International Journal of Engineering Education Vol. 33, No. 4, pp. 1345–1360, 2017
Printed in Great Britain

0949-149X/91 $3.00+0.00
© 2017 TEMPUS Publications.

Software Quality Standards and Lean Approach in Teaching and Learning Programming*

MILOŠ MILIĆ, SINIŠA VLAJIĆ, ILIJA ANTOVIĆ, DUŠAN SAVIĆ, VOJISLAV STANOJEVIĆ and SAŠA LAZAREVIĆ
University of Belgrade, Faculty of Organizational Sciences, 154 Jove Ilića, Belgrade, Serbia. E-mail: mmilic@fon.bg.ac.rs, vlajic@fon.bg.ac.rs, ilijaa@fon.bg.ac.rs, dules@fon.bg.ac.rs, vojkans@fon.bg.ac.rs, slazar@fon.bg.ac.rs

The aim of this research is to improve the process of teaching and learning programming. We have considered ISO/IEC 9126 software quality standard and the Lean software development method applied to the process of teaching and learning programming. Taking into account that each software system is characterized by the software syntax correctness, software semantic correctness, and software quality, we argue for incorporating a software-metrics driven practice in the process of teaching and learning programming. In this context, each software system should be in compliance with a software development process. The fundamental principle applied in Lean programming education process is the detection of waste (in terms of software quality violations, partially done work, motion, and defects) in the education process. In this way, it is possible to improve the process of teaching and learning programming through a continuous inspection and improvement. We have developed a software tool in order to improve the process of teaching and learning. To evaluate this approach, we conducted an experiment with a total of 30 undergraduate students in which we investigated the violations of software metrics in the students' software projects. Although the number of participants in the experiment was limited, our findings confirmed that software quality standards and the Lean software development method can be successfully applied to the process of teaching and learning programming.

Keywords: programming; software quality; software metrics; ISO/IEC 9126; lean; education

1. Introduction

Software engineers are encountering different problems in software development process. Business applications are distributed on multiple machines and within multiple components. In addition, they must be scalable and able to concurrently support the work of a large number of users. At the same time, applications should be easy to maintain. These circumstances constantly impose the need to deliver software faster, at lower cost and with fewer defects [1, 2]. For these reasons, even in the earliest periods of software development experts saw the development of software systems as a process with precisely defined activities.

Software development is a complex process involving a number of models, methods, strategies, and activities [3]. Therefore, a proper approach to a software engineering education process is of a high importance. Some of the questions that are in the focus of researchers are related to software engineering curricula, specific courses, teaching and learning methods, and industrial relevance [4–7].

Programming is important part of a software construction, which refers to the detailed creation of working software through a combination of coding, debugging and testing [8]. The main objective of initial programming courses is to develop students' analytical and problem-solving skills, which could be expressed using a specific programming language [9, 10]. There are many proposals for teaching programming [9–14] that differ in scope, purpose, tools and techniques used. However, different learning problems are widely reported. The lack of problem solving abilities and finding bugs in programs are identified to be among the reasons that cause learning problems [9, 15, 16].

In this paper, we introduce the Lean approach in teaching programming with a focus on creating value for students and teachers. This is achieved by a continuous inspection and improvement of the quality of software produced by students. Software quality standards define the software quality attributes in terms of software metrics. Software metrics directly measure the fit of software quality attributes, and at the same time measure the level of quality of the overall software product. Lean approach incorporates software quality standards and software development methods in the process of teaching and learning programming. We promote the idea of teaching students to develop programs in a manner compliant with the procedures specified in software quality standards. In order to verify the feasibility of the proposed approach, SilabMetrics tool have been developed as an extension of SonarQube, a software quality tool based on the ISO/IEC 9126 software quality standard.

The paper is organized in 7 sections. Section 2 presents the importance of software quality in the software development process, as well as the Lean approach, which advocates that the most important goal of the production is creating a value for the

** Corresponding author.

* Accepted 28 February 2017.

customer. Section 3 presents Lean approach in the process of teaching and learning programming. Section 4 presents the experimental method used for verifying the feasibility of the approach. Threats to validity are presented in Section 5. The discussion is presented in Section 6. Finally, we summarize the results of the research, identify the limitations, and present the main conclusions and further research directions in Section 7.

2. Background

2.1 Software quality standards

The application of software systems to today's business is quite diverse, and their proper functioning is of the key importance for business success, safety and economy [17]. For each software development project, it is important to define its specific meaning of software quality during the early planning phase [18]. Software quality improvement methods have a valuable role in the software engineering practice [19]. Some of these methods include code inspections, design walkthroughs, prototype simulations, and measurement-based analyses [20].

Software quality assurance does not represent an optional activity in software development. In the extreme case, a defect in a software system can cause major problems, e.g., increasing the cost of software maintenance, which can lead to dissatisfaction of customers [21]. One of the methods used by many organizations is to comply with standards in writing programming codes. This ensures uniformity and compliance with the best practice, and consequentially maximizes the quality attributes of the software system in accordance with the defined quality model.

The quality of a software system can be checked by static and dynamic analyses of software quality. Static analysis refers to software quality analysis without running the program. On the other hand, a dynamic analysis is the analysis of the attributes of a running program and covers areas such as software performances. Dynamic and static analyses are complementary techniques [22, 23]. Figure 1 shows the tools that can be used for static and dynamic analyses of software quality. Since the paper considers the quality of the software in the context of teaching and learning programming and promotes continuous inspection and continuous improvement of all parts of the software (which does not have to be executable), the focus will be directed to the static analysis of software quality. Using tools for the static analysis of software quality enables faster identification of bugs and defects in the software, as well as their correction in the early phases of software development process, when it is much easier and cheaper to correct them [24].

ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission) are international organizations whose activities are directed towards establishing international standards in different areas. The standards ISO/IEC 25000 [25], ISO/IEC 14598 [26] and ISO/IEC 9126 [27] are especially important for software engineering. ISO/IEC 14598 gives a general framework for the evaluation of software products using the model in ISO/IEC 9126 standard [28]. Standard ISO/IEC 9126 is used to define the software metrics that are used for measuring the performances of software system. This standard is also used to define the characteristics (or quality attributes) of software system and adequate metrics are being defined for each characteristic accordingly [29]. ISO/IEC 9126 standard includes four parts [27, 30]: ISO/IEC 9126-1 defines the Quality model, ISO/IEC 9126-2 defines External metrics, ISO/IEC 9126-3 defines Internal metrics and ISO/IEC 9126-4 defines Quality in use metrics. The ISO/IEC 25000 series of standards will replace and extend ISO/IEC 9126 and ISO/IEC 14598 [28, 31].

Standards define the quality model (e.g., ISO/IEC 9126 quality model, ISO/IEC 25000 quality model), whereas each quality model contains several quality characteristics. On the other hand, each characteristic contains several subcharacteristics, while each subcharacteristic contains several software metrics, as shown in Fig. 2.

The cost of a completion of software project is a very important aspect in software development process. Given that the production of software is a series of intellectual and technical activities performed by highly-educated engineers, it becomes

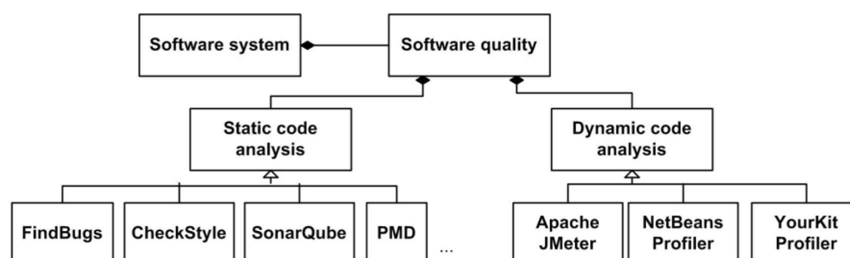

Fig. 1. Tools for static and dynamic code analyses of software quality.

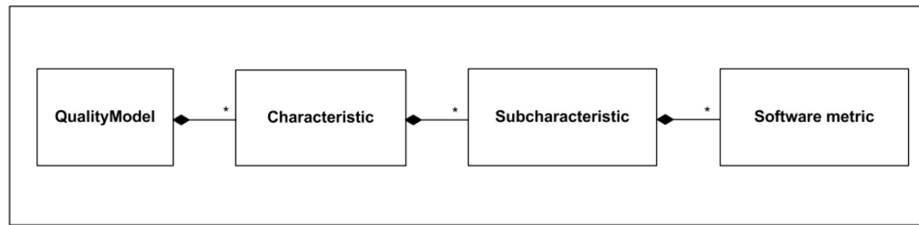

Fig. 2. The relationship between quality models, characteristics, subcharacteristics and software metrics.

clear that the cost of the system greatly depends on the time and effort required for its production [32]. By measuring various characteristics of the software product and its development process, actions can be taken to increase software quality and reliability [20]. In order to ensure the corresponding quality, it is necessary to perform a proper specification and evaluation of software quality. This can be accomplished by defining the corresponding attributes of software quality, taking into account the purpose of the application of the software. Software attributes can characterize software quality of both the product and software development process at the same time [20].

2.2 *Lean approach*

The development of manufacturing has created a need for different methods and approaches in production in order to increase the level of quality and assure continuous improvement of the production process. Lean approach postulates that the most important goal of the production is creating a value for the end customer. Any consumption of resources for any other purposes is therefore a waste and should be eliminated [33, 34]. From the customer's perspective, the value represents any action or process that a customer is prepared to pay (the customer is certainly not prepared to pay losses and problems that occur in the production process). Lean approach has had a significant impact on other approaches and, in addition to its application to production, it is implemented in the administration, services, education, software engineering etc.

We can consider the software development process in this context. A contemporary software product is very complex; over the time it has evolved from a product that represents the result of work of an individual (or a small number of developers) to a product that represents the work of hundreds of multidisciplinary professionals who are often geographically dislocated. In addition, software development is a complex process involving a number of models, methods, strategies, activities, techniques, and tools. Hence there is a need for good organization of the software development process by selecting a software development approach that best suits a particular software product.

Software engineering is becoming increasingly agile, focusing on activities that impact on how software is developed on a daily basis [35]. The agile methods of software development (e.g., Extreme programming, Feature Driven Development, Adaptive Software Development, Scrum, Lean Software Development) are widely used in software development process [36-38]. Lean is a software development method based on the transfer of positive experiences from the classic manufacturing into the software development industry [33]. It is primarily oriented on people and teams, which means that it puts software engineers, who directly produce software, in the central role. Lean, as agile method, promotes software development in several small iterations (usually from one to four weeks). Each iteration requires teamwork during all activities of software development life cycle, including gathering requirements, analysis, design, implementation, and testing phases. In addition, each iteration should provide the required functionalities, without bugs in the code or logic, in order to make further customer requirements relate to future software versions rather than to correct errors in the current software version.

The fundamental principle applied in the Lean software development is the elimination of waste [33, 37]. Authors Marry and Tom Poppendieck in [36] define seven types of waste in the software development process and compare the list against the waste of the manufacturing process, as shown in Table 1.

Table 1. The Seven Wastes of Manufacturing and The Seven Wastes of Software Development [36]

The Seven Wastes of Manufacturing	The Seven Wastes of Software Development
Inventory	Partially Done Work
Extra Processing	Extra Processes
Overproduction	Extra Features
Transportation	Task Switching
Waiting	Waiting
Motion	Motion
Defects	Defects

The waste in Lean software development represents everything that does not bring the value to the product from the end-user's perspective:

- Partially Done Work refers to partially done software development. The big problem with partially done software is that a software development team might have no idea whether or not it will eventually work.
- Extra Processes refer to processes that consume resources and slow down response time. Software development team should concentrate on value-adding processes to the final software product.
- Extra Features refers to adding new features to the software product that are not required. Taking into account that every line of code increases complexity and is a potential failure point, every extra feature is a serious waste.
- Task Switching refers to assigning people to multiple projects. Belonging to multiple teams or multiple projects usually causes more time in order to switch from one task to another. Task switching time should be eliminated.
- Waiting refers to delaying in software development process (e.g., delays in starting a project, delays in testing, and delays in deployment). These delays increase time required for software development.
- Motion refers to moves required in order to finish the assigned task (e.g., if a software developer has a question, how many motion does it take to find out the answer).
- Defects refer to defects in a software product. Defects can have high impact on further software development and maintenance. Therefore, it is very important to detect defects as soon as they occur.

It is clear that these wastes e.g., "Partially Done Work" or "Defects" do not represent the value for the end-user; these defects must be eliminated by a software development team. In this context, the first step in waste elimination is its detection [36]. The Lean software development is based on the continuous improvement of people, processes, and technology and acting in accordance with that. In this way, it is possible to correct defects in the early phases of software development when it takes less time and effort for their correction.

We examine the Lean approach in the process of teaching and learning programming which is instructed by the teacher. Taking into account that students are fully committed to the course and their software projects (each student has its own project), there are no "Extra Processes" and "Task Switching". Students have precisely defined assignments (previously approved by the teacher), which eliminates "Extra Features" and "Waiting". In this context, we will consider "Partially Done Work", "Motion", and "Defects" as wastes that should be eliminated.

2.3 Lean approach in engineering education

Engineering Education is facing more and more challenges like: the proliferation of information, the need for multidisciplinary for technological development, the globalization of markets etc [39, 40]. These challenges require appropriate teaching and learning methods and strategies in order to bridge the gap between academy and industry.

Lean thinking principles are accepted internationally among many companies [34]. Therefore, Lean provides ideal platform to educate engineers [40]. Lean in Engineering Education includes a systematic, student-centered and value-enhanced approach to educational service delivery [41]. This enables students to develop skills by integrating comprehension, appreciation and application of tools and concepts of engineering fundamentals and professional practice [41].

Providing engineering students with knowledge of Lean principles has positive impact on engineering education; in this way students can apply theoretical knowledge in solving real-world problems [42]. This could be achieved through Project-based learning (PBL), which is a process of learning through the practical application of theoretical knowledge [42–44]. This is one of the major trends in software engineering education [43].

3. Lean approach in the process of teaching and learning programming

3.1 Description of the proposed approach

Lean approach can be applied in teaching and learning programming. This is achieved by incorporating software quality standards in the process of teaching and learning programming. More precisely, the process of teaching and learning programming is guided by software quality standards. In this context, we examine the transformation of the programming task to task solution (represented by students' program code), as shown in Fig. 3.

In other words, software quality standards are used in the specification and evaluation of software quality of a students' program code-task solution. Our approach defines a model of software quality and software metrics for measuring the level of software quality. This implies that a student's program code has to be compliant with the software metrics specified in the software quality model.

In the process of learning programming students receive programming tasks and transform them to program codes. To facilitate the proposed Lean approach to teaching and learning programming,

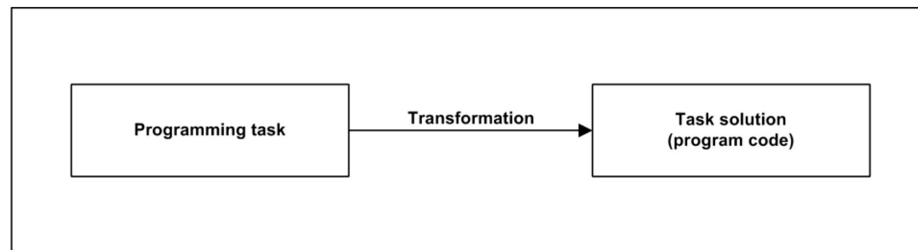

Fig. 3. Transformation of the programming task to task solution guided by software quality standards.

we have developed the SilabMetrics software quality tool. Using the SilabMetrics software quality tool students have the opportunity to analyze the program code and thus obtain feedback on its quality. Feedback includes information about the analyzed software classes, lines of code where a software metric has been violated as well as the description of the problem. In addition, the feedback includes a detailed explanation of the problem, usually with examples of compliant and non-compliant solutions. For example, if a student uses a hard-coded value in a statement, SilabMetrics will explain that this program code is hard to test and maintain, and will suggest defining a constant. Consequently, students can better understand the mistakes they make and what should be done to create error-prone code.

Teachers can also use SilabMetrics tool to analyze the program code written by their students. In this way teachers can understand where students make mistakes in the process of learning programming, or what they should do to improve the applied instructional design. In this context teachers can improve their own knowledge of students' learning process as well as assist their students in developing programming knowledge and skills. Consequently, students will learn to pay more attention to the quality of the written code while learning programming.

Writing and analyzing small segments of program code lead to immediate detection of problems in teaching and learning programming. In Lean manufacturing this concept is called Poka-Yoke [34, 45]. Poka-Yoke includes any mechanism in the manufacturing process that prevents participants from making mistakes. The analysis of software quality is a good example of application of the Poka-Yoke concept in the process of teaching and learning programming. In this way, it is possible to correct partially done work, motion, and defects in the early phases when it takes less time and effort for their correction. It directs students to write a program code that is in compliance with defined software quality model.

Software quality standards provide students and teachers with good insight into software systems quality. Therefore, we can say that software quality standards are related to all stakeholders in the process of teaching and learning programming.

Although the implementation of Lean approach in software engineering is not new [33, 36, 38, 46], it is not typically applied to teaching and learning programming. The proposed Lean approach in the process of teaching and learning programming is shown in Fig. 4. In order to present the process of teaching and learning programming we have used role/task-oriented perspective [47] in which a set of tasks is performed by two different roles: student and teacher. We can observe different tasks related to each role as well as different artifacts that are used within specific tasks. Consequently, it is possible to improve the process of teaching and learning programming through continuous inspection and continuous improvement, as shown in Fig. 4.

As discussed in Section 2, the most important concept of Lean approach is the elimination of waste and the first step to achieve this goal is waste detection. In the context of learning programming, detection of waste refers to detection of partially done work, motion, and defects in code and software metrics violations. To this end the Lean approach in learning programming involves the implementation of the following concepts:

- Continuous inspection; refers to a continuous inspection of the software quality, students' knowledge, but also teachers' knowledge and the overall building process knowledge. We can analyze the current result (program code), recognize the shortcomings in the software quality, and thus create the basis for improving the software quality and the knowledge of all the participants in the process of teaching and learning programming. In our approach, Continuous inspection is supported by SilabMetrics tool.
- Continuous improvement; involves a constant effort to improve the software quality and the process of teaching and learning programming. As a result of using SilabMetrics tool we obtain quantitative indicators on software quality (e.g., compliance with the software metrics, violated software metrics, number of defects, defects

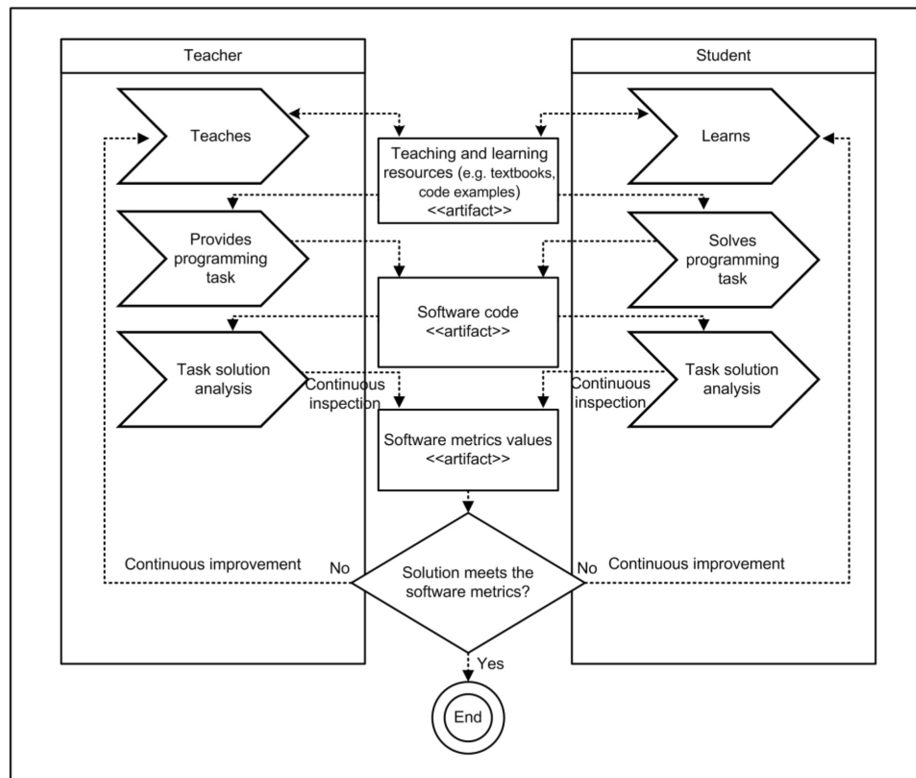

Fig. 4. Lean approach in the process of teaching and learning programming.

severity). These software metrics are in compliance with a defined software quality model. In other words, SilabMetrics makes it possible to perform the analysis of a programming tasks right after its completion, realize its shortcomings and make the required improvements. The new solution can also be analyzed and improved in order to obtain the desired level of software quality.

Continuous inspection and continuous improvement of software quality allows changes in software at the time when they are needed, which leads to less waste and explicitly affects the software quality. In addition, continuous inspection and continuous improvement of program code prevents one programming error from repeating several times.

3.2 SilabMetrics software quality tool

The described approach is independent of the programming language, problem domain as well as the size and type of software systems that students develop. We have developed the SilabMetrics software quality tool that performs static analysis of program code to prove the feasibility of this approach.

SilabMetrics tool is language dependent: it supports static code analysis in Java which is object-oriented language. This implies that we currently support object-oriented programming paradigm. On the other hand, Java is one of the most popular programming languages used in software development. The tool is integrated with NetBeans development environment to improve the learning process. It is based on SonarQube software quality tool that contains SQALE quality model [48, 49]. SQALE quality model is based on the principles defined in ISO/IEC 9126 standard. The following software characteristics are defined by the SilabMetrics software tool: Changeability, Efficiency, Maintainability, Portability, Reliability, Reusability, Security, and Testability. SilabMetrics supports software quality characteristics and software metrics previously defined in SQALE standard. In this way students can develop theoretical and practical knowledge related to industrial software quality standard and the best practice in the software development process.

These characteristics are applicable to every software system and thus ensure consistent terminology in defining the quality of a software product as well as the framework for the specification of the software quality requirements. SilabMetrics tool includes over 100 software metrics (e. g. Depth of Inheritance Tree, Coupling Between Classes, Cyclomatic Complexity of the Class/Method) that are used for measuring the quality of software system. These characteristics and software metrics can be

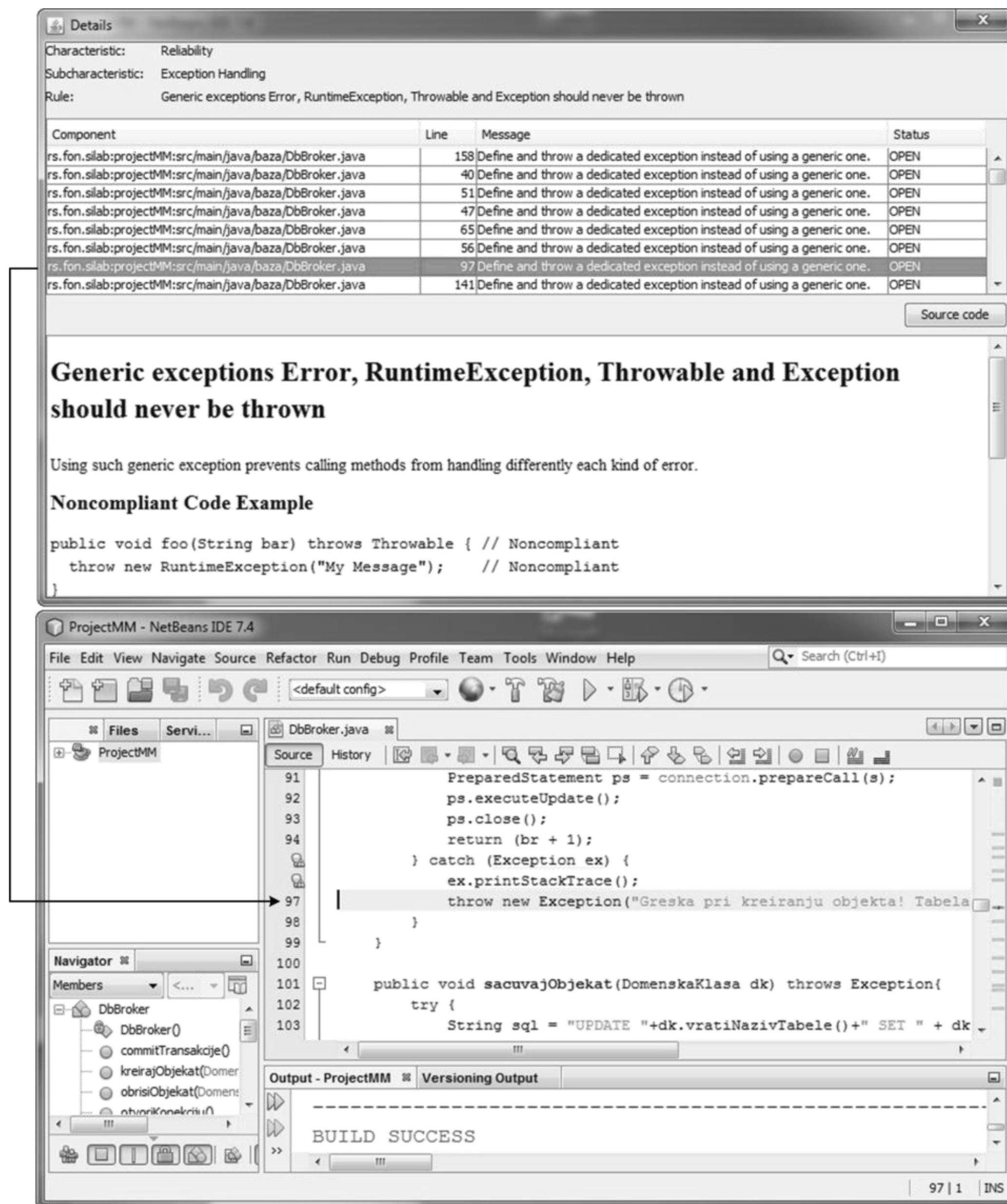

Fig. 5. SilabMetrics tool and NetBeans IDE integration—a double click on the violation in the SilabMetrics tool opens a specific line of code in NetBeans IDE.

configured by a teacher in order to improve the process of teaching and learning programming. This could be a time consuming activity; on the other hand, teachers can use predefined software quality model.

A double click on the violation in the SilabMetrics tool opens a specific line of code in NetBeans development environment, which enables detailed analysis of the violation and its correction, as shown in Fig. 5. Thus, we can reduce motion and discovery cost [38], i.e. the time it takes a student to find the appropriate program code that can be used as information or help in solving a given problem.

4. Evaluation

In order to verify the feasibility of the proposed approach and assess its benefits for students learning to program, we conducted an empirical study. The experiment was conducted with the students of the Department of Software Engineering of a large university. The experiment was part of a course on Software Design where students acquire theoretical and practical knowledge of the software development process. The experiment was driven by the following research questions:

- RQ1: Are software quality standards applicable

in the process of teaching and learning programming?
- RQ2: Is it possible to improve the process of learning programming through continuous inspection and continuous improvement of software quality?
- RQ3: Can a software quality tool help students develop programming skills?

4.1 Participants

There are many proposals related to empirical studies in software engineering [50–54]. Some of the questions that are in the focus of researchers are related to an experimental design, the number and kind of participants, data collection and analysis procedures [53, 54].

The experiment included a total of 30 undergraduate students who took our Software Design course in the 2013–2014 academic year. The students were randomly divided into experimental (15 students) and control (15 students) groups. The students had different GPA scores achieved in previously taken courses. In addition, there were no differences between experimental and control groups regarding GPA scores. The number of participants corresponds to a usual number of students in a computer classroom who attend classes in this and similar courses in the software engineering field. This ensures that all the participants in the study have similar experience [53].

Complex relationships between students and teachers (researcher) have raised several ethical dilemmas in empirical studies with students. Some countries have legal regulations that obligate a researcher to provide the Subject consent document for participants [51]. However, our students volunteered to participate, so this kind of consent was not used in this survey. Previously, all the students were informed about the research goals, and research policies, i.e. guaranteed anonymity, as well as the fact that their participation will not affect grades in any way. In this way, we have tried to contribute to the research ethics, although our laws do not provide any specific rules that pertain to this area.

4.2 Procedure

Both groups of students had the same traditional lectures (theoretical part and practical laboratory sessions). Laboratory sessions cover basic and advanced concepts of object-oriented programming in Java: classes and methods, inheritance, abstract classes, interfaces, strings, exception handling, graphical user interface, database programming, concurrent programming, and network programming. In addition, the experimental group received additional classes on software system quality. Students from the experimental group were also introduced to SilabMetrics tool for a static analysis of software quality. The goals of the course are completely harmonized with the research goals, which improves the pedagogical value [50] of the research.

There are two main weaknesses that are related to empirical studies with students [50]. The first relates to the fact that the results of surveys conducted on student population cannot be applied to experienced software engineers, and the second is that in this type of research results are often based on mini projects, but such results cannot be applied to real industry projects. It is important to point out that this research does not suffer from any of mentioned weaknesses. The results of research are related specifically to the students as well as student projects, not on experienced engineers or industrial projects. This makes the goal of the research in full compliance with the environment in which the research results shall be applied.

The experiment was focused on a programming task to be solved and implemented in the Java programming language. The students used the NetBeans development environment, and the teachers provided them with all the necessary instructions and explanations. Each student project was given a grade. We used Larman's software development method [55], which includes the following phases: gathering requirements, analysis, design, implementation, and testing. The first two phases, gathering requirements and analysis, were conducted by following the teachers' instructions. As a result of these phases, user requirements, the structure and the behavior of a software system were defined. The structure of the software system was defined with the domain model and the relational model, while the behavior of the software system was defined with the system operations. Based on this, students were working independently on design, implementation, and testing phases. Each student project includes 10 use-cases previously approved by the teacher. Architecture of a student project is shown in Fig. 6. We can observe Model-View-Controller architecture. In addition, software development process is guided by software quality standards.

The students' projects from the experimental group were reviewed from two perspectives: from the teacher's perspective and from the student's perspective. After the students implemented the first version of their software systems, the teacher performed a static analysis of the student's solutions using SilabMetrics tool. The teacher looked at the indicators of software metrics violations, and thus realized where students failed in the knowledge development process. Based on these insights, the teacher gave the students the necessary explanations and instructions how to improve the quality of their software systems. Next, students conducted the

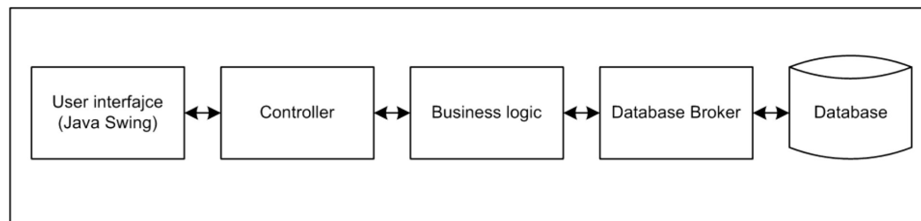

Fig. 6. Architecture of a student project.

analysis of their program code using SilabMetrics tool both before and after they improved their code. Since the tool was integrated with NetBeans development environment, students had the opportunity to directly analyze and eliminate software metrics violations. In addition, the experimental group was requested to fill in a post-course questionnaire.

4.3 Instruments

In the static analysis process, we used the SilabMetrics software quality tool. Also, we used a questionnaire-based survey. The questionnaire was based on a five-point Likert-like scale with values ranging from 1—strongly disagree to 5—strongly agree. The questionnaire contained eight closed questions related to the Lean approach and the usability of the SilabMetrics software quality tool. The question statements are given in Table 4.

4.4 Data analysis

We analyzed the violation of software metrics in the students' program code using descriptive statistics, namely a mean and standard deviation. Software metrics are objective, formalized measures used for software system analysis [3]. In this way, we can obtain feedback on a student's software systems quality. In addition, we calculated Cyclomatic complexity [56] per class and per function. Cyclomatic complexity is a software metric used to indicate the complexity of a software system. It is a quantitative measure related to different software quality attributes, e.g., software maintenance [57] and software testing [58]. Cyclomatic complexity measures control flow statements within a method and/or class (e.g., if-then, if-then-else, case, break, return, and continue). A method with high Cyclomatic complexity contains multiple control flow statements and vice versa; in this way, we can calculate student projects complexity.

We also calculated Technical Dept. It is a quantitative measure which aggregates all software metrics and software quality attributes into a single value. Technical Dept represents remediation cost of all violations in the code [49]. Hours were used as a measurement unit, but other units could be used as well [48]. Taking into account these characteristics, Technical Dept gives good insight into a student's project quality in terms of the remediation cost.

4.5 Results

According to the predefined model of software quality, 120 software metrics were used to measure the quality of a software system. By analyzing the final version of the students' program code, we observed violations of 47 software metrics. Different severity levels (Info, Minor, Major, Critical and Blocker) are defined for violation of software metrics. By analyzing the students' programming code, we made some interesting observations. Fig. 7 shows the distribution of a metrics violation according to the severity level for a control group. If we observe the control group, most violations are at the Major level, then at Minor and Critical level, and finally at Info and Blocker level. On the other hand, Fig. 8 shows the distribution of a metrics violation according to the severity level for experimental group. Most violations in the experimental group are at the Minor and Major level.

Table 2 shows the quality of students' software projects, separately for control and experimental groups. In particular, the table presents Cyclomatic complexity (per class and per function) and the number of violations in students' software systems. We can observe lower occurrence of metrics violation in the experimental group where the SilabMetrics tool was used for a static code analysis. Mean score of software metrics violations in the experimental group is 27.07, while the mean score of software metrics violations in the control group is 194.73. Also, we can observe that Cyclomatic complexity in the experimental group is somewhat lower than Cyclomatic complexity in the control group.

In addition, Table 3 shows Technical Dept of the students' software projects. This measure is calculated using SonarQube software quality tool. Technical Dept represents the time required to fix all software metrics violations in the code. This means that a student in the experimental group will need 5.90 hours to fix all issues and create error-prone code. On the other hand, a student in the control group will need 129.40 hours to remove all viola-

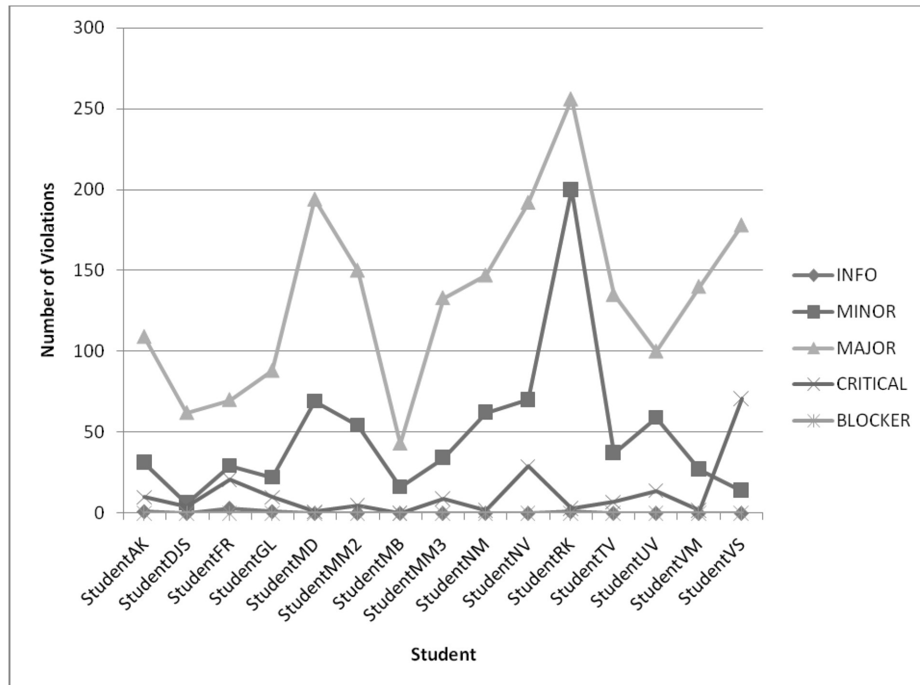

Fig. 7. The distribution of a software metrics violation according to the severity level—Control Group.

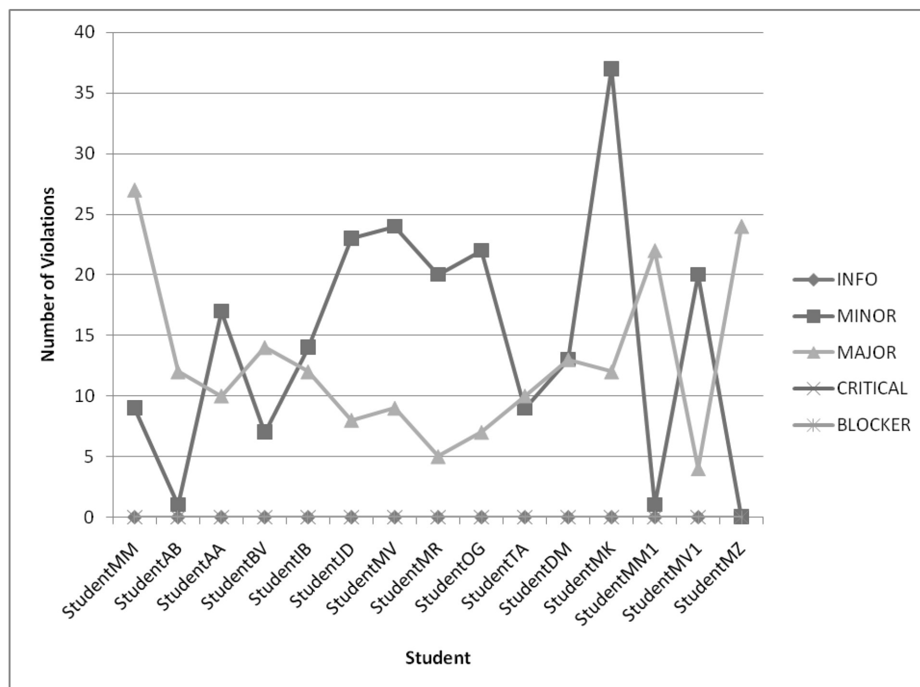

Fig. 8. The distribution of a software metrics violation according to the severity level—Experimental Group.

Table 2. The quality of students' software projects

Control group				**Experimental group**			
	Cyclomatic complexity Per Class	**Cyclomatic complexity Per Function**	**Violations**		**Cyclomatic complexity Per Class**	**Cyclomatic complexity Per Function**	**Violations**
Mean score	8.08	2.01	194.73	Mean score	7.18	1.88	27.07
Standard deviation	2.83	0.58	99.02	Standard deviation	1.22	0.15	8.27

Table 3. Technical Dept. of students' software projects

Control group		Experimental group	
	Technical Dept (in hours)		**Technical Dept (in hours)**
Mean score	129.40	Mean score	5.90
Standard deviation	70.53	Standard deviation	5.54

Table 4. Questionnaire results for the experimental group

Question	Mean score	Standard deviation
This approach contributes to a better understanding of the programming.	4.93	0.26
This approach contributes to a better understanding of the software quality.	4.80	0.56
This approach contributes to a better understanding of the course lessons.	4.93	0.26
Knowledge developed in this way can be applied in the development of other software systems.	4.87	0.52
Integration of the tool for static analysis of software quality with NetBeans development environment improves the process of learning programming.	4.87	0.35
Tool for static analysis of software quality is simple and easy to use.	4.40	1.12
Tool for static analysis of software quality makes finding bugs in code easier and faster.	4.93	0.26
This kind of tool would be useful and applicable to other phases of software development process (e.g., gathering requirements, software design).	4.87	0.52

tions. This is an expected result taking into account that students in the experimental group have to fix lower number of issues.

Finally, 15 students from the experimental group completed the post-course questionnaire. The results are presented in Table 4. Lower value for the question "Tool for static analysis of software quality is simple and easy to use" indicates that the usability of SilabMetrics tool should be improved. Overall, the results suggest that the use of the software quality standards and the tool for static analysis of software quality enable students to learn and develop programming skills easier.

5. Threats to validity

The small number of the participants in the experiment limits any strong conclusions. Control and experimental groups should contain more students and students' software projects should be more complex in order to retrieve more reliable results.

In the process of teaching and learning programming SilabMetrics tool has been used. This software quality tool is language dependent: it supports static code analysis in Java and object-oriented programming paradigm. It would be useful if SilabMetrics tool supported multiple programming paradigms and multiple programming languages. On the other hand, the proposed approach is independent of the programming language and programming paradigm: in another paradigm, different software quality indicators would be used.

In addition, students from the experimental group were requested to complete the post-course questionnaire in order to retrieve feedback about proposed process of teaching and learning programming. Although students were encouraged to express their opinions, the questionnaire results present subjective measure. Experimental group should contain more students and questionnaire statements should be improved in order to obtain more reliable feedback.

6. Discussion

The experiment results indicate that the Lean software development method and software quality standards can be incorporated in the process of teaching and learning programming. A fundamental principle applied in teaching and learning programming is the detection and elimination of waste (in terms of software quality violations, partially done work, motion, and defects) in the education process.

Gomes and Mendes identify a lack of problem solving abilities that many students show as one of the reasons that cause learning problems [9]. We believe that Lean approach, continuous inspection, and continuous improvement can help detecting and eliminating these problems. Students and teachers apply a proactive approach to the learning and teaching programming thus encouraging greater engagement in the education process itself [43]. This allows students to gain practical experience and gives the teachers an opportunity to instruct the course in order to achieve the goals of the course [42, 50].

The introduction of project based learning

enables students to apply theoretical knowledge in solving real-world problems [42, 44]. Inclusion of the semester project in the course had a positive impact on the students' knowledge in learning course concepts and promotes active learning [42, 43].

The experiment results show significantly lower occurrence of metrics violation if software quality tool is used. Consequently, students will need less time to fix all software metrics violations in the code. Instead of analyzing the quality of the final software product we promote Lean manufacturing principle of building-in quality into the product at the source [59]. In this way, it is possible to timely detect programming errors, make program improvements, and learn on the base of these activities [9, 60].

In addition, the questionnaire results indicate that software quality standards and software quality tools could be interesting and motivating to students in order to develop programming skills and improve critical thinking. This could be especially important for the initial programming courses [9, 13], as well as later courses taught to IT majors only [11]. The results also indicate that the SilabMetrics tool should be improved. We will improve the user interface with the visualization of software metrics violations and simplify installation and integration with NetBeans development environment.

We are in favor of incorporation of software metrics-driven practice in teaching and learning programming. It can complement test-driven learning approach [11, 61], which promotes test-first perspective [62, 63]. In this way, it is possible to perform static and dynamic analyses of software quality, indicating a potential for increasing the quality of student code [63, 64].

In addition, each software system is characterized by software syntax correctness, software semantic correctness, and software quality:

- Software syntax—provides a basis for the development of a software system. If we observe imperative, static-typed programming languages, software syntax correctness can be verified at the compile time using compiler and software system building tools. If the software system is syntactically incorrect, the compiler will terminate the software compiling process. In this case the compiler displays a description of the problem, as well as the instructions on how to resolve the problem. Consequently, syntax errors can be easily corrected.
- Software semantics—can be verified using a software debugger tool, software testing, code inspections, design walkthroughs, prototype simulations etc. In this way, we can verify that the software system is implemented in compliance with functional requirements.
- Software quality—can be defined as "an effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it" [3]. In this way, we can verify that the software system is implemented in compliance with non-functional requirements. Non-functional requirements are usually called quality attributes of the software system [65].

A precondition for software semantic analysis and software quality analysis is syntax correctness of the software system. Unlike syntax errors, semantic errors and software quality errors are much harder to notice: they cannot be noticed by compiling the software system. On the basis of software syntax, software semantic, and software quality we can observe four different situations, as shown in Fig. 9:

1. Software syntax correctness is verified; software semantic correctness and software quality are not verified: this means that *a software system contains software development gap*.
2. Software syntax correctness and software semantic correctness are verified; software quality is not verified: this means that *software system contains a software quality gap*.
3. Software syntax correctness and software quality are verified; software semantic correctness is not verified: this means that *software system contains a software semantic gap*.
4. Software syntax correctness, software semantic correctness, and software quality are verified: this means that the *software is in compliance with a software development process*.

On the other hand, Lahtinen, Ala-Mutka, and Järvinen have argued that finding bugs in programs is one of the most difficult issues in programming courses [16]. The SilabMetrics software quality tool allows students and teachers to focus on program code, software metrics, and software metrics violations. In this context, teaching and learning programming is based on software quality standards, which is supported by industrial software development tools. Taking into account that one of the major educational objectives in teaching programming is to teach students to develop good programs and not only to learn programming syntax [66], students can develop theoretical and practical knowledge related to best practices applied in the software development industry, which consequently leads to more "ready-to-perform" graduates [35]. Furthermore, students can easily adapt to

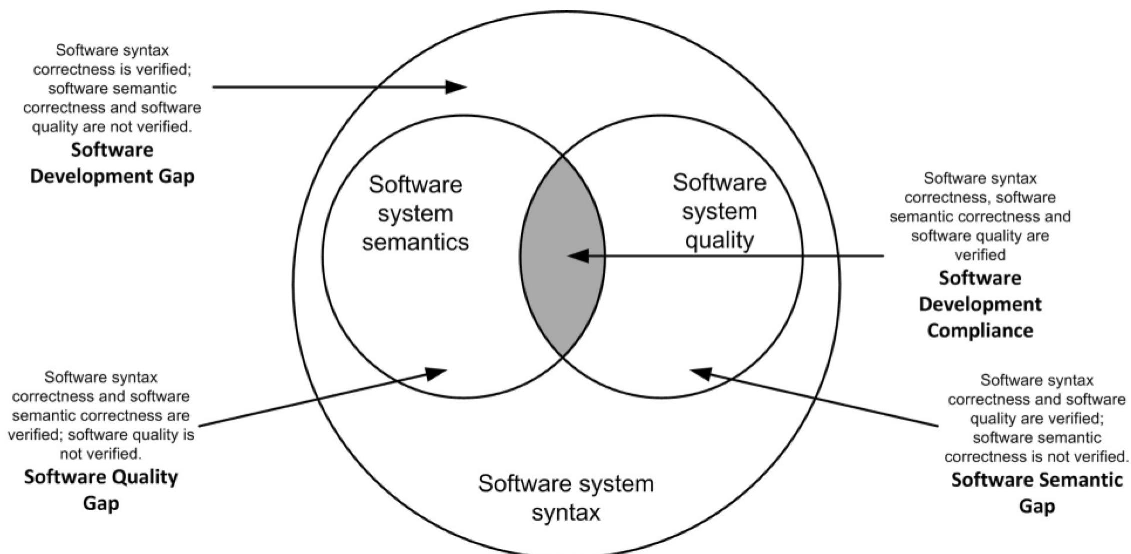

Fig. 9. Syntax correctness, semantic correctness, and software quality of the software system.

new emerging software engineering technologies [11].

In software engineering education process, we can observe multiple key stakeholders: students, teachers, researchers and industry [4, 67]. We strongly believe that the implementation of the Lean approach and software quality standards in software engineering education process can significantly contribute to all stakeholders.

We believe that software quality analysis if applied in the learning process may bring numerous benefits to students:

- Students can efficiently apply the acquired knowledge,
- Students are able to understand where they are wrong in the learning process,
- Through the feedback, students can improve their knowledge,
- Students become aware of the software quality importance and can be quickly incorporated into the software industry with respect to company standards and best practices related to the software development process.

We also believe that software quality analysis in the teaching process may bring numerous benefits to teachers:

- Teachers are able to understand where students make mistakes in the learning process,
- Teachers are able to understand where they are wrong in guiding the knowledge development,
- Through the feedback teachers can improve their knowledge,
- Through the feedback teachers can build students knowledge in a different way.

7. Conclusions

This paper presents software quality standards and the Lean software development method and their application in the process of teaching and learning programming. We have implemented the Lean approach with a focus on creating a value for students and teachers, which is achieved by continuous inspection and continuous improvement of software quality. In this way, it is possible to timely detect defects, make program improvements, and learn on the base of these activities. In addition, teachers could modify the course in order to guide knowledge development.

Although the experiment was limited in scope and number of participants, the results indicate that the software quality standards and the Lean approach can be successfully implemented in the process of teaching and learning programming. The introduction of project based learning enables students to apply theoretical and practical knowledge to solving engineering problems. Students are guided to write program codes that are in compliance with a defined software quality model, which increases the level of software quality and indicates the improvement of the process of teaching and learning. On the other hand, programming is part of software construction which is just one of the knowledge areas defined in the Software Engineering Body of Knowledge. Therefore, in further research we will investigate how this approach can be applied within other knowledge areas in software engineering education process (e.g., Lean approach in software requirements gathering, Lean approach in software testing, Lean approach in software maintenance). We will continue to explore the importance of soft-

ware quality standards for the learning process and design tools to support this process.

References

1. I. Sommerville, Software Process Models, in A. B. Tucker (Ed.), *Computer science handbook*, CRC press, 2004.
2. P. Kruchten, The rational unified process: an introduction, *Addison-Wesley Professional*, 2004.
3. R. S. Pressman, Software engineering: a practitioner's approach, *Palgrave Macmillan*, 2005.
4. J. Carver, L. Jaccheri, S. Morasca and F. Shull, Issues in using students in empirical studies in software engineering education, *Proceedings of the Ninth IEEE International Software Metrics Symposium (METRICS'03)*, 2003, pp. 239–249.
5. B. Kitchenham, D. Budgen, P. Brereton and P. Woodall, An investigation of software engineering curricula, *Journal of Systems and Software*, **74**(3), 2005, pp. 325–335.
6. T. C. Lethbridge, What knowledge is important to a software professional?, *Computer*, **5**, 2000, pp. 44–50.
7. T. C. Lethbridge, R. J. LeBlanc, A. E. K. Sobel, T. B. Hilburn and J. L. Diaz-Herrera, SE2004: Recommendations for undergraduate software engineering curricula, *IEEE Software*, **23**(6), 2006, pp. 19–25.
8. P. Bourque and R. E. Fairley (Eds.), Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0, *IEEE Computer Society Press*, 2014.
9. A. Gomes and A. J. Mendes, An environment to improve programming education, *Proceedings of the ACM international conference on Computer systems and technologies 2007*, 2007, pp. 88–95.
10. A. G. Gein, Informatics in schools: Problems of content, *Programming and Computer Software*, **37**(6), 2001, pp. 284–287.
11. N. R. Boyer, S. Langevin and A. Gaspar, Self direction & constructivism in programming education, *Proceedings of the 9th ACM SIGITE conference on Information technology education*, 2008, pp. 89–94.
12. J. Long, Just For Fun: Using Programming Games in Software Programming Training and Education, *Journal of Information Technology Education*, **6**(1), 2007, pp. 279–290.
13. A. Robins, J. Rountree and N. Rountree, Learning and teaching programming: A review and discussion, *Computer Science Education*, **13**(2), 2003, pp. 137–172.
14. J. Sorva, V. Karavirta and L. Malmi, A review of generic program visualization systems for introductory programming education, *ACM Transactions on Computing Education (TOCE)*, **13**(4), 2013, p. 15.
15. C. Bravo, M. J. Marcelino, A. J. Gomes, M. Esteves and A. J. Mendes, Integrating Educational Tools for Collaborative Computer Programming Learning, *Journal of Universal Computer Science*, **11**(9), 2005, pp. 1505–1517.
16. E. Lahtinen, K. Ala-Mutka and H. M. Järvinen, A study of the difficulties of novice programmers, *ACM SIGCSE Bulletin*, **37**(3), 2005, pp. 14–18.
17. H. F. Guo, A semantic approach for automated test oracle generation, *Computer Languages, Systems & Structures*, **45**, 2016, pp. 204–219.
18. Y. A. Alsultanny and A. M. Wohaishi, Requirements of Software Quality Assurance Model, *Second IEEE International Conference on Environmental and Computer Science (ICECS'09)*, 2009, pp. 19–23.
19. N. F. Schneidewind, Body of knowledge for software quality measurement, *IEEE Computer*, **35**(2), 2002, pp. 77–83.
20. Y. Liu, T. M. Khoshgoftaar and N. Seliya, Evolutionary Optimization of Software Quality Modeling with Multiple Repositories, *IEEE transactions on software engineering*, **36**(6), 2010, pp. 852–864.
21. M. Jørgensen, Software quality measurement, *Advances in engineering software*, **30**(12), 1999, pp. 907–912.
22. T. Ball, The concept of dynamic analysis, *7th European Software Engineering Conference ESEC/FSE'99*, France, Springer Berlin Heidelberg, 1999, pp. 216–234.
23. G. Della Penna, A type system for static and dynamic checking of C++ pointers, *Computer Languages, Systems & Structures*, **31**(2), 2005, pp. 71–101.
24. B. Johnson, Y. Song, E. Murphy-Hill and R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, *35th IEEE International Conference on Software Engineering (ICSE)*, 2013, pp. 672–681.
25. ISO/IEC 25000, Software Engineering—Software product Quality Requirements and Evaluation (SQuaRE), http://www.iso.org, Accessed 15 January 2016.
26. ISO/IEC 14598: Information Technology—Evaluation of Software Products Standard, http://www.iso.org, Accessed 15 January 2016.
27. ISO/IEC 9126, Software Engineering Standard, http://www.iso.org, Accessed 15 January 2016.
28. N. Bevan, International Standards for HCI, In Claude Ghaoui (Ed.), *Encyclopedia of Human Computer Interaction*, Idea Group Publishing, 2005.
29. H. Schackmann, M. Jansen and H. Lichter, Tool Support for User-Defined Quality Assessment Models, *Proceedings of the Software Metrik Kongress (MetriKon2009)*, 2009.
30. Y. Kanellopoulos, P. Antonellis, D. Antoniou, C. Makris, E. Theodoridis, C. Tjortjis and N. Tsirakis, Code Quality Evaluation Methodology Using The ISO/IEC 9126 Standard, *International Journal of Software Engineering and Applications*, **1**(3), 2010, pp. 17–36.
31. T. Galli, F. Chiclana, J. Carter and H. Janicke, Towards introducing execution tracing to software product quality frameworks, *Acta Polytechnica Hungarica*, **11**(3), 2014, pp. 5–24.
32. I. Antović, S. Vlajić, M. Milić, D. Savić and V. Stanojević, Model and software tool for automatic generation of user interface based on use case and data model, *IET Software*, **6**(6), 2012, pp. 559–573.
33. P. Middleton, Lean software development: two case studies, *Software Quality Journal*, **9**(4), 2001, pp. 241–252.
34. J. P. Womack and D. T. Jones, Lean thinking: banish waste and create wealth in your corporation, *Simon and Schuster*, 2010.
35. K. A. Cary, The software enterprise: Practicing best practices in software engineering education, *International Journal of Engineering Education*, **24**(4), 2008, pp. 705–716.
36. M. Poppendieck and T. Poppendieck, Lean software development: an agile toolkit, *Addison-Wesley Professional*, 2003.
37. J. Shore and S. Warden, The art of agile development, *O'Reilly Media, Inc.*, 2007.
38. J. O. Coplien and G.Bjørnvig, Lean architecture: for agile software development, *John Wiley & Sons*, 2011.
39. A. Rugarcia, R. M. Felder, D. R. Woods and J. E. Stice, The future of engineering education I. A vision for a new century, *Chemical Engineering Education*, 2000, **34**(1), pp. 16–25.
40. A. C. Alves, F. J. Kahlen, S. Flumerfelt and A. B. S. Manalang, Lean engineering education: Bridging-the-gap between academy and industry, *International Conference of the Portuguese Society for Engineering Education (CISPEE)*, 2013.
41. S. Flumerfelt, F. J. Kahlen, A. C. Alves and A. Siriban-Manalang, The future of lean engineering education: Sustainability, systems and ethics competency, *ASME Press*, 2013.
42. D. Kanigolla, E. A. Cudney, S. M. Corns and V. A. Samaranayake, Enhancing engineering education using project-based learning for Lean and Six Sigma, *International Journal of Lean Six Sigma*, **5**(1), 2014, pp. 45–61.
43. J. E. Froyd, P. C. Wankat and K. A. Smith, Five major shifts in 100 years of engineering education, *Proceedings of the IEEE*, 2012, 100 (Special Centennial Issue), pp. 1344–1360.
44. J. E. Mills and D. F. Treagust, Engineering education—Is problem-based or project-based learning the answer?, *Australasian journal of engineering education*, **3**(2), 2003, pp. 2–16.
45. J. Tierney, Eradicating mistakes from your software process through Poka Yoke, *6th International Software Quality Week, Software Research Institute*, 1993, pp. 300–307.
46. M. Poppendieck, Lean software development, *In Companion to the proceedings of the 29th International Conference on*

Software Engineering, *IEEE Computer Society*, 2007, pp. 165–166.
47. A. R. da Silva, J. Saraiva, D. Ferreira, R. Silva and C. Videira, Integration of RE and MDE paradigms: the ProjectIT approach and tools, *IET software*, **1**(6), 2007, pp. 294–314.
48. J. L. Letouzey, The SQALE Method for Evaluating Technical Debt, *Proceedings of the Third International Workshop on Managing Technical Debt ICSE 2012*, *IEEE Press*, 2012, pp. 31–36.
49. J. L. Letouzey and M. Ilkiewicz, Managing technical debt with the sqale method, *IEEE software*, 2012, (6), pp. 44–51.
50. [50] J. C. Carver, L. Jaccheri, S. Morasca and F. Shull, A checklist for integrating student empirical studies with research and teaching goals, *Empirical Software Engineering*, **15**(1), 2010, pp. 35–59.
51. J. Singer and N. G. Vinson, Ethical issues in empirical studies of software engineering, *IEEE Transactions on Software Engineering*, **28**(12), 2002, pp. 1171–1180.
52. C. B. Seaman, Qualitative methods in empirical studies of software engineering, *IEEE Transactions on software engineering*, **25**(4), 1999, pp. 557–572.
53. B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam and J. Rosenberg, Preliminary guidelines for empirical research in software engineering, *IEEE Transactions on software engineering*, **28**(8), 2002, pp. 721–734.
54. A. Jedlitschka and D. Pfahl, Reporting guidelines for controlled experiments in software engineering, *IEEE International Symposium on Empirical Software Engineering*, 2005.
55. C. Larman: Applying UML and Patterns: an introduction to object-oriented analysis and design and iterative development, *Pearson Education*, 2004.
56. T. J. McCabe, A complexity measure, *IEEE Transactions on Software Engineering*, **2**(4), 1976, pp. 308–320.
57. G. K. Gill and C. F. Kemerer, Cyclomatic complexity density and software maintenance productivity, *IEEE Transactions on Software Engineering*, **17**(12), 1991, pp. 1284–1288.
58. A. H. Watson, T. J. McCabe and D. R. Wallace, Structured testing: A testing methodology using the cyclomatic complexity metric, *NIST special Publication*, **500**(235), 1996, pp. 1–114.
59. R. B. Detty and J. C. Yingling, Quantifying benefits of conversion to lean manufacturing with discrete event simulation: a case study, *International Journal of Production Research*, **38**(2), 2000, pp. 429–445.
60. R. Duque, L. Bollen, A. Anjewierden and C. Bravo, Automating the Analysis of Problem-solving Activities in Learning Environments: the Co-Lab Case Study, *Journal of Universal Computer Science*, **18**(10), 2012, pp. 1279–1307.
61. D. Janzen and H. Saiedian, Test-driven learning in early programming courses, *ACM SIGCSE Bulletin*, **40**(1), 2008, pp. 532–536.
62. S. H. Edwards, Rethinking computer science education from a test-first perspective, *18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2003, pp. 148–155.
63. H. Erdogmus, M. Morisio and M. Torchiano, On the effectiveness of the test-first approach to programming, *IEEE Transactions on Software Engineering*, **31**(3), 2005, pp. 226–237.
64. S. H. Edwards, Using software testing to move students from trial-and-error to reflection-in-action, *ACM SIGCSE Bulletin*, **36**(1), 2004, pp. 26–30.
65. L. Chung and J. C. S. do Prado Leite, On non-functional requirements in software engineering, *In Conceptual modeling: Foundations and applications*, Springer Berlin Heidelberg, 2009, pp. 363–379.
66. G. Fischer and J. W. von Gudenberg, Improving the quality of programming education by online assessment, *ACM 4th international symposium on Principles and practice of programming in Java*, 2006, pp. 208–211.
67. B. Boehm, A. Egyed, D. Port, A. Shah, J. Kwan and R. Madachy, A stakeholder win-win approach to software engineering education, *Annals of Software Engineering*, **6**(1–4), 1998, pp. 295–321.

Miloš Milić is a teaching assistant at University of Belgrade, Faculty of Organizational Sciences, Department of Software Engineering. He has taught undergraduate and graduate level courses: Introduction to Programming, Software Design, Design Patterns, and Java Programming Language. His research interests include software quality, software design, and software testing. He is the author or co-author of several publications on national and international conferences and journal papers. He is a PhD student at the University of Belgrade.

Siniša Vlajić is an associate professor of software engineering at University of Belgrade, Faculty of Organizational Sciences, Department of Information Systems. He has taught undergraduate and graduate level courses: Introduction to Programming, Introduction to Information System, Software Design, Design Patterns, Programming Methodology, and Java Programming Language. He wrote many books and publications about C++, Java, software design, software patterns, databases and information systems. His main research interests include: software process, software design, software maintenance, software pattern formalization, and programming methodology. He is one of the founders of the Laboratory and Department of the Software Engineering at Faculty of Organizational Sciences.

Ilija Antović is an assistant professor at Software Engineering Department—Faculty of Organizational Sciences, University of Belgrade. His research interests are: Automation of User Interface Development, Modeling and Metamodeling, Model Driven Engineering, Requirement Engineering, Software Patterns, and Code Generation. He lectures at undergraduate and graduate level courses in his area. He is the author or co-author of several publications on national and international conferences and journal papers.

Dušan Savić is an assistant professor on Faculty of Organizational Sciences at the Software Engineering Department. He has interests in the following areas: modeling and meta-modeling, model driven engineering, requirement engineering, software development, software design, domain specific languages, and automation of user interface development. He has taught undergraduate and graduate level courses in his area. He is the author or co-author of several publications on national and international conferences and journal papers.

Vojislav Stanojević is a teaching assistant of software engineering at University of Belgrade, Faculty of Organizational Sciences, Department of Information Systems. He has taught undergraduate and graduate level courses: Introduction to Programming, Introduction to Information System, Software Design, Design Patterns, Programming Methodology, and

Java Programming Language. He wrote publications about Java, software design, software patterns, application frameworks and domain specific languages. His main research interests include: software design, application frameworks, business rules, and domain specific languages.

Saša Lazarević is an associate professor of software engineering at University of Belgrade, Faculty of Organizational Sciences, Department of Software Engineering. Undergraduate and graduate level courses: Introduction to Programming, Programming Principles, Software Design, Software Construction, Software Testing, and Software Quality; Databases, Information Systems Design. His main research interests include: software process, software design, software testing, software quality, databases and programming on .NET platform. As an academician and educator, he authored or co-authored over 50 scholarly articles and published three books, while mentoring a number of students and supervising a few dissertations. He is one of the founders of the Laboratory and Department of the Software Engineering at Faculty of Organizational Sciences.