Empir Software Eng (2017) 22:3057–3087
DOI 10.1007/s10664-017-9508-2

Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time

Vard Antinyan[1] · **Miroslaw Staron**[1] · **Anna Sandberg**[1]

Published online: 9 March 2017

Abstract Code complexity has been studied intensively over the past decades because it is a quintessential characterizer of code's internal quality. Previously, much emphasis has been put on creating code complexity measures and applying these measures in practical contexts. To date, most measures are created based on theoretical frameworks, which determine the expected properties that a code complexity measure should fulfil. Fulfilling the necessary properties, however, does not guarantee that the measure characterizes the code complexity that is experienced by software engineers. Subsequently, code complexity measures often turn out to provide rather superficial insights into code complexity. This paper supports the discipline of code complexity measurement by providing empirical insights into the code characteristics that trigger complexity, the use of code complexity measures in industry, and the influence of code complexity on maintenance time. Results of an online survey, conducted in seven companies and two universities with a total of 100 respondents, show that among several code characteristics, two substantially increase code complexity, which subsequently have a major influence on the maintenance time of code. Notably, existing code complexity measures are poorly used in industry.

Keywords Complexity · Measure · Survey · Internal quality · Maintainability

Communicated by: Nachiappan Nagappan

✉ Vard Antinyan
vard.antinyan@cse.gu.se

Miroslaw Staron
miroslaw.staron@cse.gu.se

Anna Sandberg
anna70sandberg@gmail.com

[1] Division of Software Engineering, University of Gothenburg, Hörselgången 11, 41756, Gothenburg, Sweden

1 Introduction

The internal quality of software influences the ability of software engineers to progress software development. A major aspect of internal quality is the code complexity, which directly affects the maintainability and defect proneness of code (Banker et al. 1993; Subramanian et al. 2006). Therefore, research interest on the topic of code complexity has been high over the years. Particularly, code complexity measures have been designed to measure and monitor complexity in practice (Zuse 1991; Abran 2010; Fenton and Bieman 2014). Complexity measurement allows quantifying complexity and understanding its effect on maintainability and defect proneness. The concept of code complexity, however, is not an atomic concept, so it is difficult to design a single measure that quantifies code complexity thoroughly. Instead, several complementary measures are designed to measure different aspects of complexity. Consequently, the insight that is provided by this combination of measures is expected to provide a fair assessment of the magnitude of complexity for a given piece of code.

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 (Weyuker 1988; Schneidewind 1992; Briand et al. 1995; Kitchenham et al. 1995). Theoretical frameworks for creating measures are justifiably necessary because they propose a common foundation upon which complexity measures should be designed. Nevertheless, we believe that theoretical frameworks alone are unsatisfactory since detailed knowledge supporting the design of measures needs to be extracted from empirical data. Specifically, to design a complexity measure one must:

- Identify specific code characteristics that should be considered for complexity measurement.
- Understand whether these characteristics are actually measurable in practice.
- Evaluate the contribution of these characteristics to complexity increase.
- Observe existing complexity measures and determine how well they capture code characteristics that influence complexity.
- Evaluate the usefulness and popularity of existing complexity measures in practice.
- Assess the influence of complexity on code maintenance time.

Since these factors are not addressed fully in the design of complexity measures, existing measures are usually perceived as being only moderately accurate in complexity measurement. A typical example of this is when two source code functions have the same cyclomatic complexity value, but intuitively we understand that one of the functions is more complex because, for example, it has more nested blocks (Sarwar et al. 2013). These kinds of issues are apparent in many well-recognized complexity measures and have been discussed previously (Shepperd and Ince 1994; Fenton and Neil 1999a; Kaner 2004; Graylin et al. 2009). In practice, certain modules of code are perceived to be intrinsically more complex and, therefore, more difficult to maintain despite their relatively small size (Munson and Khoshgoftaar 1992; Fenton and Neil 1999b)).

We believe that the aforementioned knowledge required for designing measures can be partially or fully answered if we consider the collective viewpoint of software engineers, which can provide an insightful contribution for academics when designing complexity measures and measurement-based prediction models.

The aim of this study, therefore, was to acquire such knowledge using the following five research questions (RQ):

RQ 1: Which code characteristics are perceived by software engineers to be the main triggers of complexity?
RQ 2: How frequently are complexity measures used in practice?
RQ 3: How strongly do software engineers perceive complexity to influence code quality?
RQ 4: How much does complexity affect maintenance time?
RQ 5: Do the responses to RQ 1 to RQ 4 depend on the demographics of respondents?

Here, we present the evaluation results of code characteristics as complexity triggers, the extent to which complexity measures are used in industry, and the evaluation results of complexity influence on internal quality and maintenance time based on a survey of 100 software engineers from Scandinavian software industry. The survey included both structured and open questions that allowed for comments. In summary, the results of the five questions (RQ1–RQ5) showed that:

1) Of the eleven proposed code characteristics, only two markedly influence complexity growth: *the nesting depth* and *the lack of structure*
2) None of the suggested nine popular complexity measures are actively used in practice. Size and change measures as forms of complexity measures are used relatively more often, although not for complexity or quality assessment purposes
3) Complexity is perceived to have strong negative influence on aspects of internal quality, such as readability, understandability and modifiability of code
4) The statistical mode (most likely value) of the software engineers' assessment indicates that complex code requires 250−500 % more maintenance time than simple code of the same size
5) The demographics of the respondents do not influence the results of RQ 1–RQ 4.

These results suggest that managing complexity has a crucial role in increasing product quality and decreasing maintenance time. Moreover, the results provide insight as to which code characteristics should be considered in code complexity measurement and management. Importantly, however, the reasons why existing complexity measures are not actively used in complexity management activities need thorough investigation.

2 The Landscape 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, as no generally accepted definition of complexity that would facilitate its measurement exists, every discipline has its own rough understanding on how to quantify complexity.

C*ode complexity*, the subject of this study, is not an exhaustively defined concept either. In the IEEE standard computer dictionary (Geraci et al. 1991), 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". According to Zuse (1991), the true meaning of code complexity is the difficulty in understanding, changing, and maintaining code. Fenton and Bieman (2014) view code complexity as the resources spent on developing (or maintaining) a solution for a given task. Similarly, Basili (1980) defines code complexity as a measure of the resources allocated by a system or human while interacting with a piece of software to perform a given task. An understanding of how to measure complexity and make code less complex is not facilitated by these definitions because they focus on the effects of complexity, i.e.,

the time and/or resources spent or experienced difficulty, and thus do not capture essence of complexity. Briand et al. (1996) 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 origin of complexity.

To outline a landscape of the sources of code complexity that would facilitate the design of the survey questions and the interpretation of the results, we chose two general definitions of system complexity that consider complexity to be an intrinsic attribute of a system, and then we adopted it to the code. The first definition is provided by Moses (2001), who defines complexity as "an emergent property of a system due to its many *elements* and *interconnections*". The second definition is provided by Rechtin and Maier (2010), 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 directly influence code complexity and thus complexity measurement. Based on these two definitions, we can imply three things: (i) The more elements and interconnections the code contains, the more complex the code; (ii) Since the elements and interconnections always have some kind of representation (for reading, understanding and interpreting), the complexity depends on this representational clarity; and (iii) 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, function calls, etc.). The interconnections of elements can be expressed by mathematical operators (e.g., addition, division, multiplication, etc.), 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 Complexity arises from unclear representation of the code. This is concerned with how clearly the elements and interconnections are presented to demonstrate their intended function. 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.

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 (Section 3.2).

These three sources of complexity comply with the definitions of Moses (2001) and Rechtin and Maier (2010), and are directly observable on the code, hence our term "direct sources of complexity". In addition, there are several other, indirect sources of complexity, such as those described by Mens (2012). These are not directly visible in the source code, although they somewhat influence complexity. Examples include the:

- Complexity of the problem to be solved by the programme
- Selected design solution for the given problem
- Selected architectural solution for the given problem
- Complexity of the organization where the code is developed
- Programming language
- Knowledge of developers in programming
- Quality of the communication between developers and development teams
- Managerial decisions
- Domain of development.

In this study, we did not consider the indirect sources of complexity, their measures and influence on the internal quality; this requires additional study.

In summary, we perceive complexity to be an emergent property of code that is magnified by the addition of more elements and/or interconnections, changing the existing elements and interconnections, or not clearly specifying the function of existing elements. We consider that the origin of code complexity is outlined primarily by the three aforementioned sources. Since the other factors are not direct sources of complexity, they should not be included in the landscape of code complexity sources.

3 Research Design

To address the research questions (RQ 1–RQ 5), we conducted an online survey (Rea and Parker 2014) with software engineers in Industry and Academia (see: https://goo.gl/forms/kaKYQ5VTfEzOE5o93). Most data were collected using structured questions of which there were 25 in total organized by six-point Likert scales. An even number for the scale values avoided a scale mid-point, thereby ensuring that respondents could choose a higher or lower estimate than average. Additionally, three open questions allowed respondents to add choices that might otherwise have been missed in the structured questions. The survey consisted of five logical parts:

- Part 1: Identified the demographics of participants
- Part 2: Estimated the extent to which different code characteristics make code complex
- Part 3: Evaluated the influence of complexity on internal code quality attributes
- Part 4: Evaluated the most commonly used complexity measures in Industry
- Part 5: Assessed the influence of complexity on development time

Survey participants were software engineers from seven large software development organizations and two universities, all of which were a part of a research consortium called Software Centre. The seven collaborating companies were: Ericsson, Volvo Group, Volvo Car Group, Saab and Axis Communication all from Sweden, Grundfos from Denmark, and the Swedish branch of Jeppesen in the United States. The two universities were University of Gothenburg and Chalmers University of Technology. The companies represented a variety of market sectors, namely telecommunication equipment, trucks and cars, the air defence system, video surveillance system, liquid pumps, and the air traffic management system. All of these systems used both small and large software products, which had been developed using different development processes, such as Lean, Agile, and V-model. Complexity was an actively discussed topic in these companies so many software engineers were motivated to participate in the survey. Since we were involved in previous research with these companies and knew the products and development organizations, we found this selection of companies rational from the perspective of construct validity.

We shared the online address of the survey with the collaborating managers or organizational leaders in the companies, who then distributed the survey within their corresponding software development organizations, targeting software engineers who worked intensively with software development. Our objective was to collect at least 100 responses in order to be able to reason in terms of percentage so that 1 answer is less than or equal to 1 %. One initial request and one reminder were sent to prompt a response from the participants. In total, however, 89 responses were received from the companies. In addition, 11 responses were received from the two universities. We selected university respondents who worked in close collaboration with software companies and had developed software products themselves earlier in their careers. In contrast to the companies, the survey link was distributed in universities directly to potential respondents. The response rate was estimated by counting the number of potential respondents who received the survey link from corporate contacts and directly from us. Approximately 280 people received the survey link, 100 of whom responded, resulting in a response rate of approximately 36 %.

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 Ericsson and University of Gothenburg was also used to improve the survey and the choice of assessment scales. 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.

3.1 Demographics and the Related Questions

The first part of the survey investigated the participant demographics. Five fields were given for information related to demographics, as presented with the specified options in Table 1.

Data for the five fields were collected using the following five statements:

1. Select your education
2. Select your job title
3. Select your domain
4. Select the years of experience that you have in software development
5. Select the programming languages that you usually work with.

Table 1 Specified fields and options for acquiring demographic data

Education	Job Title	Domain	Experience	Programming Language
Computer Science (31)	Developer (49)	Telecom (30)	<3 years (10)	Python / Ruby (30)
Software Engineering (37)	Tester (5)	Automotive (23)	3−5 years (11)	Java / C# (43)
Information systems, Informatics (7)	Architect (13)	Defence (10)	6−10 years (20)	C++ (42)
Computer Engineering (11)	Team leader, Scrum master (14)	Enterprise Systems (14)	11–15 years (20)	C (57)
Management (2)	Product owner (2)	Web Development (2)	>15 years (40)	JavaScript/PHP (15)
Economics (0)	Project manager (1)	Health Care (0)		Perl / Haskell (10)
Electrical, Electronic Engineering (38)	Researcher (12)	Academia (11)		TTCN / Tcl / Shell (11)
Other (12)	Other (4)	Other (16)		Other (21)

In the cases of "Job Title" and "Experience", options were given by radio buttons with a "one-choice-only" option. Checkboxes were specified for all other fields to enable respondents to select more than one option. In Table 1, the number of responses obtained per demographical category is shown in brackets. In addition, graphical representations of these results can be found in Section 4.1.

3.2 Selected Code Characteristics as Complexity Triggers

The second part of the survey concerned code characteristics with the objective of understanding the extent to which each code characteristic increases code complexity. We proposed eleven code characteristics that can potentially increase code complexity based on our previous study conducted with Ericsson and Volvo Group (Antinyan et al. 2014a). In that study, we were designing code complexity measurement systems for these companies in which approximately 20 software engineers were involved. Based on regular discussions on topics like the origin of complexity and which code characteristics are usually considered in complexity measurement, we determined the eleven 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.

Nine of the code characteristics are easily observable in the code. Although two of the characteristics—"complex requirement specification" and "many developers"—are difficult to observe in the code, they still directly influence complexity:

1) Many requirements in industry are written in a very detailed manner, such as pseudocode or detail diagram. Such detail specifications do not allow developers to consider the design of the code, but merely translate the specification into a programming language so the specification complexity is largely transferred into the code.
2) Many developers who make changes on the same piece of code add a new dimension on the *code change* as a type of complexity. The information needed to learn about the change in this case comes from multiple developers.

Table 2 Code characteristics and descriptions

Three sources of complexity	11 Code Characteristics	Description of the Characteristic
Elements and interconnections	Many operators	All mathematical operators (e.g., =, +, -, /, mod, sqrt)
	Many variables	Both local and global variables in the code
	Many control statements	Control statements in the code (e.g., "if", "while", "for", etc.)
	Many calls	All unique invocations of methods or functions in the code
	Big nesting depth	The code is nested if there are many code-blocks inside one another
	Multiple tasks	Logically independent tasks that are solved in one code unit
	Complex requirement specification	This relates to detail requirement specification that the developers use to design software
Representational clarity	Lack of structure	This relates to correct indentations, proper naming and using the same style of coding for similar patterns of code
	Improper or not existing comments	This relates to code that does not have any comments or the existing ones are misleading
Intensity of 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

To investigate the effect of these eleven characteristics on code complexity, one statement (question) per characteristic was formulated to be answered using the specified Likert scale. For example, the statement for function calls is shown Fig. 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 in Fig. 1. In most of the statements, we intentionally emphasized that "*many*

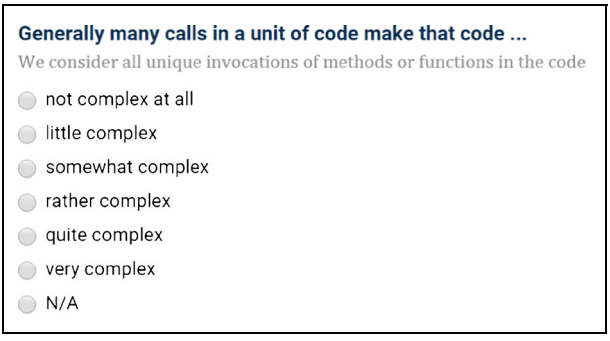


Fig. 1 Example of a question regarding a given code characteristic

Table 3 Internal quality attributes and descriptions

Internal code quality attributes	Explanation
Readability	The visual clarity of code that determines the ease for reading the code
Understandability	The conceptual clarity and soundness of code that ease the process of understanding the code
Modifiability	The logical soundness and independence of code that determine the ease of modifying the code
Ease of integration	The ease of merging a piece of code to a code development branch or to the whole product

of something" makes code complex, i.e., *many* operators, *many* variables, *many* control statements, etc. In other statements, we used different methods of framing, for example, the lack of structure, the frequent changes, etc. At the end of this part of the survey, an open question was included to allow respondents to suggest other code characteristics that they believed could significantly increase complexity.

3.3 Complexity and Internal Code Quality Attributes

The third part of the survey was designed to acquire information on the extent to which code complexity influences internal code quality attributes that are directly experienced by software engineers. Note that by internal code quality attributes we do not mean the emergent properties of code, such as size, length, cohesion, coupling and complexity itself, but the quality attributes that arise from the relationship between the intrinsic properties of code and cognitive capabilities of engineers, namely readability, understandability and modifiability. We added "ease of integration" to these three attributes, however, since we consider it also plays an important role in code development and can be influenced by complexity. The four internal code quality attributes and their brief descriptions are shown in Table 3.

The questions were organized to have six possible values of the Likert Scale plus an additional option to allow a "no answer" option. One of the four questions is shown in Fig. 2 and depicts the organization of the rest of the questions.

In this section, only selected internal code quality attributes concerned with cognitive capabilities of the engineers working with the code were covered. Other internal code quality attributes, such as error-proneness or testability were not considered in this study because they are not directly experienced by software engineers when working with the code.

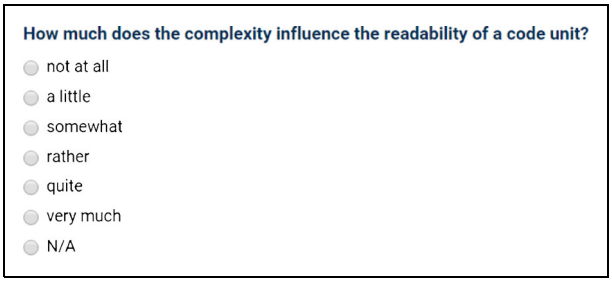


Fig. 2 Example of a question regarding the influence of complexity on internal quality

Table 4 Selected measures and descriptions

Name of the Measure	Description
McCabe's cyclomatic complexity (1976)	The number of linearly independent paths in the control flow graph of code. This can be calculated by counting the number of control statements in the code
Halstead measures (1977)	Seven measures completely based on the number of operators and operands
Fan-out (Henry and Kafura 1981)	The number of unique invocations found in a given function
Fan-in (Henry and Kafura 1981)	The number of calls of a given function elsewhere in the code
Coupling measures of (Henry and Kafura 1981)	Based on size, fan-in, and fan-out
Chidamber and Kemerer OO measures (1994)	Inheritance level and several size measures for class
Size measures	Lines of code, number of statements, etc.
Change measures, e. g., Antinyan et al. (2014b)	Number of revisions, number of developers, etc.
Readability measures, e. g., Tenny (1988), Buse and Weimer (2010)	Line length, indentations, length of identifiers, etc.

3.4 Selected Complexity Measures

The fourth survey section investigated the use of the most actively investigated complexity measures from the literature. Measures were selected based on their popularity in the literature, and particularly how often they are used for maintainability assessment and defect predictions. The measures and their descriptions are shown in Table 4.

To acquire information on the use of the measures, the frequency of use was assessed using a Likert Scale; an example of these questions is shown in Fig. 3.

The last option in this "multiple choice" question was "never heard of it", which essentially differs from that of "never used it" because in the former case, the reason why the measure is not used differs substantially from the latter. If a respondent selects "never heard of it", this implies that no conclusion can be made on whether the measure is useful or not. In contrast, if a respondent answers "never used it" this can indicate a problem with the measure itself. An additional field was included at the end of this section that allowed respondents to add more complexity measures, which they used, but was not included in our list.

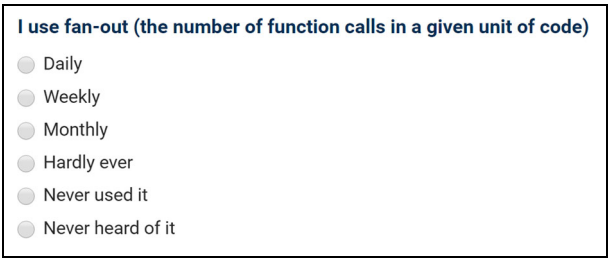


Fig. 3 Example of a question regarding the use of measures

3.5 Complexity and Maintenance Time

The fifth part of the survey concerned the influence of the code complexity on code maintenance time. Here the objective was to obtain quantitative information on how much time is spent unnecessarily on maintaining complex code. The quantitative information was based purely on a perceptive estimation of respondents; therefore, we expected the summary of the answers to be a rough estimation. The only question posed in this section of the survey is shown in Fig. 4.

The question assumes that two pieces of code of the same size can differ significantly in complexity. The respondents were expected to estimate the additional time required to maintain a piece of complex code compared to the maintenance time of simple code of the same size. The answer was not expected to be based on any quantitative estimation, but rather on the knowledge and experience of respondents. At the end of this question, a field for free comments on respondents' thought processes when making the estimates was added.

3.6 Data Analysis Methods

Data was analysed using descriptive statistics and visualizations. As regards descriptive statistics, percentages and statistical modes were used, whilst visualizations included tables and bar charts to summarize data related to the code characteristic, the use of complexity measures and the effect of complexity on the internal quality of code. Colour-coded bars were used to enhance graph readability. Pie chart was used to visualize the complexity influence on maintenance time. The fields that had been specified for free text were analysed by classifying answers into similar categories. As regards the code characteristics, the number of respondents who proposed a specific characteristic to be a significant complexity trigger was counted. With respect to measures, the number of respondents who mentioned a specific complexity measure that was not included in our list was counted. Respondents, who listed specific tactics for assessing complexity influence on maintenance time, was done by listing the tactics used by respondents for their assessment, as well as counting the number of respondents per proposed tactic.

In addition to the aforementioned analysis, cross-sectional data analysis was also conducted to investigate whether the demographics of the respondents significantly influenced the results. We hypothesized that demographics do not influence the results and conducted

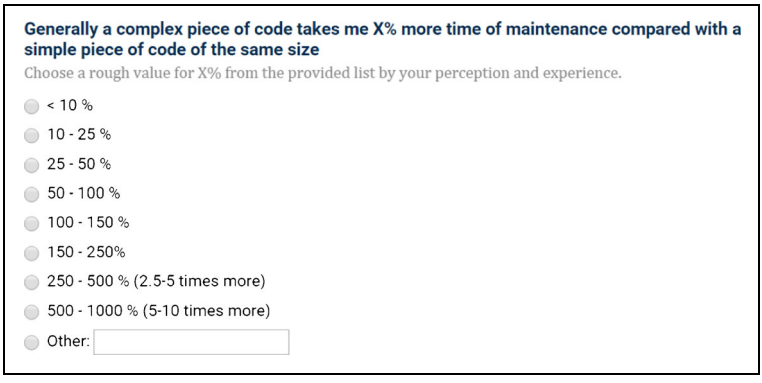


Fig. 4 Question investigating the influence of complexity on maintenance time

statistical tests to either reject or confirm this hypothesis. Since the number of responses was only 100, it was not possible to divide the data into many groups to obtain meaningful results because some groups had too few data point for meaningful statistical analysis. Data, therefore, were divided into fewer groups for such analyses.

Table 5 shows all the possibilities for cross-sectional data analysis. The first row depicts the four main categories of data. The first column shows the five units of demographics. Every cell of the table indicates whether cross-sectional analysis for a pair of "demographical data" and "category of result" was conducted in this study.

Three of the survey questions in the Demographics Section were specified by checkboxes. These questions concerned *education, domain* and *programming language.* Since these were specified by checkboxes, one respondent could select several choices concurrently, such that a statistical test to analyse the effect of demographics on the results could not be conducted. The results concerning*complexity measures* and *complexity influence on code quality attributes* were so polarized over the categorical values that it was not possible to do any cross-sectional data analysis for these two categories either. The remaining four cells of Table 5, however, show the four pieces of cross-sectional analysis that were done. Methods for each of the analyses are presented in the following subsections.

3.6.1 Evaluating the Association Between Job Type and Assessment of Code Characteristics

Here, the objective was to understand whether the type of job has any association with the assessment results of code characteristics. Therefore, the type of job was divided into two groups, developers and non-developers. Developers were respondents who marked their role as "developer" in the survey, whilst the non-developers were those respondents who marked any role other than "developer". Hence, the variable *type of job* has two possible categorical values: developer and non-developer. This division of jobs is motivated by the fact that developers work directly with the code and they themselves influence code complexity, whereas non-developers are only influenced by the code complexity. Therefore, we expected a statistical difference between these two groups. Similarly, we derived two values

Table 5 Cross-sectional data analysis table

		Four categories of results			
		Complexity characteristics	Complexity influence on quality attributes	The use of complexity measures	The influence of complexity on maintenance time
Demographics	Education	✗	✗	✗	✗
	Job	✓	✗	✗	✓
	Domain	✗	✗	✗	✗
	Experience	✓	✗	✗	✓
	Programming language	✗	✗	✗	✗

Table 6 Original six values and derived two values of "assessment" for code characteristic

Original values	Derived values
Not complex at all	Little influence
Little complex	
Somewhat complex	
Rather complex	Much influence
Quite complex	
Very complex	

of "assessment" for code characteristic from the original six values. The original six values and the derived two values are presented in Table 6.

The two values for both "type of job" and "assessment" allowed us to develop a contingency table based on which Chi-Square test was conducted to determine whether there was a statistical difference in assessment of code characteristics by people with different jobs. An example is given for *nesting depth* in Table 7.

Because the variables have categorical values, the Chi-Square test was used to assess whether the type of job and assessed influence were associated. To perform this analysis for all eleven code characteristics, eleven tables similar to that of Table 7 were developed.

3.6.2 Evaluating the Association Between Experience and Assessment of Code Characteristics

Two values of "experience" from the original five values were derived to conduct this analysis (Table 8), namely, "much experience" and "little experience". The analysis was conducted in exactly the same way as in Section 3.6.1.

3.6.3 Evaluating the Association Between Type of Job and Assessment of Complexity Influence on Maintenance Time

Two values for "type of job" (*developer* and *non-developer*) were used to analyse the effect of complexity on maintenance time. Originally, the variable that showed the assessment values of complexity influence on maintenance time had eight categorical values, but to ensure sufficient data points for a meaningful Chi-Square test, three categorical values were derived from these eight values. Three values were chosen because reducing eight values into two might lose too much information. The original eight values and the three derived values are shown in Table 9.

Based on the three values of "assessment" in Table 9 and two values of "type of job", a contingency table was made (Table 10). Using this table, a Chi-Square test was performed to determine whether there was a significant difference between "assessment" and "type of job".

Table 7 Contingency table for "type of job" and "assessment of code" characteristics

	Developer	Non-developer
Little influence	9	7
Much influence	42	41

Table 8 The original five-scale assessment and the derived two-scale assessment

Original values	Derived values
<3 years	Little experience
3–5 years	
6–10 years	Much experience
11–15 years	
>15 years	

3.6.4 Evaluating the Association Between Experience and Assessment of Complexity Influence on Maintenance Time

Based on two values of "experience" and three values of "complexity influence on maintenance time", a contingency table was developed (Table 11). A Chi-Square test using data in Table 11 was done to determine whether there was an association between "experience" and assessed "complexity influence on maintenance time".

4 Results and Interpretations

The results are divided into six sections. The first section shows demographic data of all respondents. The subsequent four sections present results on (i) code characteristics, (ii) complexity influence on internal quality, (iii) the use of complexity measures in industry, and (iv) the influence of code complexity on the maintenance time of code. These four sections answer the first four research questions (see RQ 1–RQ 4 in Introduction). Section 6 shows the cross-sectional data analysis when slicing data according to the demographic data and answers the fifth research question (RQ 5).

4.1 Summary of Demographics

This section presents data from the five demographical dimensions of the respondents, i.e., the type of *education* of respondents, the type of *job* the respondents had, *the* software development*domain* the respondents worked in, and the group of *programming languages* they used.

The educational background of the respondents is shown in Fig. 5. Since this question was based on checkboxes, respondents could select multiple answers.

Table 9 Original eight values of assessment and three derived values of assessment

Original values	Derived values
0–10 %	Little influence
10–25 %	
25–50 %	
50–100 %	Much influence
100–150 %	
150–250 %	Very much influence
250–500 %	
500–1000 %	

Table 10 Cross-sectional data for "type of job" and assessment of "complexity influence on maintenance time"

	Developer	Non-developer
Little influence	8	10
Much influence	13	16
Very much influence	29	22

In total, 100 respondents gave 138 ticks, indicating that several respondents had more than one educational background. Figure 5 shows that the majority of respondents had received education in electrical/electronic engineering, software engineering or computer science. The popularity of electrical/electronic engineering can be explained by the fact that many respondents were from car and pump industries, which traditionally demand competence in electrical engineering. The increasing importance of software in these industries has created a favourable environment for electrical engineers to become software development specialists over time.

Figure 6 shows the job titles of respondents. Almost half of respondents (n = 49) are developers, but there was also large number of architects, development team leaders and researchers. This was unexpected in that there were few software testers among the respondents, although this could be explained by the fact that many respondents are working in Agile development teams, which have no specific testers and developers. Notably, these two jobs ("testers" and "developers") often are interchangeable and both are known as "developers" in some organizations.

Figure 7 presents the domain of respondents. In total, 105 answers were given by the 100 respondents, i.e., five or fewer respondents had worked in more than one domain. Fifty three respondents alone worked in the telecom and automotive domains.

Figure 8 presents the experience of respondents in software development. Thirty nine respondents had more than 15 years of experience and, generally, only 10 respondents had little experience (less than 3 years).

Finally, Fig. 9 shows the programming languages used by respondents. According to the responses, C language was dominant in these industries, partly due to embedded development and partly because all products were old and mature having been developed for many years and traditionally relying on C language.

In total, 100 respondents gave 171 ticks for programming languages, indicating that many of the developers used several programming languages.

4.2 Code Characteristics as Complexity Triggers

Figure 10 shows the eleven characteristics and their evaluated influence on complexity. The vertical axis shows the number of respondents, and every bar represents one characteristic. Bars are colour-coded, with the darkest red indicating that the given characteristic made code very complex. The darkest green colour indicated that the given characteristic did not make the code any complex. Overall, it can be inferred from Fig. 10 that two

Table 11 Cross-sectional data for "experience" and assessment of "complexity influence on maintenance time

	Little experience	Much experience
Little influence	3	15
Much influence	7	22
Very much influence	11	40

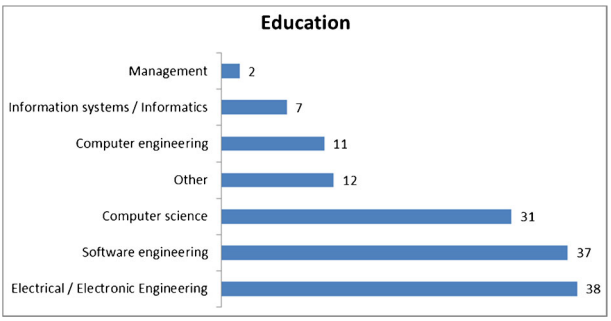


Fig. 5 Respondents' educational background

characteristics—the *lack of structure* and *nesting depth*—are separated from all other characteristics due to their estimated magnitude of influence. If the red-orange area is considered to be an area of *major influence,* then the majority of respondents (over 80 %) believed these characteristics to have *major influence* on complexity. The influence level of the rest of the characteristics decreased gradually along the horizontal axis. Approximately 50–60 % of respondents believed that *control statements, misleading comments* and *many developers* have a major influence on complexity, whilst about 35−45 % of respondents believed that *multiple tasks, frequent changes* and*complex requirements* did so.

The larger grey area for "Complex Requirements" may indicate that respondents found it difficult to evaluate this characteristic's influence on complexity.

The exact numbers of estimates are presented in Table 12. The statistical modes of the evaluations per characteristic are emphasized by colour so that what values the modes have on the assessment scale are easy to read. This alternative representation of the results enables greater understanding of the influence of characteristics on complexity. The characteristics have been divided into four groups based on their modes. The most influential characteristics are *nesting depth* and *lack of structure,* the modes of which reside in the categories of *very complex* and *quite complex.* The next three characteristics have modes categorized as *rather complex*, making them the second most influential characteristics. The rest of the characteristics are interpreted similarly.

In addition to the evaluation of code characteristics, respondents were also able to provide qualitative feedback on what other characteristics they considered might significantly influence code complexity. Eight respondents mentioned that it is preferable to separate

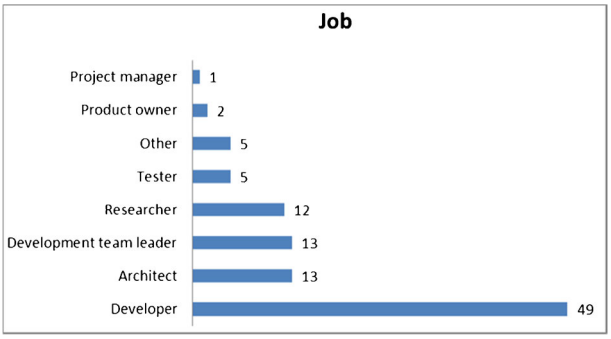


Fig. 6 Respondents' job description

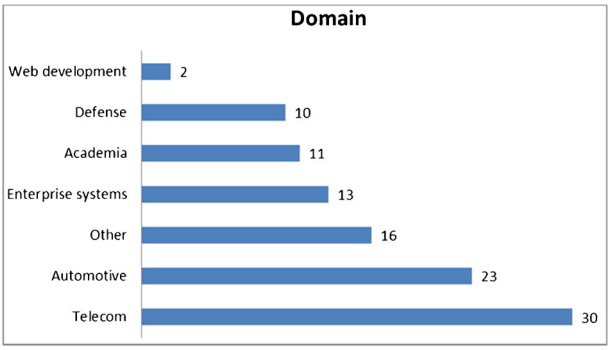


Fig. 7 Software development domain of respondents

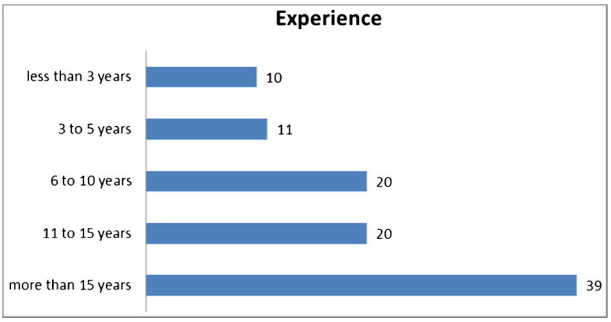


Fig. 8 Experience of respondents in software development

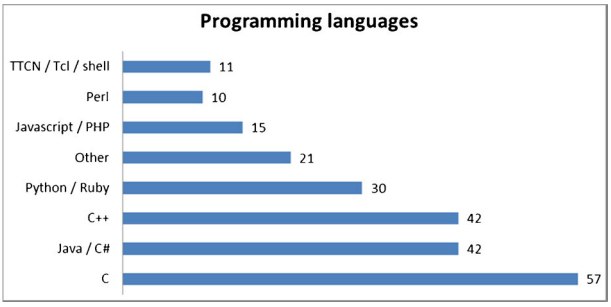


Fig. 9 Programming languages used by respondents during their entire experience

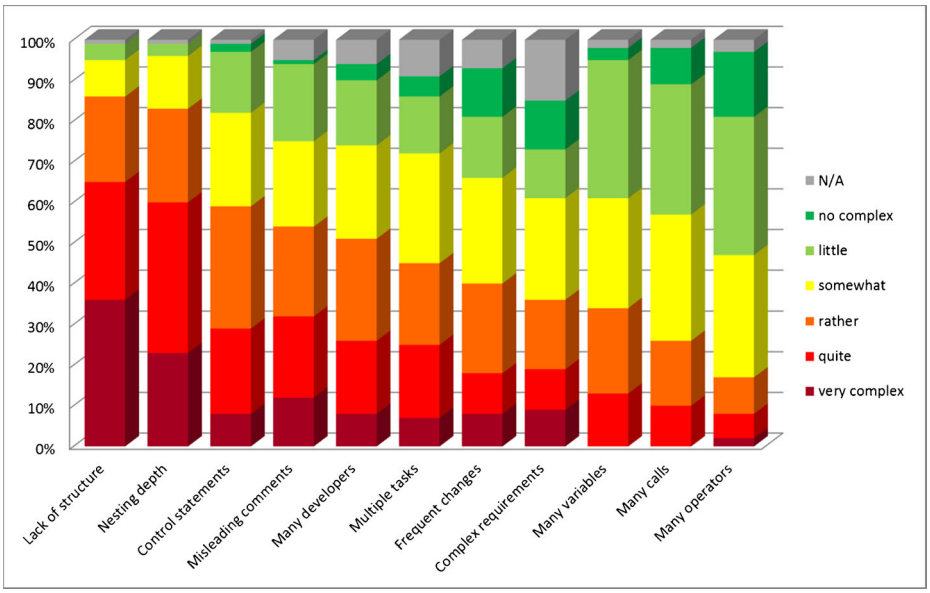


Fig. 10 Influence of code characteristics on complexity

categories of *"missing comments"* and *"misleading comments"* since they influence complexity differently, i.e., missing comments are not considered a problem if the code is well-structured and written in a self-explanatory manner; however, misleading comments can significantly increase the representational complexity of the code. One respondent stated that it is always good practice to incorporate the comments into the names of functions, variables, etc. because it is highly likely that over time and with the evolution of software, comments become misleading because they are not always updated.

Four respondents mentioned that they prefer global and local variables to be separated since global variables introduce significantly higher complexity than local variables. According to respondents, the extensive use of global variables can cause high complexity and decrease the ability to find serious defects. A case study conducted in Toyota also supports this line of argument (Koopman 2014).

Three respondents mentioned that multiple levels of inheritance with functions overloaded at many different levels can significantly increase complexity. In such cases, it is hard to understand which piece of code is actually executed. Another three respondents mentioned that the extensive use of pre-processors, macro-code and many levels of pointers can also significantly influence complexity.

As well as comments regarding code characteristics, respondents also reflected on other issues of code complexity. For example, several recognized that there are two types of complexity: essential and accidental, the former being inherent to the problem and the latter arising from non-optimal methods of programming, and that sometimes it is difficult to understand whether the complexity is essential or accidental.

4.3 The Influence of Complexity on Internal Code Quality Attributes

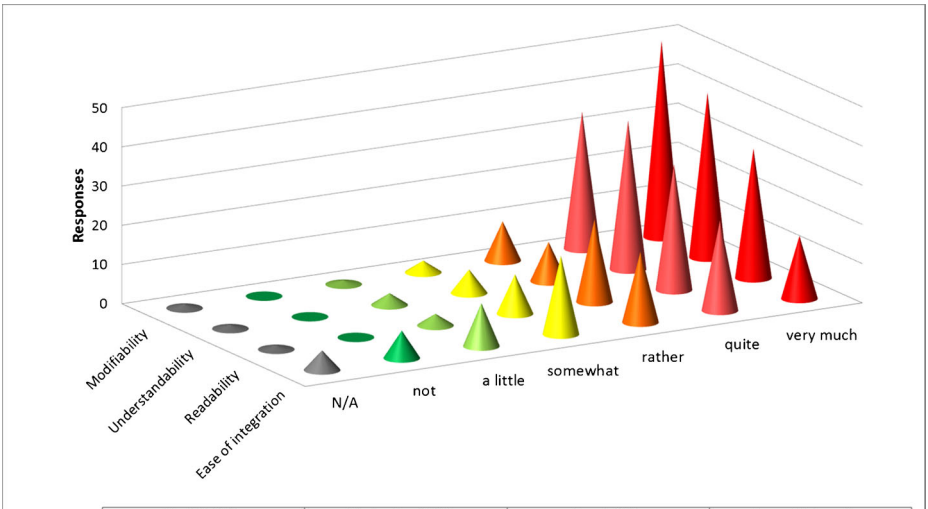
This subsection presents the negative influence of code complexity on internal code quality attributes, such as *readability, understandability, modifiability* and*ease of integration*.

Table 12 Influence of code characteristics on complexity with the modes emphasized

	very complex	quite	rather	somewhat	little	no complex
Lack of structure	36	29	21	9	4	0
Nesting depth	23	37	23	13	3	0
Control statements	8	21	30	23	15	2
Misleading comments	12	20	22	21	19	1
Many developers	8	18	25	23	16	4
Multiple tasks	7	18	20	27	14	5
Frequent changes	8	10	22	26	15	12
Complex requirements	9	10	17	25	12	12
Many variables	0	13	21	27	34	3
Many calls	0	10	16	31	32	9
Many operators	2	6	9	30	34	16

Figure 11 shows the evaluation results for the influence of code complexity on internal code quality attributes. The diagram shows that the majority of respondents agree that complexity has a huge influence on three attributes: *readability, understandability* and *modifiability.*

Modifiability, which can be considered the essential constituent of code maintainability, is influenced by complexity the most. Ninety five respondents believed that the complexity has major influence on code modifiability. Only four respondents believed otherwise, and



	Modifiability	Understandability	Readability	Ease of integration
N/A	1	1	1	5
not	0	0	0	7
a little	1	3	3	11
somewhat	3	6	10	20
rather	10	10	21	18
quite	35	38	32	23
very much	50	42	33	16

Fig. 11 Influence of code complexity on internal code quality attributes

one respondent did not answer the question. Every cell of the table in Fig. 11 shows the number of responses obtained per pair of internal code quality attribute and magnitude of influence, and the first row shows the "N/A" option.

The last three rows of this table tend to show greater numbers than the first three rows, indicating that the huge influence of complexity on internal code quality attributes. One of the attributes, *"ease of integration"*, is believed not to be influenced by complexity as much as the other three, which is intuitive because integration often concerns making the specified piece of code work with the rest of code without understanding its content in detail, and thus without actually dealing with complexity.

4.4 The Use of Complexity Measures

The use of code complexity measures in industry is presented here. Nine complexity measures (or groups of complexity measures) and their popularity are presented in Fig. 12.

Figure 12 shows that none of the nine measures are widely used according to respondents. On the left-most side are three relatively recognized and well-studied complexity measures, i.e., the Chidamber and Kemerer measures for object oriented languages, Halstead measures, and Henry and Kafura coupling measures. These were found to be rarely considered or used in industry and more than 60 respondents stated that they had never heard of these measures.

Figure 12 also shows that the next three measures (McCabe's cyclomatic complexity, fan-in, and fan-out) were slightly used. Only two groups of measures (*size measures* and *change measures*) were moderately used by respondents, although this does not necessarily mean that they were used as a means of quality assessment, but for other purposes, such as effort estimation or productivity measurement. Table 13 presents a more detailed view of the use of these measures. The modes of the first five measures in the table indicate that many respondents had never heard of the specified measures. The rest of the measures appear to be known by many, but never used in any systematic way.

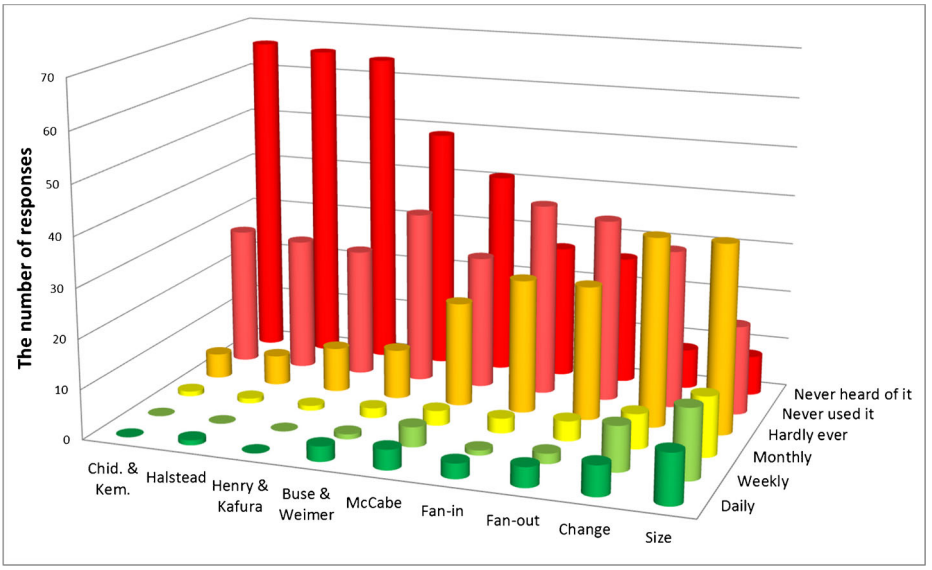


Fig. 12 Use of complexity measures in Industry

Table 13 Measures and their use represented by statistical modes

	Daily	Weekly	Monthly	Hardly ever	Never used it	Never heard of it
Chidamber & Kemerer	0	0	1	5	28	66
Halstead	1	0	1	6	27	65
Henry & Kafura	0	0	1	9	26	64
Buse & Weimer	3	1	2	10	35	49
McCabe	4	4	3	21	27	41
Fan-in	3	1	3	27	39	27
Fan-out	4	2	4	27	37	26
Change	6	9	7	38	32	8
Size	10	14	12	38	18	8

Table 14 more concisely represents the data, classifying the frequency of use into three categories: *regularly used, not used,* and *never heard of it.* We consider a measure is used regularly if it is used *daily*, *weekly*, or *monthly*, and a measure is *not used* if it is classified as*hardly ever* or*never used it.* The latter two categories mean that respondents knew of the measure and had even have tried to use it, but for some reason did not consider using it regularly. We have ascertained reasons for this through informal talks from software engineers in the participating companies; these vary and are inconclusive. For example:

- Company regulations either do not consider using the measure or another measure is the accepted standard
- Developers do not believe that use of the measure can compensate for the time spent on the measurement
- The measure is not a good indicator of complexity
- The measure is a good indicator of complexity, but of little help in understanding how to improve code
- Tool support is unsatisfactory, particularly in minimizing the spent time on the measurement and facilitating an understanding of the measurement output.

Considering these reflections, we can conclude that not only are measures potentially unhelpful, but also that company regulations and non-optimal tools thwart the full adoption of measures.

Table 14 Measures and their use represented in three categories

	Regularly used		Never heard of it
Chidamber and Kemerer	1	33	66
Halstead	2	33	65
Henry and Kafura	1	35	64
Readability measures	6	45	49
McCabe	11	48	41
Fan-in	7	66	27
Fan-out	10	64	26
Change	22	70	8
Size	36	56	8

The modes of responses in Table 14 show that the first four measures in the table are the least known. Nearly two-thirds of respondents did not know about the first three measures. Similarly, although the last five measures of the table were known by most respondents, they have never been used systematically.

Besides the measures that we suggested, respondents also mentioned several measures that they had used; however these were either alternatives of size measures (e.g., number of methods) or measures unrelated to complexity.

4.5 Influence of Complexity on Maintenance Time

Understanding the influence of complexity on maintenance time is necessary in order to make decisions on conducting complexity management activities. If complexity has a relatively small influence on maintenance time, it would be difficult to decide whether it is worth spending effort on complexity reduction. The results in this section aim to increase understanding of the complexity influence on maintenance time. They are rough estimates, however, as the estimates are based on educated guesses rather than quantitative assessment methods. *Such an estimate is subjective, and cannot be used as is. Its value, however, is that it provides an insight into the scale of complexity influence. Does complexity increase maintenance time by 10–20 %, or 60−80 %, or two-fold, or multi-fold or another order of magnitude?*

Figure 13 presents the results of the influence of code complexity on maintenance time of code. The statistical mode of the estimates is 27 % corresponding to 250–500 %. Twenty seven respondents believed that complexity roughly increases maintenance time by a factor of 2.5–5 times. Generally, 62 % of respondents believed that complex code takes more than twice as much maintenance time as simple code takes. In fact, only seven respondents thought that the code complexity has insignificant influence on maintenance time. Two respondents found it difficult to make such an estimate. Generally, the estimates indicate that there is a high likelihood that complexity increases maintenance time by multiple times.

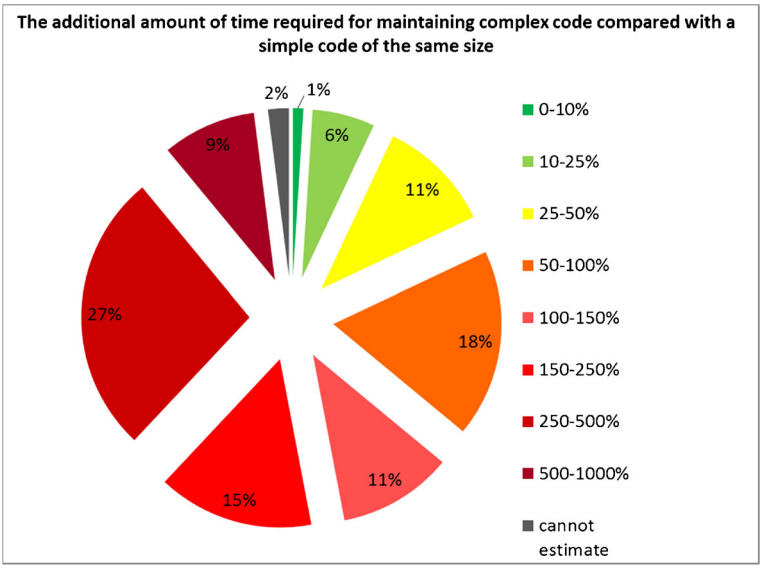


Fig. 13 Influence of complexity on maintenance time of code

This result means that complexity management activities are necessary because a significant reduction in complexity promises to decrease the maintenance time multiple times.

The respondents also commented on how they had estimated complexity influence on maintenance time. Four stated that they remembered some examples of simple code and complex code that they had modified in their practice. They remembered roughly how much time code modification took and made general estimates. One respondent noticed that in her/his experience, complex code (usually defect-prone) took a multi-fold longer time to correct defects than modifying the given code. One respondent stated that she/he purely speculated in her/his assessment.

4.6 Cross-Sectional Data Analysis Results

Here, we investigate whether the demographic data significantly affect the results presented so far. These data correspond to the four pieces of statistical analyses described in Section 3.6.

4.6.1 Type of Job and Assessment of Code Characteristics

This section presents results on whether the assessment results of code characteristics are associated with type of job. Table 15 presents the code characteristics and corresponding p and Chi-Square values for every characteristic. The significance level for p-value is $p<0.05$.

P-values for "many operators" (0.014) and "many calls" (0.016) attained statistical significance, indicating that there is indeed a difference between the assessments of "developers" and "non-developers". In both cases, the data suggest that according to the developers' assessment, "many operators" and "many calls" have less influence on complexity increase compared to that of "non-developers". All other p-values are large ($p>0.05$), indicating no significant difference between the assessments of "developers" and "non-developers".

4.6.2 Respondents' Experience and the Assessment of Code Characteristics

The results here show whether the assessment results of code characteristics are associated with respondents' experiences. Table 16 presents code characteristics and corresponding p and Chi-Square values for every characteristic.

The p-value for "multiple tasks" is small (0.04), indicating a statistical difference between assessments of "more experienced" and "less experienced" respondents. In this case, the data suggest that according to "more experienced" respondents, the number of

Table 15 Chi-Square test results per code characteristic: type of job and assessment

Name of characteristic	Lack of structure	Nesting depth	Control statements	Misleading Comments	Many developers	Multiple tasks
P-value	0.438	0.679	0.804	0.076	0.317	0.249
Chi-sq.	0.602	0.171	0.062	3.151	1.002	1.327
Name of characteristic	Frequent Changes	Complex Requirements	Many Variables	Many calls	Many operators	
P-value	0.654	0.196	0.472	0.016	0.014	
Chi-sq.	0.201	1.673	0.518	5.808	6.005	

Table 16 Chi-Square test results per code characteristic: experience and assessment

Name of characteristic	Lack of structure	Nesting depth	Control statements	Misleading Comments	Many developers	Multiple tasks
P-value	NA	0.686	0.213	0.407	0.111	0.040
Chi-sq.	NA	1.164	1.550	0.687	2.538	4.216
Name of characteristic	Frequent changes	Complex requirements	Many variables	Many calls	Many operators	
P-value	0.667	0.460	0.160	0.176	0.659	
Chi-sq.	0.185	0.547	1.971	1.834	0.195	

"multiple tasks" in a unit of code has more influence in complexity increase compared with the assessment of "less experienced respondents". The rest of the p-values are statistically significant, showing no association between assessment results and respondents' experience. In the case of "lack of structure", one of the values was less than five when calculating the estimated frequencies of its contingency table so it was not possible to conduct a meaningful test (marked NA in the table).

4.6.3 Type of Job and Assessment of Complexity Influence on Maintenance Time

The results here show whether the assessment results of "complexity influence on maintenance time" is associated with respondents' "type of job". The Chi-Square test that was performed based on Table 10 shows a large p-value, $p = 0.484$ (Chi-Sq. $= 1.453$), indicating no statistical significance. This means the assessment results of complexity influence on maintenance time are not statistically different across different jobs.

4.6.4 Respondent' Experience and Assessment of Complexity Influence on Maintenance Time

The results here show whether the assessment results of "complexity influence on maintenance time" is associated with respondents' "experience". The Chi-Square test that was performed based on Table 11 shows a large p-value, $p = 0.831$ (Chi-Sq. $= 0.831$), indicating no statistical significance. This means the assessment results of complexity influence on maintainability cannot be statistically different due to respondents' experience.

5 Discussion

Code Characteristics as Complexity Triggers (RQ 1) We have proposed eleven code characteristics in this survey, two of which,*nesting depth* and *lack of structure*, strongly influenced complexity. Compared to other characteristics, these two are usually avoidable because deeply nested blocks can be averted by using the "return" statement or creating additional function calls. It is also possible to write highly structured code by using meaningful names of function and variables, maintaining line length within good limits, keeping indentations consistent, etc. Other characteristics, such as the number of operators, control statements or function calls, usually cannot be avoided since they are tightly associated with problem complexity.

Our results show that the main two complexity triggers might instead be related to accidental complexity, which can arise due to suboptimal design decisions. Our results also closely relate to a report by Glass (2002) that for every 25 % increase in problem complexity, there is a 100 % increase in complexity of the software solution. A natural question then follows: is it the accidental complexity that quadruples the increased complexity in the solution domain? We believe that there is great value in investing effort to answer this question with a further research because the results of RQ 4 show that complexity has a substantial influence on the maintenance time, which consumes 90 % of the total cost of software projects (Seacord and et al 2003).

Figure 10 clearly shows that different complexity triggers (code characteristics) have significantly different levels of influence on complexity increase. This suggests that when creating a complexity measure, the relative differences of such influences should be considered otherwise the complexity measure will miss-estimate the perceived complexity of the given measurement entity. Moreover, when calculating complexity, the weighting for different characteristics can be derived from empirical estimates of code characteristics as complexity triggers. In our case, for example, the *nesting depth* will have a higher coefficient in complexity calculation than the *number of operators*.

The Influence of Code Complexity on Internal Code Quality Attributes (RQ 2) The results suggest that readability, understandability and modifiability of the code are highly affected by complexity. These results, and those of RQ 1, entail a straightforward conclusion: nested blocks and poorly structured code are the main contributors (at least among the proposed eleven characteristics) in making code hard to read, understand and modify. This conclusion may provide good insight for programmers in order to develop understandable code.

The Use of Complexity Measures in the Industry (RQ 3) This part of the survey included only the popular code complexity measures; however, there was an empty field where respondents could register other measures that they used. The results show that all of the measures are used rarely in the collaborating companies, and that respondents have never considered any other complexity measures. There are at least two clear arguments for these results:

1. Either the measures are not satisfactorily good at predicting problem areas,
2. Or the measures are good enough (particularly when used in combination), but software engineers need help in understanding how they can optimally use these measures to locate problem areas and improve the code.

There are also valid perspectives to support both arguments:

1. Designing measures should not be based merely on theoretical frameworks because the weighting for different complexity triggers that are considered in complexity measurement can only be derived from empirical data.
2. Complexity measures should be evaluated not only for defect prediction, but also for how well they can both locate complex code areas and indicate necessary improvements.

The Influence of Complexity on Maintenance Time of the Code (RQ 4) If we were to believe the statistical modes of the results then clearly, complexity management can potentially decrease maintenance time by a multiple factor.

Cross-Sectional Data Analysis (RQ 5) The cross-sectional data analysis results support the argument that results obtained for RQ 1−4 of the survey are most likely not associated with respondents' demographics. It was particularly intuitive to believe that certain jobs not largely related to core development activities would tend to underestimate the complexity effect on maintenance time. Our results, however, show that this is not so, which might imply that practitioners who are not working directly with software design are, nevertheless, well aware of the complexity effect on maintenance time.

Future Work We are planning two further studies to directly follow this study; specifically, we will:

1) Circulate the survey to a wider range of software developers, including the open source community, to gather results from a wider arena of products and development paradigms, and
2) Design a complexity measure that takes into consideration the assessed influences of code characteristics.

6 Validity Threats

Notably, when analysing the results obtained on code characteristics as complexity triggers, these results are limited to the eleven characteristics proposed in this study, which creates a construct validity threat. If more code characteristics had been used in the study, the influence of characteristics on complexity would differ in Fig. 10. For example, if we had added more characteristics (e.g., "inheritance level" and "usage of macro-code") to the survey, the number of the most influential characteristics might have increased. This means that "nesting depth" and "the lack of structure" might not be the only important characteristics to influence complexity. This should be considered when applying these results in practise. Nevertheless, adding more characteristics will not change the estimated influence of the current code characteristics, which means that *nesting depth* and *lack of structure* remain very influential characteristics.

There is also a possibility that several respondents had worked in the same organization/team. A common practice in software development organizations is to decide the standard tools to be used by the organization. Using software measures also complies with this practice. Therefore, if five respondents from the same organization answered the survey, they might all indicate that they use the same measure. Whilst this does not mean that this measure is used more often than others, it does mean that in a particular organization the given measure is adopted for regular use. By including seven companies (including several organizations within each) and two universities in this study, this threat has been significantly minimized. Nevertheless, employing a wider range of companies or domains in this survey would likely result in a markedly more accurate picture of the use of measures. It would be particularly interesting to determine those measures used in open source product development because there the use of measurement tools is fundamentally regulated in a different way. While tool choice is often affected by corporate regulations and standards (Xiao et al. 2014), open source developers are more likely to have greater freedom in their choice of tools.

Another construct validity threat arises due to the possibility that respondents did not actually understand the measures investigated in the survey. It is possible that respondents use a tool that shows values of complexity using a certain measure, yet despite using these

values, they still do not know the name of the measure. Thus, when encountering this measure in the survey, they might have marked it as "have not heard of". In the survey, we have partially mitigated this validity threat by providing explanatory text on what a given measure actually shows. It may well be the case that even these explanations do not shed light on whether the given measure was actually known, although this is unlikely.

The four internal quality attributes of code in Section 3.3 were chosen based on two important points. Firstly, the attributes should be simple and direct to enable respondents to make a clear logical connection between them and a complexity otherwise a validity threat of misinterpreting the attribute and the entire question could occur. For example, if we used *conciseness*, respondents might have difficulty in understanding what "conciseness of code" is and thus might provide a flawed answer. Secondly, as we are interested in internal quality attributes that directly affect developers' work on maintainability, we did not want to expand the survey to explore the effect of complexity on any quality attribute in particular.

We designed even-point, Likert scale questions to avoid mid-point values. We argue that mid-point values should not be used because some respondents might opt for them if the question is perceived as difficult and requires more thought. The survey questions did not imply the necessity of mid-point values so we believe that the six-point scale was adequate.

Two factors can cause a construct validity threat when estimating the influence of complexity on maintenance time (RQ 5). The first factor concerns the interpretation of what is simple code and what is complex code. We suggested comparing the maintenance time spent on simple code with that spent on complex code. Since respondents could have their own interpretations of *complex code* and *simple code* in our survey (RQ 5), such a comparison is based on a purely subjective interpretation of the definition of complex/simple code. The second factor concerns the estimation itself, which is neither quantified in any way nor derived from a specified mechanism that was used by respondents. These results are derived only from what respondents believe based on their experience and knowledge so we acknowledge that these results should be used cautiously when making inferences or predictions.

The classification of *developers* and *non-developers* for the cross-sectional data analysis might not have been an optimal choice because the non-developers' group contains several categories of jobs. Unfortunately, we were unable to classify the data based on more categories and conduct meaningful statistical tests due to data scarcity. Therefore, the fact that no statistical significance was attained in this piece of analysis might be due to over-simplification of this category.

In conclusion, the assessment of code characteristics and their influence on maintenance time is entirely based on the knowledge of software engineers. While a summary of this knowledge can be valuable, it should not be taken for granted. Evidence based on alternative and more objective measures would be markedly beneficial for this type of study (Devanbu et al. 2016).

7 Related Work

A comprehensive list of code characteristics that influence complexity can be found in the work of Tegarden et al. (1995), who separate code characteristics for several entities, including variables, methods, objects and subsystems. They differentiate nearly 40 distinct code characteristics that can influence complexity differently. They propose that some of these characteristics can be combined as they are similar; however they leave this up to the

user of their list to decide on how to do so. Their work is valuable because it provides a comprehensive list of characteristics that can be used to design complexity measures. Gonzalez (1995) identifies seven sources of complexity that should be considered when designing complexity measures: control structure, module coupling, algorithm, code nesting level, module cohesion, and data structure. Gonzales also distinguishes three domains of complexity: syntactical, functional and computational. Syntactical is the most visible domain, although it can reveal information about the other two domains of complexity.

In addition to the nine measures of complexity in our study, there are also several other measures reported in literature that are more or less accurate for complexity assessment, notably the Chapin (1979) complexity measure based on data input and output. Munson and Kohshgoftaar (1993) have reported measures of data structure complexity, whilst cohesion measures have been described by Tao and Chen (2014) and Yang et al. (2015). Moha et al. 2010 have designed measures for code smells, where "code smells" can be regarded as an aspect of complexity. Kpodjedo and et al (2011) have proposed a rich set of evolution measures, some of which were considered in our study. Wang and Shao (2003), followed by Waweru et al. (2013) proposed complexity measures based on the weighted sum of distinct code characteristics. Earlier, we discussed that weighting can provide a more accurate measure of complexity; however the weighting should not merely be based on the perception of the measure's designer, but on empirical estimates to provide sensibly accurate weights. From this perspective, we believe that our study can provide valuable information for studies that design measures of complexity. Keshavarz et al. (2011) have developed complexity measures, which are based on software requirement specifications and can provide an estimate of complexity without examining existing source code. Al-Hajjaji et al. (2013) have evaluated measures for decision coverage.

Suh and Neamtiu (2010) have demonstrated how software measures can be used for proactive management of software complexity. They report, however, that the measurement values they obtained for existing measures provided inconclusive evidence for refactoring and reducing complexity. They observed many occasions when developers reduced values of complexity measures in the code with no reduction in actual perceived complexity as had been expected. *The results of this study support the argument that existing software measures are still far from satisfactory for software engineers when not used in combination with each other.*

Salman (2006) has defined and used a set of complexity measures for component-oriented software systems. Most of the measures that these introduce are more like size measures (the number of components, functions, etc.). There are also measures similar to fan-in and fan-out, but at the component level. Most importantly, the study shows that complexity has major influence on code maintainability and integrity and that there is lack of empirical data on how existing complexity measures actually perform in industry. Kanellopoulos et al. (2010) have proposed a methodology for code quality evaluation based on the ISO/IEC 9126 standard. This work is distinguished by the fact that they use expert opinions for weighting code measures and attributes for more accurate evaluation of code quality. In two of our previous studies, we have developed measurement systems in Ericsson and Volvo Group Truck Technology (Antinyan et al. 2014a). We investigated several complexity measures and chose to use a combination of two measures as a predictor of maintainability and error-proneness. Since we had the close collaboration of a reference group of engineers, we received valuable feedback on how these engineers viewed

the introduced complexity measures. One of the most important points they made was that the introduced complexity measures, such as cyclomatic complexity, fan-in, and fan-out, are too simplistic for complexity measurement. According to them, there were stronger characteristics of complexity that needed to be weighed in measurement. This feedback was taken into consideration in the design of this current survey.

8 Conclusions

In this study, we have conducted a survey to: (i) investigate code characteristics and their contribution to complexity increase; (ii) evaluate how often complexity measures are used in practice; and (iii) evaluate the negative effect of complexity on the internal quality and maintenance time. Our results show that: (i) the two, top-prioritized characteristics for code complexity are not included in existing code complexity measures; (iii) existing code complexity measures are poorly used in practice; and (iii) code complexity has a major influence on internal quality and maintenance time. This study shows that the discipline concerning code complexity should focus more on designing effective complexity measures; in particular, data from empirical observations of code characteristics as complexity triggers should be used. More work is necessary for a greater understanding of how software engineers can use existing complexity measures for effective complexity management and for the ultimate need of cutting down the maintenance time.

References

Abran A (2010) Software metrics and software metrology. Wiley

Al-Hajjaji M et al. (2013) Evaluating software complexity based on decision coverage. LAP LAMBERT Academic Publishing

Antinyan V et al. (2014) Monitoring evolution of code complexity and magnitude of changes. Acta Cybern 21(3):367–382

Antinyan V et al. (2014) Identifying risky areas of software code in Agile/Lean software development: An industrial experience report. 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering, (CSMR-WCRE), IEEE

Banker RD et al. (1993) Software complexity and maintenance costs. Commun ACM 36(11):81–95

Basili V (1980) Qualitative software complexity models: A summary. Tutorial on models and methods for software management and engineering

Briand L et al. (1995) Theoretical and empirical validation of software product measures. International Software Engineering Research Network, Technical Report ISERN-95-03

Briand LC et al. (1996) Property-based software engineering measurement. IEEE Trans Softw Eng 22(1):68–86

Buse RP, Weimer WR (2010) Learning a metric for code readability. IEEE Trans Softw Eng 36(4):546–558

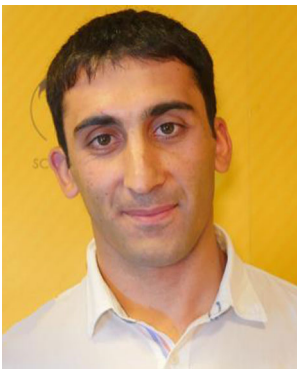
Chapin N (1979) A measure of software complexity. Proceedings of the 1979 NCC: 995–1002

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493

Devanbu P et al. (2016) Belief & evidence in empirical software engineering. Proceedings of the 38th International Conference on Software Engineering. ACM

Fenton NE, Neil M (1999) A critique of software defect prediction models. IEEE Trans Softw Eng 25(5):675–689

Fenton NE, Neil M (1999) Software metrics: successes, failures and new directions. J Syst Softw 47(2):149–157
Fenton N, Bieman J (2014) Software metrics: a rigorous and practical approach. CRC Press
Geraci A et al. (1991) IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries. IEEE Std, p 610
Glass RL (2002) Sorting out software complexity. Commun ACM 45(11):19–21
Gonzalez RR (1995) A unified metric of software complexity: measuring productivity, quality, and value. J Syst Softw 29(1):17–37
Graylin J et al. (2009) Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship. J Softw Eng Appl 2(03):137
Halstead MH (1977) Elements of Software Science (Operating and programming systems series). Elsevier Science Inc.
Henry S, Kafura D (1981) Software structure metrics based on information flow. IEEE Trans Softw Eng 5:510–518
Kanellopoulos Y et al. (2010)
Kaner C (2004) Software engineering metrics: What do they measure and how do we know? In METRICS 2004. IEEE CS, Citeseer
Keshavarz G et al. (2011) Metric for Early Measurement of Software Complexity. Interfaces 5(10):15
Kitchenham B et al. (1995) Towards a framework for software measurement validation. IEEE Trans Softw Eng 21(12):929–944
Koopman P (2014) A case study of Toyota unintended acceleration and software safety. Presentation. Sept
Kpodjedo S et al (2011) Design evolution metrics for defect prediction in object oriented systems. Empir Softw Eng 16(1):141–175
McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng 4:308–320
Mens T (2012) On the complexity of software systems. IEEE Comput 45(8):0079–0081
Moha N et al. (2010) DECOR: A method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36(1):20–36
Moses J (2001) Complexity and Flexibility. Professor of Computer Science and Engineering, MIT/ESD
Munson JC, Khoshgoftaar TM (1992) The detection of fault-prone programs. IEEE Trans Softw Eng 18(5):423–433
Munson JC, Kohshgoftaar TM (1993) Measurement of data structure complexity. J Syst Softw 20(3):217–225
Rea LM, Parker RA (2014) Designing and conducting survey research: A comprehensive guide. John Wiley & Sons
Rechtin E, Maier MW (2010) The art of systems architecting. CRC Press
Salman N (2006) Complexity metrics as predictors of maintainability and integrability of software components. Cankaya Univ J Arts Sci 1(5)
Sarwar S et al. (2013) Cyclomatic complexity: The nesting problem. 2013 Eighth International Conference on Digital Information Management (ICDIM). IEEE
Schneidewind NF (1992) Methodology for validating software metrics. IEEE Trans Softw Eng 18(5):410–422
Seacord RC et al (2003) Modernizing legacy systems: software technologies, engineering processes, and business practices, Addison-Wesley Professional
Shepperd M, Ince DC (1994) A critique of three metrics. J Syst Softw 26(3):197–210
Subramanian GH et al. (2006) An empirical study of the effect of complexity, platform, and program type on software development effort of business applications. Empir Softw Eng 11(4):541–553
Suh SD, Neamtiu I (2010) Studying software evolution for taming software complexity. Software Engineering Conference (ASWEC), 2010 21st Australian, IEEE
Tao H, Chen Y (2014) Complexity measure based on program slicing and its validation. Wuhan Univ J Nat Sci 19(6):512–518
Tegarden DP et al. (1995) A software complexity model of object-oriented systems. Decis Support Syst 13(3):241–262
Tenny T (1988) Program readability: Procedures versus comments. IEEE Trans Softw Eng 14(9):1271–1279
Wang Y, Shao J (2003) Measurement of the cognitive functional complexity of software. The Second IEEE International Conference on Cognitive Informatics
Waweru SN et al. (2013) A Software Code Complexity Framework; Based on an Empirical Analysis of Software Cognitive Complexity Metrics using an Improved Merged Weighted Complexity Measure. Int J Adv Res Comput Sci 4(8)
Weyuker EJ (1988) Evaluating software complexity measures. IEEE Trans Softw Eng 14(9):1357–1365
Xiao S et al. (2014) Social influences on secure development tool adoption: why security tools spread. ACM
Yang Y et al. (2015) Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? An empirical study. IEEE Trans Softw Eng 41(4):331–357
Zuse H (1991) Software complexity. Walter de Cruyter, USA



Vard Antinyan is a PhD candidate in Software Engineering at the University of Gothenburg, Sweden. His research includes software measurement, software complexity, software quality, and risk management.

Miroslaw Staron is an Associate Professor in Software Engineering at the University of Gothenburg. His research includes measurement and decision support in software engineering.

Anna 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.