sensed by brain is filtered through the help of transient sensory stores of the brain [4] (pp. 124-148). If the information is irrelevant for the brain it tends to disappear from these sensory stores quickly. If the information is important, however, it is stored in the short-term memory. Short-term memory is an intermediate memory storage system which can hold information for about 30 seconds in order to permit its use by working memory. Working memory then, which is the executive function of the brain responsible for decision making, uses this information for variety of purposes, including its rehearsal for storing it in the long-term memory. Long-term memory is a large memory storage which requires elaborate rehearsal of any information in order to store and activate it when working memory requires so.

Short-term memory has two important qualities that seem to have crucial impact on the effectiveness of processing information and making decisions. The first quality is its remarkably small capacity: In average short-term memory can hold no more than seven elementary concepts at a time. This means that the big amount of information obtained through human senses cannot be stored and managed at once. To manage the continuous flow of information into the short-term memory the working memory decomposes information into small and relatively isolated units. Since a decomposed unit is not completely independent, its linkages with the directly related units are also stored. The more isolated a unit is, the less linkages there are to remember, and therefore, the easier it is to process the information. A summary of experiments on the capacity of short-term memory was reported by Miller [5], who also observed that irrespective the subject of information the capacity remains in the same limit. The second quality is the possibility of the fast access to the information in the short-term memory. As opposed to the long-term memory, which requires continuous (and sometimes long-lasting) exercises for accessing information, short-term memory permits access to its total content in milliseconds.

Now let us observe how short-term memory, long-term memory, and working memory deal with code complexity when maintaining code. A developer always works on a small unit of code at a time, and the only information her/his short-term memory holds is a small number of elements and their direct interconnections with the rest of the code. When s/he is done with the necessary modifications of the current unit, s/he unloads her/his short-term memory from the current information and reloads it with the necessary information of the next unit of code. In all times her/his short-term memory holds only the information of the elements and their relations of a small unit of code. One can imagine that the short-term memory of the developer slides over the elementary code statements, continuously loading/unloading elements and their interconnections and making the necessary modifications, in all times holding no more than seven elementary concepts. If the code is difficult to decompose into small and well-isolated units, however, the short-term memory gets loaded and unloaded more intensively trying to provide thorough information about the current unit and its interconnections, so the working memory can obtain thorough understanding of the unit and make decisions. In this case the working memory (intendedly or unintendedly) conducts elaborative rehearsal process, which activates a part of the long-term memory and stores a larger content of relevant information there. Working memory then can access to the newly stored information in long-term memory relatively easier and manipulate it for decision making. The activation of long-term memory, however, takes substantial amount of time and effort, therefore, a developer will experience a sense of tiredness and time consumption. In summary, we can state that the more complex a unit of code is, the bigger area it requires from the long-term memory to be activated, and therefore the bigger
amount of time will be consumed on the maintenance. Two questions that seem to be relevant for understanding where complexity emerges from and how it can be reduced are:

- Which are the elementary characteristics of code that have high impact on complexity?
- Which of these characteristics are accidental and thus avoidable, and which of them are essential and thus unavoidable?

The next section answers these two questions.

**Evaluating Code Characteristics**

Brooks [6] claimed that complexity of code is essential because it is a corollary of essential structural characteristics of code. According to him, since code characteristics are essential, there is little chance to decrease complexity. On the other hand it is known that there are also representational and evolutionary characteristics of code, which are often accidental and can have major role in complexity decrease [7], [8]. Previously, when the authors of this paper were developing decision support systems for Ericsson and Volvo Group, a question often came to discussions with software engineers: What is it that makes code complex? This question was particularly important, because initially McCabe complexity measure was used ([9], pp. 391-394) for complexity assessment, but software engineers found it to be a simplistic measure. Thereafter, we discussed this question in numerous group meetings with overall twelve software engineers in the companies. Ten code characteristics were identified, which were omnipresent in code and were perceived to have distinctive impacts on complexity.

**Essential and Accidental Characteristics**

Here we describe the extent to which the ten characteristics are avoidable in the code. We postulate that the more avoidable a characteristic is the more accidental it is. The ten characteristics are presented in Table 1. The third column of the table gives explanations why and to what extent certain characteristics can be avoided.

The first two characteristics in the table seem to be strictly essential, because operators and variables are core constituents of program instructions. They are used to specify data of problem domain and implementing solutions. Their use in code has linear logic and it is of little use to decompose code units for reducing their numbers and distributing over several units.

The next two characteristics, calls and conditional statements, are mostly essential for the same reason as the former two characteristics. The extensive use of them, however, can create many logical paths or can abstract away the operations of the function, making the function difficult to comprehend. Even if decomposition can be used to avoid such a scenario, their overall number in the entire code is irreducible nonetheless.

When it comes to logically unrelated solutions it is difficult to decide to which extent solutions are logically unrelated and therefore it is difficult to understand when to avoid them by decomposition. Here there can be two extreme options: an isolated but non-cohesive function versus a group of coupled but cohesive functions. Depending on specifics of code the trade-off point between coupling and cohesion can be different.
<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
<th>Avoidable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many operators</td>
<td>Many mathematical operators in a unit of code</td>
<td>They cannot be avoided because they are essential for conducting operations</td>
</tr>
<tr>
<td>Many variables</td>
<td>Many variable declarations and use in a unit of code</td>
<td>They cannot be avoided because they are essential for task specifications and operations</td>
</tr>
<tr>
<td>Many calls</td>
<td>Many unique invocations of methods or functions in a unit of code</td>
<td>They mostly cannot be avoided because they are essential for code modularity. The use of decomposition, however, allows to avoid many complex functions in a unit of code.</td>
</tr>
<tr>
<td>Many conditional statements</td>
<td>Many conditional statements in a unit of code (e.g. 'if', 'while', 'for', etc.)</td>
<td>They mostly cannot be avoided because they are essential for executing different scenarios under different conditions. The use of decomposition, however, allows to avoid the extensive use of them in a unit of code.</td>
</tr>
<tr>
<td>Many solutions</td>
<td>Many logically independent solutions that are jointly solved in a unit of code (cohesion)</td>
<td>This may or may not be needed to avoid. Too little decomposition increases the internal complexity of functions, and too much decomposition increases the outbound complexity of functions.</td>
</tr>
<tr>
<td>Many changes</td>
<td>Many modifications made in a unit of code in a specified period of time</td>
<td>They may be avoided by maintaining well-modularized architecture and stable code. Many changes, however, can be a corollary of active maintenance activities.</td>
</tr>
<tr>
<td>Many developers</td>
<td>Many developers that make simultaneous changes in a given unit of code in a specified period of time</td>
<td>They usually can be avoided by using well-defined task distribution and well-modularized code architecture</td>
</tr>
<tr>
<td>Many misleading comments</td>
<td>Many misleading or obsoleted comments in a unit of code</td>
<td>They usually can be avoided by keeping the comments brief and updating them with the changing code</td>
</tr>
<tr>
<td>Deep nesting</td>
<td>Deep nesting level of conditional statements in a unit of code</td>
<td>This mostly can be avoided by refactoring the block into separate functions, combining the conditional tests, or using early returns.</td>
</tr>
<tr>
<td>Lack of structure</td>
<td>Incorrect indentations, misleading naming, and lengthy lines</td>
<td>These can always be avoided by using clean coding rules: these factors are representational and thus are not essential for code execution.</td>
</tr>
</tbody>
</table>
Without *changing code* it is not possible to conduct software maintenance or development, however, too frequent changes may indicate rather unstable code. It is usually possible to avoid too frequent changes by having well-modularized architectural design, which will prevent change waves across different code areas.

If the same unit of code is modified by *many developers* simultaneously, each of the developers needs to understand how his/her own changes are related to changes made by his/her colleagues. It is usually possible to avoid many developers making changes in the same code by implementing modular architecture and well-thought task distribution.

The next characteristic is *misleading comments*, which is possible to avoid by attentive commenting and updating comments over development time, so they do not become obsolete and thus misleading. Many software engineers in industry are inclined to believe that comments should be embedded in the names of variables and functions, so the risk of obsoleted comments can be reduced. In intensive maintenance times, however, some code areas can be changed fast and repeatedly, making the updates of comments a challenging task.

*Deep nesting* is commonly recognized as a negative characteristic, because it requires a simultaneous understanding of several conditions at once. Deep nesting can always be avoided by function calls, merging several conditions with Boolean operators, using “return”, “break”, and “continue” or equivalent operators, which will make the logical flow more linear.

*Lack of structure* of code can always be avoided by using meaningful variable and function names, correct indentations, and keeping length of code lines in an optimal limit. The names must not be too short in order to deliver its meaning as comprehensively as possible, and they should not be too long in order not to contain several units of information in one name [10].

---

**The Influence on Code Complexity**

Evaluating the impact of code characteristics on complexity is valuable because it will indicate whether code complexity reduction is a pragmatic task. In order to evaluate this impact we conducted an online survey with software engineers of seven large software development companies and two universities. Because software engineers directly work with code and develop variety of software, their collective perception could reveal profound knowledge on how code characteristics impact complexity. Seven companies were Axis Communications, Ericsson, Grundfos, Jeppesen, SAAB Defense and Security, Volvo Group Global, and Volvo Car Group. Two universities were University of Gothenburg and University of Chalmers. Eighty-nine participants from the companies and eleven participants from the universities responded to the survey. The choice of the companies was based on our previous collaborations with them, which permitted us to know their software and concerns of complexity management. The authors developed the survey questions and improved them by conducting two pilot studies with nine software engineers. As a result each of the survey questions was organized into six ordinal Likert scale values with an additional “not answered” option. Each question was targeted to find out the influence of a specified characteristic on code complexity as perceived by the respondents. For example, the question specific for *variables* was formulated so: *Generally many variables in a unit of code make that code....* The respondents were asked to fill in the ellipses by one of the following seven options:

- No complex
The rest of the questions were organized in a similar fashion.

Figure 1 presents the results of the survey. The vertical axis represents the number of answers and the horizontal axis represents the characteristics. The number of answers per ordinal value is color coded, so we can see which ordinal values are high for a given characteristic. Additionally the statistical modes per characteristic are emphasized by specifying the number of respondents on the color that represents the mode. The accidental characteristics are positioned towards the left side and the essential characteristics are positioned towards the right side of the horizontal axis.

![Figure 1: Code characteristics: influence on complexity and assortment](image)

The most important result that the figure shows is that **according to software engineers’ knowledge and experience accidental code characteristics have substantially higher impact on complexity than the essential ones.** Particularly **lack of structure and nesting depth** seems to have decisive impact on complexity. Their modes are **very complex** (36 answers) and **quite complex** (37 answers). Overall, more than 80% of the respondents selected the highest three values of Likert scale for these characteristics (red-orange area on the bars). Oppositely, three essential characteristics of code – *calls, variables,* and *operators* – are considered to have little impact on complexity. The only essential code characteristic that is considered to have significant impact is **conditional statements.** However, *conditional statements* can be used nested or not nested, and this consideration was not specified in the survey question, therefore its influence might be overestimated if the nesting factor is excluded. Additionally, two rather accidental characteristics, *many developers* and *misleading comments,* are also considered to have
significant impact. Overall the red-orange area of the left half of the figure is multiple times bigger than that of the right half. Hence, it seems that if the proposed accidental characteristics are managed in code, then a large part of complexity can be dissolved away.

**Complexity Measures**

The main goal of a complexity measure is to identify complex units of code and indicate the cause of high complexity, so the developers can reengineer and simplify them. The most popular complexity measures, however, have not been successful in fulfilling this goal [2]. Two major problems with this lack of success are presented in the next two subsections.

*The problem of measuring accidental code characteristics*

The first problem has to do with how well complexity measures quantify accidental code characteristics. The first row of Table 2 presents the ten code characteristics discussed in this article. The first column of Table 2 illustrates ten software measures. The first eight of the measures are probably the most popular measures that researchers and software engineers have been using so far: size ([9] pp. 339-344), change (e. g. [11]), coupling – fan-in, fan-out, and the measures of Henry and Kafura ([9] pp. 410-412), McCabe cyclomatic complexity ([9] pp. 391-394), object oriented measures of Chidamber and Kemerer ([9] pp. 421-422), and Halstead measures of software science ([9] pp. 344-346). These measures have often been evaluated for defect prediction and maintainability assessment, therefore many standard software measurement textbooks include them.

Table 2 presents the following information about these measures:

- If a measure fully quantifies a given characteristic, then in their intersection cell a green tick is put
- If a measure partly quantifies a given characteristic then an exclamation mark is put
- If a measure does not quantify a given characteristic then the intersection cell is left empty

What is interesting in the table, its left side does not contain any green tick, indicating that the most popular measures do not quantify the proposed accidental characteristics. The last two measures, which are designed by Buse and Weimer [8] and Harrison and Magel [7] conformably, are included to show that there have been endeavors to measure such accidental characteristics as *lack of structure* and *nesting*. The created measures, however, did not fully quantify these characteristics. Measurement of *nesting* probably is difficult because it is difficult to define the measurement entity. Traditional measurement entities are functions, files, and components, for which most of the measures are defined. Measuring nesting for such entities, however, is not straightforward. As for the *lack of structure*, it is a more sophisticated characteristic, defined by the length of identifiers, length of lines, indentations, etc., therefore, it is difficult to quantify it accurately.
Overall the results of Table 2 is worrisome, because the most popular measures poorly quantify the proposed accidental code characteristics, even though these characteristics seem to have multifold greater impact on complexity than the essential ones. Most of the measures quantify the essential characteristics, which neither have big impact on complexity nor are reducible due to their essentiality. Here it becomes clear why it is unlikely that using these measures could help software engineers to design simpler software.

**The problem of size-based measures**

The second problem is the fact that most of the existing measures have mixed nature, indicating both complexity and size at the same time. Size is an essential code characteristic, because generally the bigger size indicates more functionality of software, and therefore it cannot be reduced. Moreover, size cannot distinguish complex units of code.

When closely observing the measures of Table 2, one can understand that most of them are mixed measures. In their essence they are based on counting occurrences of characteristics, which makes them also size measures. For example, McCabe complexity measure is based on counting the number of conditional statements. Halstead measures are based on counting the number of operators and operands. Henry and Kafura measure is based on counting in and out invocations (fan-in and fan-out) and lines of code, a measure of Chidamber and Kemerer is based on the number of methods in a class. While these measures are not as simplistic as size measures, by definition, however, they contain size factor in them. Increasing number of invocations, conditional statements, etc., indicates more increasing size than complexity of code. Furthermore, with increasing size, a unit of code will not be considered as an elementary unit for short-term memory. Such a non-elementary unit will be decomposed into smaller units by working memory automatically, so the short-term memory can hold every small unit once at a time. A typical example can be a large function containing many independent non-nested “if” and “else” statements. Even though this function has a big McCabe complexity number, each of...
its conditional statements is logically separable and easily modifiable. Bigger numbers of these measures most of the time indicate bigger functions or files, which by no means have to contain complex units of code.

There are only few measures which are size-independent and also indicate pure complexity. Examples are nesting depth of blocks ([9], pp. 381-383), depth of inheritance, which is one of the measures of Chidamber and Kemerer, and the length of names of functions and variables [8].

- **Nesting depth** indicates the level of nesting of conditional statements. A statement that is in a lower level of nesting requires 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 the limit of short term memory (about seven), working memory has to go back and forward over the conditions intensively in order to make sense out of it. In this case long-term memory has to be activated to help out with the task.

- **Depth of inheritance** indicates the hierarchy level of inheritance for classes. A class that is in the lowest level of hierarchy enforces short-term memory to remember its dependences on the ancestor classes, so modification can be done. Similar to nesting depth, the increasing number of ancestor classes will increase the involvement of long-term memory in the maintenance process.

- **The length of a variable’s name** directly influences the representation of that variable. If the name is too short short-term memory cannot obtain conclusive information about the variable. Oppositely, if the name is too long, short-term memory will perceive it as several elementary concepts in one name.

We shall emphasize that thus far size-based measures orchestrate code complexity measurement discipline, probably because they are easier to design. And unfortunately the necessity of size-independent complexity measures has not been emphasized enough in the literature, to prompt their development.

**Concluding Remarks**

Two key conclusions of this study are:

- Among the evaluated ten characteristics, those that influence complexity the most are more of accidental characteristics and therefore are manageable
- Most of the well-known complexity measures are based on essential code characteristics: that is why they are of little help in simplifying source code

How can we significantly improve current practices of code complexity management? Two follow-up steps are straightforward:

- Evaluate a more exhaustive list of code characteristics so the entire source of complexity can be well-understood
- Start designing measures which are size-independent and can quantify accidental code characteristics. A good start can be by looking into nesting, depth of inheritance, naming conventions, incorrect indentations, obsoleted comments, and the number of designers who change the same code simultaneously
Keywords
Software development, Software engineering, Software metrics, Complexity, Technical Debt

Bios
Vard Antinyan is a PhD candidate in Software Engineering at the University of Gothenburg, Sweden. His research includes software measurement, software complexity and quality. Contact him at vard.antinyan@cse.gu.se

Anna Börjesson Sandberg is an Associated Professor connected to the University of Gothenburg, Sweden. Her main focus is to drive software change and improve productivity in the businesses she works in. Anna is a member of the IEEE. Contact her at anna70sandberg@gmail.com

Miroslaw Staron is an Associate Professor in Software Engineering at the University of Gothenburg. His research includes measurement and decision support in software engineering. Contact him at miroslaw.staron@gu.se

Photos
To be sent on request

References