Program Comprehension and Quality Experiments in Programming Education

Maria Medvidova
Department of Computers and Informatics, Technical University of Kosice, Slovakia

Jaroslav Porubän
Department of Computers and Informatics, Technical University of Kosice, Slovakia

Abstract

The paper deals with the design of a new experimental method designed to measure the understanding of the code of subjects who do not know any programming language in connection with the implementation of empirical and analytical study. The aim of this work is the analysis of students' knowledge before and after the course Basics of Algorithmization and Programming at Technical University in Kosice, Slovakia, and the subsequent static analysis of their codes from one of the assignments. The theoretical part provides a look at the various models and ways to measure program comprehension, code quality metrics, examines the most common analysis tools, suggests recommendations for improving comprehensibility, and provides a closer look at program comprehension issues in the teaching context.

2012 ACM Subject Classification General and reference → Surveys and overviews

Keywords and phrases Program comprehension, static code analysis, empirical software engineering, code as a story, students

Digital Object Identifier 10.4230/OASIcs.ICPEC.2022.14

Funding This work was supported by project VEGA No. 1/0630/22: Lowering Programmers' Cognitive Load Using Context-Dependent Dialogs.

1 Summary of basic information about the program comprehension

During software development and maintenance, developers spend a significant amount of time in the process of understanding the program [12]. Studies suggest that understanding the code is a major maintenance activity because it absorbs approximately 50 percent of the cost. Problems in understanding the code are known from the time of the first developed software. The partial goal of this work is to observe these challenges from the student's point of view. As software engineering teaching expands, computer science teachers are encouraged to help students develop their understanding. Today we know that even if a programmer puts together valuable code, he may not understand it. Precisely because of this, our work is devoted to evaluating the level of understanding of code as an important part in the life of a developer [2]. We gradually analyze their ability to understand from the beginning to the end of the semester using a new empirical story method that we have developed for this purpose only, as most current studies unfortunately focus only on professionals due to the need to maintain existing software and lack student studies. In our research, we focus on various metrics that we can use to quantify code quality. We are thus preparing a unique opportunity to observe the development of participants in the subject Basics of Algorithmization and Programming, which we can later use in the evaluation of students and their codes representing potential security risks in the event of poor program implementation. Thanks to this measurement, we monitor the improvement or deterioration of potential students who may have difficulty mastering this subject in the future, and we can thus set preventive steps for the successful completion of the course [9]. In the first part, we cannot generalize their knowledge in a specific programming language, as everyone underwent a

different teaching in high school. Therefore, there will be general questions in order to determine the level of the student's understanding of the code using the proposed "code as a story" method. To measure understanding in the second stage, we decided to use the "understand code" protocol. Subsequently, we evaluate the level in understanding the program before and after completing the course. In both sections, the participant was measured from the time the question was displayed until they were answered using a programmed form using Google Apps Script.

1.1 Models of understanding of program

Models are created using theories based on empirical studies by programmers who perform tasks that require them to read and understand the program.The differences between human understanding models lie in the terminology that describes the content of the knowledge base and each approach to the process of assimilation.

- Top-down model: The process begins with a hypothesis about the general nature of the program. This initial hypothesis is then refined hierarchically by creating sub-hypotheses. Sub-hypotheses are refined and evaluated in depth first [1].
- Bottom-up model: Bottom-up program understanding theories assume that programmers first read code statements and then mentally divide or group these statements into higher-level abstractions [10].
- Cognitive model: The basic structure of cognitive models consists of four components. Target system, which is to be understood. The second component is the knowledge base, which represents the previous experience of the person. The third component is a mental model that shows the current state of understanding. The last component is an assimilation process that interacts with the other three components and updates the state of understanding [4].

1.2 Complications in understanding the program

In this section, we examine the scientific understanding of code odors. A clear example from the studies we are discussing is that long methods are particularly problematic. They make it harder for developers to understand what's going on and make changes, are more prone to error, and require more effort to change. The code smells like "blob class" and "spaghetti code", which are clearly related to it.

1.2.1 Code smells

The term "odor code" was popularized by Martin Fowler. Any symptom in the code that indicates a deeper problem can be considered "odors" in the code. This problem often cannot be seen right away, but can be uncovered if the code is thoroughly analyzed. Code smells are not program errors. They only show software design issues that make it difficult to understand the code and then make it difficult to develop or extend it. Code smells also increase the likelihood of future errors in this software. The presence of odors in the code is a sign of poor code quality. There are many potential odors. The most common code smells and thus the causes of poor code level in terms of understanding include:

- Duplicate code: If you see the same code structure in multiple places, you can be sure that your program will be better if you find a way to unify them.
- Shotgun Surgery: Occurs when one change requires adjustments in many classes.

- Feature Envy: This is code smell that describes when an object accesses another object's fields to perform an operation, instead of just telling the object what to do.
- Middle Man: Occurs when a function has too much delegation to other functions and methods and does almost nothing else because the work gradually moves to other methods.
- Extremely short identifiers: The name of a variable should reflect its function, unless the function is obvious.
- Too complex conditions: Extensive nested conditions tend to grow more and more complicated over time as developers constantly add conditions and more levels. The deeper the nesting, the more time it will eventually take to remodel.
- Spaghetti code: The nested if, for, do, while, switch, and try statements are a key component in creating what is known as "spaghetti code." Such code is difficult to read, refactor and therefore maintained.
- Unnecessary comments: Unused code should be deleted and, if necessary, retrieved from the source check history.
- Dead code: These are methods and classes that are no longer used.

1.2.2 Technical debt

The technical code debt is an insufficient and misspelled code that violates best practices and coding rules.Technical debt is like a collection of code odors along with bad architectural decisions. Resolving a technical debt usually takes longer. It is most often increased by adding another method to an already too long class, copying and pasting an existing class or method, but with a different name. But whatever the cause, technical debt will inevitably cause more work in the future by not introducing better solutions now. That's why Ward Cunningham coined the term "debt." The more you pay, the more interest you pay [3]. And in this analogy, refactoring is the process of paying off this debt.

1.3 Code comprehension measurement concepts

In most software engineering processes, a complete understanding of the entire program is unnecessary and often impossible. Change requests are often formulated in terms of domain concepts, such as "Add the ability to save a trusted device to the authorization system." An important task is then to understand where and how the relevant terms are implemented in the code. We know many protocols designed to measure it, the most well-known are think out loud protocol, remember, understand the code.

1.4 Approaches to improve the comprehensibility of existing code

There are a number of factors that can be helpful in improving the accuracy, correctness, completeness, and simplicity that a program can understand. In this subsection, we list the most common ones.

- Syntax highlighting: The function displays text, especially source code, in different colors and fonts depending on the category of expressions. It is commonly used to highlight different code constructs using different colors and fonts for different identifiers. Although this is a very basic approach, it undoubtedly contributes greatly to improving comprehensibility.
- Expertise in the domain: Many empirical studies have shown that expertise is important in building fast and accurate comprehensibility. Well-founded knowledge of the problem domain, troubleshooting, and programming languages make it easier to understand the program [6].

- Cross-references: Support is used to provide a link between an identifier definition and its use in different places in the source code.
- Annotations and comments: Annotation is another way to help a programmer easily understand code. These are virtually added comments that did not originally exist in the code, but are often more useful.

1.5 Tools helping to understand programs

In the early days of programming, source code was often written using a standard editor such as Emacs or Vi. However, as the complexity of the programs grew, the standard text editor was no longer enough, so the need for integrated development environments grew rapidly. The area of program understanding research has resulted in many different tools that help in understanding the program, and in this subsection we discuss the features of the tools that support program understanding. The tools for understanding the program can be roughly categorized according to three categories:

- extraction,
- analysis and
- presentation.

Extraction tools include analyzers and data collection tools. Analytical tools perform static and dynamic analyzes to support activities such as clustering, concept assignment, feature identification, transformations, domain analysis, division, and metric calculations. Presentation tools include code editors, browsers, hypertext, and visualizations.

It is best to use the integrated development environment (IDE) and its built-in application add-ins, which combine common development tools into a single graphical user interface and provide these features. However, if that's not enough, we provide a brief overview of the tools that help to understand. Rigi supports multiple views, cross-references, and bottom-up comprehension queries. Reflection tool supports a top-down approach by generating and validating hypotheses and helping to bridge the gap between source code and high-level models.The Bauhaus tool has features to support clustering and concept analysis for software architecture, software maintenance and program understanding. And Codecrawler is a search tool specifically designed for use to search for source code. We can classify the mentioned tools mainly in the category of analysis but also partially in the extraction.

2 Student and understanding of code

As the teaching of computer science expands, computer science teachers are encouraged to help students develop an accurate understanding. Introductory programming courses are challenging for students, and novices often show misconceptions that hamper their ability to learn and make progress. Students should test and analyze their code to identify and correct problems [8]. Here are some of the methods used to improve teaching: work by learning and learning by example, live coding, gameplay and gamification, mentor support, peer mentoring and programming, information and demonstration of bad practices leading to technical debt [7].

We think that the key knowledge for a beginning student to learn to program is the ability to solve programs, math skills, abstraction and creativity. A good knowledge of English also forms a solid foundation, due to the great support of the community in the English language, available material also in the form of videos, courses, forums but also English keywords in programming languages. Beginning programmers have trouble understanding

abstract concepts as real-life equivalents, the syntactic side of the language, and the actual execution and debugging of the program. On the other hand, the main challenge for teachers is to maintain motivation, relationship and choice of the right tools and language [11]. We will need more new professionals in the field than will be produced next year. The need for individuals in the field of informatics is expected to increase faster than the average of all professions.

3 Source code analysis

Code quality is important because it affects the overall quality of the software. And quality affects the extent to which your code base is secure, seamless, and reliable. When the code is of low quality, it can pose security risks. If the software fails due to a security breach or security flaw, the consequences can be catastrophic or even fatal, so high quality should be the goal of the entire development process.

3.1 Most used code metrics

There is no specific way to measure code quality, but there are several metrics that can be used to quantify code quality. Different teams may use different definitions and metrics based on the context of their area [6]. Code that is considered good can mean one thing to a car developer. And for a web application developer, that might mean something else. For this reason, we explain what code quality is, how to improve code quality, what important code quality metrics are, and how code quality tools can help, but we don't list just one specific procedure or metric. The chapter introduces the more well-known metrics used and their value.

The most widely used and discussed metric for estimating project scope is the number of lines of source code, referred to as LOC. It is currently used as a basis for other metrics and is used to evaluate the size of a software system or the effort in writing code, but it depends significantly on the programming language used. The basic premise is that the larger the software system, the more difficult it will be to understand and maintain it. LOC is a size metric, another simple alternative may be the number of functions. Halstead's metrics link metrics calculated from the line count and syntax elements of the program's source code.

In addition to indicators for evaluating the volume of work on a project, indicators for assessing its complexity are also very important for obtaining objective estimates for a project. As a rule, these indicators cannot be calculated at the earliest stages of project work, as they require at least a detailed design.

Cyclomatic complexity measures the number of linearly independent paths through the program source code and is calculated using a program flow control graph. The graph nodes correspond to the indivisible groups of program commands, and the leading edge connects the two nodes if the second command could be executed immediately after the first command. It can also be applied to individual functions, modules, methods or classes within a program.

Control flow metrics are a separate group of software metrics. These are based on the assumption that the more complicated the management flow, the more complex the program. The best known is the Chepin metric. The calculation is performed by analyzing the nature of the use of variables from the input-output list and serves to determine the degree of difficulty in understanding programs based on input and output data. The calculated weightings are used to express the different effects on the complexity of each functional group's program.

3.2 Static code analysis

Static analysis is best described as a debugging method by automatically examining the source code before running the program. The popularity of software quality analysis tools has increased over the years, with special attention paid to tools that calculate technical debt based on a set of rules, detect code odors, and usually also provide an estimate of the time required to correct technical debt. Of course, this can also be achieved by manual code checking. However, using automated tools is much more efficient.

There are many static analyzers on the market. Examples of some tools are CodeScene, Clang Static Analyzer, SonarQube, DeepSource, OCLint, Embold, Klocwork, Coverity Scan, Fortify Static Code Analyzer. Based on the above list of synthetic analyzers, SonarQube was selected for research. It is suitable for evaluating programs taking into account the desired metrics to be evaluated in this work at the same time is efficient and easy to use.It has a paid and unpaid version, which we used for our needs.

4 Study design

In designing the empirical study, we identified the key variables to be measured during the study, the appropriate subject population for the study, and the reference tasks [5]. These steps will be described in more detail in this section. We used the most suitable tool selected in 3.2 and then evaluated all the collected data.

4.1 Division of the study

The study is divided into empirical and analytical parts using different methods. In the analytics, student codes will be examined and measured using quality metrics for one and the same task. The course of research will be divided into 2 main parts, namely the data collection before and after the introductory course. With regard to the above division, research questions will also differ. In the first part, we cannot generalize students' knowledge of a specific programming language, as everyone underwent a different teaching in high school. Therefore, there will be general questions in order to determine the level of the student's understanding of the code using the proposed code as a story method. In both stages, the participant will be measured the time from the display of the question to his answer using the programmed form in Google Apps Script. The second part of the study will be carried out after the completion of Basics of Algorithmization and Programming course, so it is assumed that students' knowledge should be unified. In this introductory course, the aim is to acquire basic knowledge and skills of programs in procedural language C. Questions will therefore use programming language C.

4.2 Studied population

One of the goals of our study is to find out how students perform activities related to understanding the program. The research was focused on full-time students of the introductory course Fundamentals of Algorithmization and Programming at the Technical University in Kosice in the academic year 2021/2022, who were given the opportunity to answer questions electronically at the beginning and end of the course. The majority of participants are 18-20 years of Slovak nationality with a predominant male gender. The course has more than 400 participants every year. Although everyone was contacted, in the first stage, the form was completed by only 27 students. In the second stage, only the participants of the first were contacted, but we received feedback from the survey from 7 students. The

researched codes are available from 19 students participating in the first series of studies. In the study, the response rate was 7 percent in the first and 26 percent in the second phase of the questionnaire, so the number of people who responded to the survey was divided by the number of people in the sample. We attribute the low level of response to the lack of interest of students in the researched topic and no obligation to fill in the questionnaire.

4.3 Research questions

The purpose of this study is to examine the relationship of understanding the code of early programmers before and after the introductory course and the correlation of study performance in programming among first-year students of the Technical University who attended the Basics of Algorithmization and Programming. In view of the above objectives, we have identified the following research questions:

1. What are the students' abilities in understanding the programs?
2. Is there a difference in students' understanding of the code before and after the introductory course?
3. Are there any relationships between the ability to understand the code and the learning performance in the introductory programming course?

4.4 Proposed experimental method "code as a story" and demonstration of its use in research

When designing the study, we solved one fundamental problem, and that was how to examine the understanding of code in a group with different knowledge. First-year students come with different academic training. Some have not even encountered programming yet. An experimental method has been proposed in which the code is presented as a story. It focuses on comprehension of the text, memorization, attention and detail. The text was categorized on the basis of programming properties and relationships to individual groups and represented as a text game called "Peter's Friday". The identified categories were condition (problem with nested if-else-if, switch logic problem), cycle (for cycle, while - do cycle), bits operations (AND, OR), fields and pointers. The "code as a story" method used is based on using real-life situations or their simulations in connection with programming turns and focusing the content of the story on the researched population.

The whole story begins with the student's introduction to the story and the first question whether the main character Peter will take an umbrella when the logical condition is formulated: " in the weather at the FM station they said that it would be rainless in the afternoon or my mother warned me that it would rain all day from nine, I will take the umbrella anyway ". This part is focused on bit operations AND and OR, the student must be aware of this and correctly evaluate the statement. When evaluating the whole expression, it is found that the statement takes the value 1 and therefore the correct answer should be YES.

The problem with the nested condition "if-else-if" where the goal is for the student to correctly identify two situations that are "game changer" and evaluate what outputs can arise from situation. It can help students to compile a flowchart to find out that there can be only 2 situations. The third question falls into the category of pointers that we simulated with a finger pointing situation. The array category is represented by the fourth part of the story. Specifically, it focuses on going through them from beginning to end and vice versa and thus the sequence of elements in the field. In our story, the field represents a dryer and things hung on it. The student must be aware of the sequence of things on the dryer (sequence of elements in the field) and based on that determine whether the chosen things are correct.

The fifth task is focused on understanding the main condition of the cycle. The logic of this snippet can also be understood as a while-do loop. There is a catch there, the condition states "yellow building on the right", while the text ends that Peter saw "yellow building on the left". The next passage contains a reference to the SWITCH construct. So the condition can be understood, depending on what beer I order, I pay so much for three pieces. The story states that Peter ordered a dark beer, which means that the correct answer is 7.5 euros. The last question and its text bridge to the "for" cycle, where we know the exact number of repetitions in advance and also to remember the information about the price of the product mentioned in the previous question. Determines whether the student can identify the specified number of repetitions.

4.5 Data collection

We decided to conduct the research online using a form, taking into account culminating wave of the COVID pandemic at the time of the research. A questionnaire programmed with Google Apps Script was used. It was advisable to have the time of each student's response recorded, and no free software met this condition. Completing the questionnaire online was quick, easy, participants just needed to write the answers in the appropriate field. The answers were automatically stored in a clear table in a file, which was then used to evaluate the research. In the report, we simply see the time needed to understand each task. Significant differences were excluded from the responses examined. Based on time, we ruled out answers to one question in less than 10 seconds and longer than 30 minutes.

5 Summary of results

An important step in the survey was to map the current state of knowledge of students' code with the intention of subsequent analysis of their codes for the same assignment. In this part, the collected data from the empirical and analytical part are represented.

5.1 Evaluation of questionnaire research

The total number of participants was 27, the first part deals with the evaluation of their understanding of the program using the story structure of the code and then, the second paragraph, evaluates the data from the form with the same categories submitted in C.

5.1.1 The first series before completing the course

The questionnaire was filled in on 26.10. - 16.11.2021. After analyzing the first ten answers, we decided to ask question number 2 differently, because we found that the analysis focused on the knowledge of determining the probability and not the "if-else-if" condition. A modified version of this question finds out the number of different paths, the original wanted to know the percentage of probability of the described situation. Following the change in wording helped to increase the success by 6 times. Questions from the field categories, the "switch" and the "loop for" have the most correct answers, while the least from the "while-do" and pointers have the most correct answers. Students answered the longest question in part of the fields, averaging 305 seconds, which is about 5 minutes, and the shortest, only an average of 46 seconds, a question from the "for cycle" category. For a better idea of what story form of the questionnaire looked like, we offer a sample question. The logic of this passage can be understood as a while-do loop with a focus on understanding the main condition of the

loop. There is also a catch, the condition states "yellow building on the right", while the text ends that Peter saw "yellow building on the left". Thus, the condition for ending the cycle was not met and main character of story, Peter, had to continue straight on, still moving 10 meters, and so the student has to identify whether the specified minimum number of cycle repetitions is correct or not. According to story example below, the correct answer is yes, Peter walked more than 65 meters. It is necessary to think that the questionnaire was designed to gradually introduce the student to the story and therefore, this sample is torn from the story.

> When he finally manages to find one of his T-shirts, he sets off. He and his friend Michael have long ago agreed that they will go for a beer, but since Michael did not remember the exact name of the company they were going to, he only gave Peter a description of the route and the name of the stop he is to get off: "When you leave the stop straight until you find the yellow building on the right and then immediately turn left." Each building is exactly ten meters away from the next, and the distance between the stop and the first building is exactly ten meters. Pete walked past the five buildings and began to doubt that Michael was really remembering the way when he suddenly noticed the first yellow building on the left. After a while, he finally got to the business Michael was telling him about, and he looked angrily at the distance traveled on his watch. Did Peter walk more than 65 meters from the stop until he finally found the yellow building? (Yes/No)

The research shows the expected results. Simpler concepts such as fields or cycles have a success rate of 80 percent or more with a smaller average response time. On the other hand, the concept of pointers or bit operations has a success rate of 60 percent or less with an average time of 100 seconds or more. A total of five students out of 27 answered all the questions correctly. The relatively simple concept of "if-else-if" remains an unexpected result, where the observed increase in success from 10 percent to more than 60 percent is still a relatively small ratio, and its value ranks among the less frequently answered categories of questions.

Data from the point of view of individual questions were evaluated above. Figure no. 1 shows the success rate of each student and the total response time. Significant differences in overall time are visible and it cannot be said unequivocally that completion time plays a role in success. The best results using the experimental method are shown by students s13 and s17, who completed the form in 10 minutes and at the same time achieved 100 percent success.

5.1.2 The second series after completing the course

The second series was attended by 7 students, namely students with numbers s2, s9, s10, s13, s18, s25 and s26. The form was filled in on 03.02. - 14.02.2022. The percentage of correct answers was 100 percent only for a question from the field category and, conversely, the success rate of 70 percent was achieved by questions from the categories of both cycles. Students answered the longest question in the section of cycles and the switch, namely an average of 2.5 minutes and the shortest, only an average of 48 seconds to a question from the category of the cycle "if-else-if".

The average time to complete the questionnaire is around 10 minutes, up to a significantly higher value of student s25 with a completion time of 28 minutes. Students s9, s13 and s17, who completed the form in 9 minutes and achieved 100 percent success, show the best results using the C code output method. Students with these numbers also achieved 100 percent

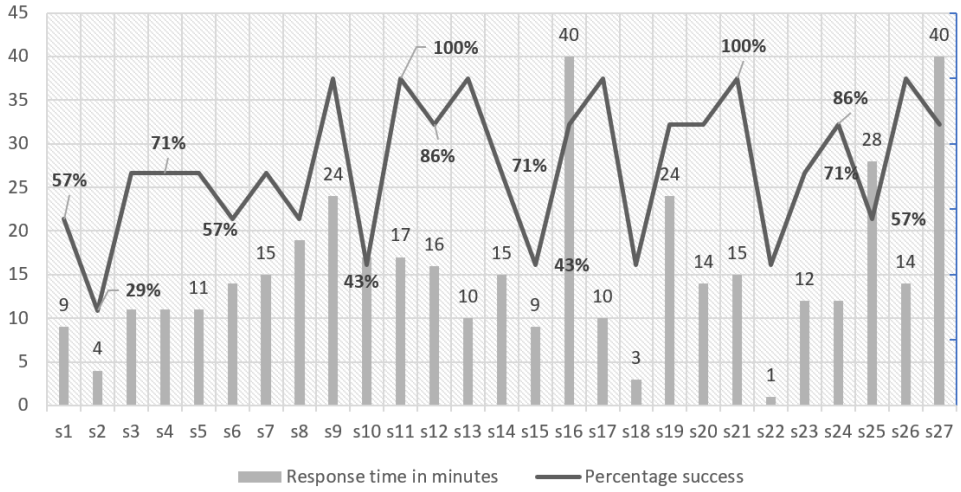

Figure 1 The success rate of each student and the total time to respond.

success in the first stage. The student with the number s2 with 57 percent achieved the lowest success rate from the monitored sample. The participant with serial numbers s25 has the worst ratio between the length of answers and the overall success rate. Below is a sample code in one of the questions and we want from students to enter the output of the program.

```
int main(){
   int a = 6;
   int b = -1;
   while (b < 2) {
     if (a <= 10) {
       a = a + 4;
     }
     else {
       a = a - 10;
     }
     printf("%d, ", a);
     b = b + 1;
   }
   return 0;
 }
```

5.2 Static analysis of student code quality

An analysis of the codes of the students involved in the research was performed on submitted codes from one of the assignments, so they all wrote program for the same problem. The results show that student s12 achieved the highest cognitive complexity 48 and the lowest student s25 only 18. Student s21 had the highest cyclomatic complexity and the lowest again s25. Except for students s16 and s24, they all have a cognitive complexity higher than cyclomatic. In the examined sample, the average cognitive complexity was 34 and cyclomatic 30.Student s11 has the highest number of comments 34.4 percent. Of course, there are also students like s24, s5 and s4 who have no comments. Although student s13 has only 129 lines, his number of logical lines is 108, which is the smallest difference between the values. The lowest number of LOCs is s25, which may be why it achieved the lowest cyclomatic and cognitive complexity.

Regarding the number of errors, we observe only four students who recorded 1 to 3 errors. These were errors performing arithmetic and logical operations with uninitialized variables. The number of odors in the code is interesting. Student s11 has the highest number, namely 29, and participant s3 has the lowest number of odors 3. This is a significant disparity between students. Student s3 performed best with 0 errors, odors 6 and the lowest technical debt 0.7 percent. Given the achieved LOCs numbers, we observe a very high incidence of odors on disproportionately small programs. The technical debt ranges from 30 to 150 minutes. All students achieve very well the results of technical debt repair within 1 or 2 hours.

Based on these data, we can evaluate that students in the first year of the introductory course of the Technical University best understand the logical turnover of fields, switches and bit operation evaluation, on the contrary, they understand the cycle and pointer the least. The students who did the worst in the form sections (s2, s10, s18, s15 and s22) did not fulfill the selected assignment from the course and therefore we could not examine their codes, which they never submitted. It is also possible to note that the nominees had one of the worst successes in the experimental method and the results were confirmed after the end of the course using a validated C output determination procedure. Comparing static code analysis with success in measuring code comprehension yields interesting data. We believe that if the students who did the worst in the empirical part submitted the assignments, they would be evaluated poorly in code quality and we can also say that the ability to program the assignment may be related to understanding results, as the worst participants by research did not submit the assignment. Furthermore, students with numbers s1, s11, s14 and s25, whose codes also have a high degree of complexity, errors and technical debt ratio, show the worst results of the submitted assignments in the odor number metric. These participants have an average ability to understand the code.

6 Conclusion

An experimental “code as a story” method was designed and used, and data from 27 students from two stages of research were processed. Finally, 19 student assignments underwent a static analysis. The biggest problem was the low turnout, but mapping this process provided some interesting insights. Research shows that more than half of students incorrectly answered one or more questions about the implementation of basic programs and the associated understanding of the code. There is a slight improvement for participants who have also completed the second stage of the forms after completing the course. Less than half of the students were able to answer trivial questions, and in interviews with them we found out that they knew the programming terms used to program the assignment, but had trouble determining the output of the program shown, so they did not understand the code. In static code analysis, we observe a correlation in bad results and non-submission. With average and above-average results in the empirical part, students have a lot of odors in the code. In 68 percent of students, 10 or more odors were identified, which is an average of LOC 137 for every 10 lines of code. The most common odors were spaghetti code, empty-body functions and methods, never used but defined variables, extremely short naming. The highest number of odors was 29 and the lowest 3. This is a significant disparity between students. We observe the appropriate use of the acquired knowledge in the possibility of improving the teaching of introductory programming courses. The path is in the integration of coding standards when passing mandatory fields. Using a coding standard is one of the best ways to ensure good code quality. The coding standard ensures that everyone uses the right style. It improves

consistency and readability, and this is the key to reducing complexity and improving quality. We recommend that students complete a text story in the first days of study to identify weaker course participants and then draw more attention to their shortcomings in order to avoid failure in the exam or credit in this subject. We see an opportunity to continue our studies, but a larger sample of students will be needed.

References

1 Rodney A Brooks. Planning collision-free motions for pick-and-place operations. *The International Journal of Robotics Research*, 2(4):19–44, 1983.

2 Malcolm Corney, Donna Teague, Alireza Ahadi, and Raymond Lister. Some empirical results for neo-piagetian reasoning in novice programmers and the relationship to code explanation questions. In *Proceedings of the fourteenth australasian computing education conference*, volume 123, pages 77–86, 2012.

3 Rosemary T Cunningham. The effects of debt burden on economic growth in heavily indebted developing nations. *Journal of economic development*, 18(1):115–126, 1993.

4 Françoise Détienne. *Software design–cognitive aspect*. Springer Science & Business Media, 2001.

5 Massimiliano Di Penta, RE Kurt Stirewalt, and Eileen Kraemer. Designing your next empirical study on program comprehension. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 281–285. IEEE, 2007.

6 Amy J Ko and Bob Uttl. Individual differences in program comprehension strategies in unfamiliar programming systems. In *11th IEEE International Workshop on Program Comprehension, 2003.*, pages 175–184. IEEE, 2003.

7 Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4):119–150, 2004.

8 Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 125–180. ACM, 2001.

9 Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, 62(2):77–90, 2018.

10 Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19(3):295–341, 1987.

11 Yizhou Qian and James D Lehman. Correlates of success in introductory programming: A study with middle school students. *Journal of Education and Learning*, 5(2):73–83, 2016.

12 Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on software engineering*, 5:595–609, 1984.