

TOWARDS PROGRAM COMPREHENSION: KNOWLEDGE PROFILES FOR C PROGRAMMERS

Emília PIETRIKOVÁ

Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice,
Letná 9, 040 01 Košice, Slovakia, tel. 055/602 2554, E-mail: emilia.pietrikova@tuke.sk

ABSTRACT

Following up on the issue of programmer knowledge profiles, the present article deals with the generation of profiles for C language programmers. The main idea is based on static analysis of source codes, assembling the most significant data about their author. Such a profile can point out some abilities and habits of the programmer. The process of static analysis applies methods and techniques of size metrics, complexity metrics, and clone detection. We also present an experiment focused on novice programmers. The results show that knowledge profiles provide an easy way of novice programmers' progress tracking. The main contribution is believed to be in the area of program comprehension.

Keywords: *program comprehension, programmer knowledge profile, static code analysis, software quality metrics, clone detection*

1. INTRODUCTION

Software development is strongly affected by program comprehension. Programmers are frequently tasked with modifying and enhancing applications, so understanding the existing code is a matter of course [1]. During the comprehension process, programmers form mental representations of the code they are working with. Therefore, program comprehension is considered to be essential for software reuse, debugging, testing, maintenance, and evolution [2].

Many modern studies on observe professional programmers and their strategies [3]. In the following article, we focus on novice programmers. Unlike [4], who dealt with expertise predictions, our goal is to track novices' progress in the process of knowledge acquisition.

Static source code analysis is a result of our efforts to better source code comprehension. The goal is to understand the code at a deeper level than the implemented functionality. The reason is that there are many different ways of writing the same program functionality. However, code analysis allows to examine the quality of the source code and thus reveal its potential weaknesses that may affect the program behavior in specific situations.

The need to ensure a certain source code quality resulted in analyzers' creation to provide this service through several criteria, such as program complexity, clarity, and others. Through these categories, it is possible to detect system security vulnerabilities or overly complex pieces of code that are prone to errors. Due to the high complexity, these blocks of code have become confusing and detection of the potential problem has become harder. High complexity also makes it difficult to overcome an already identified problem. However, this is just one of many potential complications, increasing the importance of source code parsers and code comprehension research [5].

Source code analysis has significant use in the educational process. Unlike experienced programmers, novices have not yet developed their programming habits, either positively or negatively, and their study is crucial in creating good habits.

Thanks to source code analysis tools in student projects evaluation, it is possible to detect novices' bad habits when

writing code over time. The identified problems can then be reduced or eliminated by targeted inclusion of the issue in teaching, or by deepening knowledge in this area.

We have addressed the issue of programmer knowledge profiles in several previous studies [6] [7] [8]. Our tools support static source code analysis, mostly in Java. In the present study we focus on the C programming language, its support and results. The support is based on a group of metrics, providing a better comprehension of the C code.

2. KNOWLEDGE PROFILES

In a study [7], we defined knowledge as a description of the knowledge and links between their elements needed to master a particular task.

The purpose of the knowledge profile is to provide data based on which it is possible to verify the programmer's knowledge to solve the task and visualize how deeply he/she comprehends the code. This process is related to missing knowledge identification. For this reason, the profile is structured.

Knowledge profiles are constructed from input artifacts, source codes. That is, the profile is bound to the programming language. In the study [7], we worked with two types of profiles. In the process of further research, we reworked them into three profiles [8]:

- *Standard profile* – It is created based on the static code analysis. It provides information about language constructs, keywords, and the frequency of their occurrence.
- *Group profile* – Based on several projects, the aim is to view the programmer's progress between the first and the last project.
- *Comparing profile* – Performs analysis by comparing two profiles. One of these profiles is the master profile, representing a sample solution. The second one is the programmer's profile. The analysis is based on the two projects similarity.

To create a more complex profile, the assumption that the programmer understands the language construct if used

once only is not enough. Evidence of a deeper comprehension can be obtained through metrics. The assumption is that the more method calls, the deeper understanding.

2.1. Related Work

Regarding related work, we have not much to compare, as there is no available literature dealing with program analysis the same way. However, we recently inspired one research group, creating their own program profiler [9]. Their perception of profiles is on a scale and their goal is to determine how skilled is the programmer. The skill level is the profile and if considered to be weak, they try to motivate the programmer. Moreover, specific tasks must be solved.

On the contrary, our perception of profile is more general. We are able to generate profiles from different codes (not specific tasks) and our goal is to understand how the author comprehends his/her code. Therefore we believe the main contribution of this paper is in the area of program comprehension and static code analysis.

2.2. Applied Metrics

In general, there are many metrics for evaluating software quality, but we have selected only some. According to the measured aspect of source codes, metrics for source code analysis are mostly divided into size and complexity.

According to [10], there are several types of size-oriented metrics: Lines of Code (LoC), Comments Lines of Code (CLOC), Number of Methods, Method Lines of Code, and Comments Density. It is not possible to realistically estimate the code quality based on *size* metrics only, since more complex languages are currently in use. The answer to this problem is the creation of metrics, dealing with *program complexity*.

Nested Block Depth – More nested blocks lead to poorer readability and a more complex solution. The maximum nesting depth expresses the depth of the most nested block within one function. The maximum value of this metric is considered to be 3-4¹. If the deepest nesting is 5 or more, the function is evaluated as critical.

Cyclomatic complexity – It is obtained by summarizing the number of linearly independent paths within a code or a method. Each method has a minimum complexity value of 1. The calculation of cyclomatic complexity varies according to the language. This is because different programming languages also differ in functionality and keywords².

Condition coverage – The number of executed statement blocks and conditions divided by their total number. Here each condition counts twice, which may result in a large number of possible outcomes in a complex decision³.

NPath Complexity – Study [11] defines a new analogy of the original cyclomatic complexity on the C language commands for program flow control. The reason was the lack of cyclomatic complexity, due to which the graph blocks complexity was not taken into account. The problem occurred in the case of blocks containing cycles or branches. The

new variant works with the complexity of conditions inside conditions, cycle headers etc.

Cognitive complexity – Code nesting, and especially multiple nesting, is considered a bad approach in practice as it indicates a bad code design. This is why SonarQube [12] has developed a new metric – cognitive complexity. In addition to the functionality of cyclomatic complexity, it can also evaluate the cyclomatic complexity of individual operands in the meaning of branches and cycles, the complexity of nesting code, and others.

Henry's and Kafura's Informational Flow Metrics – Authors of [13] have defined a different variant of cyclomatic complexity, which is often referred to as the Fan-in Fan-out metric. This metric also takes into account the interconnectedness of the program when calculating the complexity. It was designed based on source code analysis of the UNIX operating system. The authors managed to find a link between the high value of this metric and the problems in the same part of the code where it was applied.

In general, complexity metrics provide a better view of code quality, as the number of LoC usually provides only the stylistic information about the author. This is why all the mentioned metrics are implemented in our experiment, extending knowledge profile generator from [6].

2.3. Visualization

In the standard profile, metrics are visualized by hierarchical view, table, and heat map. The *hierarchical view* is represented by a multilevel pie chart of metrics at selected level. Depending on the given size for a specific graph part, the size is derived. E.g. with LoC, code parts with the largest number will occupy the largest graph area (Fig. 1). Each additional level has descendants of given symbols.

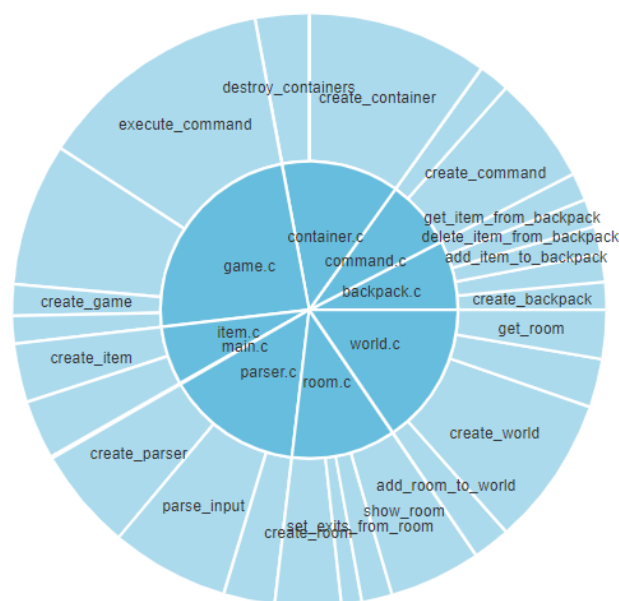


Fig. 1 Standard profile: Hierarchical view for selected level

¹Function level metrics. Source: <https://hub.codebeat.co/docs/software-quality-metrics>, accessed Aug-02-2021

²Metric definitions. Source: <https://docs.sonarqube.org/latest/user-guide/metric-definitions>, accessed Aug-02-2021

³Source: <https://www.froglogic.com/coco/condition-coverage>, accessed Jan-24-2022

The *table view* shows the measured metrics at the file or method level. The individual rows of the table are grouped according to their parent symbol.

The *heat map* is provided for selected table rows. An example is depicted in Fig. 2, in which the individual metrics are color-coded. If the metric does not have pre-defined limits, the color moves on a scale of blue, the intensity of which depends on the value distance from the statistical maximum or minimum of the metric. In the case of defined limits, the cell can be green, yellow, or red, depending on the limits.

All the abbreviations in the heat map refer to metrics described in sec. 2.2:

- Complexity metrics:
 - CC – Cognitive complexity,
 - HK – Henry & Kafura, Fan-in Fan-out,
 - NPATH – NPath complexity,
 - NEST – Nested block depth,
 - COND – Condition coverage,
 - CYCLO/LOC – Cyclomatic complexity and lines of code relative,
 - CYCLO – Cyclomatic complexity.
- Size metrics:
 - MLOC – Method lines of code (size),
 - COMMENT – Comments density (size),
 - LLOC – Logical lines of code (size),
 - LOC – Lines of code (size).

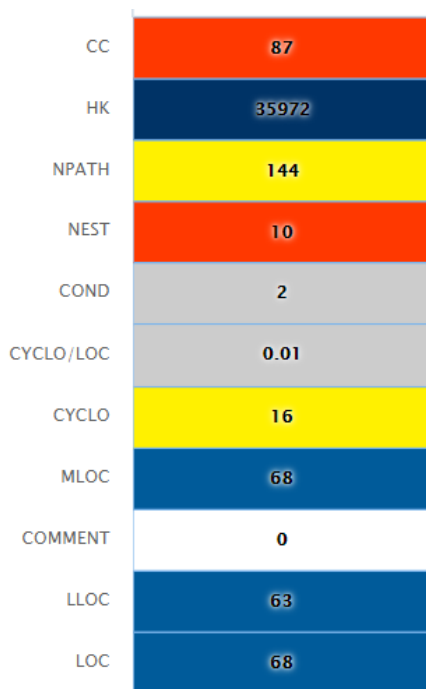


Fig. 2 Standard profile: Heat map for metrics of selected row

2.4. Clone Detection

According to [14], clone detection is a technique that can use metrics to detect a function that is a copy of another function or represents a mutation of that function.

In most cases, clones are created by copying a certain function and subsequent modifications to the resulting copy. Cloning is a consequence of insufficient use of process reusability within the system.

A large number of clones can cause side effects in the program. One such side effect may be an increase in the resources required by the software. The second effect caused by a large number of clones is the increased maintenance of the system. For example, if problems are resolved in the original version, the same problem may occur in a clone, that is, an earlier version of the feature being fixed.

We use the detection method based on tokens, where we rely on the results of [15]. The principle is to compare parts of codes, specifically to compare the sequences of lexical symbols of code pairs. Since lexical symbols represent only specific elements of a programming language (without context), methods based on token sequence comparisons can detect both complete and parameterized clones.

In [16], we addressed this issue in more detail. Clone detection is also part of our experiment and C source code analysis.

3. EXPERIMENT

In order to verify the contribution of knowledge profiles to program comprehension, we have already experimented with students in Java [8]. As we have enhanced the existing profiler tool and implemented the metrics for C code, we conducted an experiment with students in C. The counters provide a better view of the programmer's habits but are not sufficient to assess the quality of the code. Therefore, the experiment also includes size metrics and complexity metrics described in sec. 2.2. The evaluated metrics provide a much better view of the programmer's tendencies in programming. For example, in the case of a metric expressing the density of comments concerning the LoC, it is possible to judge whether the author cares about code documentation, which is of great importance for the code sustainability in the future.

Cyclomatic complexity further reveals potential weaknesses in the program if its value exceeds a predefined recommended limit. With higher cyclomatic complexity, the complexity of a given code grows, increasing error probability. Too complex program also reduces clarity, which can result in complications in locating and debugging.

The experiment is focused on novice programmers: 5 students of Informatics in the Programming course (CS1) focused on procedural programming and the C language. The measurements include two assigned projects. One project is focused on implementing modified version of the 2048 game, and another one is a text game. The main goal is to generate and compare student programming profiles using our tool, and understand how they comprehend their code.

3.1. Code Quality

Using the standard profile (sec. 2), both projects were analyzed for every student. Subsequently, data were collected on higher weight metrics such as cyclomatic complexity, cognitive complexity, maximum code nesting, and the number of paths that could be performed.

In the first step, the goal was to evaluate the percentage of exceeded limits of the mentioned metrics, which makes it possible to obtain the percentage of success in following the recommended practices. The second step was to find out, whether the student had made progress in code quality during the semester.

For this reason, the first analyzed project was assigned at the beginning of the semester, when the students were newly introduced to the C language and its possibilities. The second comparative project was assigned at the end of the semester as there is a presumption that the students acquired new knowledge and skills during the semester.

The analysis results are represented in Fig. 3. The graph columns represent the percentage of violations of the defined limits for the metrics mentioned above.

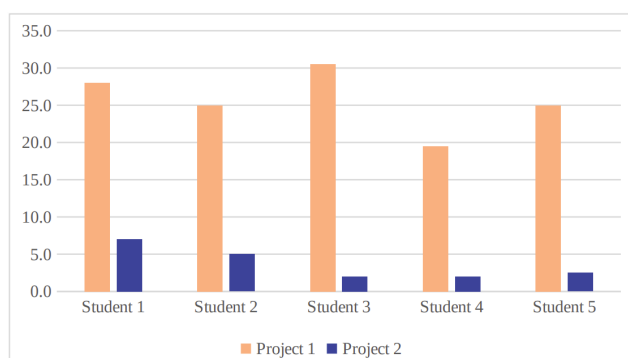


Fig. 3 Percentage of defined metric limits violation

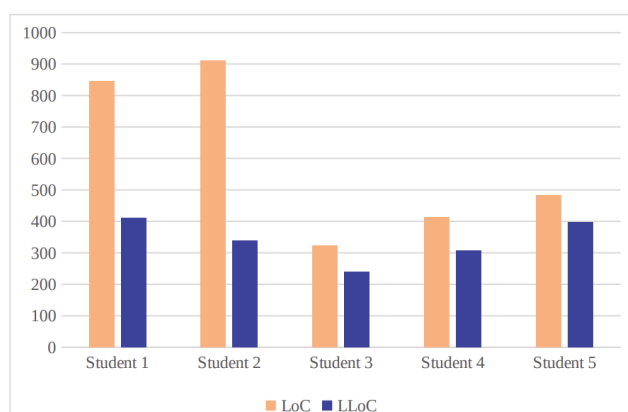


Fig. 4 Group comparison of LoC and LLoC metrics results

Based on the results, we can argue that authors of all analyzed projects have made significant progress in code quality. The improvement could be the result of a targeted effort, but also of a subconscious impact on the student.

This subconscious influence could be a result of attending lectures and working on exercises, where the students solve various tasks under the guidance of the teacher. In this way, students are confronted with common practices.

In Tab. 1 it is possible to see the points of students received through automated tests in the Arena system [17]. By comparing the points with the qualitative success of individual projects, it is clear that in the first one all students from the test sample achieved a high score, but the quality of the code reached a low level. On the contrary, in the case of the second project, students 1, 4 and 5 again received a relatively high score, while the quality of their code increased significantly. Using these observations, it can be argued that even poor code quality can provide the same level of functionality as lower code quality. Lower code quality could occur later when the code is further extended, where code weaknesses occur.

Table 1 Student score for analyzed projects

Student	Project 1	Project 2
1	100%	100%
2	80%	60%
3	100%	70%
4	100%	90%
5	90%	80%

Score of student 3 indicates that in the first project he/she achieved similar score as other students. The difference occurred in the second project. Compared to other students, better code quality was achieved, however, the score was lower due to failure in meeting functional requirements. After comparison with other metrics (logical number of rows shown in Fig. 4), low metric values were likely achieved by the insufficient implementation of all necessary functionalities.

3.2. Group Profile

The role of the *group profile* is to simultaneously compare a group of projects. Unlike the *standard profile*, the group profile does not provide the same level of detail in an overview of the analysis results, as it is focused more on the comparison of projects than on their detailed analysis.

In the following experiment, all five student projects were compared based on graphs and tables. Due to the objective comparison, all projects had the same task assignment (text game project).

Fig. 4 represents the results of a group analysis at the project level in the number of LoC and Logical LoC. Based on the light bar, it can be deduced that students 1 and 2 reached values well above the average, which may be the cause of too long code. The second variant is a reflection of the stylistic habits of the author, such as formatting or comment writing. It is also likely that student 5, who achieved

values significantly below the average, did not implement all the necessary functionalities listed in the project.

The first assumption can be verified employing dark bar, which shows the measured number of LLoC. Here, both students, who achieved above-average values in Fig. 4, came significantly closer to the average value. Therefore the comparison shows that in this case, both students used more comments than the other students.

The experiment results and the project score (assessment) point to the connection between the low number of LLoC and the missing functionalities defined in the assignment.

4. RESULTS

Based on the experiment results, we can argue that static analysis evaluated whether size metrics, complexity metrics, or clone detection can provide an overview of the author's programming habits. Programming habits are meant to be stylistic habits, for example, the ratio of comments to the number of LoC, the use of duplicate code, and the programmer's way of algorithmic thinking, reflected in the level of program complexity.

All these factors have great use in evaluating and monitoring the quality of novices' code. Individual findings can subsequently influence the educational process. For example, if high values of complexity metrics are recorded for a large number of students, topics related to the issue may be included in the lesson, allowing the student to improve the quality of their code and thus reduce the likelihood of errors in the source code. At the same time, comparing analysis results of one student provides the opportunity to monitor his/her progress in code quality.

Static analysis can be used in the educational process when evaluating projects, but it is also necessary to take into account the evaluation of the program functionality. An example is student 3, who achieved a high success rate in measured metrics but had a relatively low success rate in the evaluation of implemented functionalities.

Compared to similar experiment in Java [8], solution in C code are less complex, yet the results provide good but not better insight of the source codes. Since not all students achieved the same score, the metrics limits violations were influenced by different programming skills. More testing is planned to be sufficient enough for strong conclusions, since experiment in Java revealed much more findings and habits.

5. CONCLUSION

If available, programmers often read documentation or codebase to understand a software structure. Both approaches are tedious, time-consuming, and inefficient [18]. A programming language, together with the programming paradigm it supports, is a strong factor that profoundly impacts how programmers comprehend code [19].

The present article focused on enhancement of an existing programming profiler tool [7] to support the C programming language, and an analysis of the extent to which novice programmers comprehend their C code. Method of

generating programmer knowledge profiles was used [6], using techniques of various qualitative metrics and clone detection. In the experiment, we observed a group of novice programmers – students. The generated knowledge profiles allowed us to record their progress and level of program comprehension during one semester.

Based on the presented results, we believe the method of generation of programmer knowledge profiles contributes to the area of program comprehension and static code analysis.

ACKNOWLEDGEMENT

This work was supported by Slovak KEPA Agency under Grant no. "002TUKE-4/2021 Implementation of Modern Methods and Education Forms in the Area of Cybersecurity towards Requirements of Labour Market."

REFERENCES

- [1] DÉTIENNE, F.: *Software Design-Cognitive Aspect*. Springer Science & Business Media, 2001, ISBN 978-1-4471-0111-6
- [2] CORNELISSEN, B. et al.: *A systematic survey of program comprehension through dynamic analysis*, IEEE Transactions on Software Engineering **35**, No. 5 (2009) 684–702, doi: 10.1109/TSE.2009.28
- [3] MAALEJ, W. et al.: *On the Comprehension of Program Comprehension*, ACM Transactions on Software Engineering and Methodology **23**, No. 4 (2014) 31:1–31:37, doi: 10.1145/2622669
- [4] AHSAN, Z. – OBAIDELLAH, U.: Predicting expertise among novice programmers with prior knowledge on programming tasks. IEEE Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC), 2020, pp. 1008–1016
- [5] SULÍR, M. et al.: *Large-Scale Dataset of Local Java Software Build Results*, Data **5**, No. 3 (2020) 1–11, doi: 10.3390/data5030086
- [6] PIETRIKOVÁ, E. – CHODAREV, S.: *Towards Programmer Knowledge Profile Generation*, Acta Electrotechnica et Informatica **16**, No. 1 (2016) 15–19, doi: 10.15546/aei-2016-0003
- [7] PIETRIKOVÁ, E. – CHODAREV, S.: *Profile-driven Source Code Exploration*, Annals of Computer Science and Information Systems **15**, No. 1 (2015) 929–934, doi: 10.15439/2015F238
- [8] VEHEC, I. – PIETRIKOVÁ, E.: Metrics for Student Source Code Analysis. IEEE 18th International Conference on Emerging eLearning Technologies and Applications, 2020, pp. 739–744, doi: 10.1109/IC-ETA51985.2020.9379264
- [9] ARAGAO, L. – PEREIRA, M. – HENRIQUES, P.: Scaling up a Programmers' Profile Tool. Symposium on Languages, Applications and Technologies (SLATE), 2019, pp. 11:1–11:8, Schloss-Dagstuhl, doi: 10.4230/OASIs.SLATE.2019.11

- [10] OLIVEIRA, M. et al.: Software quality metrics and their impact on embedded software. *IEEE 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, 2008, pp. 68–77, doi: 10.1109/MOMPES.2008.11
- [11] NEJMEH, B.A.: *NPATH: A Measure of Execution Path Complexity and Its Applications*, *Communications of the ACM* **31**, No. 2 (1988) 188–200, doi: 10.1145/42372.42379
- [12] CAMPBELL, G.A.: Cognitive complexity: A new way of measuring understandability. *SonarSource*, Version 1.5, 2021
- [13] LI, W. – HENRY, S.: *Object-oriented Metrics That Predict Maintainability*, *Journal of Systems and Software* **23**, No. 2 (1993) 111–122, doi: 10.1016/0164-1212(93)90077-B
- [14] LI, B. et al.: Semantic Code Clone Detection Via Event Embedding Tree and GAT Network. *IEEE 20th International Conference on Software Quality Reliability and Security (QRS)*, 2020, pp. 382–393, doi: 10.1109/QRS51102.2020.00057
- [15] NASIRLOO, H. – AZIMZADEH, F.: Semantic code clone detection using abstract memory states and program dependency graphs. *IEEE 4th International Conference on Web Research (ICWR)*, 2018, pp. 19–27, doi: 10.1109/ICWR.2018.8387232
- [16] CHODAREV, S. et al.: Haskell clone detection using pattern comparing algorithm. *IEEE 13th Engineering of Modern Electric Systems*, 2015, pp. 145–148, doi: 10.1109/EMES.2015.7158423
- [17] ŠŤASTNÁ, J. et al.: *Security Measures in Automated Assessment System for Programming Courses*, *Acta Informatica Pragensia* **4**, No. 3 (2015) 226–241, doi: 10.18267/j.aip.71
- [18] ALANAZI, R. et al.: *Facilitating program comprehension with call graph multilevel hierarchical abstractions*, *Journal of Systems and Software* **176**, No. 1 (2021) 110945:1–110945:15, doi: 10.1016/j.jss.2021.110945
- [19] VALDECANTOS, H.A.: An Empirical Study on Code Comprehension: Data Context Interaction Compared to Classical Object Oriented. *IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 275–285, doi: 10.1109/ICPC.2017.23

Received August 2, 2021, accepted March 15, 2022

BIOGRAPHY

Emília Pietriková is Assistant Professor at the Department of Computers and Informatics, Technical University of Košice, Slovakia. She received her M.Sc. with honors in 2010 and her Ph.D. in 2014 in Informatics from Technical University of Košice. Her doctoral thesis title was "Abstraction and generation in programming languages". Her internships include the Department of Telematics at the Norwegian University of Science and Technology, Norway (2010), and the Department of Computer Architecture at the University of Málaga, Spain (2011). The subject of her research is source code analysis, programming techniques, and quality of higher education.