Introduction to Software Engineering – Challenges and Responses in Course Composition*

Petar Rajković[1,*,†], Anđelija Đorđević[1,†] and Dragan Janković[1,†]

[1] *University of Niš, Faculty of Electronic Engineering, Aleksandra Medvedeva 4, 18104 Niš, Serbia*

Abstract

Introduction to Software Engineering courses serve as the foundation for preparing future professionals in a key aspect of the software industry. This study examines the challenges inherent in course composition and highlights pedagogical responses designed to overcome these obstacles. The course was established in the Faculty of Electronic Engineering in Niš in 1995 and, due to various influences, underwent numerous changes. Each influence, ranging from constant updates in technology and theory to reforms in Serbian higher education, variations in related courses, and adjustments to the course's position within the curriculum, requires a timely response that helps students keep pace and creates a solid foundation for future roles. In our evaluation, the evolution of the course will be presented alongside its current state, followed by an analysis of students' results, acceptance rates, and a discussion of the feedback they provided. During this period, the primary focus was on developing pedagogical models that engaged students and bridged the knowledge gap between fundamental programming courses and advanced topics in Information Systems and Software Engineering. Project-based learning initiatives involve students in realistic software development cycles, fostering skills through iterative design, coding, and testing. The case of the course being moved in the earlier 2021 semester was specifically evaluated. Initial responses were lower, but through the engagement of the teaching staff and adjustments to the course, the acceptance numbers and average grades returned to normal within two years. One of the main approaches was the flipped classroom model, which uses asynchronous online content and interactive in-class sessions to deepen understanding and facilitate problem-solving. Given the course's position in the curriculum, the approach is more individualized, aiming to help students enhance their knowledge by providing a personalized educational experience that supports diverse learning needs and promotes continuous improvement. By synthesizing all mentioned aspects, this work outlines a comprehensive experience-based approach for addressing the multifaceted challenges in introductory software engineering courses.

Keywords

Software Engineering, Education, Course Design, Course Evaluation

1. Introduction

Software engineering [1] is a dynamic discipline characterized by rapid technological change and evolving industry practices [2]. Consequently, introductory courses in software engineering face the dual challenge of delivering robust theoretical foundations while also emphasizing practical, industry-relevant skills [3]. This work examines key challenges in course design, including curriculum alignment with industry trends, the course's position within the curriculum, and student diversity in experience, as well as the balance between theory and practice [4][5][6]. It explores innovative pedagogical responses, ranging from project-based learning [7] to agile curriculum updates [8].

The additional challenge is the course's position within the curriculum [9], particularly for students in the Electrical Engineering and Computer Science program, and the preceding set of courses is not fully software-oriented [10]. Next, the historical context of the course, dating back to

* *NWISEd 2025: 3rd Workshop on Co-Creating New Ways of Information Systems, September 10–11, 2025, Maribor, Slovenia*
[*] Corresponding author.
[†] These authors contributed equally.

petar.rajkovic@elfak.ni.ac.rs (P. Rajković); andjelija.djordjevic@elfak.ni.ac.rs (A. Đorđević); dragan.janković@elfak.ni.ac.rs (D. Janković)
0000-0003-4998-2036 (P. Rajković); 0000-0002-4760-8641(A. Đorđević); 0000-0003-1198-0174 (D. Janković)

previous educational programs, also plays a role to some extent – the change in position of such a course to earlier semesters could also be an administrative struggle, followed by endless discussions.

However, the purpose has always been and will continue to be to provide a solid foundation in the principles and practices of software engineering [11]. The goal is to equip students with knowledge of the software development life cycle (SDLC) [12][13] best practices in design and coding, and collaborative skills essential in modern engineering environments [14]. By the end of the course, students should be able to articulate fundamental concepts, apply methodologies, and work effectively in team-based project settings.

The main aim of this paper is to present the experience gained through thirty years of teaching a course that should act as an introduction to software engineering and to discuss the changes in the content and overall software development scene when the position within the curriculum and correlation with prerequisite subjects changed.

2. Course Evolution

The introduction of this course in the Faculty of Electronic Engineering in Niš dates to the mid-1990s. In the initial version, from 1995, the course was titled "Software Development Techniques and Methods" and was placed in the eighth semester of a 5-year electronic engineering curriculum (**Figure 1**). The focus was to introduce students to software development models (like Incremental, Spiral, Iterative, and Waterfall [15]), and then, through a specific practical lesson, teach them some technology that they could use to develop their projects. The focus of the projects was to create a Windows Forms or web-based application by following the process defined by the offered methodologies and creating proper documentation. All these facts must be considered with the understanding that thirty years ago, at our university, Software Engineering was a two-semester course in postgraduate studies, and there was no perception that the topic was suitable for graduate studies.

All the mentioned problems led to the first course redesign in 1999, when the course was moved one semester earlier and the name was slightly changed to “Software Development and Design Methods.” This opens the door to introducing new concepts. Unfortunately, the course remains relatively late in the curriculum (in the fourth year), with a focus still on electronic engineering. The first course redesign introduced UML diagrams [17] as the design tool, along with software metrics such as cyclomatic complexity, lines of code per class, and maintainability index. Application design patterns such as Windows' two-class and three-class were introduced, marking a slight shift towards software engineering.

The biggest drawback in the first two versions of the course was that students aimed to complete the project as quickly as possible and focused on documentation. There was an immense amount of copied code, and the code quality was very questionable [16]. Notably, there was a tendency among the students to put everything in one class. Also, the often-repeated remark was, “In PHP, there are no classes, so do we need them at all?” For reference, PHP was one of the most widely used technologies among Serbian software companies in the late 1990s and early 2000s. PHP introduced full object-oriented support starting with version 5, released in 2004.

The next course redesign occurred in 2004, when the Bologna process [18] was adopted in Serbian universities, and the study was reorganized so that all courses became one-semester courses. The curriculum underwent a significant shift to a 3-year program with a focus on computer science. Additionally, software engineering was moved to the sixth semester, creating an ideal opportunity for us to follow our course. The course is renamed to “Object-Oriented Design”, and the selection of design patterns is introduced, accounting for one-third of the course topics. Additionally, the practical part of the exam was revised from a single project to five laboratory exercises. The tasks within the exercises were to design applications using design templates (such as Model-View-Controller, or MVC [19]) and to apply various software development patterns (such as the GoF patterns [20][21]). The course was well-positioned and provided students with sufficient knowledge for later project-based courses, such as “Information Systems”, “Distributed Systems”, or “Software

Engineering". Additionally, the course followed courses such as Programming Languages, Data Structures, and Databases, equipping the students well for the tasks to come.

The following curriculum updates (2007 and 2013) extended bachelor's studies to a 4-year model, keeping our course in the same place. There was a room to consolidate teaching materials and update the development environment to newer versions of Qt and Visual Studio, allowing students to create more visually compelling projects. The number of projects was increased to six, covering a complete semester with tasks.

The latest curriculum updates (from 2019, as well as the one which is currently "in design", aimed to come in 2027) moved "Object Oriented Design" to the fourth semester since it was required to bring introduction knowledge from software engineering earlier. It was envisioned to be beneficial for several later project-based courses, especially those in the fifth semester, which require knowledge from our course (such as Information Systems and Web Development). The major problem we faced with this change was that students had only completed Object-Oriented Programming before taking this course, while they were also attending Data Structures and Programming Languages in parallel. This reduced, to some extent, our ability to choose and suggest technology, and shifted the focus slightly away from the application design.

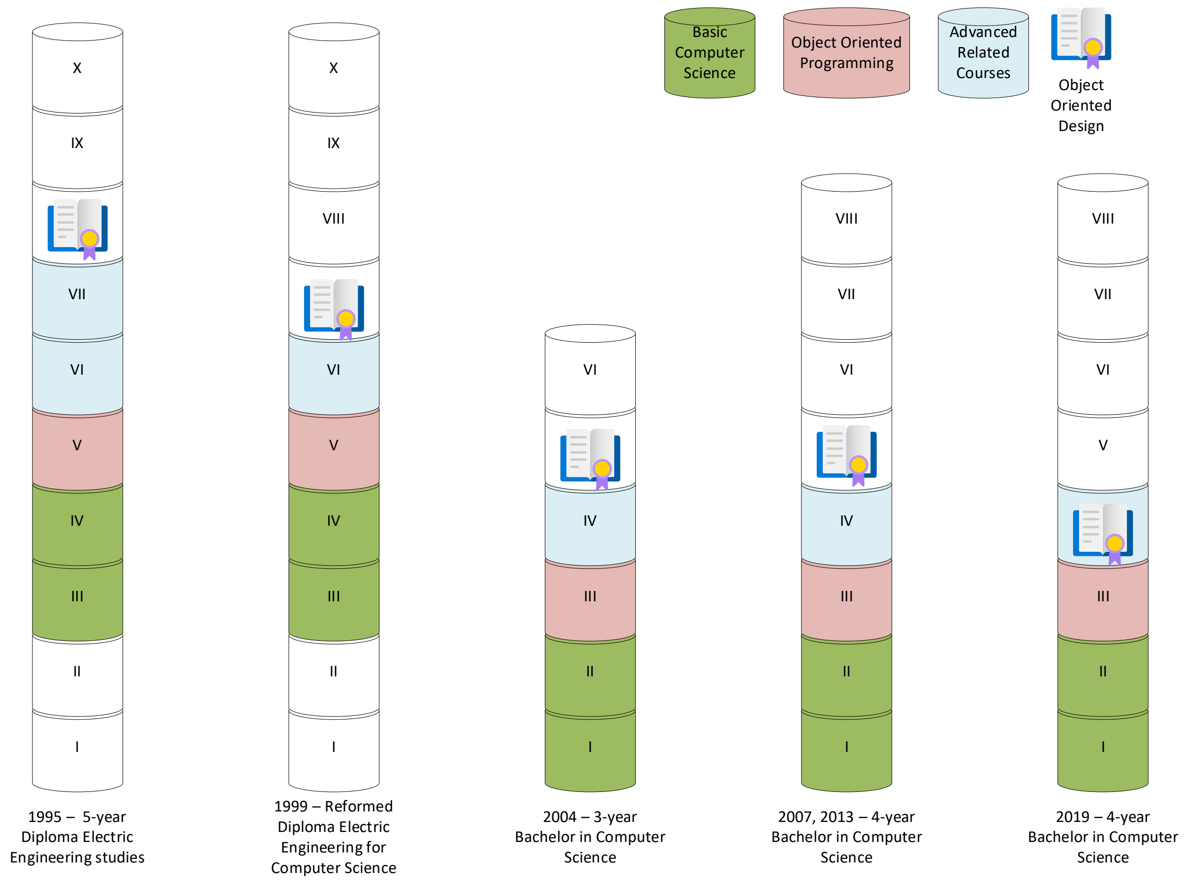
Basic Computer Science
Object Oriented Programming
Advanced Related Courses
Object Oriented Design
X
IX
VIII
VII
VI
V
IV
III
II
I
1995 – 5-year Diploma Electric Engineering studies
1999 – Reformed Diploma Electric Engineering for Computer Science
2004 – 3-year Bachelor in Computer Science
2007, 2013 – 4-year Bachelor in Computer Science
2019 – 4-year Bachelor in Computer Science

Figure 1: Change in course position over the years

3. Current Course Composition

The current course composition has been revised and updated to serve as a bridge to Object-Oriented Programming and Software Engineering. As mentioned, the course is now in the fourth semester and runs in parallel with Programming Languages and Data Structures. For this reason, we cannot rely any longer on the programming experience our students had before, but need to adapt to help them even with the two mentioned courses. Due to this request, the course is organized around lectures, practical and laboratory exercises, evaluating the necessary student's effort with 6 ECTS

(European Credit Transfer and Accumulation System) credits. This value has not changed since the course's foundation.

The course is running in the fourth semester alongside Data Structures, Programming Language Foundations, Computer architectures, as well as chosen topics from supporting areas of Mathematics. It comes after courses (third semester) covering databases and object-oriented programming, and before Information Systems (fifth semester) and Software Engineering (sixth semester).

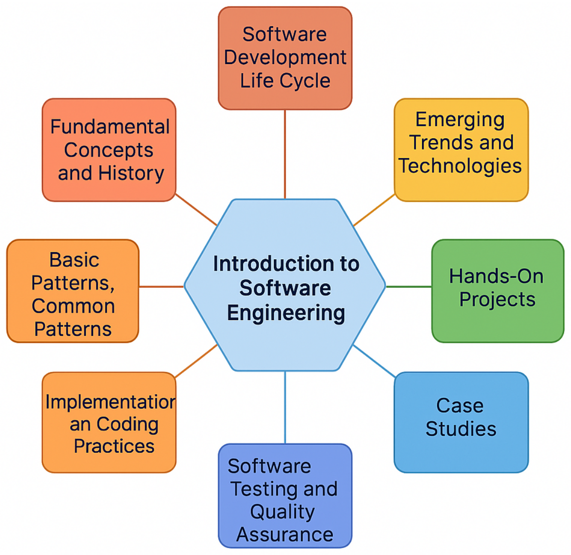
Software Development Life Cycle
Fundamental Concepts and History
Emerging Trends and Technologies
Basic Patterns, Common Patterns
Introduction to Software Engineering
Hands-On Projects
Implementation an Coding Practices
Case Studies
Software Testing and Quality Assurance

Figure 2: Main elements of the current course composition

The design of our course follows the general recommendations for introductory software engineering courses (such as [23]), but it is also based on previous experience, requests, and insights that come from the industry. Compared to the current recommendations for computer science curricula [23], this course predominantly covers topics from the software engineering and design (chapter: SE-Design: Software Design), enriched with coding styles, practical usage of advanced debuggers, and best practices from SE-Construction: Software Construction, and SDF-Practices: Software Development Practices. Overall, within the course, the following content areas are covered (**Figure 2**):

A. Fundamental Concepts and History. Definition, scope, and importance in today's technology landscape. Historical evolution and emerging roles within the field.
B. Software Development Life Cycle (SDLC) and major models – Waterfall, iterative, agile (Scrum, Kanban), and spiral models. Phases: requirement analysis, design, implementation (coding), testing, deployment, and maintenance. Comparative Analysis: situations for optimal use of each model, with examples from industry case studies.
C. Software Design and Architecture. Design Principles: Modularity, encapsulation, abstraction, reusability, and scalability. SOLID principles. Use of UML diagrams for representing system components and interactions.
D. Design Patterns: Basic patterns, Common patterns (e.g., Singleton, Observer, MVC), and their applicability.
E. Implementation and Coding Practices. Coding Standards: Best practices for readable, maintainable, and efficient code. Use of IDEs and Collaboration Tools (such as QT and Visual Studio). Practical labs on coding, code reviews, and debugging techniques.
F. Software Testing and Quality Assurance. Introduction to testing frameworks used within IDEs. Quality Metrics: Techniques to measure and ensure software quality.

G. Emerging Trends and Technologies: Artificial Intelligence and Software Engineering: The impact of AI-driven tools like code assistants, automated testing solutions, and intelligent debugging.
H. Practical, Hands-On Projects and Case Studies. Real-World Scenarios: Projects that simulate complete software development cycles.
I. Case Studies: Analysis of successful and failed projects to extract lessons on best practices and common pitfalls.

The initial lectures cover the basics of UML. The start is with a class, sequence, and state diagram. The intention is that through drawing basic integrations between objects, static class structure, and defining states of the object, our student improve their understanding of classes and relations between them. Each lecture is followed by homework, which enables students to assess their knowledge immediately. Homework acts as a preparation for the lab exercise.

The following section introduces the SOLID principles and basic design patterns [22]. SOLID principles are intended to help students gain a better understanding of essential UML diagrams they have already learned, and to improve their knowledge of relations between interfaces, classes, and objects, method calls, and how to define a combination of property values that would act as the state for the object. Basic design patterns, such as the factory method, null object, and composite method, are covered in the first part of the course. The goal of this part is to develop a simple board game, such as Minesweeper, incorporating the elements learned so far.

The next part of the course focuses on further developing the students' ability to analyze problems and provide structured solutions. It consists of UML-related lectures and demonstrations of IDE capabilities that students can use to execute the following lab exercises. The students will be introduced to additional UML diagrams (excluding the initial three types) and learn how to use QT or Visual Studio as the appropriate environments for C++. At this stage of the course, students will continue to use C++ as the coding language, as previously introduced in classes such as Object-Oriented Programming. It must be noted that there is considerable room for discussion regarding whether C++ is the best choice at this stage of programming. However, since it was used in a previous course and also demonstrated in Data Structures, the course that runs in parallel is an obvious choice. It is also worth mentioning that in the practical part of the Programming Languages course, students use Java or C# as examples of languages that run on the software virtual machine. As a result of the second part of the course, the students will have two programming projects to complete. The projects are based on understanding well-known processes (such as libraries and driver's licenses) and implementing a simple software solution that yields results in the form of a Windows Forms application.

In the final part, students are taught various design patterns, starting with organizational patterns such as MVC, and progressing to Abstract Factory, Bridge, Command, Observer, Decorator, and others. Up to this point, we have utilized their progress in Java and C# to assign tasks that will enable them to develop the application with a more vivid graphical interface, which will have multiple layers, making the implementation challenges more interesting.

4. Pedagogical Approaches and Assessment Strategies

Along with standard lectures covering theoretical aspects, the primary approach employed in the course is Project-Based Learning (PBL). It is essentially applied through five to six projects that should be completed throughout the semester. Mentioned projects are defined to mimic the almost complete software development life cycle (excluding maintenance). Students begin with a basic set of requirements and must complete them through a series of requirements gathering sessions. After that, they move through design, implementation, testing, and creation of basic documentation. Documentation sets are not equal for each project, but they slightly change throughout the semester. The constant focus is on modeling, but in some projects, the emphasis is more on the end user, while in others, it is on technical implementation documents.

Since this course serves as a bridge from object-oriented programming to later engineering-intensive courses, a few key points, such as iterative development and interdisciplinary integration, were not in focus. Besides these two points, which are essential to software engineering, they have been left to Information Systems and Software Engineering. The main goal of our course is to move students' mindset from "I'm typing a code and creating a program" to "I'm designing a software project which will be implemented and documented with proper industry standards".

The general approach for the course is based on "Flipped and Blended Classroom" models. They include pre-class engagement, active in-class sessions, and post-class assignments (such as projects). Unlike the preceding courses, which were usually designed more strictly and classically, the introduction to software engineering requires a higher student involvement. In a flipped classroom approach, students review lecture material, tutorials, or short videos before coming to class, freeing in-class time for discussions, coding challenges, and collaborative problem-solving. Our teaching is based on the Microsoft Teams platform, which integrates all necessary elements and provides students with good collaborative tools.

Class preparation helps better organize active in-class sessions, which utilize class time for hands-on activities—such as coding exercises and debugging sessions—and reinforce theoretical concepts while catering to diverse learning speeds. Luckily, software engineering itself is primarily practical, which drives the course definition.

Since we are focused on improving students' programming and design skills, all projects are based on a one-man-only setup. In this sense, we support digital integration by facilitating face-to-face interactions, which can be further extended through online forums and discussion groups, thereby enhancing learning beyond the classroom. Collaborative and Team-Based Learning is reduced to peer discussions and code reviews, which we see as the weakest point in the course. Eventually, the focus on this topic will be addressed in later courses, which should further enhance students' team-based skills.

The next point that we plan to integrate into the course is Technology-Enhanced Learning (TEL), which means that this should include adaptive learning platforms, virtual labs, and simulations, and possibly start using more options of the already used interactive development tools (integration with Visual Studio and Git, i.e.). Leveraging AI-driven learning platforms can offer personalized content delivery and real-time feedback, ensuring that all students—from beginners to those with advanced backgrounds—remain engaged and challenged. Utilizing virtual lab and simulation environments, container orchestration simulations, or cloud-based labs enables students to gain practical skills in a controlled, reproducible setting. Tools such as collaborative coding environments, real-time debugging sessions, and version control simulations (e.g., using Git in a classroom setting) enhance experiential learning.

Assessment strategies are based on formative and summative assessments. Formative assessments are done through regular quizzes and assignments. During the semester, the goal is to take frequent, low-stakes quizzes to reinforce key concepts and reinforce learning. Small software design and development tasks serve as checkpoints that enable both students and instructors to assess ongoing progress. Not to forget, but one of the most essential parts that helps with better understanding is in-class participation. Activities such as live coding sessions, group discussions, and in-class problem-solving exercises provide immediate feedback and encourage active participation.

Summative Assessments are done through capstone projects and a final exam, as mentioned. Comprehensive projects that require students to apply most of the software development cycle assess their ability to integrate learned concepts into a cohesive product. The final exam, which combines theoretical questions (typically 6) with practical problem-solving scenarios (usually 2), assesses overall understanding.

5. Results and Discussion

This section is intended to present the results of the study (Table 1). Since our research is related to the course adaptation, we chose a particular moment to discuss the students' achievement on the exam. This specific moment dates to 2021, when the new curriculum was approved, and it should be fully in force starting from 2022. The significant change for our exam is that the course was moved from the fifth to the fourth semester. Within the same school year, we offered the same course in both semesters – for students in the fifth semester in the winter and for those in the fourth semester in the summer.

After that, we had new students only for the fourth-semester curriculum, while in the fifth semester, the course was organized only for students who, for any reason, had not yet completed their third year. For example, in the actual school year (2024/2025), 42 students are still enrolled in the course in the fifth semester, while the regular class of the summer semester has 245 students.

Between 2013 and 2020, the course was only offered in the fifth semester, and the acceptance rate among students was exceptionally high by local standards. Since it is an obligatory course, the number of students is determined by the quota for third-year students, which has varied between 180 and 240 over the years. The average grade was 7.65 out of 10, and the average result for the in-semester project completion was 60,58% of the total points. The number of students who managed to complete the course and pass the exam in the same year was 71,89% on average.

Considering all the reasons mentioned above, we recognized that the transition to the fourth semester could be a source of stress for new students. The biggest problem seemed to be a lack of programming experience that previous generations had gained through lab exercises in Data Structures and Programming Languages. To prepare for the new setup, we held several promotional lectures during the winter semester and adapted some of the examples, retaining the well-known C++ as much as possible.

In the 2021/2022 school year, we had the opportunity to have two generations in parallel. The older (third-year, winter semester) and younger generation (second-year, summer semester) had a similar setup. The adaptation was made according to the recommendations we received from the Department, and the result was satisfactory. The additional programming experience of one more semester yielded significantly better results for older students. There were 173 students in the older generation, compared to 233 in the new generation. Additionally, there was no indication that the younger generation is in any way worse; however, the results were significantly different. The course was completed by 76.30% of students in the older generation, compared to 62.66% in the younger generation. The number of students who completed all programming tasks was 78.61% in the older group, compared to 54.94% in the younger group. An additional two parameters favored the older generation – average points in projects (58.25% vs. 50.40%) and average grade (7.82 vs. 7.21).

Table 1
Student achievement by year and semester

Year-semester	Total students	Passed exam (%)	Finish projects (%)	Average project points (%)	Average grade
2013-2019 avg.	208	149.43 (71.89)	154.57 (74.31)	60.58	7.65
20/21-winter	218	153 (70.18)	135 (61.93)	61.65	7.82
20/21-summer	-	-	-	-	-
21/22-winter	173	132 (76.30)	136 (78.61)	58.25	7.63
21/22-summer	233	146 (62.66)	128 (54.94)	50.40	7.21
22/23-winter	87	32 (36.78)	6 (6.90)	43.30	6.53
22/23-summer	213	145 (68.08)	124 (58.22)	54.90	7.61
23/24-winter	55	10 (18.18)	2 (3.64)	37.50	7.00
23/24-summer	255	173 (67.84)	170 (66.67)	60.75	8.05
24/25-winter	42	-	-	-	-
24/25-summer	242	-	-	-	-

This does not look like a too dramatic change, but the general feeling was that many students decided to take the exam in the next year, and that a significant number struggle with the tasks.

Additionally, it is worth noting that the previous generation informed them that the course is “interesting, but lab-intensive,” as it should be. There were not many warnings that it could be challenging to follow, again, because the students from the 2013-2021 generation had more experience when they started our course.

What we immediately decided was to review the projects (6 of them) that students must deliver throughout the semester. The project definition was broken down into detailed tasks and structured into phases with clearly defined deliverables and deadlines, ensuring that students are always clear about what needs to be done and how this will affect their grades. Next, projects are modified to yield more interesting results and are shifted more towards entirely visual (Windows Forms) applications. For students, it is more interesting to develop simple games, such as Memory or Minesweeper, rather than perform, for example, some text evaluation of the same computational complexity.

In the years to come, we restructured the course in a way that immediately began with more, but simpler, programming tasks. Students received weekly assignments along with regular projects to complete. Due to the reaction, in the 2022/2023 school year, the percentage of students who passed the exam increased to 68.08%, which is much closer to the results from previous years. Average grade rose to 7.61, which is almost at the level of pre-curriculum change (7.65). Looking at the table, one can see that the data for the students remaining for the winter semester has dropped, but it must be noted that they are leftovers from the 2021 generation and older. In the current year, we still have 42 of them.

In the last completed year, 2023/2023, we added more consultative lectures and increased the time needed to explain and understand the development environment and debugging process. Additionally, we have added a few invited talks (outside of the lectures) from industry professionals who should promote software engineering in general. The numbers were improved compared to the previous years. The number of students who finished projects rose by two-thirds, and the average success rate was again more than 60%. One interesting fact was that the average grade on the final exam rose to 8.05, which is one of the best results ever.

The most significant change that we noticed is that students with better previous academic achievement rates put in more effort for this exam. Consultations helped students understand the theory more effectively, and the lectures from the external lecturer served as an invitation to those interested in software engineering. For the year to come, we expect better results, since some basic parts of UML and improved debugging techniques are moved to courses from previous semesters (Programming and Object-Oriented Programming). Additionally, we unified programming tools and IDEs, which provide the necessary foundation for the students' personal development.

At this moment, it seems that we are on the way to match pre-curriculum change numbers. However, since we continually promote software engineering, we plan to incorporate additional elements into the course in the years to come. Initially, we consider presentations and demonstrations. Oral presentations or software demonstrations, often accompanied by technical documentation, assess a student’s ability to communicate complex ideas and defend their design choices. The idea is to replace one or two of the current projects with one presentation and demonstration.

Overall, the course itself cannot be viewed as disconnected from the rest of the curriculum, but needs to be carefully positioned not only within it, but also in the broader scientific context, and its goals need to be clearly defined. If all these aspects are well-defined, the course can run successfully with a high acceptance rate. However, during evolution, various potential challenges may arise. In the case of the course we presented, which should serve as an introduction to software engineering, we described the response to a situation where the course significantly changed its position, requiring students to work harder. As a response, we restructured the course and adapted the evaluation scheme, which gave results after a period.

For a design of such a course, extra effort is needed to follow recommendations from the reference international organizations, such as ACM (Association for Computing Machinery). Besides the course's primary role of introducing software engineering, it also needs to cover a practical part, and its outcome must also be on the improvement of software design and development skills that students need to develop. For this reason, we try to add all necessary elements of software design practices to allow students to create more interesting pieces of software and to further promote software engineering as a discipline that connects all aspects of computer science.

6. Conclusion and Future Work

Effective course design in the introduction of software engineering relies on a continuous alignment between teaching methods and assessment strategies, taking into account the course's position in the curriculum. For example, a project- or task-based learning approach should incorporate both formative assessments that monitor the students’ progress and summative assessments that evaluate the final level of understanding of all course elements. Additional challenges arise when the course environment changes. In our study, we presented a scenario in which our course was moved earlier into the curriculum and examined the resulting reactions. After two years, we are on the way to returning to the goals that we were able to achieve in previous years.

Due to the new position, students start the course with less experience in general programming. As a response, we restructured the course and adapted the evaluation scheme, which gave results after a period. For the future, we plan to focus on technology-enhanced learning tools that can simultaneously deliver personalized content and track individual contributions, supporting both teaching and assessment. This holistic approach ensures that all aspects of a student’s learning experience—technical knowledge, practical implementation, and ethical considerations—are covered up to the expected level.

Next steps could include further incorporating emerging trends, such as gamification elements, into both teaching and assessments to boost engagement. Additionally, consider integrating AI and data analytics to personalize learning paths further and provide targeted interventions based on student performance data.

The main guidelines that we can state here are to come as close as possible to the students, try to understand their point of view, promote the course using all possible means, emphasize periodic checks, and keep students constantly active. All these aspects, together with joint student and teacher endeavor, could bring the best possible results.

Acknowledgements

The Ministry of Science, Technological Development and Innovation of the Republic of Serbia supported this work [grant number 451-03-137/2025-03/200102].

Declaration on Generative AI

During the preparation of this work, the author(s) used Grammarly to improve Grammar and for spelling checks.

References

[1] I. Sommerville, Software Engineering, 10th Edition, Pearson, 2015.
[2] O. Cico, L. Jaccheri, A. Nguyen-Duc, H. Zhang, Exploring the intersection between software industry and Software Engineering education - A systematic mapping of Software Engineering Trends, J. Syst. Softw. 172 (2021) 110736. doi:10.1016/j.jss.2020.110736.
[3] S. Tenhunen, T. Männistö, M. Luukkainen, P. Ihantola, A systematic literature review of capstone courses in software engineering, Inf. Softw. Technol. (2023) 107191. doi:10.1016/j.infsof.2023.107191.

[4] S. Ouhbi, N. Pombo, Software Engineering Education: Challenges and Perspectives, in: 2020 IEEE Global Engineering Education Conference (EDUCON), IEEE, 2020. doi:10.1109/educon45650.2020.9125353.

[5] V. D. V David, Software Engineering: A Roadmap, J. Sci. Technol. 7.10 (2022) 86–92. doi:10.46243/jst.2022.v7.i10.pp86-92.

[6] G. Regev, D. C. Gause, A. Wegmann, Experiential learning approach for requirements engineering education, Requir. Eng. 14.4 (2009) 269–287. doi:10.1007/s00766-009-0084-x.

[7] P. Morais, M. J. Ferreira, B. Veloso, Improving Student Engagement With Project-Based Learning: A Case Study in Software Engineering, IEEE Rev. Iberoam. Tecnol. Del Aprendiz. 16.1 (2021) 21–28. doi:10.1109/rita.2021.3052677.

[8] A. Saad, The Effectiveness of Project Based Learning with Computational Thinking Techniques in a Software Engineering Project Course, J. Contemp. Issues 12.1 (2022) 65–79. doi:10.37134/jcit.vol12.1.6.2022.

[9] A. Holland-Minkley, J. E. Barnard, V. Barr, G. Braught, J. Davis, D. Reed, K. Schmitt, A. Tartaro, J. D. Teresco, CS2023: Computer Science Curriculum Guidelines: A New Liberal Arts Perspective, ACM Inroads 16.1 (2025) 40–52. doi:10.1145/3700773.

[10] I. Trunova, S. Arhun, A. Hnatov, P. Apse-Apsitis, N. Kunicina, V. Myhal, Sustainable Approach Development for Education of Electrical Engineers in Long-Term Online Education Conditions, Sustainability 15.18 (2023) 13289. doi:10.3390/su151813289.

[11] V. Garousi, G. Giray, E. Tüzün, C. Catal, M. Felderer, Aligning software engineering education with industrial needs: A meta-analysis, J. Syst. Softw. 156 (2019) 65–83. doi:10.1016/j.jss.2019.06.044.

[12] O. E. Olorunshola, F. N. Ogwueleka, Review of System Development Life Cycle (SDLC) Models for Effective Application Delivery, in: Information and Communication Technology for Competitive Strategies (ICTCS 2020), Springer Singapore, Singapore, 2021, pp. 281–289. doi:10.1007/978-981-16-0739-4_28.

[13] G. Sawarkar, D. Rajput, Comparative Analysis of Various Software Development Life Cycle, Int. J. Comput. Sci. Mob. Comput. 11.8 (2022) 1–8. doi:10.47760/ijcsmc.2022.v11i08.001.

[14] A. Kovari, J. Katona, Effect of software development course on programming self-efficacy, Educ. Inf. Technol. (2023). doi:10.1007/s10639-023-11617-8.

[15] I. Trunova, S. Arhun, A. Hnatov, P. Apse-Apsitis, N. Kunicina, V. Myhal, Sustainable Approach Development for Education of Electrical Engineers in Long-Term Online Education Conditions, Sustainability 15.18 (2023) 13289. doi:10.3390/su151813289.

[16] H. Keuning, B. Heeren, J. Jeuring, Code Quality Issues in Student Programs, in: ITiCSE '17: Innovation and Technology in Computer Science Education, ACM, New York, NY, USA, 2017. doi:10.1145/3059009.3059061.

[17] H. Koç, A. M. Erdoğan, Y. Barjakly, S. Peker, UML Diagrams in Software Engineering Research: A Systematic Literature Review, Proceedings 74.1 (2021) 13. doi:10.3390/proceedings2021074013.

[18] B. WACHTER, The Bologna Process: developments and prospects, Eur. J. Educ. 39.3 (2004) 265–273. doi:10.1111/j.1465-3435.2004.00182.x.

[19] K. Sharan, Model-View-Controller Pattern, in: Learn JavaFX 8, Apress, Berkeley, CA, 2015, pp. 419–434. doi:10.1007/978-1-4842-1142-7_11.

[20] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Abstraction and Reuse of Object-Oriented Design, in: ECOOP' 93 — Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 1993, pp. 406–431. doi:10.1007/3-540-47910-4_21.

[21] R. Martin, Functional Design: Principles, Patterns, and Practices, Addison Wesley Professional, 2023.

[22] B. Joshi, SOLID Principles, in: Beginning SOLID Principles and Design Patterns for ASP.NET Developers, Apress, Berkeley, CA, 2016, pp. 45–85. doi:10.1007/978-1-4842-1848-8_2.

[23] A. N. Kumar, R. K. Raj, Computer Science Curricula 2023 (CS2023): The Final Report, in: SIGCSE 2024: The 55th ACM Technical Symposium on Computer Science Education, ACM, New York, NY, USA, 2024. doi:10.1145/3626253.3633405. https://dl.acm.org/doi/pdf/10.1145/3664191