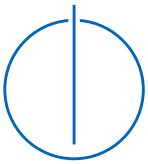
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH



Interactive learning - A Scalable and Adaptive Learning Approach for Large Courses

Dr. rer. nat. Stephan Krusche

Habilitation

Chair: Univ.-Prof. Dr. Tobias Nipkow
Department of Informatics, Technical University of Munich

Examiner: 1. Univ.-Prof. Dr. Bernd Brügge
Department of Informatics, Technical University of Munich

2. Univ.-Prof. Dr. Maria Bannert,
School of Education, Technical University of Munich

Abstract

In university courses with hundreds of students, instructors cannot interact with each student individually. This leads to the problem that students engage less or not at all in the learning activities and build up misconceptions due to lack of guidance. Existing teaching approaches such as blended learning, experiential learning, or active learning focus on instructor-student interaction in smaller courses.

This habilitation introduces a new teaching philosophy called interactive learning, a scalable and adaptive learning approach specifically for larger courses, and the associated teaching platform Artemis. Interactive learning allows instructors to teach small pieces of content in short cycles. Students can practice and reflect on the taught concepts in real-time based on the instructor's individual feedback. Interactive learning encourages creativity and does not limit students to sample solutions. Artemis reduces the overhead of correcting and providing feedback in exercises and exams. It automatically grades programming and quiz exercises and uses a supervised machine learning approach to grade modeling and text exercises based on similarity analysis semi-automatically.

Artemis has been deployed in 63 courses across ten universities and has been used by 30,000 students in these courses. In addition, 8,500 students completed their exams on Artemis in 31 online examinations with realistic exercises based on constructive alignment. Interactive learning and Artemis have been empirically evaluated in multiple case studies in computer science courses using a design-based research process with four hypotheses.

The case studies show that interactive learning is scalable to courses with more than 1,500 active students (H1) and increased student engagement by up to 165 % compared to traditional courses (H2). Interactive learning improved the students' learning outcome in modeling exercises by up to 87 % (H3). The semi-automatic grading process with individual feedback reduced the grading effort by up to 80 % (H4). These results show that individual interaction between instructors and students is possible even in larger courses. The teaching platform Artemis enables future research projects in the field of learning analytics and adaptive learning.

Zusammenfassung

In Universitätskursen mit Hunderten von Studierenden können die Lehrenden nicht auf jeden einzelnen Teilnehmer eingehen. Dies führt zu dem Problem, dass sich die Studierenden weniger oder gar nicht auf die Lernaktivitäten einlassen und aufgrund mangelnder Anleitung falsche Vorstellungen aufbauen. Bestehende Lehransätze wie integriertes Lernen, erfahrungsbasiertes Lernen oder aktives Lernen konzentrieren sich auf die Interaktion zwischen Lehrenden und Studierenden in kleineren Kursen.

Diese Habilitation stellt eine neue Lehrphilosophie namens interaktives Lernen vor, ein skalierbarer und adaptiver Lernansatz speziell für größere Kurse, und die dazugehörige Lehrplattform Artemis. Interaktives Lernen erlaubt es den Lehrenden, kleine Teile des Inhalts in kurzen Zyklen zu vermitteln. Die Studierenden können die vermittelten Konzepte auf der Grundlage des individuellen Feedbacks des Lehrenden in Echtzeit üben und reflektieren. Interaktives Lernen fördert die Kreativität und beschränkt die Studierenden nicht auf Musterlösungen. Artemis reduziert den Aufwand für Korrekturen und Feedback in Übungen und Prüfungen. Es bewertet automatisch Programmier- und Quizaufgaben und verwendet einen überwachten Ansatz zum maschinellen Lernen, um Modellierungs- und Textaufgaben auf Basis einer Ähnlichkeitsanalyse halbautomatisch zu bewerten.

Artemis wurde in 63 Kursen an zehn Universitäten eingesetzt und von 30.000 Studierenden in diesen Kursen genutzt. Darüber hinaus haben 8.500 Studierende in 31 Online Klausuren mit realitätsnahen, auf konstruktivem Abgleich basierenden Übungen ihre Prüfungen mit Artemis abgelegt. Interaktives Lernen und Artemis wurden in mehreren Fallstudien in Informatikkursen empirisch evaluiert, wobei ein designbasierter Forschungsprozess mit vier Hypothesen verwendet wurde.

Die Fallstudien zeigen, dass interaktives Lernen auf Kurse mit mehr als 1.500 aktiven Teilnehmern skalierbar ist (H1) und das Engagement der Studierenden im Vergleich zu traditionellen Kursen um bis zu 165 % erhöht (H2). Interaktives Lernen verbesserte den Lernerfolg der Studierenden bei Modellierungsaufgaben um bis zu 87 % (H3). Der halbautomatische Bewertungsansatz mit individuellem Feedback reduzierte den Bewertungsaufwand um bis zu 80 % (H4). Diese Ergebnisse zeigen, dass eine individuelle Interaktion zwischen Lehrenden und Studierenden auch in größeren Kursen möglich ist. Die Lehrplattform Artemis ermöglicht künftige Forschungsvorhaben im Bereich Lernanalytik und adaptives Lernen.

Acknowledgements

Many people influenced me during the four years of research and teaching while working on the habilitation. I want to thank them and acknowledge their support.

First and foremost, I would like to express my deep gratitude to my mentor and role model Bernd Brügge, who inspired me again and again with new ideas, endless enthusiasm, and support during the time of my habilitation. You created an environment of opportunity and growth, trusted me and my work, supported my thoughts, provided valuable feedback, encouraged new ways of thinking, and always gave me opportunities to grow and the freedom to try new approaches. I am excited to follow in your footsteps and continue to develop our visions of excellent teaching.

I also want to thank Maria Bannert for accompanying my habilitation research and for her valuable feedback, and Tobias Nipkow for chairing the habilitation project. I am thankful to the dean and the faculty of Informatics for the possibility of writing this habilitation. I would like to thank Manuela Fischer-Russenberger for her outstanding support in all administrative questions regarding the habilitation.

I am very grateful to all members of the Chair for Applied Software Engineering. I learned a lot from all of you, and I am thankful for all discussions, feedback, fun, and encouragement. I want to thank Jan Philip Bernius, Andreas Seitz, Nadine von Frankenberg, and Lara Marie Reimer for their great support in the different courses we have taught. Without your help, it would not have been possible to improve education and to win prizes. I want to thank Helma Schneider, Monika Markl, and Uta Weber for their support in all administrative, financial and personal questions. I am also grateful to Ruth Demmel, Andreas Jung, and the whole team of the computer operations group (RBG) of the Informatics faculty for live streams, video recordings, and infrastructure support for the operation of Artemis.

I also like to thank all co-authors of papers and articles who contributed to this habilitation: Andreas Seitz, Anna Kovaleva, Bernd Brügge, Cecil Wöbker, Christopher Laß, Dora Dzvonyar, Han Xu, Irina Camilleri, Jan Philip Bernius, Jürgen Börstler, Kirill Krinkin, Lara Marie Reimer, Nadine von Frankenberg, and Sami Afifi.

Many people have contributed to the development of Artemis and I am indebted to the following people (in alphabetical order) for their time and passion into developing the teaching platform Artemis: Adem Khachnaoui, Alex Mardale, Alexander Karpp, Alexander Ungar, Alexander von Trostorff, Andi Turdiu, Andreas Seitz, Anh Montag, Birtan Gültekin, Can Sarpkaya, Christian Femers, Christian Ziegner, Christopher Lass, Clemens Zuck, Daniel Crazzolara, David Otter, Dominik Fuchs, Dominik Münch, Filip Gregure-

vic, Florian Glombik, Francisco De las Casas Young, Hanya Elhashemy, İsmet Melih Özbeyli, Ivan Chimeno, Jan-Thilo Behnke, Johann Rottenfusser, Johannes Stöhr, Jonas Petry, Jonas Schulte-Coerne, Josias Montag, Julian Christl, Julian Frielinghaus, Julian Willand, Katie Sebina, Kilian Schulte, Linus Michel, Lukas Franke, Mai Ton Nu Cam, Mario Amah, Marius Schulz, Martin Wauligmann, Matthias Linhuber, Maximilian Meier, Minh Tran, Moritz Issig, Nadine von Frankenberg, Niclas Schümann, Nicolas Ruscher, Patrik Zander, Philipp Bauch, René Lalla, Riccardo Padovani, Sascha Beele, Sebastian Jagla, Simon Leiß, Stefan Klöss-Schuster, Stefan Kreuzer, Stefan Waldhauser, Tessa Ruckstuhl, Tiffany Cauthen, Tobias Priesching, Tobias Schamel, Valentin Schlattinger, and Varnika Tyagi.

I am grateful to all students who participated in courses and exams and who provided valuable feedback. In addition, I would like to thank more than 300 tutors who have supported these courses and made their first teaching steps.

Finally, I want to express my love and gratitude to my family and my wife Anke, my daughter Lina, and my son Felix. Writing a habilitation requires even more than the researcher's full attention. I am indebted for your love and devotion, your understanding, and your support. Without you, this habilitation would not have been possible!

Contents

Publication Preface

This habilitation is based on the following 10 publications (in chronological order). The four most significant publications are highlighted in blue. Chapter 8 provides an overview of the publications and describes the main contribution of each paper for this habilitation.

1. Publication [KSBB17]

Stephan Krusche, Andreas Seitz, Jürgen Börstler, and Bernd Bruegge (2017). Interactive Learning – Increasing Student Participation through Shorter Exercise Cycles. In Proceedings of the 19th Australasian Computing Education Conference. ACM.

DOI: https://doi.org/10.1145/3013499.3013513

2. Publication [KvFA17]

Stephan Krusche, Nadine von Frankenberg, and Sami Afifi (2017). Experiences of a Software Engineering Course based on Interactive Learning. 15. Workshop Software Engineering im Unterricht der Hochschulen.

Proceedings: http://ceur-ws.org/Vol-1790

3. Publication [KBC+17]

Stephan Krusche, Bernd Bruegge, Irina Camilleri, Kirill Krinkin, Andreas Seitz, and Cecil Wöbker (2017). Chaordic Learning: A Case Study. In Proceedings of the 39th International Conference on Software Engineering. IEEE.

DOI: https://doi.org/10.1109/ICSE-SEET.2017.21

4. Publication [KS18]

Stephan Krusche and Andreas Seitz (2018). ArTEMiS - An Automatic Assessment Management System for Interactive Learning. In Proceedings of the 49th Technical Symposium on Computer Science Education. ACM.

DOI: https://doi.org/10.1145/3159450.3159602

5. Publication [KDXB18]

Stephan Krusche, Dora Dzvonyar, Han Xu, and Bernd Bruegge (2018). Software Theater — Teaching Demo Oriented Prototyping. Transactions on Computing Education. ACM

DOI: https://doi.org/10.1145/3145454

6. Publication [KS19]

Stephan Krusche and Andreas Seitz (2019). Increasing the Interactivity in Software Engineering MOOCs - A Case Study. In Proceedings of the 31st Conference on Software Engineering Education and Training (52nd Hawaii International Conference on System Sciences). ScholarSpace.

DOI: https://doi.org/10.24251/HICSS.2019.915

7. Publication [LKvFB19]

Christopher Laß, Stephan Krusche, Nadine von Frankenberg, and Bernd Bruegge (2019). Stager: Simplifying the Manual Assessment of Programming Exercises. 16. Workshop Software Engineering im Unterricht der Hochschulen.

Proceedings: http://ceur-ws.org/Vol-2358

8. Publication [KvFRB20]

Stephan Krusche, Nadine von Frankenberg, Lara Marie Reimer, and Bernd Bruegge (2020). An Interactive Learning Method to Engage Students in Modeling. In Proceedings of the 42nd International Conference on Software Engineering. ACM.

DOI: https://doi.org/10.1145/3377814.3381701

9. Publication [BKKB21]

Jan Philip Bernius, Anna Kovaleva, Stephan Krusche, and Bernd Bruegge (2020). Towards the Automation of Grading Textual Student Submissions to Open-ended Questions. In Proceedings of the 4th European Conference on Software Engineering Education. ACM.

DOI: https://doi.org/10.1145/3396802.3396805

10. Publication [BKB21]

Jan Philip Bernius, Stephan Krusche, and Bernd Bruegge (2020). A Machine Learning Approach for Suggesting Feedback in Textual Exercises in Large Courses. In Proceedings of the 8th Conference on Learning at Scale. ACM.

DOI: https://doi.org/10.1145/3430895.3460135

Chapter 1

Introduction

> "Tell me and I will forget. Show me and I will remember. Involve me and I will understand. Step back and I will act."
>
> — Chinese Proverb

The demand for computer scientists has grown significantly in the last few years. As a result, it is common to have more than 1000 students in bachelor courses and more than 500 students in master courses[1]. The high number of students makes it challenging for the instructors to keep the quality of education high because it is impossible to interact with all students individually and adapt lectures and exercises to heterogeneous student groups.

In university courses with hundreds of students, instructors still tend to use a unidirectional teaching approach. They only explain concepts and theory to students instead of involving the students in the educational activities. The interaction between students and instructors is then limited or non-existing, leaving students confused and lost if they cannot ask clarification questions and do not receive feedback. In addition, content delivery in lectures and content deepening in exercises are separated, leading to a limited learning experience.

There are learning approaches such as active learning [JJS91], computer-based learning [GK04], and experiential learning [Kol84], which try to overcome these problems. However, their use by instructors in large courses is challenging and limited due to increased effort and scalability problems. For example, while it makes sense to activate 20 students in a small classroom with a group discussion, it is impossible to have a group discussion with 1.000 students distributed into multiple lecture halls that the instructor still guides.

Nevertheless, there is clear evidence that guidance is essential to facilitate learning and prevent misconceptions [KSC06]. Therefore, it is vital to involve students in the learning activities, even in large courses. Exercises and examples are essential elements in teaching and learning [Van96]. Carefully developed and integrated examples increase the

[1] The numbers of the computer science faculty of the Technical University of Munich show that in the academic year of 2020/21, there are 2.508 first-year students: https://www.in.tum.de/en/the-department/profile-of-the-department/facts-figures/facts-and-figures-2020

learning outcome [SC85, TR93]. In particular, in complex problem spaces, like software development, "[l]earners may learn more by solving problems with the guidance of some examples than solving more problems without the guidance of examples" [TR93, p. 42].

Computer science and software engineering are problem-solving activities that require creativity, collaboration, and practical application of knowledge [Whi07, Sha04]. Current higher education practices lead to a multitude of rules, guidelines, and order, which in turn makes "motivated, creative teachers and students feel less and less in place" [Mul15, p. 2]. Creative activities such as modeling are essential in software engineering but hard to teach and assess, particularly in large courses. There is usually not only one correct solution to a modeling exercise.

Much expertise is necessary to assess whether the model is complete, unambiguous, and correct and whether it abstracts the essential aspects. In such cases, it would be problematic only to distribute one sample solution and generic feedback. Students with a different solution could assume that their solution is wrong, even if it is correct. This reduces creativity and stimulates learning by heart instead of understanding the good and problematic aspects of the solution and the iterative aspects of improving the solution based on feedback.

Another problem is that large courses are typically not adaptive to heterogeneous student groups. However, due to the increasing complexity in the world, there is a need for diverse employee profiles. The number of study programs increases, while it is impossible to offer specialized courses for each study program. Many courses are offered for minor subjects where students might not fulfill all prerequisites of a course. This leads to courses with heterogeneous participants who have varying prior knowledge and interests in the course.

While there are approaches to individualizing students' learning experience [Son05], it is particularly challenging to implement them in large courses. In a classroom with 20 students and direct interaction, an instructor could identify weaker and stronger students more efficiently and separate them into multiple, smaller groups. This is no longer feasible for instructors who work with hundreds of students in different locations.

Instructors also face challenges during the assessment of students in large courses. Many instructors focus on lower cognitive skills to reduce the grading effort because exercises are easier to assess when students only need to repeat the concepts they have learned before. The assessment of explanations, differentiations, or new creative solutions to existing problems creates a high effort. However, not including exercises in the final exams that request higher-level learning goals stimulates learning by heart and reduces the students' learning outcome for the desired competencies and skills.

The principles of constructive alignment [Big03] have been promoted as a powerful way to circumvent these issues [HVVP21]. Nevertheless, instructors struggle to implement constructive alignment because there is a gap between the taught learning outcomes, the learning activities, and the assessment in the examination, which should be aligned with each other. Paper-based exams force students to write little code on paper while learning and practicing to implement more complex programs in integrated development environments.

1.1 Objectives

To solve these challenges, we developed a new teaching philosophy **interactive learning** for larger courses, and the associated teaching platform **Artemis** which supports interactive learning with individual feedback.

The main goal is that instructors can apply a student-focused teaching and learning approach in courses with hundreds of participants to increase the involvement of students. Instructors teach and exercise small chunks of content in short cycles using technology. They provide immediate feedback so that students can reflect on the content and increase their knowledge incrementally.

Immediate, individual feedback can increase the required effort for instructors in large courses significantly. For example, instructors have to grade submissions with similar aspects and repeatedly apply the same grading criteria. To prevent this and to increase the quality, consistency, and fairness of individual feedback, Artemis should automate repeated and cumbersome activities during the grading process. Then, instructors have more time for direct interaction and communication with students.

A focus on lower cognitive skills in exercises and examinations can also reduce the grading effort in larger courses but limit students' learning outcomes and creativity. However, it is essential that instructors can offer exercises based on higher cognitive skills without increasing the grading effort significantly. Therefore, we want to investigate and integrate supervised machine learning approaches. This approach would allow Artemis to learn which aspects of an exercise are correct and wrong during manual assessments. Based on similarity analysis, Artemis should then suggest feedback for similar elements in other submissions. Using this approach, instructors could stimulate higher cognitive skills, including creative aspects, while still providing individual feedback and allowing multiple solutions.

Instructors should also be able to stimulate higher-order skills in examinations based on constructive alignment. Otherwise, students would not be able to learn the competencies needed when they start working after university. Therefore, Artemis should offer a digital exam mode based on the same exercise types as the course. Instructors can then more easily align the learning activities in the course with the assessment in the exam exercises. In addition, automatic assessment can reduce the organization and grading effort in exams. This would make it more likely that instructors choose examination exercises with higher cognitive skills.

To address the challenge of a heterogeneous, large audience, we want to integrate adaptive learning into Artemis. Students should be able to choose individual learning paths based on their existing skills. Artemis should be able to offer exercises in multiple difficulty levels. Inexperienced students should not lose the motivation to work on the exercise. Experienced students should still feel challenged and learn new aspects. Therefore, Artemis should be able to choose the correct exercise difficulty for each individual student.

1.2 Research Process

This habilitation follows a design-based research process [Col92, Hoa02]. We develop interactive learning and Artemis as interventions in a formative approach [Sta03] to iteratively and incrementally apply them in case studies in actual courses to evaluate and to enhance them. This allows refining the objectives for interactive learning and the requirements for Artemis according to the feedback of the involved instructors and students. This is necessary because the objectives and requirements can change over time and based on user feedback. The research process includes the following activities in no particular order due to its formative nature:

- Description of the teaching philosophy interactive learning
- Implementation of Artemis based on the objectives defined in the previous section
- Application of interactive learning and Artemis in multiple courses and case studies
- Dissemination of interactive learning and Artemis in other universities
- Empirical evaluation of the benefits of interactive learning and Artemis

After the initial description of interactive learning and the initial implementation of Artemis, we will apply them in multiple instances of three large courses and one massive open online course (MOOC) at the Technical University of Munich:

1. Project Organization and Management (POM)
2. Patterns in Software Engineering (PSE)
3. Introduction to Software Engineering (EIST)
4. MOOC: Software Engineering Essentials (SEECx)

The application of interactive learning and Artemis in actual courses (field studies) reveals essential insights that help refine the teaching philosophy and further implement the teaching platform. Instructors can use Artemis during the lecture period (around three months) and refine it in the lecture-free period (around two months). This allows us to empirically evaluate the benefits in natural settings that are not limited by the artificial nature of experiments. The empirical evaluations analyze the effectiveness of interactive learning concerning the scalability, engagement, learning outcome, grading effort, and feedback quality. We design qualitative and quantitative evaluations to investigate the following five hypotheses in the following courses:

H1 **Scalability:** Interactive learning can be used in large courses with more than 1,500 active students participating in exercises simultaneously.
H2 **Engagement:** Interactive learning increases the participation and motivation of students.
H3 **Learning outcome:** Interactive learning improves the learning outcome for students.
H4 **Grading effort and feedback quality:** Supervised machine learning reduces grading effort while improving feedback quality.
H5 **Adaptability:** Interactive learning adapts the difficulty of a course to each individual student by using machine learning.

To validate these hypotheses, we collect quantitative data based on the usage of Artemis. We can then compare the final grades in exams with the exercise performance or analyze the manual and automatic assessments to compare their quality and consistency. In addition, interviews and surveys with the involved stakeholders and course evaluations allow us to collect qualitative data and additional insights about the personal benefits of instructors and students.

Positive validation of the hypotheses will enable the transfer to other disciplines outside our research group so that interactive learning is used practically in courses at other universities and in industry, nationally and internationally. This enables an empirical evaluation of the hypotheses where we have no direct influence on the intervention. Furthermore, the dissemination of interactive learning and Artemis will prove that they provide enough benefits so that other instructors at the same university or other universities apply them.

1.3 Outline

This habilitation is publication-based. Chapters 1 to 7 are a synopsis, i.e., a summary of the most important contributions. Parts of the text of the synopsis are reused[2] from the publications referenced in Chapter 8.

The synopsis includes an overview of existing teaching approaches (Chapter 2) and a description of related learning approaches and automatic assessment systems (Chapter 3). It defines the teaching philosophy interactive learning (Chapter 4) and the associated teaching platform Artemis (Chapter 5). Finally, it includes empirical evaluations demonstrating the benefits of interactive learning and Artemis (Chapter 6), discusses future research directions to extend the work (Chapter 7), and a reprint of the ten publications at conferences, journals, and workshops (Chapter 8).

Chapter 2 provides an overview of existing teaching approaches that have influenced interactive learning. We describe problem-based learning, cooperative learning, blended learning, experiential learning, active learning, technology-enhanced learning, and team-based learning. Finally, we introduce the framework constructive alignment, which proposes to align learning goals with learning activities and assessments together with a taxonomy of learning goals from lower-order complexity (e.g., remember) to higher-order complexity (e.g., evaluate).

Chapter 3 describes the related work in terms of other learning approaches similar to interactive learning and other automatic assessment systems that try to achieve similar objectives. A variety of automatic assessment systems exists, all with slightly different purposes. However, compared to existing systems, Artemis is unique in several aspects. First, it focuses on multiple exercise types: programming, modeling, text, quiz, and file upload. Second, it includes different assessment strategies: automatic, semi-automatic based on machine learning to allow multiple correct solutions and hybrid assessment

[2]Reuse means that some text blocks (e.g., sentences or paragraphs) are copied from the publication and enriched with additional details or reformulated to better fit into the context of the synopsis of the habilitation.

approaches (e.g., first automatic, then manual feedback). Third, it integrates assessment training, structured grading instructions, double-blind assessment, and assessment leaderboards to improve the fairness and consistency of manual assessments. Fourth, it integrates additional features such as team exercises, lectures, questions and answers, plagiarism checks, and an exam mode.

Chapter 4 explains interactive learning as a teaching philosophy, allowing instructors to teach small pieces of content in short cycles [KSBB17, KvFA17, KS19]. Interactive learning integrates aspects of active learning [BE91], blended learning [GK04], and experiential learning [Kol84]. It is based on constructive alignment [Big03] and includes creative aspects [KBC+17] and team-based learning [MWCDF82]. We describe examples for simple interactive exercises and more complex exercises spanning multiple lectures such as software theater [KDXB18].

Chapter 5 describes the main features and the architecture of the teaching platform Artemis[3], which includes multiple subsystems and external components [KS18, LKvFB19]. Artemis connects to a version control system and a continuous integration server to realize programming exercises with automatic feedback independent of the programming language. It offers quiz exercises with multiple-choice, drag and drop, or short-answer questions that are evaluated automatically. Ares[4] is a Junit testing framework supporting instructors in writing test cases for programming exercises with helpful feedback. Orion[5] is an IntelliJ plugin that simplifies the participation of students in programming exercises and the creation of programming exercises for instructors.

Modeling exercises are based on Apollon[6], a lightweight and easy-to-use online modeling editor that supports seven UML diagrams and three other diagram types [KvFRB20]. Compass[7] is integrated into Artemis and realizes semi-automatic assessments based on supervised machine learning for modeling exercises using similarity analysis. Finally, Athene[8] is an external system connected to Artemis. It implements semi-automatic assessments based on supervised machine learning and natural language processing for text exercises [BKKB21, BKB21].

Among other additional features, Artemis offers an exam mode that supports digital exams with exercise variants and plagiarism control. The exam mode allows instructors to assess the same competencies the students have learned in the course before. Moreover, students can participate from home even if the internet connection is not reliable.

[3] In Greek mythology, Artemis is the goddess of the hunt, the wilderness, wild animals, the moon, and chastity. The Artemis teaching platform is open-source: https://github.com/ls1intum/Artemis

[4] In Greek mythology, Ares is the god of courage and war. He is one of the brothers of Artemis. The Junit testing framework Ares is open-source: https://github.com/ls1intum/Ares

[5] In Greek mythology, Orion is a giant huntsman and a hunting partner of Artemis. The IntelliJ plugin Orion is open-source: https://github.com/ls1intum/Orion

[6] In Greek mythology, Apollon is the god of oracles, healing, archery, music and arts, sunlight, knowledge, herds and flocks, and protection of the young. He was one of the brothers of Artemis. The online modeling editor Apollon is open-source: https://github.com/ls1intum/Apollon

[7] A compass is a tool that shows directions and can be used for navigation. Following Greek mythology, Artemis uses a compass to find the right individual feedback automatically.

[8] In Greek mythology, Athene is the goddess of wisdom, handicraft, and warfare. She is one of the sisters of Artemis. The system to support semi-automated assessment of textual exercises Athene is open-source: https://github.com/ls1intum/Athene

Chapter 5 also explains the system architecture of Artemis and its related subsystems and external components. The Artemis server subsystem can be scaled horizontally and uses different mechanisms such as network file systems, shared caches, a registry for monitoring, and a broker to distribute messages to maintain the consistency between multiple server instances. The central server instance is responsible for scheduling events, e.g., the start of a live quiz or removing write access in the version control repositories in online exams, while the other server instances allow handling many submissions simultaneously.

Chapter 6 describes the design-based research process, consisting of case studies in actual courses and multiple quantitative and qualitative evaluations. It also explains and discusses the results found in these evaluations. Interactive learning is scalable to courses with more than 1,500 active students submitting their exercise solutions simultaneously. It increased student engagement in the last four lectures of one course by 165 % compared to a previous traditional instance of the same course. It improved students' learning outcome in modeling exercises in another course by up to 87 % when comparing the exam results with the previous course without the intervention. The semi-automatic grading process in Artemis reduced the grading effort by up to 80 % in text and modeling exercises.

Chapter 7 summarizes the main contributions of the habilitation and provides ideas for future research directions. The main contributions are the description of a teaching philosophy interactive learning specifically for larger courses, the development of an associated teaching platform Artemis, and the empirical evaluation of both in the context of multiple courses. Future research directions include integrating learning analytics to let students reflect on their performance and allow instructors to reflect on whether students have understood specific topics more easily.

Learning paths and exercises with different difficulty levels would build the first steps towards adaptive learning. Machine learning could enable the automatic configuration based on the individual skills of one student. From the technical perspective, it is desirable to migrate Artemis from a monolithic architecture to microservices [New15] and micro frontends [Gee20], allowing a more lightweight and flexible configuration during the deployment. Kubernetes clusters could further improve the automatic scaling and performance when many students use Artemis simultaneously [VSTK18].

Chapter 8 contains a summary and a reprint of the ten publications on which this habilitation is based. We describe the primary research contribution for each publication, explain how the publication relates to interactive learning, and mention how the author of this habilitation contributed to the publication.

Chapter 2

Background

"Never stop learning, because life never stops teaching"

— Lin Pernille

This chapter provides an overview of the field of learning philosophies. It summarizes the most important learning approaches, which have influenced interactive learning, and presents constructive alignment.

Problem-based learning is a technique to learn about a subject through the experience of problem-solving. Educators facilitate learning by supporting, guiding, and monitoring this process. Working in groups, students identify what they know, what they need to know, and how and where to access new information that leads to the resolution of the problem [BF98].

Cooperative learning is an educational approach that aims to organize classroom activities into social learning experiences. Students work in groups to complete tasks collectively towards a common goal. The role of the teacher changes from giving information to facilitating students' learning. As a result, everyone succeeds when the group succeeds [J+91].

Blended learning allows students to learn through the delivery of content and instructions via computer-mediated activities, digital media, and online media. While still attending traditional teaching environments, face-to-face methods are combined with computer-mediated activities. Thus, blended learning facilitates a simultaneous, independent, and collaborative learning experience [GK04].

2.1 Experiential Learning

Experiential learning is the process of learning from experience, a methodology in which educators engage with students in direct experience to increase knowledge, develop skills, and clarify values [Kol84]. Aristoteles said: "For the things we have to learn before we can do them, we learn by doing them." John Dewey followed this idea with his statement

that "there is an intimate and necessary relation between the process of actual experience and education" [Dew38, p. 6].

Experiential learning is considered to be more efficient than passive learning like reading or listening. It is in contrast to academic learning, where students acquire information through studying a subject without the necessity for direct experience. The main dimensions of experiential learning are analysis, initiative, and immersion [Kol84]. Academic learning promotes the dimensions of constructive learning [Ver98] and reproductive learning [JJWH06]. Both methods instill new knowledge, though academic learning uses more abstract techniques, whereas experiential learning actively involves the student in a concrete experience such as an exercise.

Experiential learning focuses on the learning process through reflection on doing [Fel11]. Thus, the student takes an active role compared to didactic learning, where the student plays a relatively passive role [Bea10].

Kolb developed the experiential learning cycle with four stages [Kol84].

1. **Concrete experience:** Work on a substantial task.
2. **Reflective observation:** Take a timeout from "doing" and step back from the concrete task. Instead, review what has been done and experienced.
3. **Abstract conceptualization:** Make sense of what has happened. Interpret the events and understand the relationships between them.
4. **Active experimentation:** Put the learning into practice, ideally regularly.

2.2 Active Learning

Active learning is an educational approach to increase student involvement and excitement with the subject being taught [BE91]. Instead of students acting as receivers of knowledge by passively listening, active learning emphasizes developing student skills and engaging them in activities such as small group discussions or class games.

Grabinger and Dunlop emphasize that authentic contexts encourage students to take more responsibility and engage them in learning activities that promote high-level thinking processes [GD95]. An authentic context in software engineering would, for example, be the management of a project where students experience typical activities such as meeting and task management. The students' learning progress is supported and assessed through realistic tasks such as planning and conducting a meeting or distributing tasks within the team.

Michael Prince examined the evidence for the effectiveness of active learning and discussed its common forms [Pri04]. He concluded that active learning positively influences knowledge transfer and student performance. Joel Michael reviewed the literature and found that there is evidence that active learning improves the learning outcome compared to more passive approaches [Mic06].

However, certain active learning approaches are not feasible for large classrooms. For example, it is impossible to have a group discussion with 300 students in the same lecture

hall. In addition, it is essential that instructors "place a strong emphasis on guidance of the student learning process" [KSC06, p. 1] to prevent misconceptions.

Active learning led to improved learning experiences on different cognitive skill levels of Bloom's taxonomy in university courses. It emphasizes developing skills through active participation and engagement in activities. It moves away from teacher-centered approaches, where teachers instruct and students listen passively, to a more student-centered approach, where students play an active role.

Bonwell and Eison define active learning as "anything that involves students in doing things and thinking about the things they are doing" [BE91, p. 19]. It requires students to regularly assess their problem-solving skills and understanding of the taught concepts [Mic06]. Brophy and Good identify four main premises of active learning [GB87]:

1. Students construct their meanings
2. New learning builds on prior knowledge
3. Learning is enhanced by social interaction
4. Learning develops through **authentic** tasks

Prince [Pri04] and Michael [Mic06] found support for all forms of examined active learning in their studies. They conclude that active learning improves learning outcomes compared to passive learning approaches.

2.3 Technology-enhanced Learning

Technology-enhanced learning (TEL) illustrates the application of information and communication technologies to teaching and learning [KP14]. The term is broadly used: a shared understanding has not been developed in higher education of what constitutes enhancing the student learning experience. TEL is concerned with improving the quality and outcomes of learning in all those varied circumstances where technology plays a significant supportive role [GR10].

TEL uses technologies to support learning in local (e.g., in the lecture hall) or remote (e.g., at home) locations. Traditionally, technologies have included instructional films, radio, and television [Wes10]. Today, TEL focuses on computer-based technologies, including smartphones and other smart devices.

Learning is a process whereby the student reviews concepts and ideas, assimilates these through practice, and demonstrates mastery [SL19]. Enhancements of learning can improve this practice and process. Modern technologies allow enhancements through the facilitation of learning activities by technology in various forms. TEL can offer scalability, flexibility, and new methods of facilitating learning.

Technology-enabled active learning (TEAL) is a variation established in a technology-enhanced multimedia studio, emphasizing constructivist-oriented teaching and learning [Shi12]. Benefits appear in different aspects: students are more interested in attending lectures and are more active in participating in extracurricular activities. They are more enthusiastic about and confident in helping other students. Achievements and positive responses of students improve the dedication and confidence of the instructor.

A related concept is collaborative learning, e.g., in the form of computer-supported collaborative work (CSCW), which was incorporated into education and empirically studied for many years [DJF09]. Collaborative learning is an approach to teaching and learning that involves groups of students working together to complete a task, solve a problem, or create a product [LL12]. Collaborative learning is used as a generic term for various learning approaches involving students' joint effort [GMT+92]. Students work in groups of two or more, try to search for understanding, solutions, or meanings.

2.4 Team-based Learning

One variant of collaborative learning is team-based learning (TBL) which Larry Michaelsen first introduced at the University of Oklahoma in the 1980s [MWCDF82]. The goal was to give students the benefits of learning in small teams within a large class setting in the business school environment [SYK+13]. TBL has been gaining traction at academic institutions since, especially in medical and pharmaceutical education at colleges in the US [ACF+13, HR20]. The main difference to collaborative learning is that students in teams are more strongly connected, focusing on a specific outcome, a common goal, which requires coordination over a more extended period [Ald77, FH+97].

The primary objective of TBL is to go beyond just covering the material of a course and focus on engaging the students in practicing the course concepts by applying them to solve problems. Thus, while students spent some time ensuring the mastery of course content, most class time is dedicated to team assignments, usually divided into five to seven central work units for the whole course.

One of the main challenges of TBL is the transition from learning about the course concepts to applying them since this requires a role shift for both the instructor and the students: The instructor is now no longer primarily dispensing information but instead managing the instructional process. In contrast, the students change from passive recipients of information to active participants in team-based exercises requiring them to act upon the received information.

Michaelsen identifies four essential elements that instructors need to implement in his eyes in order to achieve this transition successfully [MS08]:

1. Properly formed and managed groups: TBL intends the team formation process to be strictly overseen by the instructor so that the following variables can be controlled that will influence the effectiveness of the teams: Teams should have a similar amount of resources to draw from when it comes to completing the assignments. This means that each team should have an adequate balance concerning student characteristics that would favor or hinder performing well in the course, such as having previous course-related experience.

Team members should ideally be demographically diverse to bring in different perspectives during the problem-solving phase. Although culturally homogeneous teams tend to be more effective initially, this edge disappears after some time, with heterogeneous teams even scoring higher in specific task metrics [WKM93].

Coalitions among members of a team should be avoided since they threaten the overall development of the team. If a subset of team members already has a previous relationship, likely, insider-outsider tension will negatively affect the team cohesiveness. Since humans tend to seek out others similar to them, letting the students form teams by themselves would almost certainly lead to potentially troubling subgroups [FD84].

Formed teams need to stay the same for the entire course duration since teams need time to become high-performing and self-managed in their learning process.

2. Student accountability for individual and group work: Compared to the normal mode of working on exercises individually and only being accountable to the instructor, TBL demands more from the students in the sense that they are now accountable also to their teammates, both for the quality and quantity of their work.

The first step is to ensure that students are individually accountable for the preparation before a team exercise. If several team members are unprepared when starting the exercise, the team will likely not succeed. The next step is to account for the individual contribution of each team member to the joined work product. Finally, students evaluate the contributions in a peer assessment process since they tend to be the only ones who have enough information for an accurate assessment.

TBL suggests two factors for assessing the quality of the team's work product: The teams should develop a product that can be easily compared across teams, and such comparisons should happen frequently and on time.

3. Frequent immediate student feedback: Students' immediate feedback is seen as a **primary instructional lever** in TBL. It is essential in the retention of content [HT07] and advantageous for the development of the team [BM04].

4. Assignments that promote both learning and team development: When designing exercises suitable for engaging students in teamwork, it is fundamental that these exercises require team interaction. This is usually the case when course concepts have to be used to make decisions based on many factors and report the result of their decisions in a simple form.

If an exercise requires the students to produce a complex, lengthy work product, it leads to less interaction among the students due to simply dividing up the individual parts of the work product and then working on them alone [MS08].

2.5 Constructive Alignment

Constructive alignment proposes to align learning goals with learning activities and assessment. It was introduced by John Biggs [Big03] and is derived from constructivism and curriculum theory [MB97]. Figure 2.1 shows the three dimensions and their alignment. Biggs refers to the basic idea of constructivism that students construct their learning through learning activities instead of passively receiving knowledge from the instructor. Thus, all components in the learning system - the learning goals, the learning activities, and the assessment tasks - are aligned.

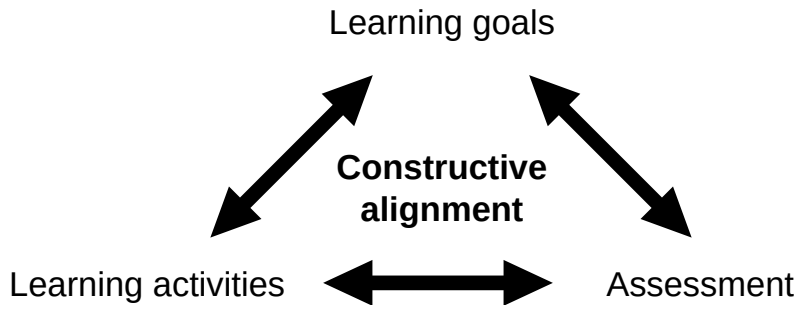


Figure 2.1: Constructive alignment proposes to align learning goals with learning activities and assessment (adapted from [Big03]).

Bloom developed a framework to classify expectations of what students should learn as the result of an instruction [BEF+56]. It serves as a common language about learning goals. An example in software engineering would be that students can describe the waterfall model. Bloom classified six major categories of cognitive processes ordered by their complexity from lowest to highest: knowledge, comprehension, application, analysis, synthesis, and evaluation [BEF+56].

1. **Knowledge:** Recalling fundamental facts, terms, or dates.
2. **Comprehension:** Understanding or translating the meaning of instructions and problems based on prior learning.
3. **Application:** Solving problems in new situations by applying acquired knowledge.
4. **Analysis:** Examining and breaking information into parts by identifying motives or causes to understand its structure.
5. **Synthesis:** Compiling parts together differently by combining elements in a new pattern or proposing alternative solutions.
6. **Evaluation:** Judging the value of ideas or materials and defending opinions by making judgments based on a set of criteria.

Krathwohl revised the original taxonomy and extended it by four knowledge dimensions ordered from concrete to abstract knowledge: factual, conceptual, procedural, and metacognitive [Kra02].

1. **Factual:** Basic knowledge to acquaint with a discipline and solve problems.
2. **Conceptual:** Connection of basic knowledge in a larger context.
3. **Procedural:** Methodology of knowledge application using skills, techniques, and methods.
4. **Metacognitive:** Knowledge about the use of particular strategies for learning or problem-solving.

The revised taxonomy includes six cognitive skills levels ranked from lower to higher-order complexity:

1. **Remember:** Retrieve relevant knowledge from memory.
2. **Understand:** Determine the meaning of instructional messages.

3. **Apply:** Carry out or use a procedure in a given situation.
4. **Analyze:** Break material into its parts and detect how the parts relate to one another.
5. **Evaluate:** Make judgments based on criteria and standards.
6. **Create:** Put elements together to form new knowledge.

Chapter 3

Related Work

"Education is the most powerful weapon which you can use to change the world."
— Nelson Mandela

This chapter summarizes related work. Section 3.1 presents other learning approaches and compares them to interactive learning. Section 3.2 presents alternative learning management systems with automatic assessments and relates them to Artemis.

3.1 Learning Approaches

There are several courses in computer science that apply active learning techniques. They report an increase in students' learning, engagement, and performance.

Kurtz et al. describe an active learning approach using micro labs [KFT+14]. Students perform short activities individually or in groups during a lecture and submit their answers to an automated grading system. They receive constructive feedback and can revise their answers. Kurtz et al. conclude that micro labs can increase the students learning experience. Their approach is comparable to in-class exercises in interactive learning. Students also have a time limit for activities and submit their solutions to an automated grading system or upload them for manual assessment if no automated assessment is possible. Both approaches have one thing in common: instructors use them in the classroom during lectures. Heckman reports about an increase in student engagement for the use of in-class labs [Hec15].

Another popular approach is *think pair share* (TPS), where students work on a problem individually, then in small groups, and finally reflect on it with the whole class. Interactive learning proposes a similar approach where students first experience a concept separately and then apply it within a team. Kothiyal et al. describe a large programming course that uses TPS [KMMI13]. The course includes programming labs and lectures with two TPS activities: students worked on questions individually first and then in pairs, while the instructor helps in case of questions. In addition, instructors facilitated class-wide discussions concerning the former tasks. The study reports an average of 83 % student

engagement for TPS-based activities. This approach parallels the course set up in the case studies since two courses introduce individual and team exercises, similar to the think and pair phases. Interactive learning also covers the sharing stage, as students could discuss with the instructor during the reflection.

Campbell et al. describe an approach based on the flipped classroom concept with video lectures, labs, and assignments [CHCG14]. They also use quizzes, contributing to the course grade as implemented in Artemis. However, the authors do not use in-class exercises and report a low lecture attendance rate. Interactive learning implements frequent homework assignments, as well as immediate feedback for activities to keep students motivated. Campbell et al. suggest the use of such an approach instead of labs for future improvement.

Gruenewald et al. focus on the challenge of conducting programming lectures as MOOCs [GMTW13] integrating active experimentation and relating to concrete experience. They developed design guidelines for MOOCs to support experiential learning. Practical exercises should improve the learning experience of students based on their existing expertise. They describe different MOOC participants ranging from passive students to meta-designing students who implement their material and share it with others. Interactive learning promotes a similar approach. Instructors can use it in university courses and online courses.

3.2 Automatic Assessment Systems

A variety of systems for automatic assessment exists [IAKS10, BFPS17]. Many of these systems have been published in different conferences, journals, and workshops. For example, the workshop "Automatische Bewertung von Programmieraufgaben" (English: automatic evaluation of programming tasks) has been offered four times in the last years[1]. Table 3.1 shows an overview of the systems found in a literature review in alphabetical order. In addition, multiple surveys have been published, summarizing and categorizing these systems. Ala-Mutka [AM05] and Douce et al. [DLO05] published the first extensive survey on this topic. They describe multiple auto assessment tools, categorize them into dynamic and static assessments, and differentiate between local and web-based systems.

The survey by Ihantola et al. focuses on identifying key features of automatic assessment tools, such as supporting different programming languages, allowing resubmissions, or providing a sandbox environment to handle malicious submissions [IAKS10]. The authors state that most systems are not open-source or available otherwise, even if a publication describes the development of a prototype. A survey by Queiros says interoperability and compatibility to other services is a critical factor for automatic assessment systems [QL12]. He concludes that many existing assessment tools do not consider this

[1]The proceedings of the workshops "Automatische Bewertung von Programmieraufgaben" are available on http://ceur-ws.org/Vol-1067 (first workshop 2013), http://ceur-ws.org/Vol-1496 (second workshop 2015), http://ceur-ws.org/Vol-2015 (third workshop 2017), and https://dl.gi.de/handle/20.500.12116/27935 (fourth workshop 2019).

factor and that future solutions have to improve this. This section summarizes essential systems mentioned in Table 3.1 and relates the described system to Artemis.

#	Name	Website	Publication
1	AA Framework		[GHV+19]
2	Arena	https://arena.kpi.fei.tuke.sk	[MBP19]
3	ASB	https://asb.hochschule-trier.de	[BHS17]
4	ASSYST		[JU97]
5	AuDoscore	https://github.com/FAU-Inf2/AuDoscore	[OKP17]
6	Aurora	https://aurora.iguw.tuwien.ac.at	[PL14]
7	Autograder.io	https://github.com/eecs-autograder/autograder.io	
8	Autolab	https://github.com/autolab/Autolab	
9	AutoLEP		[WSM+11]
10	Autotool	https://gitlab.imn.htwk-leipzig.de/autotool/all0	[Wal17]
11	BOSS	https://www.dcs.warwick.ac.uk/boss/index.php	[LJ95]
12	Codeboard	https://codeboard.io	[AHEAK+17]
13	CodeLab	https://www.turingscraft.com	[WBL06]
14	CodeOcean	https://github.com/openHPI5codeocean	[SKT+16]
15	CourseMarker		[HGST05]
16	CTpracticals		[GTR+10]
17	eduComponents		[AFR08]
18	EduJudge		[VRV+12]
19	EMSEL		[Bal16]
20	FaSt-generator		[RSZ15]
21	GAME		[BGNM04]
22	GATE	https://github.com/csware/si	[SOP11]
23	Gradescope	https://www.gradescope.com	[SKGA17]
24	Graja	https://graja.hs-hannover.de	[Gar16]
25	JAssess		[GTR+10]
26	InfoMark	https://github.com/infomark-org/infomark	
27	INGInious		[DGR+15]
28	JACK	https://jack-community.org	[MSS+18]
29	Jutge.org		[PRC+17]
30	KATTIS		[EKN+11]
31	Marmoset		[SHP+06]
32	Mooshak		[LS03]
33	Online Judge		[CKLO03]
34	OK	https://github.com/okpy/ok	[SHLD16]

#	Name	Website	Publication
35	Oto		[TGPS08]
36	PABS		[IDDBI15]
37	Praktomat	https://github.com/KITPraktomatTeam/Praktomat	[KSZ02]
38	ProgTest		[dSMB11]
39	PLWeb		[TLL13]
40	RoboProf		[Dal99]
41	SAC		[ALSCiMP08]
42	SIETTE	https://www.siette.org	[CGM+04]
43	Submitty	https://github.com/Submitty/Submitty	[PTB+17]
44	Test My Code	https://github.com/testmycode	[VVLP13]
45	TRAKLA	https://www.cse.hut.fi/en/research/SVG/TRAKLA2	[HM93]
46	ViLLE		[KHK+15]
47	VIOPE		[VA02]
48	VEA		[Gus17]
49	Vocareum	https://www.vocareum.com	
50	VPL	https://github.com/jcrodriguez-dis/moodle-mod_vpl	[Thi15]
51	Web-CAT	https://github.com/web-cat/web-cat-subsystem-Core	[Edw03]

Table 3.1: Overview of systems with automatic assessment functionalities found in a literature review (sorted in alphabetical order)

The **AA Framework** is an extension of the Code::Blocks open-source IDE[2] [GHV+19]. It supports automatic assessments for C++ programs using a differential testing approach. It compares the output of the student's solution against the predefined one of the instructors by running multiple tests and then checking if the two program outputs coincide. If a test fails, the oracle provides feedback and explains why the submitted program is not correct. Artemis also uses output-based tests but offers higher flexibility because it is independent of the programming language and better integrates the students' workflows on a learning management system.

Galan et al. used used the AA Framework for several years involving 14,944 students and found that the system prepares students to cope with more complex assignments. In addition, completion of the assignments positively influences the final exam grades. These findings are similar to the results that we found (see Chapter 6).

Arena is a system for testing and evaluating student programming assignments stored in a Gitlab repository [MBP19]. The student creates a separate repository with a defined structure for each exercise and commits and pushes the solution. Arena assesses the latest student submissions in parallel at specific intervals (e.g., every 3 hrs) using workers in containers. It first downloads the submission from the Gitlab repository, checks the structure, compiles and tests the code in the container, and writes the results into a

[2] https://www.codeblocks.org

database to display it to the students on a website. Artemis uses a similar approach for programming exercises but is easier for students because it creates the git repositories automatically based on predefined templates.

The **ASB** system is a web application used since 2006 at the University of Applied Sciences in Trier, which allows students to submit solutions to programming exercises [BHS17, SBOG17, GBSO17]. The system can automatically evaluate student programs written in Java, Python, C++, and Android using dynamic tests and static code analysis. It provides individualized feedback on submitted solutions. This enables fast feedback on the correctness of the submission compared to requirements specified by the instructor. The system is similar to Artemis, which also enables dynamic testing and static code analysis of programming submissions. However, Artemis is independent of the programming language and is based on version control and continuous integration.

ASSYST was one of the first systems developed in 1997 to support tutors assessing programs [JU97]. It is a semi-automated assessment system that uses humans to mark C and Ada programs. It offers a graphical user interface for all aspects of the grading process and considers different grading criteria in the automatic assessment: correctness, efficiency, style, complexity, test data adequacy. The system should support tutors in the assessment. However, the human is still in charge and can refuse to accept the diagnosis of the system. Jackson and Usher report that the experience with the system has been encouraging. Artemis supports the assessment based on similar grading criteria and offers a manual grading step after the automatic assessments, similar to the judgment by tutors in ASSYST.

AuDoscore is an automatic grading system for Java and Scala programs used at the Friedrich-Alexander University Erlangen-Nürnberg with up to 750 students in a course on algorithms and data structures [OKP17]. It extends JUnit and supports the creation of exercises and corresponding grading tests using lightweight annotations. Students need to upload their solution in the web-based lecture management system called "Exercise Submission Tool". Public tests are provided as a smoke test to ensure that the student solutions adhere to the expected interfaces. Secret tests check the submission in more detail. Consequently, public and secret tests contribute to the grading. Artemis also supports hidden and public tests. It additionally uses static code analysis, version control, and continuous integration for the automatic assessment.

Aurora is a learning platform used at the Technical University of Wien that facilitates many activities to structure and conduct larger courses [PL14, Luc20]. It supports contextual communication and interaction amongst students to engage with each other and the course content. It includes a peer review module. Students can submit their solutions and then review and evaluate the work of other students. Aurora follows a constructivist learning approach based on active learning and encourages collaboration between peers. Instructors act as facilitators. Artemis uses a similar learning approach to activate the students. It also includes communication (questions and answers) and collaboration features (team exercises) but not peer-review assessments. Instead, tutors, experienced students who passed the course in the previous year, provide high-quality feedback after review training.

Autograder.io is an open-source automated grading system for Python programming exercises: instructors write test cases without worrying about running them. It was developed at the Computer Science department of the University of Michigan, where it supports 5,000 students per semester in multiple courses. It offers customizable feedback systems and flexible grading policies. In addition, it allows to manually grade and annotate student source code for style and code quality. Students can work individually or in groups. Artemis offers the same functionality independent of the programming language and automatically assesses style and code quality based on static code analysis while offering a subsequent manual grading step.

Autolab is a course management service initially developed by a team of students at Carnegie Mellon University and is now used by multiple universities in the United States. It enables instructors to offer auto-graded programming assignments to their students in a large variety of programming languages. It includes a leaderboard to encourage competition between students and allows instructors to annotate the student code manually with feedback. In addition, it includes a plagiarism check based on Stanford's MOSS implementation [SWA03]. Artemis also supports many programming languages with predefined templates but integrates JPlag [PMP+02] for plagiarism checks. While Artemis does not include a leaderboard, students can see how they compare to the class average for each exercise.

Autotool is an e-learning system used at HTWK Leipzig to automatically assess exercises on algorithms and data structure [Wal17]. The system consists of a stateless semantics service for generating task instances and scoring solution attempts and provides a web interface. Instructors configure, manage and test exercises and configure points, while students can view the exercises and edit their submission to an exercise. Exercises, e.g., focus on heap-ordered trees, hash tables, or graphs. The student submits a textual representation of the solution. Intentionally, autotool does not integrate a graphical input option to get students used to scientific notation through terms in an appropriate signature. Instead, the input parsers generate helpful error messages. The erroneous position is marked. Possible continuations are printed to the output so that the student can understand the issues. Artemis does not focus on the textual representation of graphs, but instructors have used it in similar courses where students needed to implement algorithms in the form of programming exercises. Apollon, the modeling editor integrated into Artemis, supports the visual representation of graphs.

The **BOSS** online submission system is a course management tool developed starting 1993 by the computer science department at the University of Warwick [LJ95, JGB05]. It allows students to submit assignments and contains a selection of tools to allow instructors to grade assignments. It also provides an automatic testing system based on JUnit and code metrics capabilities. In addition, it integrates natural language plagiarism detection based on Sherlock [JL99]. Unfortunately, the last version was released six years ago, and BOSS is not actively maintained anymore. Artemis also uses JUnit for automatic testing but integrates a different plagiarism checker JPlag [PMP+02].

Codeboard is an open-source and web-based IDE to teach programming in the classroom [AHEAK+17]. It is cloud-based and uses regression testing. It allows students to edit and submit source code in the browser to simplify participation. It can be integrated into existing learning management solutions based on the LTI (learning tools

interoperability) standard and is, e.g., used in MOOCs on the edX platform.. Artemis also integrates an online code editor that instructors can use to set up an exercise. Tutors can use it to review code and provide feedback manually. Students can use it to participate in programming exercises. Artemis also offers integration based on LTI, is used in MOOCs [KS19], and offers many additional features such as static code analysis.

CodeLab [WBL06], formerly known as WebToTeach [AB99], is now available as a commercial product from the company Turingscraft. CodeLab is a web-based interactive programming exercise system for introductory programming courses in Python, Java, C++, C, and C# and supports short exercises. Each exercise focuses on a particular programming idea or language construct. It provides instant feedback and can be integrated into existing learning management systems. Instructors using the system state that it leads to a better understanding of programming amongst the students. Artemis is programming language independent and provides templates for more languages than CodeLab. It additionally provides static code analysis and plagiarism detection and can also be used for more complex assignments.

CodeOcean is an online system for programming exercises in MOOCs [SKT+16], e.g., on the openHPI learning platform[3]. It integrates an online code editor with syntax highlighting and the possibility to execute code directly in the browser. Code is executed on the server. The output is shown on the students' web browser. Students do not need to set up an integrated development environment. CodeOcean includes unit tests to provide feedback for students and test their code.

While the code of the unit tests is hidden, students can run their solution against the unit tests and get feedback on whether the solution passes or fails the tests. If a unit test fails, the result is shown with an error message defined by the instructors. A manual review by instructors is not feasible, and peer review of source code has not been implemented in CodeOcean. Artemis offers the same features: an online code editor with instant feedback based on tests and the display of the output of the code. It provides additional features such as static code analysis and manual grading. Both CodeOcean and Artemis can be integrated into other learning management systems based on LTI.

CourseMarker (formerly known as CourseMaster) is a computer-based assessment system at the University of Nottingham [HGST05]. It is an improved version of Ceilidh, one of the first known automated assessment tools since 1989 [FTHS99]. CourseMarker provides immediate results and feedback to students. Instructors can create a variety of programming exercises that benefit the students learning experience. Students implement the program based on the problem description provided submit it. CourseMarker compares the program output with the expected output and shows the results to the students. Students can try to resolve the problem if it is not satisfactory, upon the instructor's permission. Artemis applies a similar iterative approach but uses unit tests and static code analysis to verify the correctness of the student's program in addition to output checks.

eduComponents is a system that splits e-learning and e-assessment platforms into separate systems, allowing independent deployment and easier adoption [AFR08]. Artemis targets the same idea, but the goal is not to implement another assessment system. In-

[3] https://open.hpi.de

stead, Artemis reuses workflows provided by existing version control and continuous integration tools to achieve similar results.

EMSEL is an exercise management system for e-learning to practice HTML and JavaScript [Bal16]. It allows students to search exercises through specific criteria using a search engine. EMSEL allows instructors to create exercises with a problem statement and a solution. Students can submit their solutions and receive feedback to self-evaluate their programming skills. EMSEL is an extensible system and can be used for other programming languages such as XML, PHP, and C++. Artemis is independent of the programming language and additionally offers static code analysis and manual assessment possibilities.

GATE (generic assessment and testing environment) is an automatic assessment system to support programming education developed at TU Clausthal [SOP11, MS13]. It helps with the process of exercise management and assessment for large programming classes at the university level. Furthermore, it allows students to assess their skills and checks solutions for plagiarism. In addition, it provides additional special functions for tasks from the areas of Java programming and UML modeling [SSLP12].

MFS is a GATE subsystem and uses an expert solution against which the modeling submissions are compared but is flexible to accommodate variability characteristic to modeling tasks. However, modeling is a creative activity where multiple solutions can be correct. In contrast, Artemis uses a supervised machine learning approach to support the assessment of modeling exercises that do **not** depend on one single expert solution.

Gradescope is a system geared toward the assessment of handwritten submissions [SKGA17] by scanning papers. Instructors review the submissions online by dynamically creating grading rubrics during the assessment. They group similar submissions so that the same grade is applied or use the suggested grouping of the system. Artemis uses a similar idea by sharing feedback with groups of answers in the semi-automatic assessment approach for text and modeling exercises. However, Artemis groups individual text segments or modeling elements, whereas Gradescope groups entire student submissions. Artemis requires instructors to inspect every submission and suggests feedback that is reused from previous similar submissions.

Neither system requires a training dataset of previously assessed answers. For exercises with a limited number of possible answers, Gradescope allows the instructor to review multiple (similar) submissions simultaneously and effectively reduces the number of submissions to grade. However, this approach is unsuitable for highly creative exercises because the grouping will not work on the submission level. Then, instructors would need to review many submission groups with only one or a few submissions.

Graja is a Grader for Java programming exercises that extends JUnit 4 by providing helper classes for grading tasks and evaluating student programs' observable behavior [Gar16]. It uses annotations to improve the feedback shown to the students. It verifies functional and non-functional requirements of simple Java code. It is based on test-driven development. Graja also performs a static code analysis to provide feedback about quality characteristics. Artemis offers similar functionalities independent of the programming language. It is also based on test-driven development, where instructors

write tests and students write the actual source code so that the predefined tests pass. It also includes static code analysis.

InfoMark is an open-source solution for programming courses supporting the automatic assessment of programming assignments. It includes direct feedback for students after unit testing their homework submissions automatically in a Docker[4] sandbox. Students can drag and drop homework solutions as zip files. InfoMark includes live metrics to monitor submission failures and traffic. It has supported thousands of students and several tutors during university courses in Tübingen. Tutors see the test output and can add additional feedback.

InfoMark is language agnostic and is easy to install as it only requires Docker and docker-compose. It is scalable with background workers that can be deployed on different machines. The unit test workload will be distributed amongst the workers. Artemis supports similar programming exercises features and runs the tests in lightweight Docker containers on build agents. In contrast to InfoMark, students use version control to work on programming exercises. Artemis additionally supports team exercises and static code analysis.

JACK is an automatic exercise and examination system developed by the paluno institute at the University of Duisburg-Essen [MSS+18]. It offers the possibility to check programming tasks statically and dynamically and generate visualizations of data structures. Furthermore, instructors can use other generic task types like multiple-choice and cloze tests for exercises. Randomization of content and connection to computer algebra systems is also possible.

JACK uses graph transformation rules on a graph generated from the submitted source code. This approach allows an instructor to specify checks that a reasonable solution must pass. This approach is powerful. However, it is also rather difficult for instructors to develop these checks. JACK offers a variety of exercises types:

1. Form-based exercises for multiple-choice and cloze tasks: A form-based task requires students to make entries in fields. It consists of one or more questions of different types.
2. Automated essay scoring for text assignments. Several research prototypes of checkers can evaluate the submitted texts automatically based on regular expressions or trained models. However, these prototypes are not suitable for production use without manual review.
3. The JAVA task type is intended for programming tasks. First, students download program code templates and edit them locally. Then, they upload the files again so that JACK can evaluate them.
4. The UML task type is intended for modeling tasks. Students cannot solve such tasks directly in the browser. Instead, the instructor provides a potentially empty model file that students can download and edit in a compatible modeling editor. The students then upload the edited file so that JACK can evaluate it. So-called checkers assess the modeling submission in the file.

[4]The tool Docker allows to create, deploy, and run applications in lightweight containers that can be started faster than virtual machines: https://www.docker.com

Artemis offers a similar variety of features and exercise types but is open-source and based on modern software engineering principles such as version control and continuous integration. It offers static code analysis and interactive exercise instructions and is independent of the programming language. In contrast to JACK, Artemis offers semi-automatic assessment for modeling and text exercises based on supervised machine learning. Artemis also integrates a lightweight modeling editor directly in the browser, and students do not need to download files, edit UML models in external tools, and upload them again.

Marmoset focuses on information collected during the development process of students [SHP+06]. The system takes regular snapshots of the students' progress and requires that students install custom tools on their workstations to submit solutions. It allows the instructor to study the development process of the students and to identify common bug patterns. By using version control systems and teaching its application, Artemis achieves the same outcome. Students commit multiple iterations of their solution, resulting in a commit history that can be evaluated. This history allows instructors to identify common mistakes and study problem-solving behavior.

Praktomat is a programming exercise management system for better quality control in practical programming courses [KSZ02, BHS17, KKS+20, EFF+21]. It was first developed for practical programming courses at the University of Passau in 1998. The Karlsruhe Institute of Technology (KIT) developed the open-source software and has been using Praktomat in their first-semester programming course since 2008. Since 2011, a modern reimplementation based on the Django web framework has been used. Praktomat automatically compiles and tests the solutions to programming exercises and final tasks, mainly for Java assignments.

Administration and the automatic testing of other programming languages (e.g., Haskell or Isabelle/HOL) are also possible. In the winter semester of 2021, 862 students and tutors have used it in the programming course at KIT. Praktomat supports multiple variants of an exercise and allows mutual feedback among students. Artemis offers similar functionality in a more modern user interface. In contrast to the Praktomat, Artemis uses version control and continuous integration and offers interactive exercise instructions. Notably, the KIT is currently migrating from Praktomat to Artemis.

RoboProf is an automatic assessment system that presents notes and exercises to students [Dal99]. Students can upload submissions. RoboProf tests those submissions against test data, provides immediate and minimal feedback, and archives the results. It includes almost exclusively correct and incorrect statements about the output of student code. RoboProf uses a modular grading engine. It lacks static code analysis and does not measure code attributes such as size, speed, indentation, and commenting. On the other hand, Artemis supports rich feedback messages based on dynamic tests and static code analysis. It also allows tutors to provide additional feedback by reviewing the student code.

SIETTE is an automatic assessment system for complex programming exercises based on adaptive learning [CGM+04, CBB18]. The selection of the following question and the decision to finish the evaluation is performed dynamically based on a student profile created and updated during interaction with the system. SIETTE includes several types

of questions, from multiple-choice to open questions, and can assess mathematical content. It is extensible with user-defined plugins. In addition, it offers the assessment of computer programs or complex tasks that require student interaction, like drawing or music playing.

It uses classical test theory and item response theory to grade the results of different items of evidence obtained from students' results instead of heuristic assessment as in other systems [CBB18]. The methodology considers program proofs as items, calibrates them, and obtains the score using different procedures which measure overall validity, reliability and diagnose the quality of each item. SIETTE collects and processes all data to calculate the student knowledge level and adapts to the student's ability level. It can be integrated into Moodle[5] using a plugin.

SIETTE offers optional manual assessment and plagiarism detection based on MOSS [SWA03]. Additionally, gamification elements are under development. Artemis and SIETTE share many features, such as automatic assessments of programming and quiz exercises. However, Artemis uses a different, more deterministic approach. Artemis does not yet offer adaptive learning, but this should be added as future work. In contrast, Artemis implements semi-automatic grading for modeling and text exercises.

Submitty is an open-source course management system at Rensselaer Polytechnic Institute [PTB+17, MPAC20]. It offers many features for a modern automatic assessment system, is actively developed and maintained on Github, and is similar to Artemis in many aspects. It can automatically assess programming submissions based on tests and static code analysis and offers manual grading. It offers online exams, integrates discussion forums, distributes course material, and offers peer grading. In contrast, Artemis offers a question and answer functionality directly integrated next to the lecture or exercise to capture the context of the question. Artemis does not offer peer grading but focuses on reviewer training to increase quality and fairness. Submitty does not offer modeling exercises and also does not offer interactive exercise instructions.

Test My Code is a suite of tools developed to make the lives of instructors and students easier [VVLP13, PLVV13]. The system has been used at the University of Helsinki and Aalto University. Instructors can use automatic bookkeeping facilities and integrate automated guidance into programming exercises. As a result, they can spend more time explaining and help students with complex concepts.

Test My Code provides support for downloading and submitting code within the IDE NetBeans. Consequently, students can focus on programming, which is helpful in introductory courses, where students struggle to apply and practice what they just learned. Artemis also offers a plugin Orion for IntelliJ that students can use to work on programming exercises in the IDE directly. Interactive exercise instructions in Artemis are comparable to the automated guidance in Test My Code. In addition, Artemis offers static code analysis and optional manual grading for additional feedback on the code quality of the students.

TRAKLA supports instructors in teaching data structures and algorithms by offering individually tailored programming exercises of algorithm simulations [HM93, KMS03].

[5]Moodle is a popular open-source learning management system: https://moodle.org.

It does not assess programming coursework as such. Instead, TRAKLA automatically assesses student solutions and presents them with feedback via email. Email-based feedback is neither instant nor immediate. Artemis provides rich text feedback with interactive instructions directly in the browser.

VPL (Virtual Programming Lab) is a plugin for Moodle that offers the automatic assessment of programming exercises [Thi15]. VPL requires a separate execution server, which is also called a jail server. This jail server runs the test cases on the programs submitted by the students. If a student program crashes the jail server, Moodle is not affected and can continue to perform normally. VPL supports any language with a compiler or interpreter that runs on Linux with an executable that can output text that VPL can evaluate. Instructors have been successfully implementing VPL assignments for Python, Java, and Assembly. They can define the rubric of how the student program is evaluated and graded. Artemis offers similar capabilities regarding programming exercises and can also be integrated into Moodle using LTI. In contrast to VPL, it offers static code analysis and manual grading.

WebCAT was first created in 2003 as one of the first automatic assessment tools [Edw03]. It has been developed as open-source software and allows extensibility by plugins. In terms of assessment, it supports student-written tests, test coverage, static code analysis, and a combination of both automatic and manual grading. Artemis covers most of the features of WebCAT while removing the dependence on a single software product. Instead, it consists of multiple independent software systems that are connected using standard interfaces. This approach leads to higher flexibility because individual parts of the architecture can be replaced, for example, in favor of lower costs, superior support, larger communities, or general management decisions.

Chapter 4

Interactive Learning

> "Learning is not the product of teaching. Learning is the product of the activity of learners."
>
> — John Holt

Interactive learning is a scalable and adaptive teaching philosophy based on the constructive alignment that puts the interaction with a student into the core of the educational activities. It integrates aspects of team-based learning and creativity to stimulate problem-solving skills and soft skills.

Interactive learning decreases the cycle time between teaching a concept and practicing it in class in multiple short iterations: Instructors teach and exercise small chunks of content in short cycles and provide immediate feedback so that students can reflect on the content and increase their knowledge incrementally. Interactive learning expects the active participation of students and the use of computers (laptops, tablets, smartphones) in classrooms. Figure 4.1 shows the iterative process of interactive learning, where each iteration consists of five phases that are performed several times during lecture:

1. **Theory:** The instructor introduces a new concept and describes the theory behind it. Students listen and try to understand it.
2. **Example:** The instructor provides an example so that students can refer the theory to a concrete situation.
3. **Practice:** The instructor asks the students to apply the concept in a short exercise adapted to the individual student's existing knowledge and skills. The students submit their solutions to the exercise.
4. **Feedback:** The instructor provides immediate feedback to the student submissions using an automatic assessment system. Alternatively, the instructor can show multiple exemplary solutions and discuss their strengths and weaknesses.
5. **Reflection:** The instructor facilitates a discussion about the theory and the exercise to reflect on the first experience with the new concept.

In large education environments with hundreds of students who participate in a course simultaneously, tutors help in the conduction of the exercises. Tutors walk through the

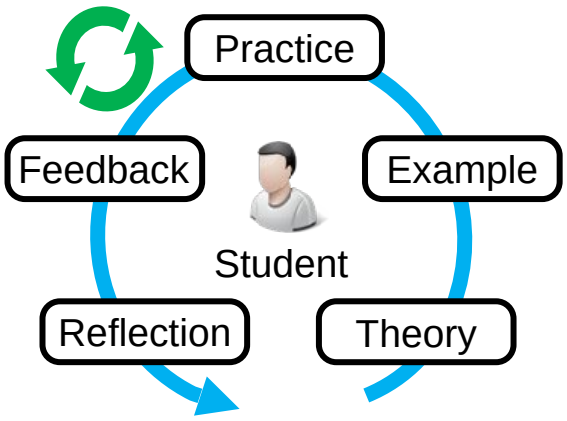


Figure 4.1: Interactive learning puts the individual student into the core of the learning activity and follows an iterative process that is conducted multiple times in lectures.

classroom, answer questions and provide help if problems occur or exercise instructions are unclear. The evaluation of the submitted solutions can be automated using tool support or manual with the help of tutors who review the submitted solutions and provide immediate feedback. The degree of automation depends on the exercise type and the format of the solution. The evaluation of programming assignments can, e.g., be automated. Instructors can apply interactive learning in individual exercises and team exercises. The degree of automation depends on the exercise type.

Interactive learning is a teaching philosophy that combines active learning, technology-enhanced learning and includes team-based learning. It also integrates creative aspects as described in [KBC+17] to emphasize the creativity of learning activities and stimulate self-organization. This chapter defines the term "Interactivity" and then shows how instructors can integrate interactive learning into the course syllabus and lectures. We also describe simple exercises and a more complex example using software theater [KDXB18] that consists of multiple exercises spanning two lectures.

4.1 Interactivity

When an instructor says, 'I am trying to make my classes more interactive', the meaning of *interactive* seems intuitive. However, an agreed-upon definition of interactivity is hard to find. The term is used in the context of various fields, such as communication, advertising, websites, the internet, and education, to name a few [LRJG09]. Since Rafaeli's statement "Interactivity is a widely used term with an intuitive appeal, but it is an underdefined concept" [Raf88, p. 110], several attempts have been made to define the concept of interactivity in its different contexts leading to the inconsistent use of the term [May05].

The term *interactivity* is rooted in the term interaction. The Cambridge dictionary defines *interaction* as "an occasion when two or more people or things communicate with or react to each other." Steffensen differentiates between interaction and interactivity: "whereas interaction captures a relation of dependence between separable systems, interactivity explores their bidirectional coupling" [Ste13, p. 198].

Jones and Gerard propose that all social interaction is goal-oriented [JG67]. They distinguish four different types of interactions according to their influence on the interaction partners:

1. **Pseudo interaction:** A sequence of actions follows predefined patterns. The actions of an involved participant are not intended to be interpreted by the other participant.
2. **Asymmetrical interaction:** One participant follows his or her intentions while another party reacts complementarily to the previous actions.
3. **Reactive interaction:** The involved parties do not interpret the intentions of the other party's actions and react in an isolated form.
4. **Interdependent symmetrical interaction:** Aligning one's action to the own intentions while considering the intentions of the others in a reciprocal fashion.

Similarly, Rafaeli argues that interactivity is best defined by considering the degree of responsiveness [Raf88]. He recognizes three levels of communication. Two-way (non-interactive), reactive (quasi-interactive), and fully interactive communication. For an interaction to be classified as two-way communication, messages must flow bilaterally. If the messages cohere with previous messages, the interaction is at least reactive or quasi-interactive. The third level, complete interactivity, references the content, nature, or presence of earlier references. Rafaeli formally defines "*interactivity* is an expression of the extent that in a given series of communication exchanges, any third (or later) transmission (or message) is related to the degree to which previous exchanges referred to even earlier transmissions" [Raf88, p. 111]. Domagk, Schwartz, and Plass [DSP10] and Johnson et al. [JBK06] identified two fundamental conditions common in interactivity research:

1. At least two participants interact with each other.
2. The actions of these participants are reciprocal[1] and responsive[2].

Yacci examined *interactivity* in the context of distance learning and computer-based teaching. He identified primary interactivity attributes [Yac00]. First, interactivity is a message loop whose messages must be mutually coherent. Second, the student's perspective is part of instructional interactivity with learning and affective benefits as outputs.

4.2 Continuous Interactive Learning

Figure 4.2 shows the idea of continuous interactive learning[3] that we adapted from Scrum [Sch95] and experiential learning [Kol84]. The course syllabus consists of high-level learning goals typically structured into lectures giving them meaningful boundaries

[1] Reciprocal means that actions of one participant trigger responses from the other and lead to change in the first.

[2] Responsive means that actions and reactions are related and sustain the continuity of the interaction.

[3] We started to integrate continuous interactive learning into courses in 2016 [KRTB16].

in the learning activities. Each lecture consists of more detailed learning goals based on the students' competencies following constructive alignment. The instructor teaches each learning goal in a learning sprint, a cycle that starts with theory and examples.

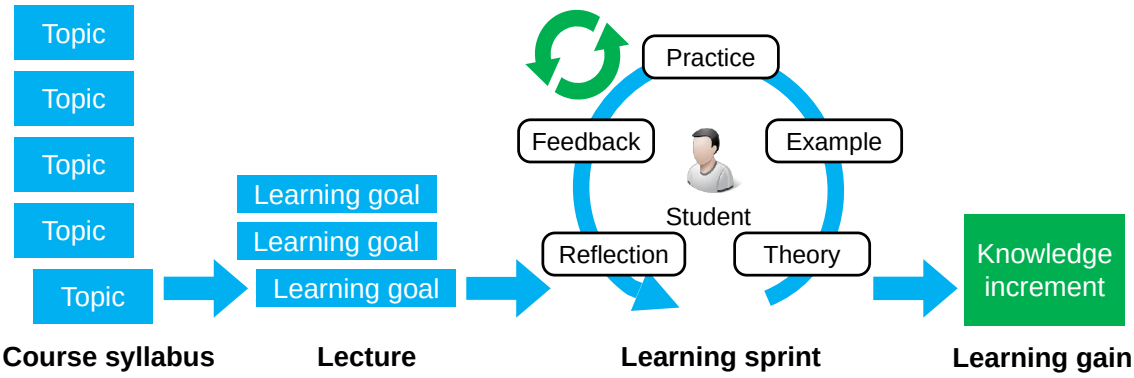


Figure 4.2: Continuous interactive learning embedded into a course consisting of lectures, each with a number of learning goals. Each learning goal is taught in a learning sprint through theory, example, exercise, feedback and reflection and leads to a new knowledge increment (adapted from Scrum [Sch95] and experiential learning [Kol84]).

Students then work on an exercise and receive immediate feedback building a second small cycle (green in Figure 4.2) that allows them to improve their solution to the exercise iteratively. After the exercise, the instructor stimulates reflection to relate their experience in the exercise with the taught theory. This reflection phase closes the cycle of the learning sprint and leads to a learning gain, which we call knowledge increment, concerning the taught learning goal. This knowledge increment is comparable to the potentially shippable product increment in Scrum.

Depending on the complexity of the concepts behind the learning goal and the accompanying exercises, a learning sprint can be between 20 minutes and 120 minutes. Figure 4.3 shows an example of a lecture with three learning sprints and breaks in-between.

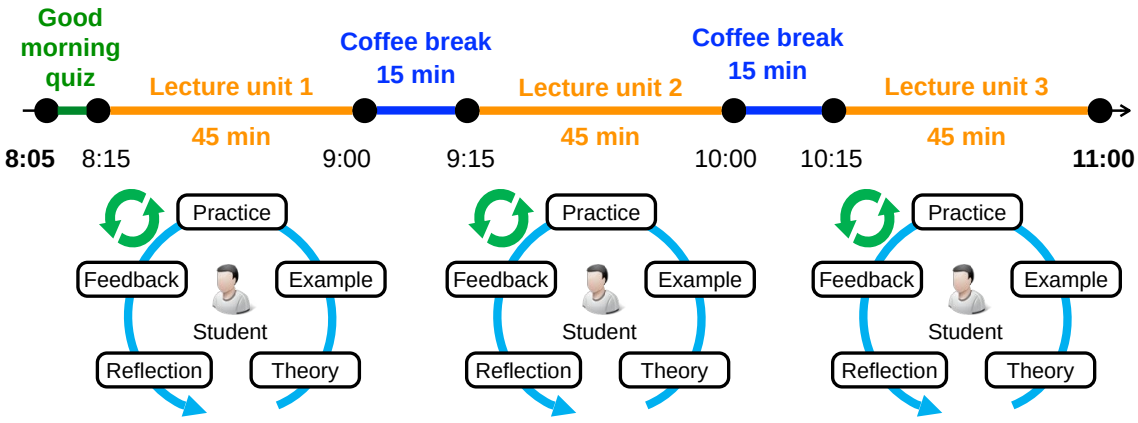


Figure 4.3: Example of a lecture with three learning sprints and breaks in-between. The lecture starts early in the morning and includes a good morning quiz about the lecture content of the previous lecture so that the students get started exciting.

Figure 4.4 shows another lecture example with four learning sprints, two before and two after the lunch break. Each lecture unit covers one learning goal (competency) and includes practicing the newly learned knowledge.

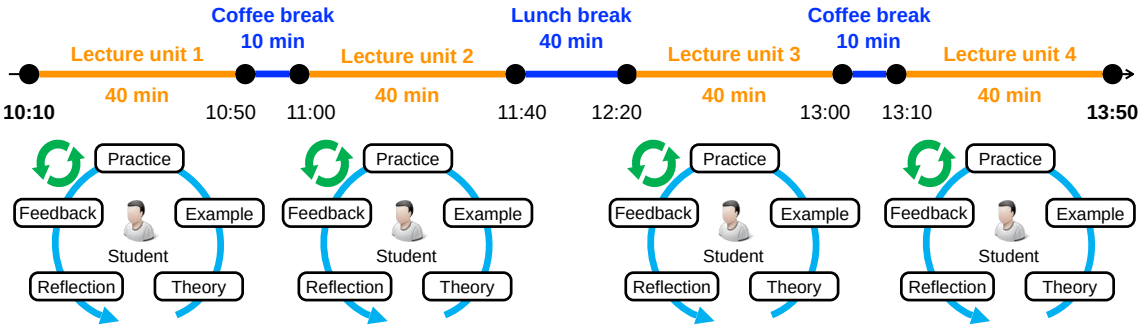


Figure 4.4: Example of a lecture with four learning sprints and breaks in-between. The lecture has two learning sprints before the lunch break and two learning sprints after the lunch break.

Examples and exercises are essential elements and play a central role in the early phases of cognitive skill acquisition [Van96]. Carefully developed and integrated examples increase the learning outcome [SC85, TR93]. Dynamic exercises with context-sensitive feedback enable a richer learning experience. Continuous interactive learning focuses on applying knowledge in various exercise types, e.g., programming and modeling with instant feedback. This approach supports the cognitive skill acquisition [Van96] on all levels of Bloom's taxonomy shown in Figure 4.5.

Multiple-choice quizzes focus on the first two levels. Programming and modeling address the four higher and more complex levels[4]. Instant and context-sensitive feedback at the end of the learning sprint provides guidance. If feedback can be generated automatically (e.g., through test cases in programming exercises) or by other students (e.g., through peer review in modeling exercises), it is scalable to many students. This approach requires higher effort during the exercise creation but reduces the assessment effort.

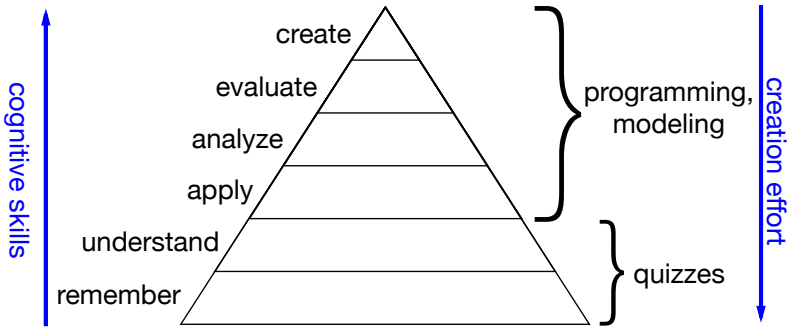


Figure 4.5: Mapping of exercises to cognitive skills

Depending on the type of the exercise, the student's submission is automatically assessed, or a manual review of the solution is carried out, potentially involving other

[4]While it might be possible to create multiple-choice tests for higher cognitive skills, it is difficult and does not reflect software engineering working practices: software engineers do not answer multiple-choice questions in their daily work when applying, analyzing, evaluating or creating something. Williams and Haladyna recommend to limit multiple-choice tests to lower cognitive skills [WH82].

students (e.g., in peer reviews). The assessment leads to manual or automatic feedback, which needs to be context-sensitive to be meaningful. Students can use it to improve and submit another solution. Feedback motivates students and allows them to reflect on their learning progress.

We developed the concept of interactive instructions that visually explain the problem to be solved. Such instructions are dynamic and provide continuous and granular feedback with self-updating elements, e.g., tasks and UML diagrams concerning the structure of the exercise. In addition, these elements respond to the interaction of students by changing their color from red to green to indicate that the solution is correct, as shown in Figure 4.6.

An interactive task is dynamically updated based on the student's progress. Tasks are associated with the assessment, e.g., a test or a peer review. For example, an interactive task in a programming exercise is completed when all associated tests are passing. This association allows referring the student to the problem in the source code when the user clicks on the unfulfilled, red task. After completion, the task is displayed in green and ticked off.

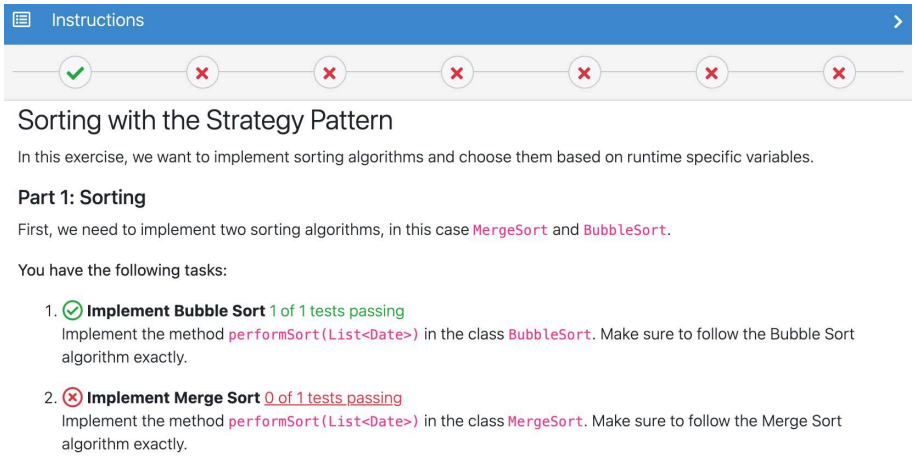


Figure 4.6: Interactive tasks provide immediate feedback to students about the correctness (red, green) of their solution

As shown in Figure 4.7, an interactive diagram is dynamically created and updated based on the student's progress. A UML class diagram consists of multiple elements, such as classes, attributes, or methods. A diagram element can be associated with an assessment and a source file. The implementation of a method is, e.g., associated with its method name in the class diagram. Based on the test results, the color of this diagram element changes to green if all associated tests succeed or to red if at least one test fails. Thus, students can immediately identify which parts of their exercise are correctly or incorrectly solved. In addition, the associated feedback includes context-sensitive information, why a test failed, and refers to the theory learned in lectures, slides, and handouts.

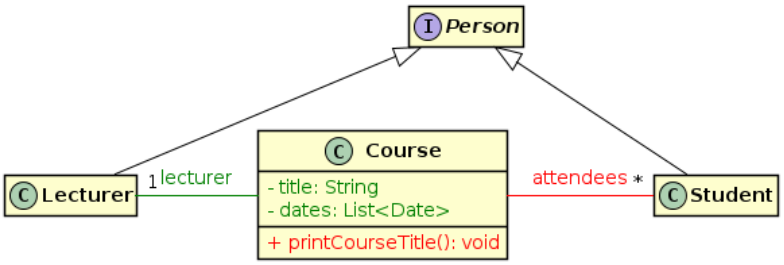


Figure 4.7: Interactive UML diagrams provide immediate feedback to students about the correctness (red, green) of their solution

4.3 Examples

Interactive exercises in lectures can have different types. Simple examples of four **individual** exercises (1-4) and one **team-based** exercise (5) are:

1. A quiz with a couple of multiple-choice questions
2. Interactive tutorials with step by step instructions, e.g., the introduction of a new technique that the instructor demonstrates during the lecture with a simple example
3. Interactive coding challenges to solve programming assignments, e.g., a new programming language concept demonstrated in the integrated development environment
4. An open-ended text exercise in which students explain the similarities and differences of two related techniques based on examples
5. Project work including communication and collaboration aspects, e.g., an exercise in a team project for a given problem statement where students have to write down the functional requirements

Instructors can also design more complex exercises with related parts that span multiple learning goals or even multiple lectures. One example is software theater, a technique taught in project-based courses. It outlines an approach to present visionary scenarios using techniques borrowed from theater and film, including props and humor. We describe the whole idea of software theater in detail in the journal article [KDXB18]. It involves a more engaging, dynamic way of presenting software prototypes to demonstrate the context of the software usage.

Software theater consists of several steps integrated into a lecture following interactive learning or solved as homework. The process starts with a visionary scenario that the customer typically creates. The development team is then responsible for the following steps:

1. Formalize the scenario
2. Create a demo backlog
3. Write a screenplay
4. Select the participating subsystems

5. Identify the participating methods
6. Identify the participating objects
7. Create action items
8. If the demo is too ambitious and not realizable, modify the screenplay and continue in step 4
9. Realize (i.e., implement) the demo
10. Present the demo
11. Incorporate the feedback
12. If needed, modify the scenario and start a second iteration in step 1

Each step can be a lecture exercise or homework (e.g., between two lectures). The whole software theater workflow is integrated into a team project. It is performed towards the end of the project when the students have already implemented parts of the software. It typically involves two lectures with examples and feedback.

Chapter 5

Artemis

> "Learning is more than absorbing facts, it is acquiring understanding."
>
> — William Arthur Ward

Artemis is a teaching platform that supports interactive learning and is scalable to large courses with immediate and individual feedback. It is open-source[1] and used by multiple universities and courses.

5.1 Functionalities

Artemis includes several functionalities to implement interactive learning. In the following section, we present and discuss the essential features. Instructors can create different **exercises**: programming, modeling, quiz, text, and file upload. Artemis offers different **assessment** modes: automatic, semi-automatic, and manual. It automatically assesses programming and quiz exercises and provides a semi-automatic assessment approach based on machine learning for modeling and text exercises.

Artemis also integrates team-based learning by offering **team exercises** to work collaboratively on the solution to the given tasks. In addition, instructors can incorporate live streams, recordings, and slides of **lectures** and embed exercises directly into them using lecture units. Artemis also offers an **exam mode** for online exams. The exam mode includes additional instructor functionalities, such as exercise variants, plagiarism checks, and offline support.

5.1.1 Programming Exercises

Artemis implements programming exercises using version control and continuous integration. It uses generic interfaces to connect to an existing version control server such

[1] https://github.com/ls1intum/Artemis

as Bitbucket Server[2] and Gitlab[3] and to connect to an existing continuous integration server such as Bamboo[4] and Jenkins[5]. Test cases and static code analysis assess the student submissions automatically. This approach allows providing feedback to students in real-time in interactive instructions that change their status and color based on students' progress. Completed tasks and correctly implemented model elements are marked in green. Incomplete and not yet implemented ones are marked in red. This highlighting helps students to identify which parts of the exercise they have already solved correctly and improves the understanding of the source code on the model level. When they submit their current solution, the interactive instructions dynamically update.

The programming exercise workflow is shown in a simplified dynamic model in Figure 5.1 and works as follows: An instructor sets up a version control repository containing the exercise code (template) handed out to students and test cases to verify students' submissions (*template repository*). It can include a small sample project with predefined classes and dependencies to external libraries. The instructor stores the tests for the auto-grading functionality in a separate *test repository*, which is not accessible to students. A combination of behavioral (black-box), structural (white-box) tests, and static code analysis allows checking for functionality, implementation details, and code quality of the submitted code.

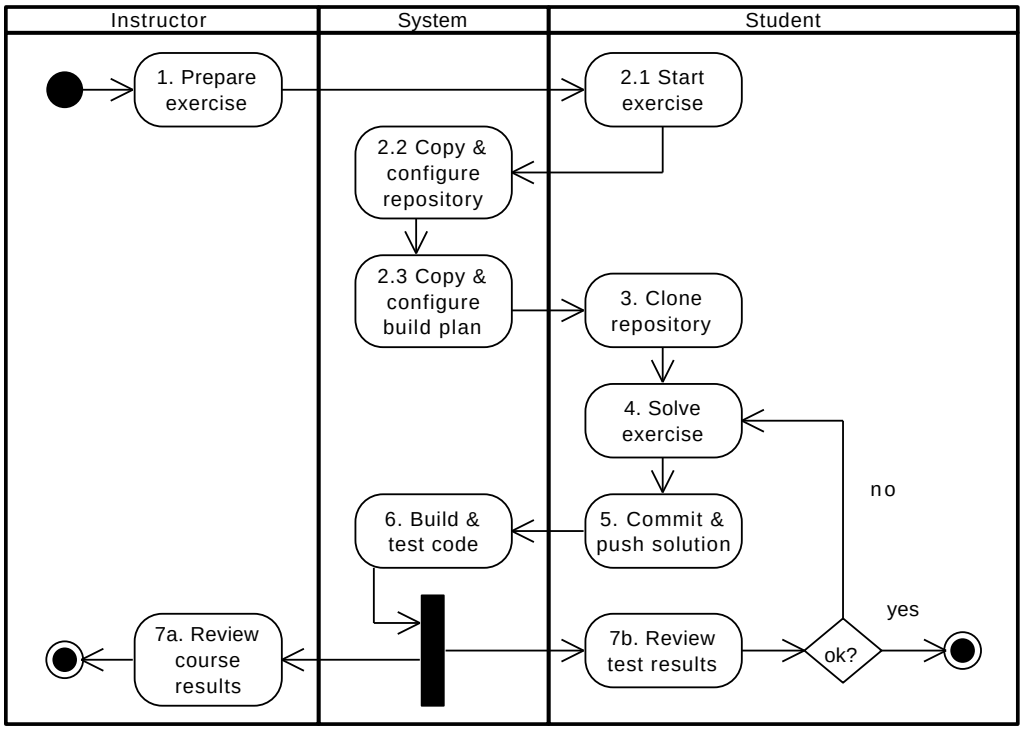


Figure 5.1: Simplified process for conducting programming exercises with Artemis as a UML activity diagram [KS18]. The instructor creates a programming exercise. Then, the students try to solve the exercise and receive automatically generated feedback that helps them improve.

[2] https://www.atlassian.com/software/bitbucket
[3] https://about.gitlab.com
[4] https://www.atlassian.com/software/bamboo
[5] https://www.jenkins.io

After setting up the template, test, and solution repositories, the instructor configures the build plan on the continuous integration server. A build plan compiles and tests the exercise code using the previously defined test cases and the static code analysis configuration (*template build plan*). The build plan includes a task to pull the source code from the template repository and the test repository whenever changes occur. It combines both repositories and executes the compilation and tests in the second step. A final task notifies Artemis about the new result.

A student starts an exercise with a single click, triggering the setup process: Artemis creates a personal copy of the template repository (*student repository*) and grants access only to this student. It also creates a personal copy of the template build plan (*student build plan*) and configures it to be triggered when the particular student uploads changes to this personal student repository. The students cannot access the build plan because it is not required for their participation, so Artemis hides the complexity of continuous integration from students. Personalized means that each student gets one repository and one build plan. When 2,000 students participate in an exercise, Artemis creates 2,000 student repositories and 2,000 student build plans. Students only have access to their repository. They cannot access other student repositories.

After the setup is complete, Artemis allows the student to work in a local IDE or open the online editor's exercise. When the student submits a new solution to the personalized repository, the build plan compiles the code and executes the tests defined by the instructor in a docker container. It uploads the results to Artemis in a few seconds so that the students can immediately review the feedback and iteratively improve their solution. In case of an incorrect solution, the feedback shows a failure message for each failed test. The student can reattempt to solve the exercise and submit a new solution. The instructor can review results, gain insights on the progress, and react immediately to errors and problems. Instructors can add a manual review step. Reviewers can then provide additional manual feedback (activity left out for simplicity in Figure 5.1).

Artemis includes an online editor that allows inexperienced students to participate in exercises without dealing with the complex setup of version control and integrated development environments. In addition, it supports the manual review of student submissions after the due date Tutors can see the automatic feedback through tests and static code analysis and enhance them with manual feedback. This manual review phase makes it possible to automatically review aspects that are difficult to assess, e.g., the internal structure and specific code quality issues.

Figure 5.2 visualizes how students participate in a programming exercise and receive code quality feedback for their submissions. The actors in this process are the student and Artemis. The student starts a programming exercise, which prompts Artemis to set up the participation. Artemis copies the repository and the build plan of the template participation to create a personal build plan and personal repository for the student. The student accesses the code skeleton of the assignment by cloning this repository. The option to access the personal repository using the online code editor is not depicted in the diagram for simplicity. The student writes and submits code to solve the exercise. The repository stores the files and initiates the creation of a submission. Simultaneously, the update triggers the execution of the build plan.

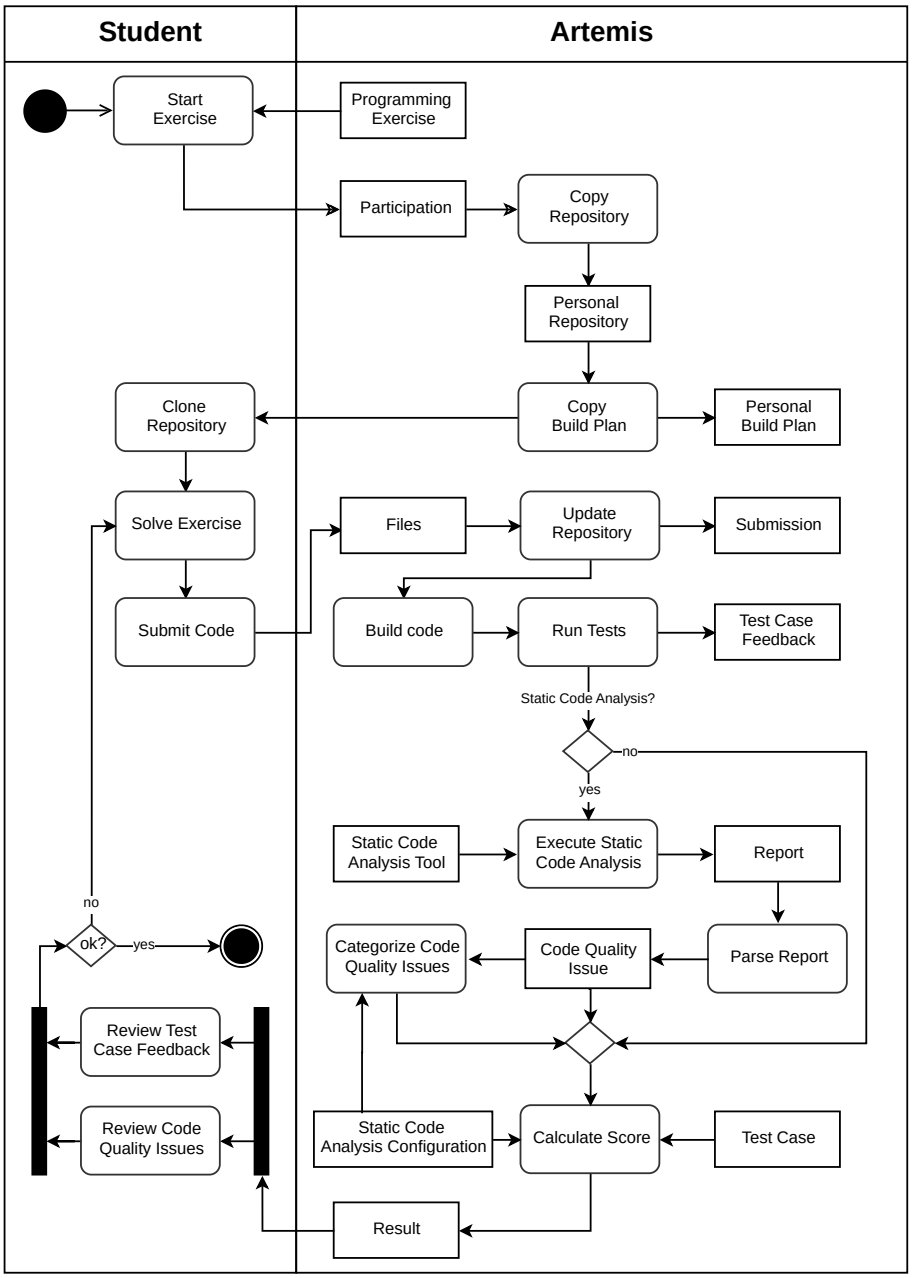


Figure 5.2: The UML activity diagram shows the interactive feedback loop during a programming exercise participation. Based on the configuration of tests and static code analysis, it grades the submission. It grades the submission according to the test case and static code analysis configuration. Students explore the feedback to improve their submissions and programming skills iteratively.

The build plan compiles the code. Then, it executes test cases to check the functionality of the resulting program. The build plan can also run static code analysis tools to generate reports, which contain information about code quality issues. Artemis groups these issues into categories according to the static code analysis configuration made by the instructor, who can follow a default configuration. Artemis calculates the scores for the test cases, deducts the penalties for code quality issues as defined in the static code analysis configuration, and presents the result to the student. The student verifies that the submission passes the test cases and can inspect the identified code quality issues.

The student stops working on the exercise if the result is satisfying. Otherwise, the student attempts to solve the exercise again. This time the student benefits from feedback that helps to improve the code and learn from mistakes. Thus, Artemis establishes a feedback loop that continuously teaches students to write high-quality functional code.

5.1.2 Modeling Exercises

Artemis integrates an online modeling editor Apollon that is open-source[6] and available as a standalone and free web application[7]. Apollon supports seven UML diagrams: class diagrams, object diagrams, activity diagrams, use case diagrams, communication diagrams, component diagrams, and deployment diagrams. It also supports three additional diagram types: Petri nets, syntax trees, and flowcharts. Apollon is lightweight and easy to use to lower the entrance barrier of digital modeling. It focuses on the learning experience of students. Figure 5.3 shows an example of a UML communication diagram. Students drag the model elements from the right into the diagram and double-click on an element to edit it in a small popup. They can create relationships (e.g., control flow) between model elements.

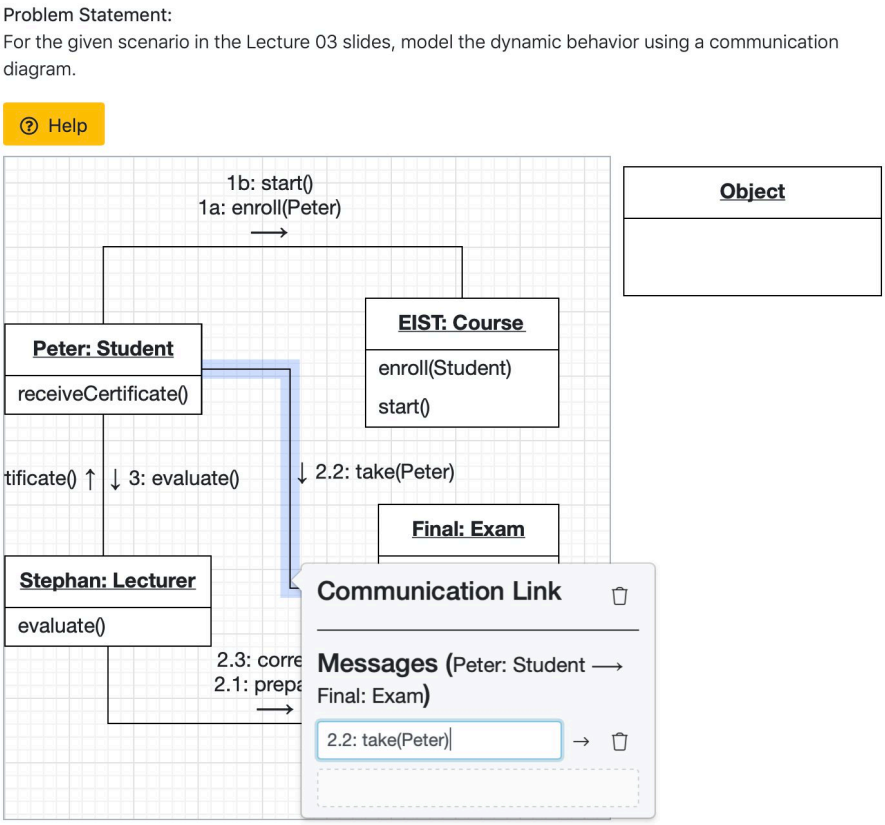


Figure 5.3: The online modeling editor Apollon is integrated into Artemis and supports the easy creation and assessment of digital models.

[6] https://github.com/ls1intum/Apollon
[7] https://apollon.ase.in.tum.de

Instructors and tutors provide feedback directly in Apollon. They double click on a model element and assess it in a popup with a score in points and with additional feedback comments to explain why a model element is correct or wrong. In addition, they can provide general feedback about the whole model or missing elements. Students can see this feedback directly in place next to the model elements and learn from it. Section 5.1.5 explains more details about assessing submissions to modeling exercises.

Apollon is also available as a standalone version on https://apollon.ase.in.tum.de for free and without the need for an account. It should be as easy to use as possible and reduce the barrier to get started with modeling. It omits complex menus and user interfaces which make modeling complex. Beginners, who learn to model, e.g., as part of their university program, should be able to use Apollon without detailed instructions or user manuals. Figure 5.4 shows that students can select predefined patterns which are then editable.

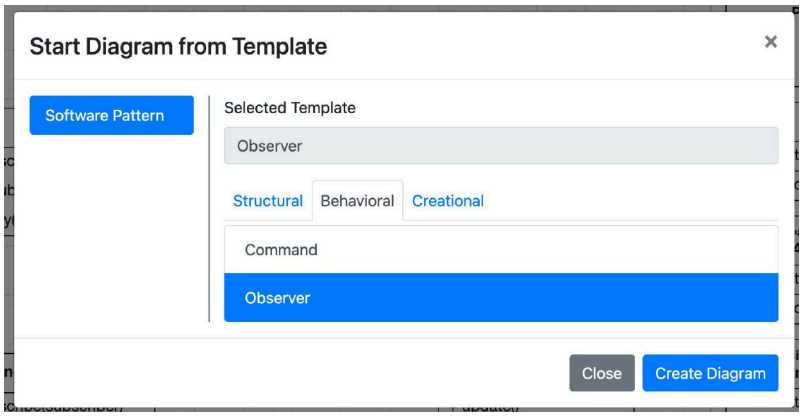


Figure 5.4: Users can choose predefined patterns as templates to get started. They can adapt the created UML diagrams (based on the chosen pattern) to their own needs.

Students can adapt the created UML diagrams for the chosen patterns to their own needs and integrate them into the problem they are working on. Figure 5.5 shows an example of the observer pattern in Apollon.

Apollon allows working on models collaboratively. Figure 5.6 shows how users can create a link to use the sharing functionality even without accounts. Users can create a sharing link and ask others to edit a copy of the diagram. Students can use the sharing functionality to ask tutors for feedback. After tutors have entered the feedback, they can share the diagram again with the students. Another possibility is to work with multiple users on the model simultaneously using the real-time collaboration mode.

5.1.3 Quiz Exercises

Artemis allows instructors to compose quizzes with different question types:

1. Short-answer question (cloze)
2. Drag and drop
3. Multiple-choice

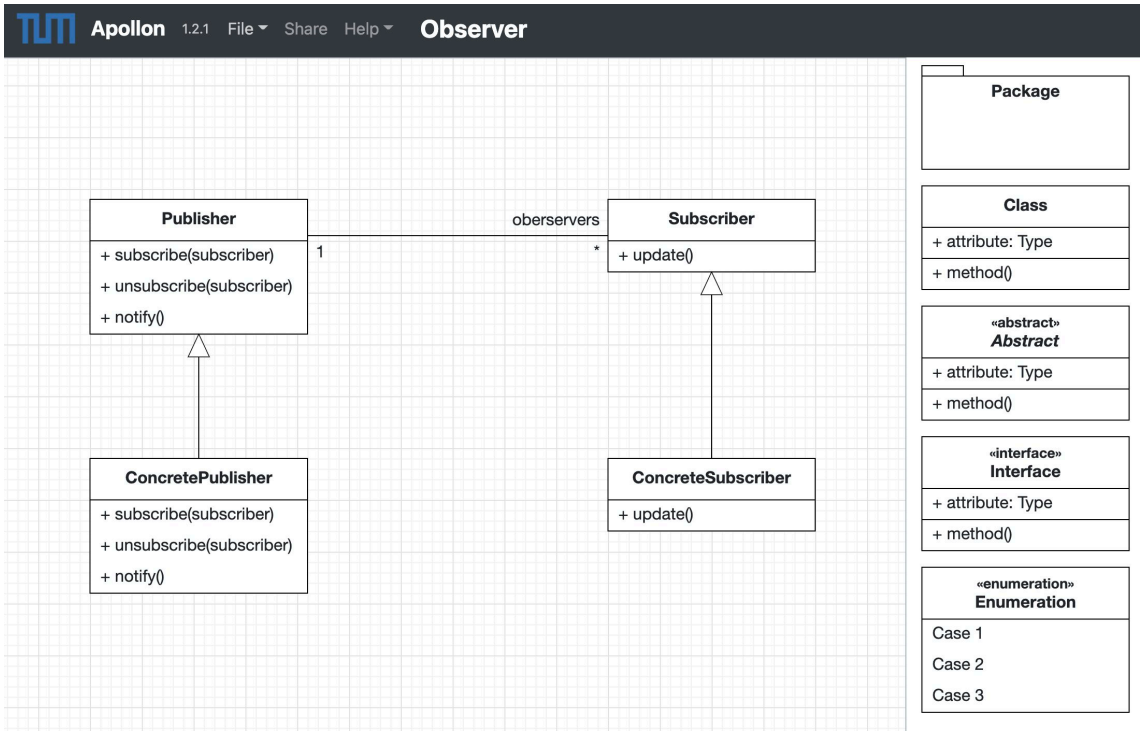


Figure 5.5: The Apollon standalone modeling editor should be easy to use and lightweight. Users can start directly without the need to create an account or log in.

Figure 5.6: Apollon allows to share models based on a link. Users can choose whether the receiver can edit a copy of the model, provide feedback or see the inserted feedback.

Figure 5.7 shows an example of a short-answer question. Students need to fill the correct text into the text boxes. Instructors can add multiple correct options for the same text box and can determine whether typos or capitalization are considered in the evaluation of correct answers.

Figure 5.8 shows an example of a drag and drop question. Instructors can easily create the drop areas interactively. They can map the drag items (either text or an image) to the drop areas. Artemis verifies that instructors do not enter an illegal mapping by accident, making the quiz unsolvable for a student.

Artemis allows instructors to create modeling diagrams with Apollon, select the elements for a drag and drop question, and automatically create a modeling drag and drop exercise. This reduces the time to create such quizzes significantly and also allows modifying the quiz question more easily later on, e.g., in case the quiz question will be reused in an exam.

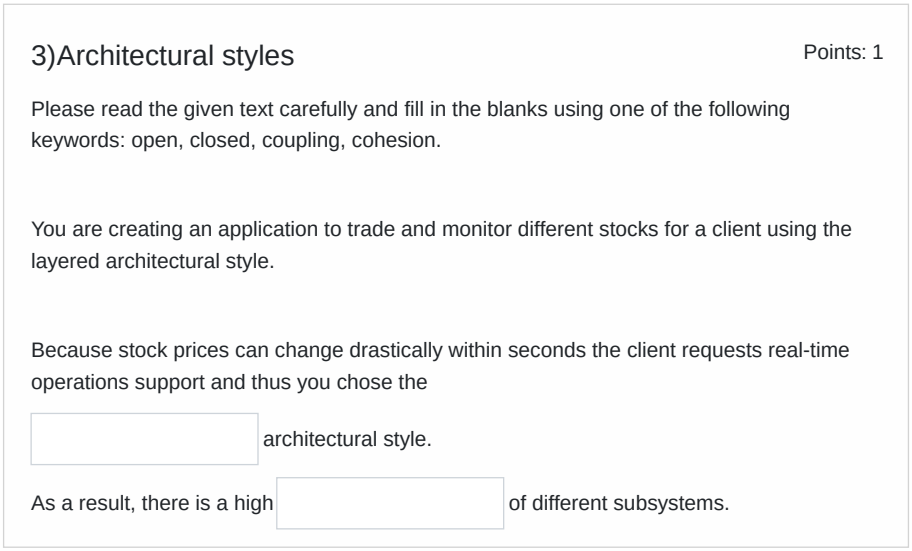


Figure 5.7: Example of a short-answer question in Artemis. Students need to fill the correct text into the text boxes.

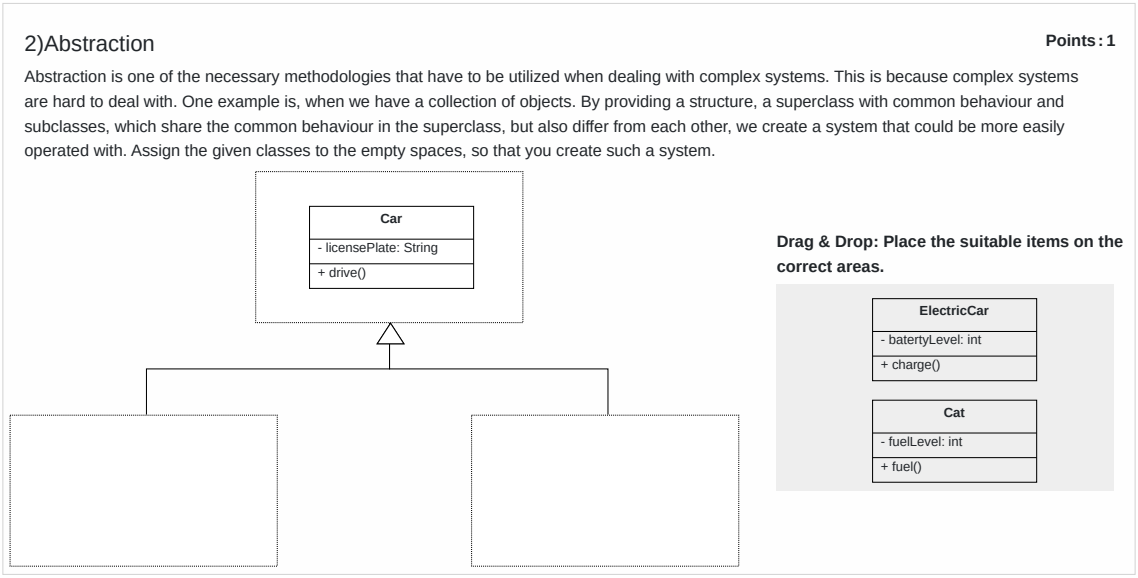


Figure 5.8: Example of a drag and drop question in Artemis. Students need to drag the text or image items on the right to the correct areas on the left.

Figure 5.9 shows an example of a multiple-choice question. Instructors can define the question and answer options using markdown, allowing basic formatting (e.g., bold, italics, coloring).

They can provide hints and explanations why answer options are correct or wrong. Hints are shown during the quiz, explanations only after the quiz has finished. Students need to select the correct answers. Incorrect answers must not be selected.

Instructors can define how many points each question is worth and how long the whole quiz should take. They can integrate live quizzes into their lecture, which are typically between five and ten minutes long. All students answer live quizzes simultaneously. At

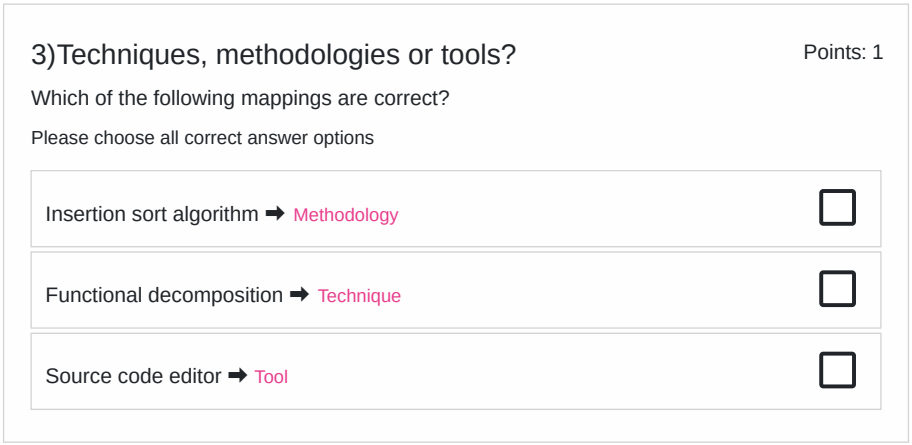


Figure 5.9: Example of a multiple-choice question in Artemis. Students need to select the correct answers. Incorrect answers must not be selected.

the end of the quiz, Artemis evaluates the results automatically and displays them to the students to provide immediate feedback. Instructors can add hints and explanations and decide if they would like to randomly show questions and answer options to make it more difficult for students to work collaboratively. Artemis offers three scoring strategies:

1. **All or nothing**: Students get all points only if they have correctly selected all options (or filled all answers). In case they have one mistake, they will get 0 points. There is no score in between.
2. **Proportional with penalty**: Every correct answer gives a fraction of points. Each mistake subtracts the same fraction of points to avoid guesswork. For example: if the score of the question is three and there are five options. Each correctly selected answer option results in 0.6 points. Each wrongly selected answer option subtracts 0.6 points. A student with three correctly and two wrongly selected answers would then receive 0.6 points.
3. **Proportional without penalty**: Every correct answer gives a fraction of points. No points are deducted for mistakes. For example: if the score of the question is three and there are five options. Each correct answer option results in 0.6 points. A student with three correct and two wrong answers would then receive 1.8 points.

After the live mode has ended, students can practice the quiz as often as they want in the practice mode. This is particularly helpful for reviewing later (e.g., shortly before the final exam) whether they have understood the concepts correctly. Results of the practice mode are not included in the overall score calculation of the course.

5.1.4 Text Exercises

To work on text exercises, Artemis provides an integrated text editor shown in Figure 5.10. When creating text exercises, instructors define the problem statement and configure how many points the exercise should give, which is weight-related to all other exercises in the same course or exam. In addition, instructors define whether the text exercises are included completely, as a bonus, or not in the overall score calculation.

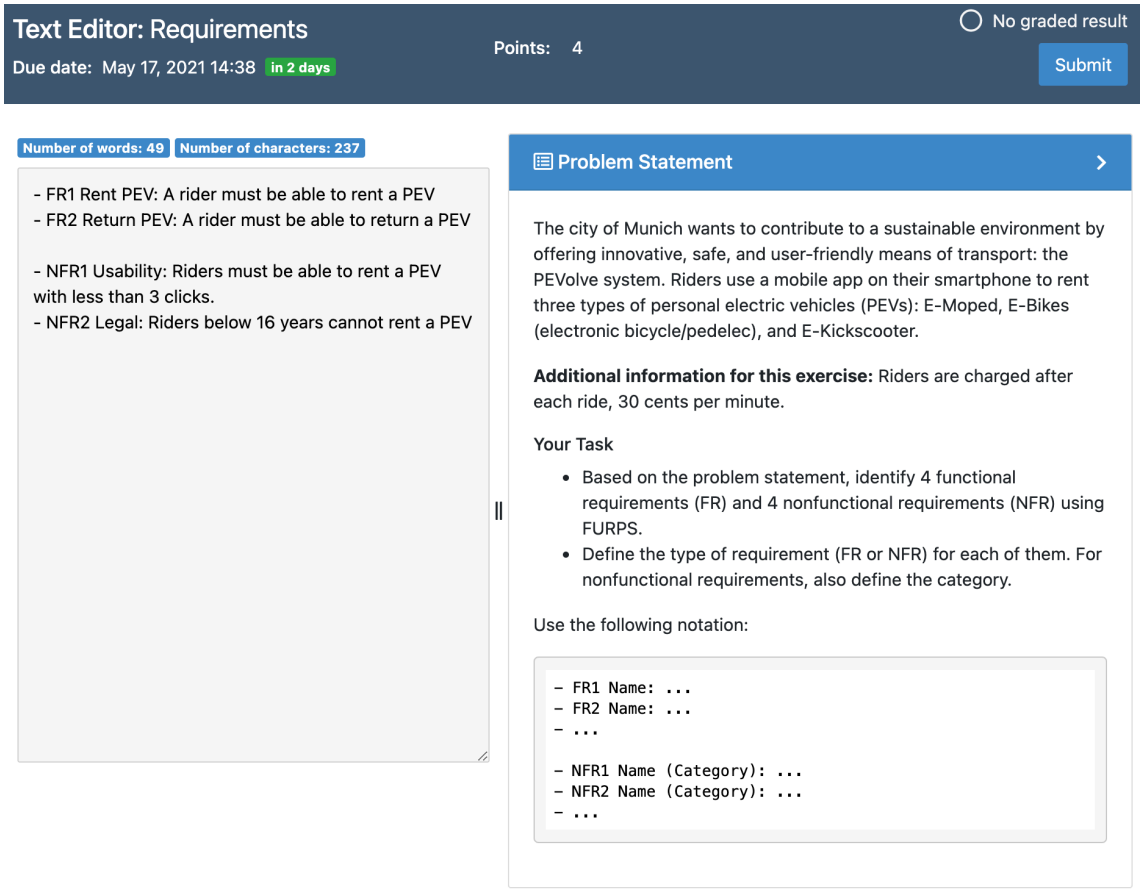


Figure 5.10: Integrated text editor in Artemis. Students can read the problem statement on the right side and answer the question on the left side.

Instructors can also create an example solution that shows one possible correct answer. Artemis specifically does not call this a "sample solution" to avoid the misunderstanding that only one solution can be correct. Instructors can also create example submissions and assessments used for the reviewer training as soon as the manual or semi-automatic assessment starts.

After the release of the exercise, students can work on it. The text editor is simple to use and easy to navigate. There is no limit to the length of the submission. The text editor automatically tracks the number of words and characters in the students' submissions to indicate the length of the answer to the students. Students can read the problem statement based on markdown formatting on the right side and answer the question on the left side. In case they need more space, they can collapse the problem statement.

Students cannot format the text using markdown to make it easier to read and parse it for the semi-automatic assessment approach. Artemis allows students to submit the text answer as often as they want until the due date In addition, Artemis saves the text regularly to avoid getting lost when students navigate to a different page before pressing the submit button.

Artemis automatically detects the language (e.g., German or English) of the answer. This makes sure that reviewers can understand the answer and assess it accordingly.

5.1.5 Assessment

Artemis supports different assessment strategies. It automatically assesses programming exercises based on tests and static code analysis rules. Instructors can define the weight of individual tests to increase or decrease their influence on the score. Artemis supports hidden tests that are not visible to students. Instructors can include or exclude specific predefined static code analysis rules from a default configuration. They can also determine how many points the system should deduct if students do not fulfill specific rules. Examples of static code analysis rules include coding style, class design, code smells, duplicate code, security issues, size metrics (e.g., length of a method), and formatting. Artemis allows instructors to add a manual assessment step after the automatic assessment to provide additional feedback, e.g., related to implementation specifics and aspects that cannot easily be assessed automatically by dynamic tests or static code analysis.

Quiz exercises are also assessed automatically based on the scoring strategy. Instructors can re-evaluate quizzes, e.g., adding additional text options to short-answer quizzes that they have not thought of when creating the quiz. In addition, instructors can set questions or answer options to invalid if they have not been clearly formulated. Invalid questions are counted as correctly answer questions. Invalid answer options are also counted as students would have chosen the correct answer.

Artemis supports semi-automatic assessment based on supervised machine learning for modeling and text exercises. During the manual assessment of reviewers, Artemis learns which aspects are correct and which are wrong. Based on similarity analysis, Artemis can then propose feedback for subsequent submissions.

The manual assessment mode of Artemis is tailored to the corresponding exercise type. Reviewers select text segments and provide feedback for text exercises. For modeling exercises, reviewers select modeling elements (e.g., attributes, methods, associations) and provide feedback. In programming exercises, reviewers can provide feedback to single lines of code or the whole file. Artemis includes reviewer training to improve the consistency and fairness of grading in larger courses with many reviewers.

In addition, it implements double-blind reviews so that students do not know the identity of the reviewer and reviewers do not know the student's identity. This prevents unconscious biases based on personal relationships or other aspects. Structured grading instructions (comparable to grading rubrics) simplify the grading process. They include predefined feedback and points. Reviewers can drag and drop the correct instruction to the text, model element, or source code line to apply it. Students can rate the quality of the feedback. Together with a reviewer leaderboard, this motivates the reviewers to provide high-quality feedback that helps the students improve their understanding and prevent misconceptions.

In the following, we describe the **semi-automatic assessment** of modeling exercises in more detail. The semi-automatic assessment for text exercises is comparable but uses different internal algorithms based on natural language processing. Those are explained in more detail in [BKKB21, BKB21].

The Unified Modeling Language (UML) is the standard for visually describing software systems [RJB99]. It consists of different diagram types, e.g., UML class diagrams to describe the system's structure. A model "captures the important aspects of the thing being modeled from a certain point of view" [RJB99, p. 13]. Thus, it is an abstraction and generalization and omits specific details. This makes models useful for software engineering because both handle the complexity by focusing on relevant details and leaving out the rest.

Teaching modeling to students is a difficult task. While the structural aspects follow rules that can be learned by heart, it is challenging for students to map a realistic problem statement to a concrete model. Including all semantics correctly is challenging and sometimes not even possible due to abstraction and simplification. Students have to decide on inevitable trade-offs when modeling a problem that includes higher-level skills.

Bloom's taxonomy classifies learning goals [BEF+56]. Modeling requires understanding and analyzing a problem, evaluating different design solutions, and combining them to create new knowledge. Designing a system and modeling it in UML allow for multiple ways how to solve the problem. The freedom in creating UML models has an impact on the assessment of modeling exercises.

When exercising modeling, multiple solutions can be correct and can show essentials aspects of a problem. There is a risk in teaching that instructors focus on one particular sample solution and that students misunderstand that only this solution is correct and all other solutions are wrong. However, the nature of modeling is abstraction by hiding complex details. It is not always easy to decide which intricate details should be hidden in a model and which aspects need to be shown to understand the core idea of the problem. Even experienced modelers will develop different solutions, which is one advantage of modeling because it stimulates discussion among the system and facilitates communication. However, this is also a challenge in teaching, especially if instructors do not have enough time to teach students these aspects of modeling, e.g., undergraduate courses.

One sample solution cannot cover all relevant aspects of a problem and can constraint the creativity of students. While it allows comparing, it is not easy to decide if a solution is still partly correct if it differs from the sample solution. Furthermore, showing a sample solution poses the risk that students learn one out of many representations of a system by heart instead of using their creativity to develop their representation. Instead, students should learn to address a problem with the help of modeling by applying the skill on their own. Instructors then take over the task of providing an individual assessment for each student.

"Formative assessments provide an opportunity for learners to demonstrate their development toward the learning outcomes and provide constructive feedback to learners about their progress to ensure they can achieve the intended outcomes by the end of the course" [SH14, p. 54]. The assessment identifies gaps between the student's understanding and the actual knowledge. "Feedback is information about the gap between the actual level and the reference level of a system parameter which is used to alter the gap in some way" [Ram83, p. 4]. Individual feedback includes information on why a

particular aspect of the solution is incorrect and facilitates learning. Students can use the feedback to learn about their mistakes, improve their solutions and identify strengths and weaknesses.

When instructors assess UML models, they typically scan the submitted model for known model elements and groups of model elements and evaluate if they are correct in their structure, semantics, and visual aspects. The result of the evaluation is individual feedback for each student. In large courses, they face similar models frequently, leading to recurring manual tasks when identifying model elements and evaluating them.

Automating such recurring tasks can reduce the correction effort. It can preserve that multiple submissions are considered correct and would not limit the modeling creativity of students. To support this idea, Artemis needs to learn from manual model grading and apply the learned knowledge to correct other model submissions enabling a semi-automatic assessment approach as shown in Figure 5.11.

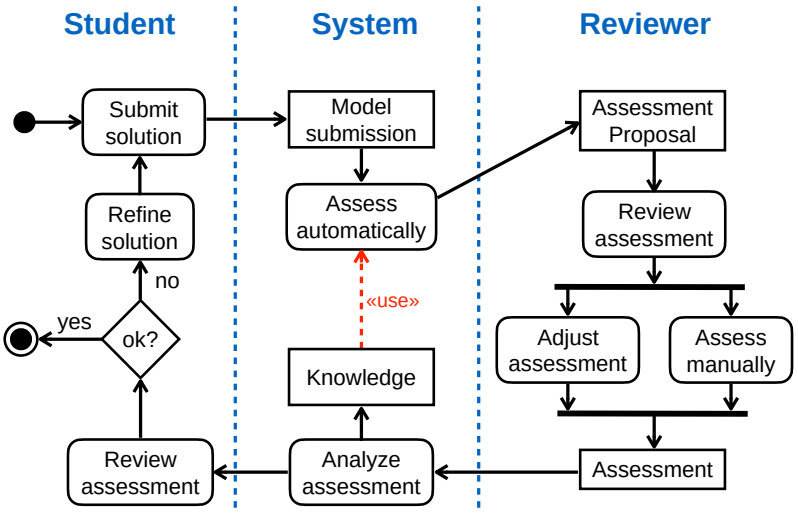


Figure 5.11: Semi-automatic assessment workflow with multiple submissions

Students submit their models as a solution to a specific exercise. In the beginning, there is not enough knowledge for the system to assess automatically so that the assessment proposal will be empty, and reviewers need to assess the complete solution manually. The system analyzes the manual assessment and generates knowledge that can be used to propose automatic assessments for subsequent submissions with similar model elements. Then reviewers start with a partly assessed submission, as shown in Figure 5.12. They can either confirm the proposed assessments if they are correct or adjust them if they are wrong for the current submission, e.g., due to a different context. Following an interactive learning approach, students can review the assessment and use it to refine their model and submit another version to the same exercise if the due date has not passed yet.

Two core concepts are necessary to enable the system to learn. First, the system needs to decompose models into smaller model elements. For example, Figure 5.13 shows the four model elements in a UML class diagram: classes, attributes, methods, and associations.

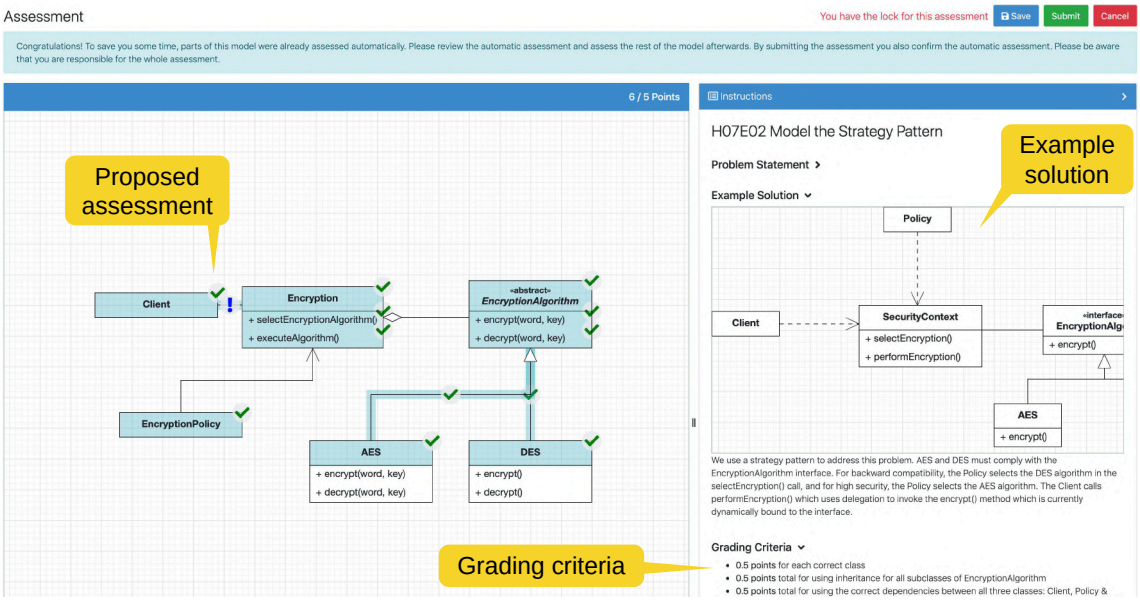


Figure 5.12: Assessment user interface in Artemis. The right side shows the instructions, including the problem statement (collapsed), one example solution, and the grading criteria. The left part is the assessment editor, which includes proposed assessments (highlighted in blue).

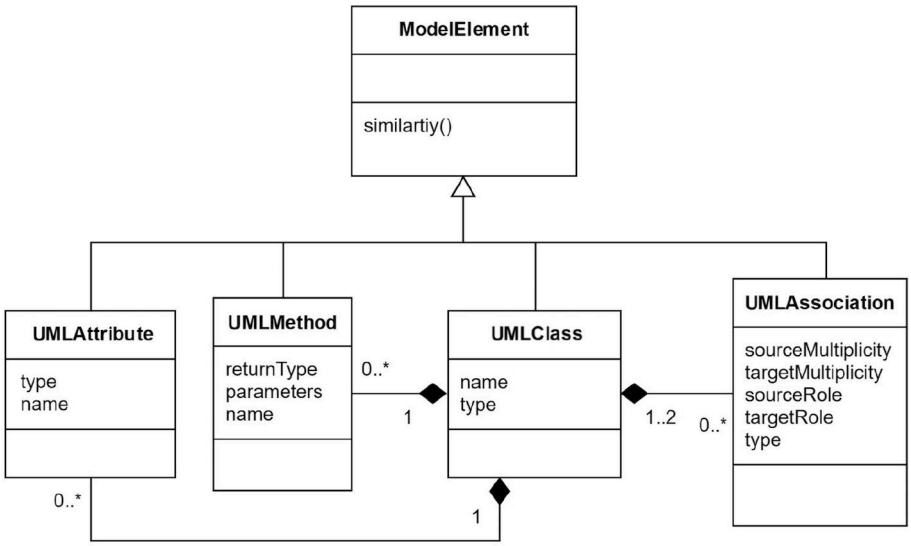


Figure 5.13: Taxonomy of model elements in UML class diagrams

Second, it needs to identify similar model elements in other submissions. Then, it can apply the existing assessment of one model element to other model elements in the same similarity set. Model elements are considered similarly based on their name and their context.

Methods and attributes belong to the same similarity set if specified in the same UML class and if they have a similar name. UML classes belong to the same similarity set if they have a similar name, a similar kind (e.g., abstract, interface), and similar associations to other classes. The last point includes the context of classes to make sure that classes are only identified as similar if defined in the same context. Associations

belong to the same similarity set if they have the same source and target class, the same kind, and similar multiplicities and roles with only minor deviations.

When a reviewer assesses one model element in a similarity set, the system learns whether the model element is correct or not and can apply the same score and feedback to all other model elements in the same set. Thus, each manual assessment creates additional knowledge in the knowledge repository, which can be used for automatic assessment proposals. The system can also suggest the following manual assessment, where the most knowledge is gained by choosing the submission in which the least amount of model elements would include an assessment proposal.

Figure 5.14 shows an example of how the system learns. The first two submissions are manually assessed and produce enough knowledge so that the system can propose assessments for all six model elements in the third submission. While it might be tempting to submit the assessment of the third submission automatically, it makes more sense that a reviewer inspects the model and takes the responsibility for the assessment. There are several reasons for this: First, the context and similarity detection are not always correct. Second, missing model elements cannot be identified with this approach. Third, even if the scores (in the example expressed with green ticks and red crosses) are proposed correctly, the textual feedback (not shown in Figure 5.14) might not fully apply.

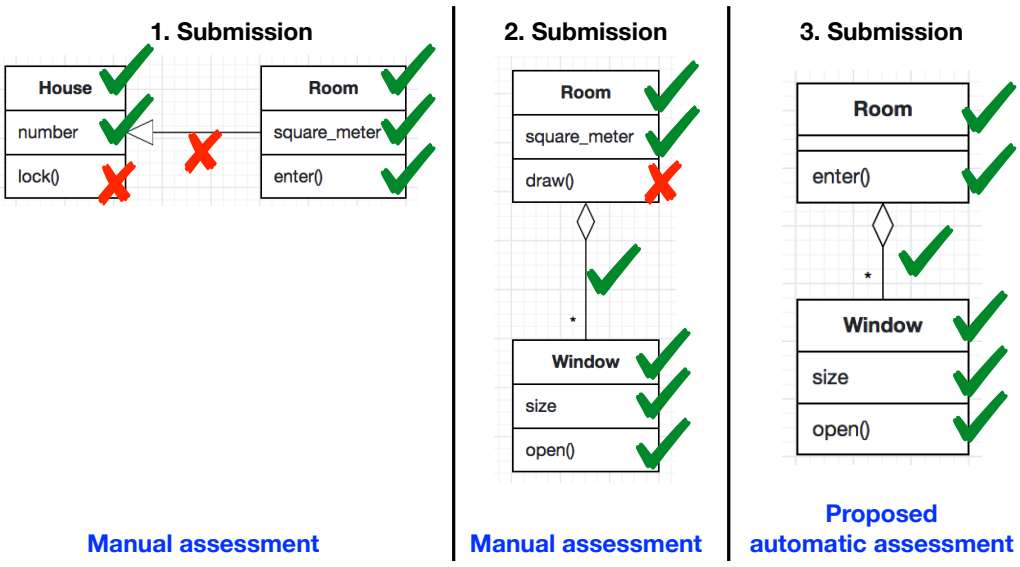


Figure 5.14: Example of how two manual assessments (left, middle) lead to knowledge for a proposed automatic assessment (right)

5.1.6 Team Exercises

Team exercises play an essential role in Artemis to support team-based learning and to facilitate soft skills. Artemis supports team-based learning for programming, modeling, text and file upload exercises. The activity diagram in Figure 5.15 shows the dynamic model of a team's participation in a programming exercise and highlights the different activities from start to finish.

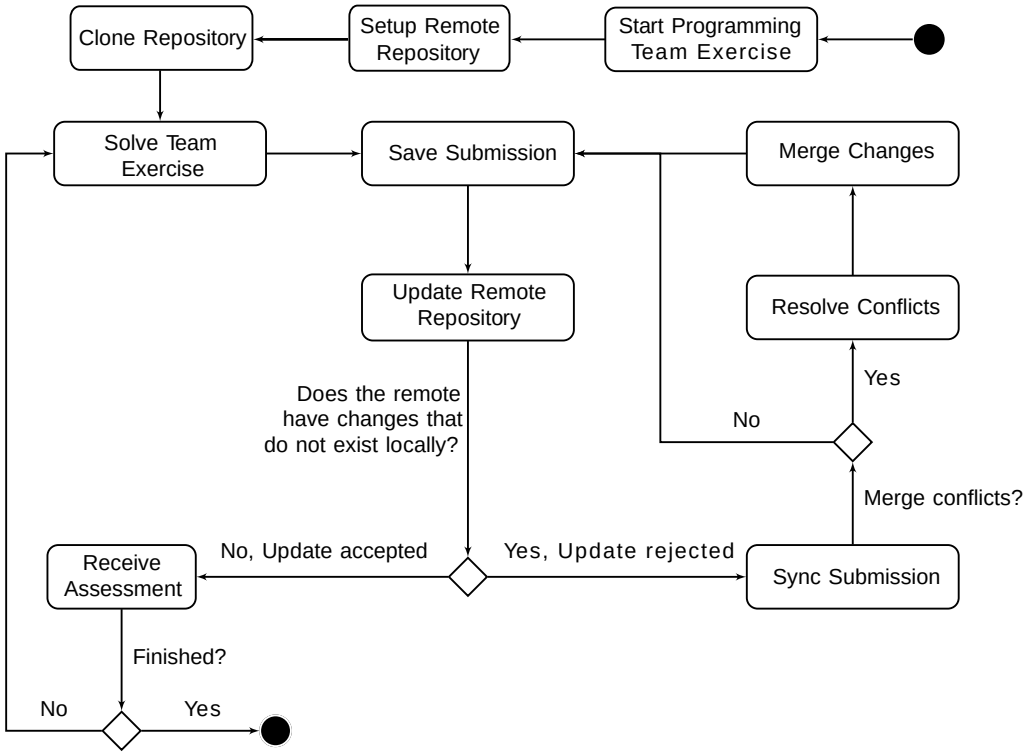


Figure 5.15: The UML activity diagram shows how students can work together on a team programming exercise in Artemis. One of the team members starts the exercise, which sets up a remote repository. All team members are then able to clone this repository and work on the exercise locally. When they save their changes to the remote repository, they might encounter a rejection, sync the submission state, and resolve the conflicts first.

This activity triggers the repository setup, which Artemis performs in the background. First, a repository will be created using the version control service. Next, the team members can copy the URL of the newly created repository to clone the repository to their local machine. Then, they can start working on the actual tasks of the programming exercise using their preferred IDE. Once they have worked on the exercise, they save their submission and push the changes to the remote repository.

There are two different scenarios now: In the first scenario, the other team members have not pushed any of their work to the shared remote repository yet. The consequence is that the push operation will be successful, and the student can continue writing code. In the second scenario, the contents of the remote repository have changed between cloning the repository and pushing the code. The remote repository will reject the push operation. This scenario is new in team-based programming exercises due to multiple students working on the same repository.

Technically, this case can also occur when students work alone, e.g., if they use multiple machines or combine their local IDE and the online code editor. However, this will usually not lead to merge conflicts. On the other hand, when working in a team, merge conflicts are likely to occur and will potentially require communication between the involved team members.

Students who pushed code after changes have been made to the remote repository will face conflicts. They need to figure out how to resolve them and push their changes,

including the merge commit. Once the team finished the work and pushed all changes, the team-based programming exercise is completed.

For modeling and text exercises, Artemis offers a real-time editing mode so that team members can collaboratively work together. Figure 5.16 shows the modeling editor with an information panel that lists all team members and shows their online status using a green circle indicator.

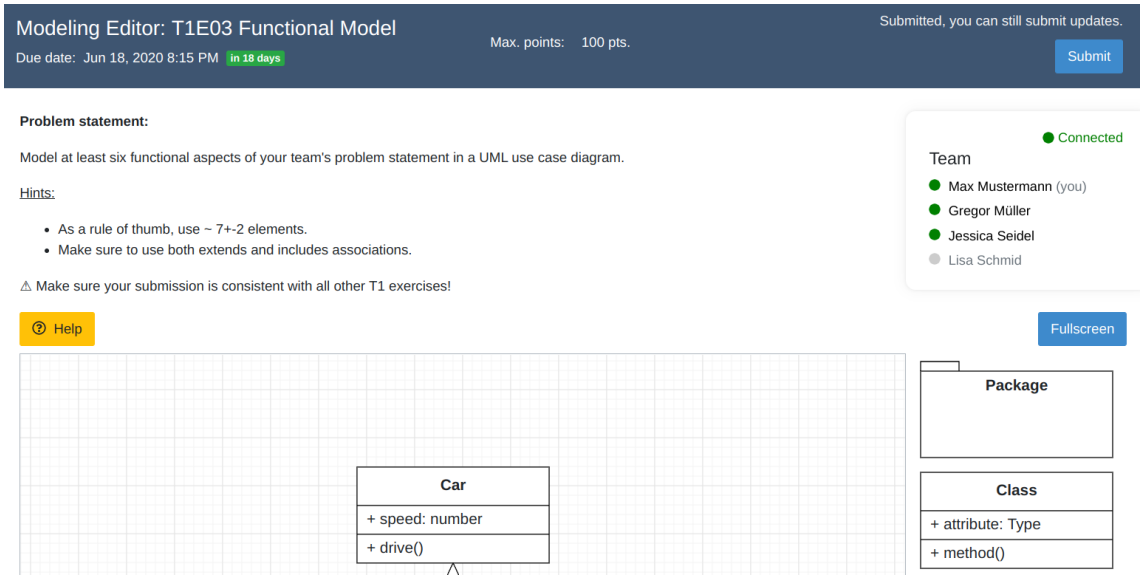


Figure 5.16: Multiple students working together on a UML diagram in a team.

Changes in those exercise editors automatically sync between the team members without the need to press save. This enables real-time collaborative modeling and text editing. Students see team-specific user interface elements next to the exercise submissions they are currently working on. They can see which team members are online and are currently editing the same exercise.

While users change the model (or text), Artemis indicates this with three points in the information panel. Clicking on the change history icon reveals the relative time since the last contribution by the respective team member.

Each team in Artemis is led by a team owner, e.g., an instructor or tutor in the respective course. Team exercises include a team overview page, which shows all teams with the team owner and all team members. Figure 5.17 shows how team owners can create teams with multiple students. Each team is assigned to one team owner who will act as the team's mentor and follow the team's progress across a set of team exercises typically built upon each other. In addition, the team owner will review the submissions of the teams and provide feedback. Using a double-blind assessment for team exercises (see Section 5.1.5) would not be feasible in this case because the tutors are closer to the team. However, it is crucial that they also consider the consistency between multiple team exercises in team projects. Therefore, the same team owners also evaluate complaints and feedback requests.

This tutor can, e.g., play the role of a customer, product owner, project leader, or Scrum Master in the team and guide the team throughout a project which consists

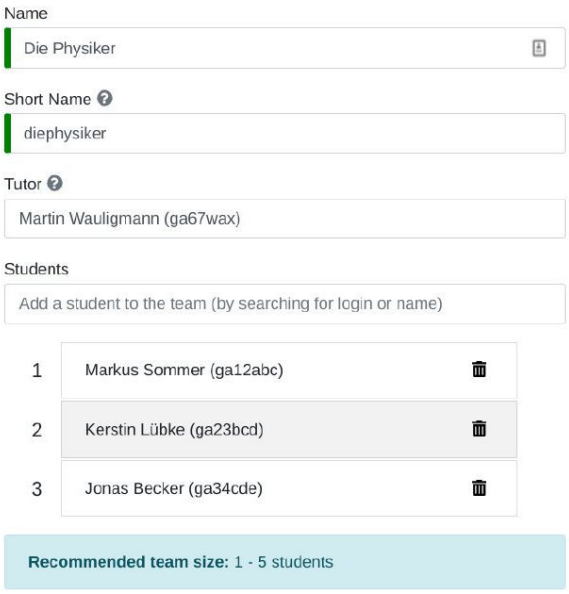


Figure 5.17: Dialog that allows a tutor to create a new team for an exercise.

of multiple exercises. Teams in Artemis are exercise-specific but can be imported into other exercises This creates duplicates using the same team name and team short name so that Artemis can track teams over multiple exercises. In addition, it allows team member changes between exercises, e.g., when a student joins the team late or when one student drops the course. Finally, tutors can manage all submissions on the team page in Artemis, as shown in Figure 5.18.

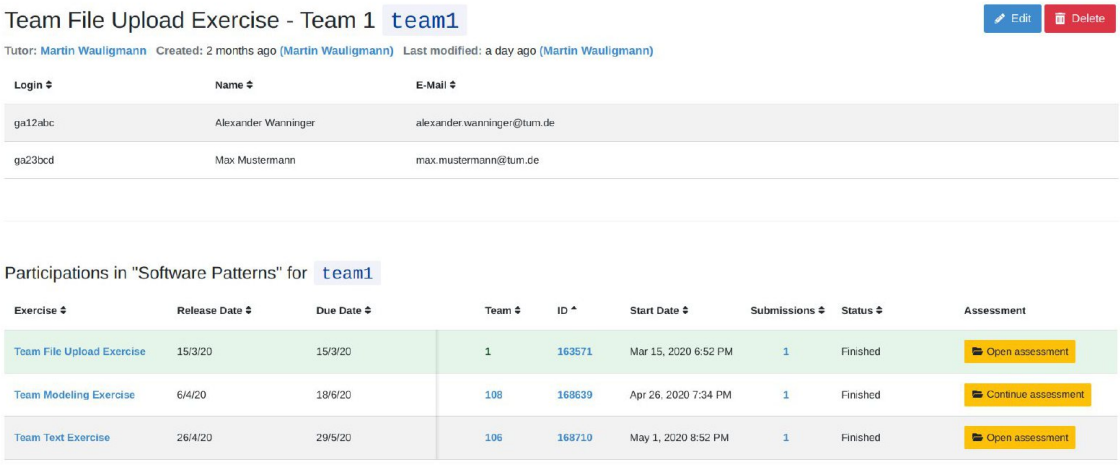


Figure 5.18: Team page that lists all team members, shows the team's participation in the course, and lets the team tutor assess their submissions.

The team page has several purposes. First, it allows the students to see who their team members are and allows them to contact them. Second, they can see the team's participation in the various team exercises in the same course. Finally, it also provides tutors with a way to assess the submissions for all team participants.

5.1.7 Lectures

Instructors can divide a lecture in Artemis into multiple lecture units following the interactive learning sprints. Figure 5.19 shows an example.

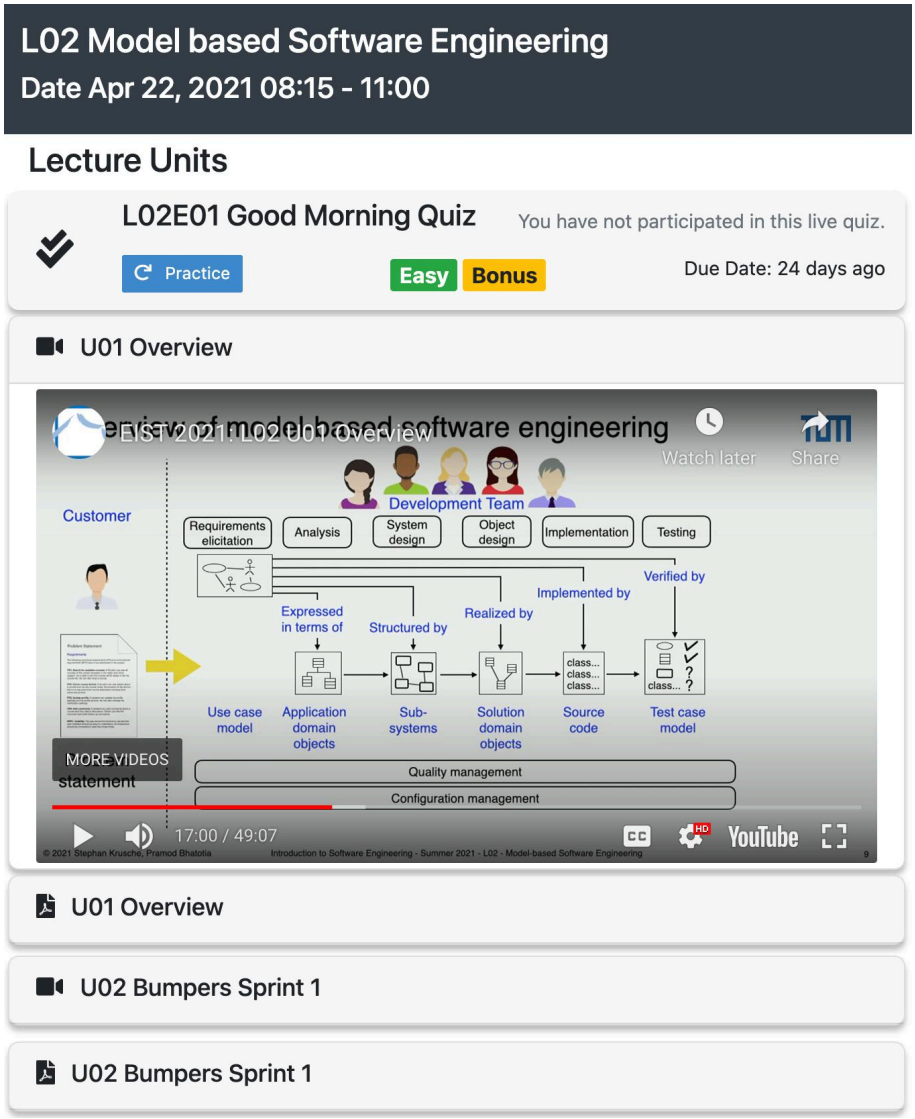


Figure 5.19: Instructors can divide a lecture in Artemis into multiple lecture units

Lectures consist of lecture units, either text, videos (live streams or recordings, e.g., stored on Youtube), or attachments (e.g., lecture slides). In addition, instructors can link exercises (e.g., L02E01 Good Morning Quiz in Figure 5.19) to integrate them in the interactive learning sprint and make it easier for students to find and participate in them during the lecture.

Instructors can also connect learning goals to lecture units, as shown in Figure 5.20. This allows instructors to make the progress of learning goals visible to students. Students can then better understand which learning goals (i.e., competencies) they have mastered.

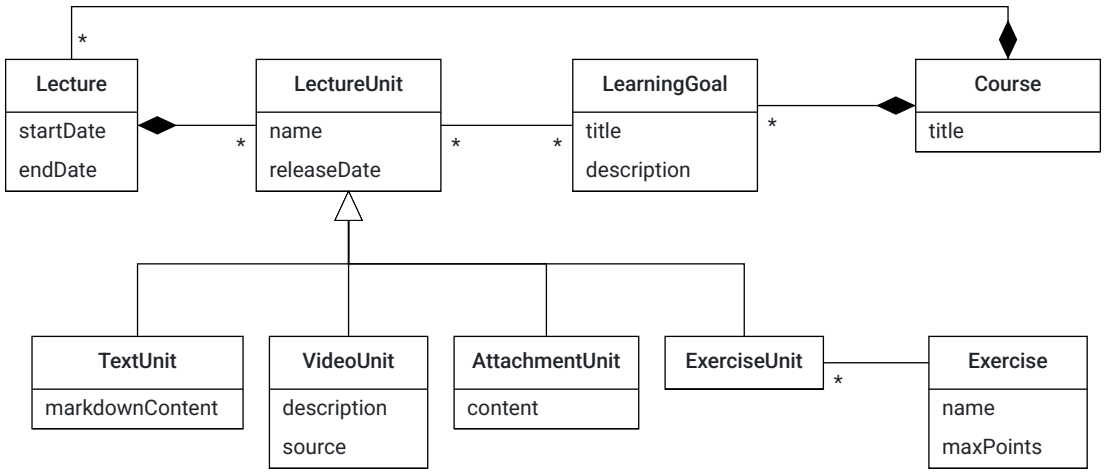


Figure 5.20: Structure and relationships of lectures and learning goals (UML class diagram)

Students can see the progress of all learning goals in the course statistics. Figure 5.21 shows an example of five high-level learning goals. Each learning goal covers multiple exercises. The percentages shown in Figure 5.21 show the weighted average of all related exercises. Instructors can decompose learning goals into smaller competencies. In a future version of Artemis, they can also model the relationships between learning goals.

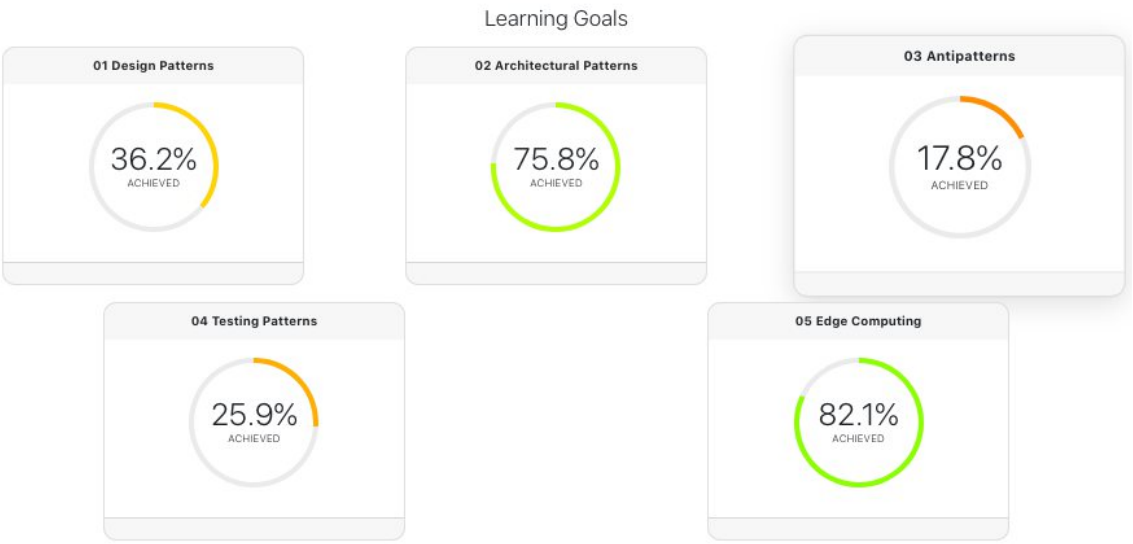


Figure 5.21: Progress of one student in five high-level learning goals.

Figure 5.22 shows the details of all exercises related to one learning goal. Students can use this information to identify which exercises they have not finished yet. Learning goal progress continuously measures the last result of the student. This can also be not graded when they participate in the practice mode of quizzes or finish a programming exercise after the due date. The progress is not meant as another assessment overview (such as the course score) but should mainly visualize the learning progress. It does not matter whether students have solved the exercise before or after the deadline.

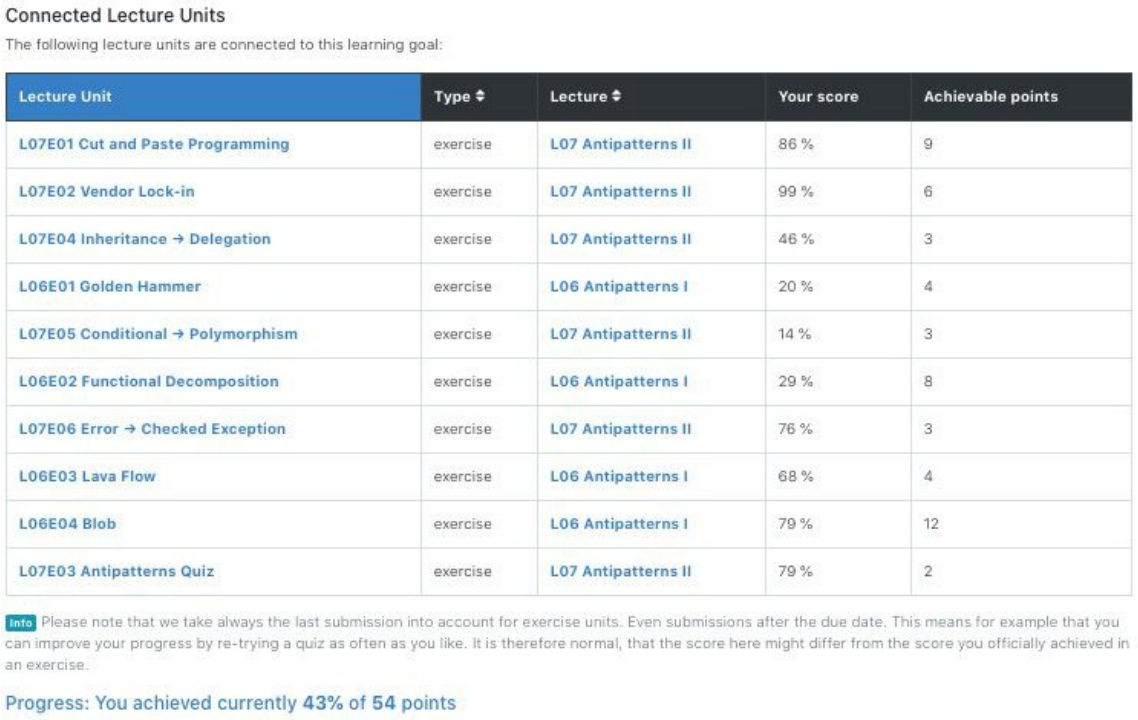


Figure 5.22: Detailed progress page for one specific learning goal.

5.1.8 Exam Mode

Artemis supports an exam mode, which offers a digital way of assessing the performance of students. Instructors can create exams for each course and register all or a selection of the participating students. Within the exam, they can set up programming exercises, modeling exercises, text exercises, file upload exercises, and quiz exercises or import existing ones. Then, before releasing the exam to the students, they can perform test runs to ensure all exercises are configured correctly.

Artemis supports individual working times in case of time extensions. To improve the reliability for students, Artemis saves all changes locally and automatically synchronizes every 30 seconds to avoid issues caused by unstable internet connections during the conduction. Once the exam is over, instructors can assess the submissions in the assessment dashboard, visualizing the assessment progress. They can choose to apply one or two correction rounds. All the features of the course exercise assessments apply to exams: double-blind reviews, reviewer training, leaderboards, structured grading criteria (see Section 5.1.5).

Once all exams have been assessed, instructors can release the results and allow all students to review their results online. If students believe they found a mistake in their assessment, they can use the complaint feature to request a re-assessment. A second reviewer will then re-assess the specific submission and decide whether to accept or reject the complaint. Artemis also creates a statistical report of the exam to evaluate and examine the students' performance.

Artemis provides the possibility to create a unique exam for every student. Instructors can create exercise groups that contain multiple exercise variants. The number of exercise groups reflects the number of exercises for an exam. When generating a student's exam, Artemis selects one exercise randomly from each exercise group and combines those to create a unique exam. Instructors can also randomize the exercise order. In addition, they can distinguish between mandatory and optional exercises to create even more randomness in the assignment. For example, Figure 5.23 shows two exercise groups, both including two variants.

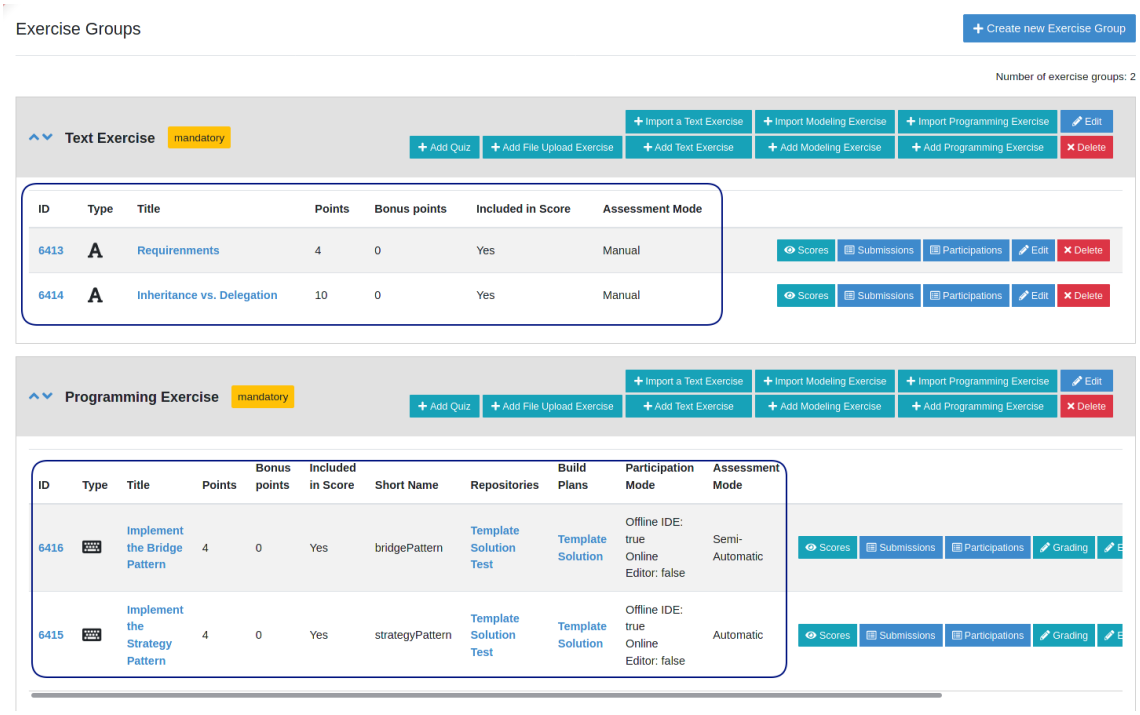


Figure 5.23: Instructor can create multiple exercise variants. Artemis randomly assigns a variant to one student and tries to generate random student exams.

Artemis can support instructors in detecting plagiarism attempts. It analyzes the similarities between all student submissions and highlighting those which exceed a given threshold. Artemis integrates the open-source plagiarism tool JPlag [PMP+02] for programming and text exercises and implements a custom comparison for modeling exercises. With the integrated plagiarism editor, instructors can compare all highlighted submissions and confirm those actual plagiarism attempts cases. In addition, instructors can download a report of accepted and rejected plagiarism attempts for further processing on external systems, notify the students, and allow them to react to the accusation. Figure 5.24 shows plagiarism checks for a modeling exercise.

The exam mode in Artemis is tolerant of internet connection issues. In case students are not connected, they can continue working on text, quiz and modeling exercises. If the internet connection recovers, Artemis will automatically send the locally saved state to the server. Students can work on programming exercises in their integrated development environment. Then, they only need to be online when they clone the repository and push their commits to the remote repository, i.e., when they submit their solution.

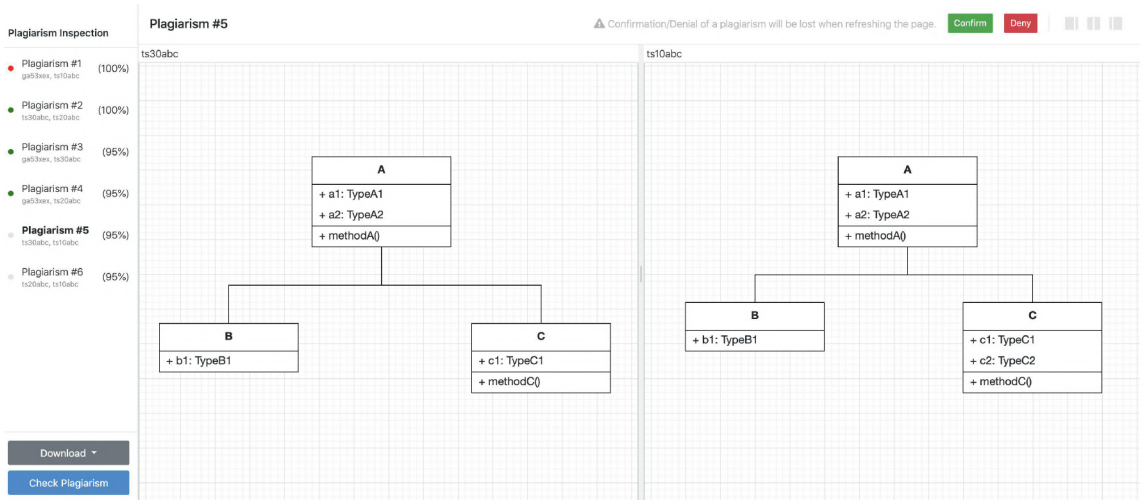


Figure 5.24: Instructors can run automated plagiarism checks. They can then evaluated the automatic findings manually to confirm or deny the plagiarism.

5.2 System Architecture

Artemis is a distributed system that consists of multiple subsystems running in processes and on separate virtual machines. Administrators can tailor the concrete architecture based on profiles and configuration settings to the specific needs of the Artemis installation. For example, they can use different implementations for the version control system (VCS) and the continuous integration system (CIS). Artemis also offers flexibility when it comes to scaling. Administrators can operate multiple Artemis servers (horizontal scaling), synchronized using a broker and a discovery subsystem, a load balancer, and a network file storage. Alternatively, they can operate a single Artemis server with a lot of CPU and RAM resources (vertical scaling) which would be easier to operate because some subsystems (broker, discovery, load balancer, network file storage) are not needed. In the following, we describe the top-level design, the deployment, and the data management of Artemis in more detail.

5.2.1 Top-Level Design

Figure 5.25 shows the top-level design of Artemis, which is decomposed into an application client (running as an Angular web app in the browser) and an application server (based on Spring Boot). For programming exercises, the application server connects to a version control system and a continuous integration system. Authentication is handled by an external user management system (UMS).

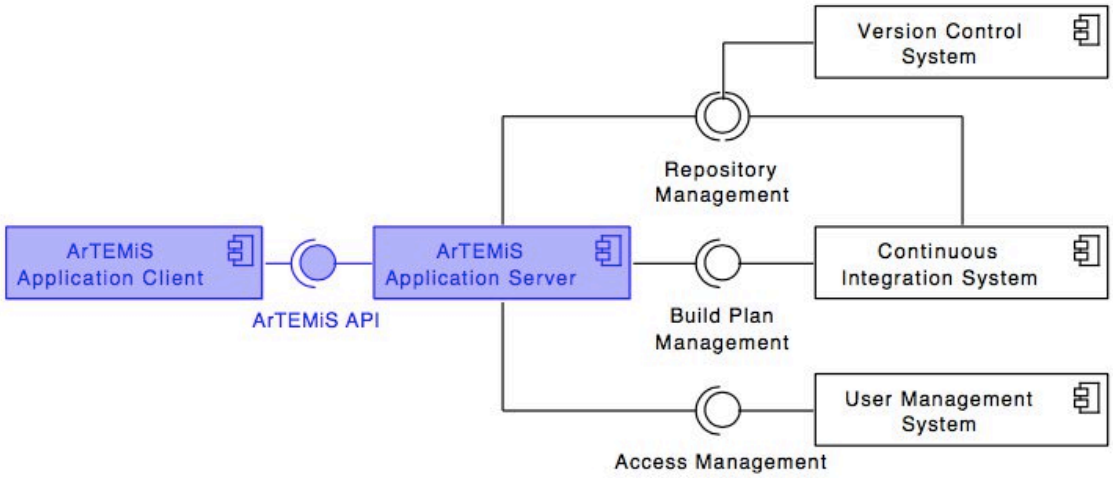


Figure 5.25: Top level design of the architecture of Artemis (UML component diagram)

While Artemis includes generic adapters to these three external systems with a defined protocol that can be instantiated to connect to any **VCS**, **CIS**, or **UMS**, it also provides multiple concrete implementations for these adapters to connect to:

- **VCS**: Bitbucket Server or Gitlab
- **CIS**: Bamboo Server or Jenkins
- **UMS**: JIRA Server (more specifically Atlassian Crowd on the JIRA Server)

Artemis allows external applications (e.g., learning management systems such as Moodle) to connect using LTI [SHH10]. The learning tools interoperability (LTI) standard describes how to quickly and securely connect learning applications and tools with learning management systems. LTI is comprised of a central core and optional services to add additional features and functions. The LTI core establishes a secure connection and confirms the tool's authenticity, while the extensions add features like the exchange of assignment and grade data between external assessment tools and the learning management system.

Figure 5.26 shows more details of the Artemis server architecture and its REST interfaces to the application client. The server application consists of three layers:

1. The web layer includes all REST and Websocket resources that handle the incoming and outgoing communication with the client. Classes in the web layer are responsible for authentication and authorization checks. They check the incoming data for validity and transform the outgoing data.
2. The application layer includes the logic for handling the incoming requests. For example, the `Exercise Participation Service` makes sure that specific steps are performed in the correct order when students start a programming exercise.
3. The data layer represents the facade to the database and includes database queries and write operations to save or delete objects.

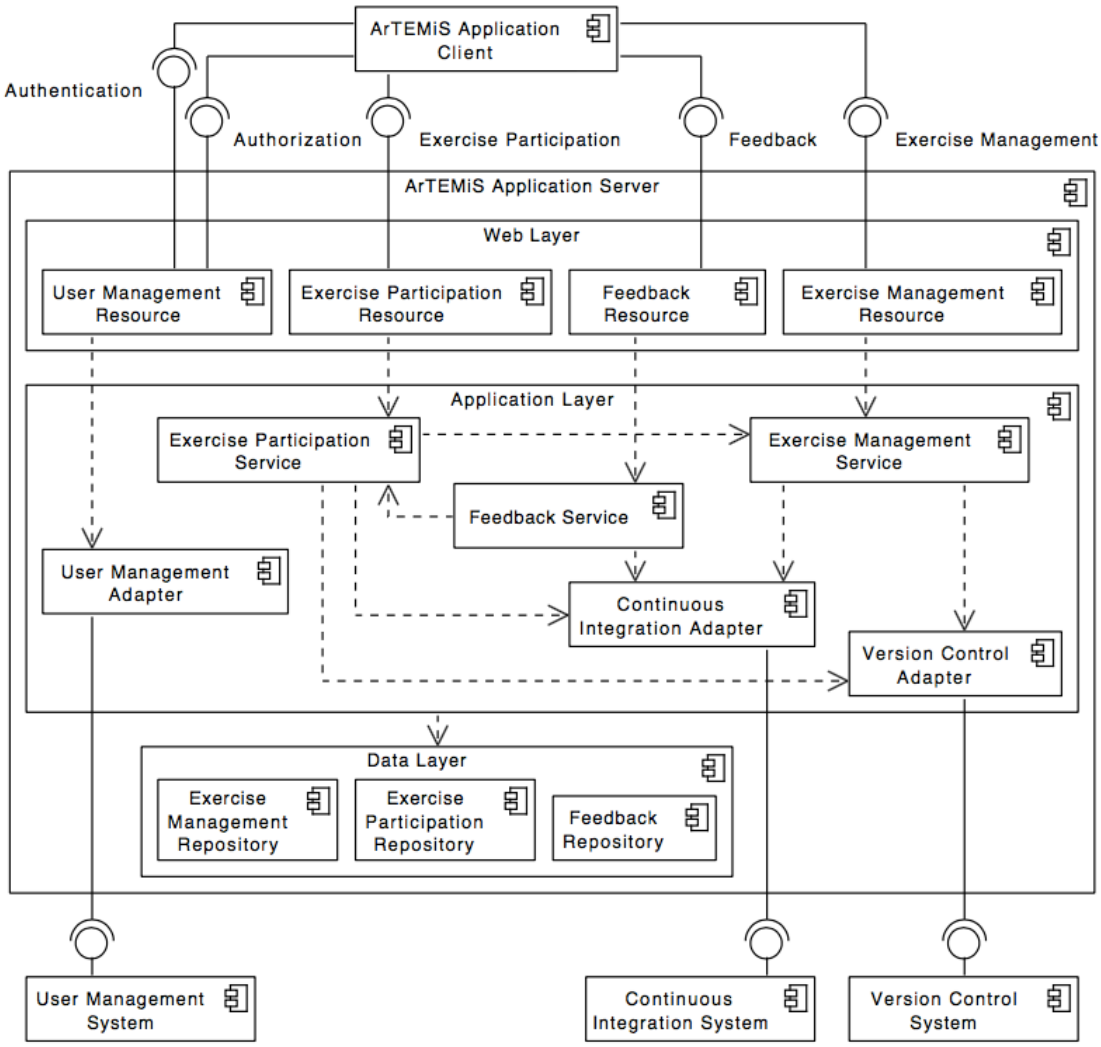


Figure 5.26: Artemis server architecture (UML component diagram)

5.2.2 Deployment

Figure 5.27 shows a typical deployment of the Artemis application server with one server node and the application client, which is accessed multiple times from different users (student, instructor, and tutor). The deployment focuses on the programming exercises feature and displays external components such as the version control server needed to conduct programming exercises. The student computers use a version control client and the Artemis client, a web application executed in the browser (e.g., Google Chrome).

The Artemis server acts as a facade to the end users and delegates specific tasks to the version control server (e.g. store repository with files, upload submissions, access control) and to the continuous integration server (e.g., compile student code, execute instructor tests, collect test case feedback). The continuous integration server typically delegates the build jobs to local build agents within the university infrastructure or remote build agents, e.g., hosted in the Amazon Cloud (AWS).

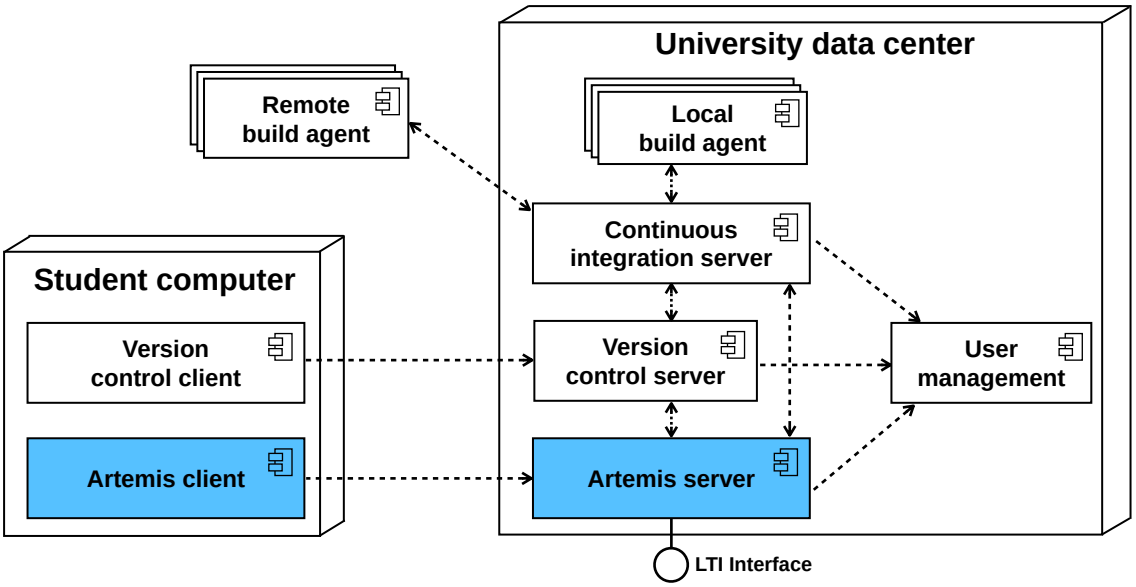


Figure 5.27: Deployment overview of Artemis (UML deployment diagram)

Figure 5.28 shows the deployment with multi-server nodes for horizontal scaling. The system's performance improves by adding more instances of the application server to distribute the load. Different machines run the different subsystems of Artemis. However, the client still communicates with the application server (through the load balancer). This diagram focuses on the central newly added subsystems (load balancer and several application server instances).

In this deployment, server administrators can perform maintenance work with less impact on the users. They can perform several tasks (such as updating Artemis) on the instances separately, allowing for higher availability. The redundancy also creates better fault tolerance, as a failure of one instance of the application server does not affect the whole system.

To allow for better scaling, all subsystems run on different virtual machines. For example, the database server runs on a separate virtual machine to increase its performance. The resources of this machine do not have to be shared with other subsystems anymore. It is possible to scale this machine better vertically because it is possible to adjust the resources for a database server.

Administrators are not limited by running several services with different requirements on the same machine. If the database server would run on the same machine as the application server, increased resource consumption would also impact the database server's performance. This setup would cause issues when several instances of the application server try to access the database. Then, one instance could slow down all other instances as they rely on the database server

The load balancer also runs on its own (virtual) machine to reduce the number of subsystems on one machine and allow for better scaling. As it handles all traffic to and from clients, the throughput of the network connection is essential. Memory and CPU are not as relevant as the load balancer is very resource-saving.

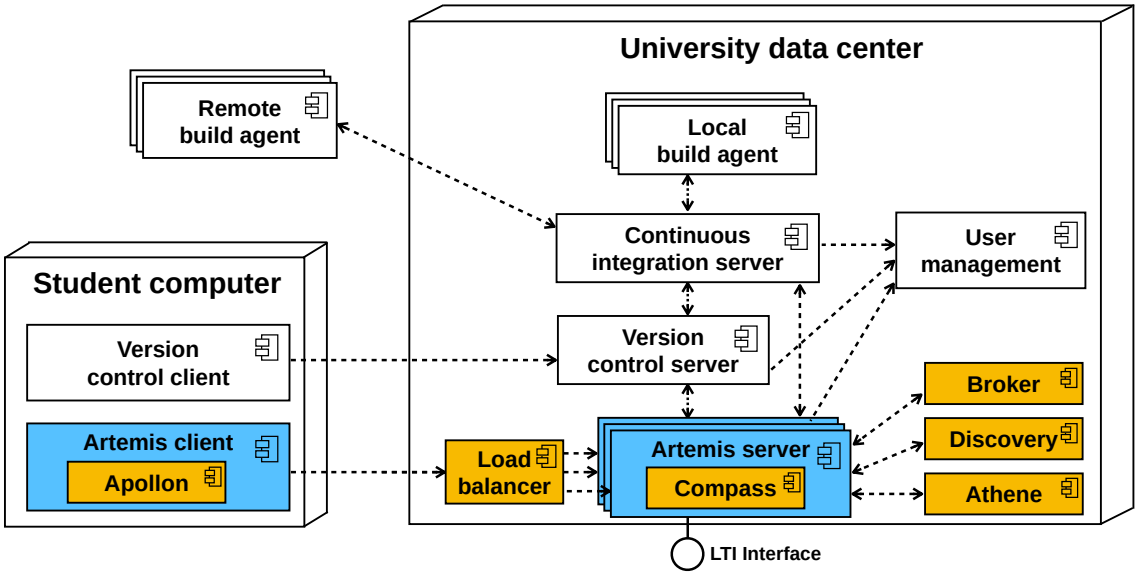


Figure 5.28: Deployment overview with several application server instances, the load balancer subsystem, and other related subsystems. The load balancer distributes requests from application clients to different instances of the application servers (UML deployment diagram).

Instances of the application server should be running on individual virtual machines so that they can neither affect each other nor different subsystems like the database server by having a high resource usage. Administrators can assign different weights to the different instances of the application server so that instances with more abundant resources (like more CPU cores) get more requests than instances with fewer resources. The application server is mainly CPU intensive. Therefore, its weight should depend on the available computation power rather than other resources like system memory.

One machine can host both the broker and the discovery service as the discovery service only requires minimal resources. The broker relays WebSocket messages between all instances of the application server. Thus, it should have a reliable connection to the machines running the application server.

Artemis uses a cache to reduce the load on the database server. Artemis uses Hibernate[8], a framework for object-relational mapping (ORM), that automatically maps the objects used within Artemis to relations stored in the database [BK05]. Hibernate offers different caching strategies and integrates with several caching providers [JF11]. The cache consists of different regions (e.g., all cached objects of a class form a region). Developers can apply different configuration options for each region.

Artemis uses EhCache[9], a commonly used cache provider for Hibernate [Win13]. It uses Hazelcast[10] as a cache provider because it supports distributed caching without manual management of the cache. Figure 5.29 shows the shared usage of the database and the cluster built by Hazelcast to offer a distributed cache. Hazelcast is an in-memory data grid that uses a peer-to-peer architectural style [Joh13]. The discovery service allows Hazelcast to form a cluster containing all instances of the application server.

[8] https://hibernate.org
[9] https://www.ehcache.org
[10] https://hazelcast.com

Hazelcast integrates into Spring and Hibernate and provides common data structures like distributed hash maps that all instances of the cluster can access. Artemis uses them for caching that is not directly integrated into Spring and Hibernate. For example, they are used to store the submissions during live quizzes. All instances of the cluster access the identical submissions so that in case one instance fails, other instances still have a copy stored.

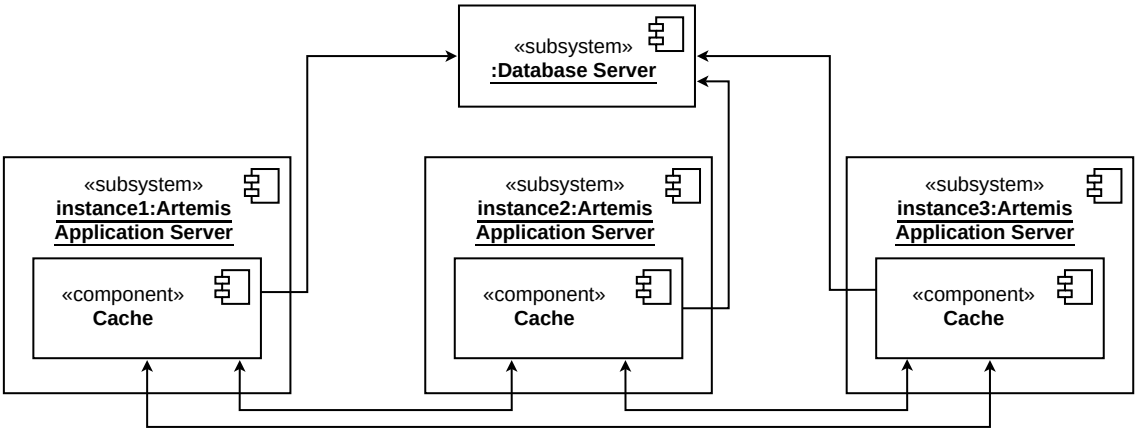


Figure 5.29: Shared usage of one database by three application server instances and communication within the distributed caching cluster. Every instance of the application server communicates directly with the database server and with all other instances to handle cache invalidation (UML component diagram).

Every instance of the application server is connected to all other instances to broadcast updates to the cache to ensure a consistent state. Thus, the cache acts as a proxy and receives all requests that interact with the database. It can either provide the requested data from the local cache (thus preventing a request to the database) or, if the data is not present locally, request the data from the database (and store the data in the local cache to prevent additional requests), implementing a proxy pattern.

5.2.3 Data Management

The Artemis application server uses a relational MySQL database. Figure 5.30 shows the Java classes stored persistently (note that the figure does not include all entities, attributes, and relationships).

Artemis supports multiple courses with multiple exercises. Each student in the student group can participate in the exercise which creates one participation. For programming exercises, a git repository and a continuous integration build plan will be created and configured for the student (`User`) or the team. The initialization state variable (enum) helps track this complex operation's progress and allows recovery from errors.

A student can submit multiple solutions by committing and pushing the source code changes to a given example code into the version control system or using the user interface. The continuous integration server automatically tests each submission and notifies the Artemis application server when a new result exists. For other exercise types, students directly submit their solution to Artemis, which creates a submission

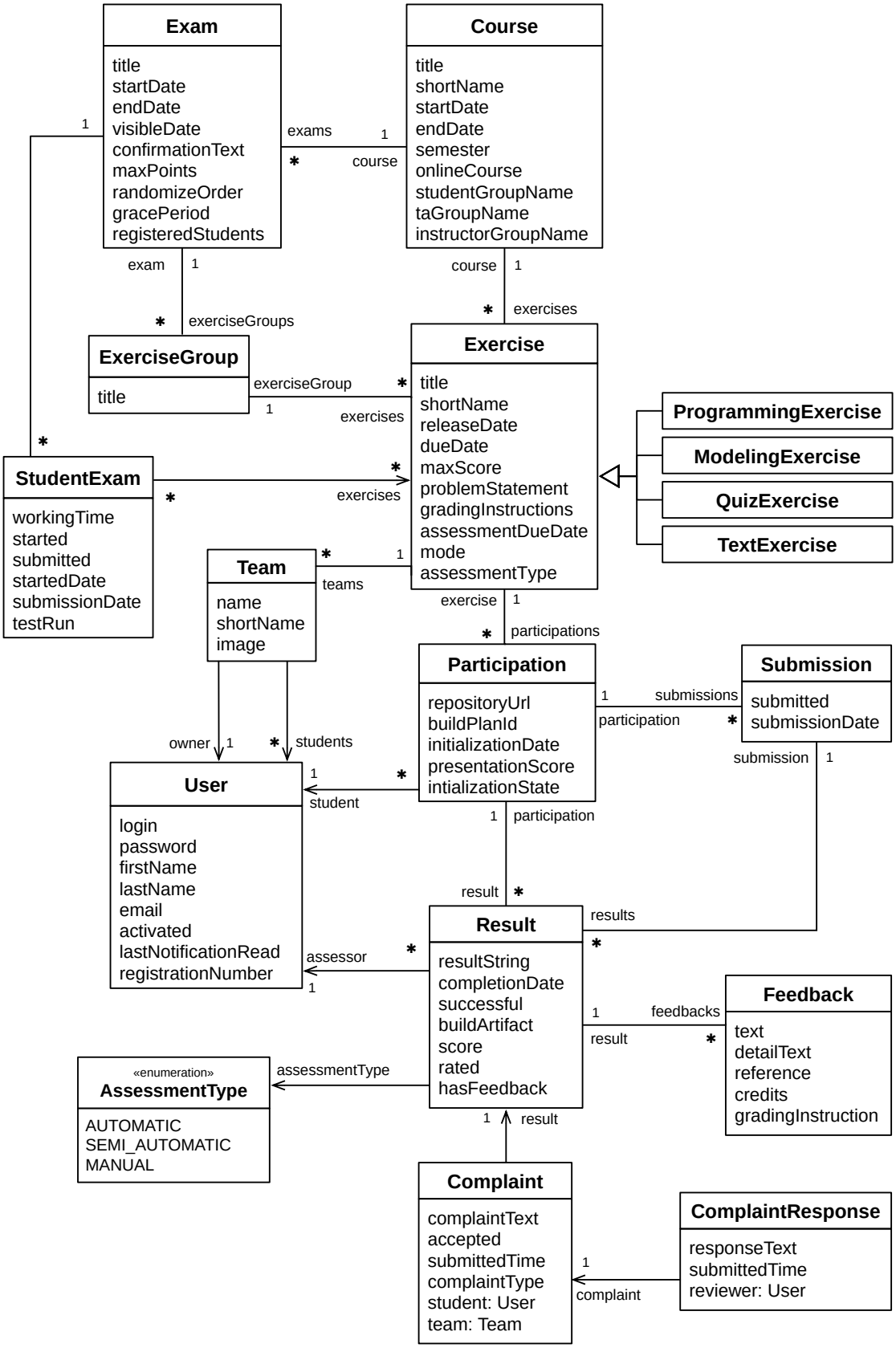


Figure 5.30: Database model of Artemis (UML class diagram)

object. Tutors can review student submissions and create manual results with feedback. Artemis distinguishes between different assessment types: automatic, semi-automatic, and manual.

Students can complain or request more feedback if they think the result is wrong or misses essential aspects. Then, a second tutor (complain) or the same tutor (feedback request) re-evaluates the submission and either accepts or rejects the complaint, respectively provides more feedback.

A course can include multiple exams. Each exam consists of multiple exercise groups, which allow the creation of multiple exercise variants. In addition, instructors generate student exams with random exercise variant combinations for each registered student. Student exams also allow instructors to grant individual working time extensions, e.g., when students provide a doctor's certificate for a specific weakness. The actual data model is more complex and supports more features such as versioning, lectures, student questions, and static code analysis.

Chapter 6

Evaluation

> "I never teach my pupils. I only attempt to provide the conditions in which they can learn."
>
> — Albert Einstein

This chapter summarizes the empirical evaluations of interactive learning and Artemis using a design-based research process [Col92, Hoa02].

Section 6.1 describes the courses EIST, PSE, POM, and SEECx offered at the Technical University of Munich (TUM) as case studies in which we have applied interactive learning and Artemis as interventions. The findings in those case studies have helped address the needs of students, tutors, and instructors and iteratively and incrementally improve Artemis based on a formative development approach. We also describe the dissemination of Artemis to other courses at TUM and nine other universities that have started to apply interactive learning with Artemis.

Section 6.2 presents the essential evaluation results based on four of the five hypotheses stated in Chapter 1. Section 6.3 discusses the threats to validity. More detailed evaluations can be found in the individual publications in Chapter 8.

6.1 Case Studies

We applied interactive learning between 2015 and 2021 in four software engineering university courses shown in Table 6.1. In previous instances of these courses, content delivery (theory) and content deepening (exercises) were separated, i.e., students learned a concept in the lecture and applied it a week later in a central exercise session. However, in the first instance mentioned in Table 6.1, we applied interactive learning and combined theory and exercises into interactive classes. As a result, students learned a concept theoretically, then immediately practiced it in a short exercise and received feedback about their progress following the interactive learning teaching philosophy described in Chapter 4 and the cycles shown in Figure 4.1. In the following, we describe the four courses in more detail.

Short	Course	Active students	Program	Instances
EIST	Introduction to Software Engineering	up to 2,100	Bachelor (2nd sem)	SS19 - SS21
PSE	Patterns in Software Engineering	up to 600	Bachelor + Master	WS16/17 - WS20/21
POM	Project Organization and Management	up to 400	Bachelor + Master	SS15 - SS19
SEECx	MOOC: Software Engineering Essentials	up to 700	Anyone	SS17 - WS20/21

Table 6.1: Courses with interactive learning used as case studies

6.1.1 Introduction to Software Engineering (EIST)

This section describes the undergraduate software engineering course EIST[1] at the Technical University of Munich. The learning objectives of EIST are to familiarize students with relevant concepts, workflows, and methods of software engineering and apply them in all phases of software engineering projects. This includes analyzing and evaluating problems, e.g., modeling the problem, reusing classes and components, and testing the software. Concerning UML, students learn to communicate using models. They learn how and when to apply which model. They understand the relationship between modeling and programming and learn to abstract. Students learn to model and implement concrete problems in software engineering.

EIST is a mandatory bachelor's course offered in the second semester for a heterogeneous group of students from computer science, business informatics, business, and other fields. A prerequisite of the course is that the students have essential programming experience, such as having completed an introductory course in computer science (e.g., CS1). Course instructors use constructive alignment [Big03] to align teaching and assessment with the course objectives. For each lecture, a set of learning goals is defined based on the six cognitive skills in Bloom's taxonomy [BEF+56]. The course focuses on higher cognitive skills so that students learn to apply the concepts in concrete situations. Students cannot pass the course by simply memorizing the course material.

Organization

One lecturer and two to three exercise instructors organize and teach the course with the help of around 45-60 student tutors. The tutors are bachelor and master students who completed the course in previous years. The course takes place in the summer semester over 12 weeks. Table 6.2 shows the course content together with the UML models that are taught in the respective lecture. Up to 2,100 active students participate in the course. There is no lecture hall with enough seats for all students. The course uses a live stream and broadcasts the lecture to additional overflow lecture halls and over the internet so that students can participate in the lecture from home.

Most students either participate actively in the main lecture hall or watch the live stream at home. Therefore, the overflow lecture halls are closed after a few weeks. All students (within the main lecture hall, overflow lecture halls, and live stream) can ask

[1] The course is called "Introduction to Software Engineering", the abbreviation EIST is based on the German translation "**E**inführung **I**n die **S**oftware **T**echnik

Week	Content	UML Model
1	Introduction	Class
2	Model-based SE	Use Case, Class
3	Requirements Analysis	Object, Communication
4	System Design I	Component, Deployment
5	System Design II	Component, Deployment
6	Object Design I	Class
7	Object Design II	Class
8	Model Transformation and Refactoring	State Chart
9	Software Lifecycle Modeling	Activity
10	Software Configuration Management	Activity
11	Testing	Class
12	Project Management	Class, Activity

Table 6.2: The course schedule of EIST includes 12 lectures. Each lecture includes specific UML models.

questions using Slack's chat platform[2]. Tutors answer these questions directly or pass on a question to the lecturer to repeat and answer it for all students.

Design

Large courses present the challenge of keeping students motivated throughout the semester (without, e.g., enforcing mandatory attendance). Students are easily distracted by off-topic conversations with other students or social media and stop paying attention to the lecture. The course includes interactive elements to activate the students and keep students engaged in dealing with such situations. The interactive components include in-class exercises, in-class quizzes, and group exercise sessions.

The university's study program does not allow the inclusion of weekly assignments when calculating the final grade. Therefore, the course uses a bonus system that motivates students to participate in its exercises: students can earn bonus points for completing in-class and homework exercises. They need to present their homework twice in tutor exercise sessions to get the bonus applied. The presentation requirement makes sure that they have worked on the homework independently and improves their soft skills.

If they pass the final exam, their exercise points are mapped to exam points added to their final exam score to improve it. The German grading system consists of marks between 1.0 (similar to *A* in the US grading system) and 5.0 (similar to *F* in the US

[2]Slack is a cloud-based instant messaging platform: https://slack.com

grading system), with 1.0 being the highest grade and 4.0 (similar to *D* in the US grading system) being the passing grade. For instance, if students score 30 % of the bonus points, they receive additional 3.0 points on top of their exam score, which improves their final grade by 0.3. If they score 100 % of the bonus points, they can receive a total bonus of 1.0. For example, they can improve from a 2.3 to a 1.3. Students have reported that this increases their motivation to participate in the exercise system actively.

The course includes quizzes, programming, modeling, and text exercises. In-class exercises include quizzes to recapture previously learned content. They also include programming and modeling exercises as guided tutorials. Tutors help with student questions and problems during the in-class exercises. Group exercises mainly encompass modeling exercises, small programming exercises, and text exercises that the students work out together in small groups during their group exercise sessions. Homework assignments include modeling, programming, and text exercises and enable students to deepen their knowledge in self-study.

Modeling Exercises

Students model a solution to concrete problems using UML. Modeling exercises stimulate higher cognitive skills and force students to analyze, evaluate and create. Apollon supports UML class, object, activity, use case, communication, component, and deployment diagrams. As shown in Table 6.2, each lecture includes different UML model types aligned with the taught content. Apollon allows students to drag and drop the model elements into the canvas, add attributes, methods, and define associations between them. The advantage of Apollon is that students cannot use a UML element other than the ones specified for the specific model type.

Quiz Exercises

Students repeat already learned content during lectures and test their knowledge. They stimulate lower cognitive skills such as remembering and understanding the concepts. A quiz question can be multiple-choice (MC), drag and drop (DnD), or short-answer. For questions related to modeling, the quizzes include MC and DnD questions where students drag elements to predefined spots on the canvas.

Programming Exercises

Students learn to make connections and see differences between models and their implementation in programming exercises. These connections stimulate their cognitive skills, and students learn to apply the knowledge when implementing source code. A UML class diagram, e.g., represents the general structure of the source code and can be used as an interactive problem statement in Artemis. Red model elements indicate that they are not implemented correctly, whereas green elements indicate correctly implemented ones. These interactive elements further help students understand what UML models should contain and what should be left out.

Text Exercises

Students need to answer questions about the learned concepts by writing open text responses. They, e.g., need to explain similarities and differences between design patterns in their own words and describe concrete situations when design patterns can be used. These exercises stimulate analysis and evaluation skills.

Team Exercises

In addition to individual in-class exercises during the lectures, group exercises during the tutor exercise sessions, and individual homework exercises, the course also includes team projects since 2020. Team projects focus applying the learned content in a more realistic example based on a short problem statement. Other exercises (in-class, group, and homework) typically focus on many different examples and are limited in that students might not understand the relationships between the taught concept, which is particularly important in software engineering. A change in the analysis model, e.g., influences the object design and the definition of test cases.

Team projects with freedom and creativity can teach those relationships to students. If the students are responsible for the consistency between multiple software engineering artifacts, they will recognize the influence of their own decisions and better understand the relation between the taught concepts. Team-based learning also improves the soft skills of the students, in particular communication and negotiation skills. Between 250 and 350 teams, each with three to five students, actively participate in the team exercises. Tutors supervise and mentor the teams and provide feedback. Table 6.3 shows the phases of the team project, which is aligned to the course schedule in Table 6.2.

#	Start	End	Phase
0	Week 1	Week 3	Team-building
1	Week 4	Week 5	Requirements analysis
2	Week 6	Week 7	System design
3	Week 8	Week 9	Object design & implementation
4	Week 10	Week 11	Testing
5	Week 12	Week 13	Build and release management

Table 6.3: Phases of the team exercises in EIST

In the team-building phase, students find peers and build a team. They choose one of three vague problem statements that involve developing a simple single-player game in Java. They have much freedom in specifying the exact requirements together with the supervising tutor. Shortly before the end of the team project, the instructors (who play the role of the customer) introduce change and ask the students to develop a multiplayer game instead. The teams have to adapt the most important models of previous phases. Late changes make the team exercises even more realistic because they also have to be considered in real software engineering projects.

Communication

The course uses Slack as a communication tool to facilitate discussions between students and teaching staff. Using instant messaging lowers the entrance barrier for students to ask questions because it feels familiar to communicate (e.g., social media chats), and students ask more questions if they notice that other students do the same. Students can communicate with each other and send direct messages to tutors and instructors if they have a question or require help during the lectures and at any other time. Slack offers the ability to use channels with specific purposes:
#announcements — Instructors post course-wide announcements (students cannot post here), e.g., reminders that a lecture is canceled on a public holiday
#organization — Questions about the organization of the course
#lecture — Questions regarding the lecture slides
#exercise — Questions regarding the exercises

The instructors further encourage students to answer questions themselves. This increases a sense of belonging (which is hard to achieve in such a large setting) when students communicate with each other but can also deepen their understanding of a topic, e.g., in discussions that pursue questions. Students receive fast replies, which increases the interactivity. Tutors help to moderate discussions and make sure all participants respect the university and course code of conduct. They ensure a positive atmosphere, reprimand, and prevent bullying. They answer questions and point students to previously asked questions if it has already been asked before.

Tutor Exercise Sessions

45-60 tutors hold 80-100 weekly occurring tutor exercise sessions, each with around 15-20 students. The main focus for tutors is to activate the students in these sessions, moderate discussions, and explain the learned concepts again in case the students ask questions. Tutors have the following responsibilities:

- Attend a weekly tutor meeting with the instructors[3].
- Provide feedback to student submissions to exercises.
- Hold one or two tutor exercise sessions per week.

Tutors also help with moderating the Slack channels, answer questions on Artemis, help students during in-class exercises, or review slides and exercise content.

In the tutor exercise sessions, students apply the knowledge acquired in the lecture. Each tutor exercise session is structured as follows:

1. **Review the previous lecture** [5 - 10 min]: students discuss the learning goals, outline, and summary.
2. **Homework presentation** [30 - 45 min]: students present their solution to homework exercises. Tutors ask questions about the solution, point out typical mistakes, and provide additional feedback.

[3] Instructors discuss issues and present the subsequent group work and homework.

3. **Group work** [30 - 45 min]: Students work on predefined group exercises in groups (3-6 students).
4. **Discussion of the next homework** [5 - 10 min]: the new homework exercises are briefly discussed.

The tutor exercise sessions review a specific topic covered in the lecture before and prepare the students for the next homework assignment. They help to deepen the understanding of the taught concepts. Group exercises show the application of the learned methods with the help of concrete problems in the different phases of software engineering. Homework assignments deepen the knowledge in self-study. Students receive individual feedback on their homework submissions, which allows them to measure their learning progress and improve their skills. In addition, the presentation of their solution improves the students' communication skills, an essential skill in software engineering.

For instance, in lecture *Object Design II*, the course covered the Strategy Design Pattern [GHJV94] using an example and the general structure. In the corresponding tutor exercise session, there was one group work, where students discussed the pattern's problem, solution, benefits, and consequences. The students modeled a real-world example of the strategy pattern as a UML class diagram in the subsequent group work. This exercise was designed to teach students how to approach a concrete problem, analyze it, and model this problem. In one homework assignment, the students were given a similar problem: model different encryption strategies as a UML class diagram, using the strategy pattern. In another assignment in a programming exercise, the students had to implement sorting algorithms using the strategy pattern.

Grading

While programming and quiz exercises are automatically evaluated, modeling and text submissions are graded manually. Artemis offers a double-blind grading system, which opts for less bias while grading. Every week on Monday at noon, the homework is published. The students then have one week to work on the exercise and submit their solutions. In the following week, students present their homework in the tutor exercise sessions. Tutors use example solutions and detailed grading criteria to provide individual feedback. In the grading criteria, instructors point out that multiple solutions to modeling exercises can be correct.

When students are given sample solutions, they often do not think about their solutions but take the sample solution as the single truth. To encourage self-reflection and revision, instructors do not distribute example solutions to students. The feedback students receive about their own solutions is crucial for them to understand how they can improve.

6.1.2 Project Organization and Management (POM)

Up to 400 students with a heterogeneous background participate actively in POM, offered in multiple study programs. Two distinct groups participated: (1) bachelor students in information science, a few with experience in software engineering, and (2) master

students in computer science, some with existing experience in the taught topics. The challenge of this heterogeneity was that students completed lecture exercises at different rates. The exercises included optional tasks specifically for more experienced students to improve this situation. In addition, the students had the opportunity to solve exercises as homework if they could not finish them in the lecture. Some tasks and exercises were also explicitly designed as homework.

The module description of POM describes the following intended learning outcomes:

1. Participants understand the fundamental concepts of software project management.
2. They can write a software project management plan, initiate and manage a software project, and tailor a software life cycle.
3. They are familiar with risk management, scheduling, planning, quality management, build management, and release management, and can apply these techniques to solve simple problems.

Table 6.4 shows the schedule and the content of the lecture.

Week	**Content**
1	Team formation
2	Project organization
3	Software process models
4	Agile methods [KABW14]
5	Prototyping
6	Proposal management
7	Branch & merge management [KBB16]
8	Contracting
9	Continuous integration
10	Continuous delivery [KA14]
11	Risk and demo management
12	Global project management [LKLB16]
13	Project management antipatterns

Table 6.4: The course schedule of POM includes 13 lectures.

Students can earn bonus points for completing exercises successfully. They can use these bonus points to improve their final exam mark. If they, e.g., earn between 60 % and 80 % of the total points, their mark in the final exam is improved by one grade. This possibility motivates the students to participate in individual and team-based exercises. The instructors use the exercise types: quizzes, interactive tutorials, and project work.

Interactive Tutorial

Students solve individual tasks on their computers. They cooperate with the instructor, tutors, and fellow students to solve particular problems. They learn from their experience in exercises and reflect on the concepts they just learned a few minutes before. The instructor conducts four extensive interactive tutorials in POM using dedicated tools:

1. Agile Methods (Atlassian JIRA[4])
2. Branch and Merge management (Atlassian Bitbucket Server[5])
3. Continuous Integration (Atlassian Bamboo[6])
4. Continuous Delivery (HockeyApp[7])

In these interactive tutorials, the instructor introduces concepts and immediately applies them in short exercises. The students complete the exercises on their computers using the mentioned tools in the browser. Students look at the detailed slides handed out at the beginning of the exercise or watch how the instructor conducts the exercise on the presentation computer. Tutors walk through the lecture hall and help students by answering questions directly.

Each interactive tutorial consists of three to five exercises which were decomposed into smaller tasks. In summary, the students have to solve between twelve and twenty small tasks in one tutorial. The instructor synchronizes the speed of the tutorial several times by asking students about their progress and by checking the number of participants and results in the tools. If more than 90 % were able to complete particular tasks, the instructor proceeded to the next exercise.

One example is executing the two exercises on continuous integration and delivery based on a release management workflow [KA14]. The instructor maps an exemplary delivery process for a mobile application to the continuous integration server Bamboo. Each student first forks a preconfigured git repository and clones a preconfigured build plan to simplify the exercises. Then, the students adapt and configure the build plan, fix existing test cases and write additional test cases. A change in the software requirements leads to a bug that is detected by Bamboo during a regression test. The students have to fix the bug so that all tests pass again at the end of the exercise. Then, they can deliver the software to their fellow students who play the role of test users and provide user feedback.

Project Work

In addition to individual exercises, students participate in a team project with five team members, a simplified version of the team projects described by Bruegge et al. [BKA15], including creative exercises such as software theater [KDXB18]. The goal of the project is that students experience the learned concepts in a more realistic environment. The instructor plays the role of the customer and provides three short problem statements about the development of mobile applications. The teams choose one of the problem statements (flight app, lecture app, reservation app) and a development environment and target platform, either Android, iOS, or Java.

The instructor arranges the students into different teams according to their self-assessment. The goal is to have balanced teams concerning the skill level of the students. Team-based exercises are also built on experiential learning techniques. However, they

[4] https://www.atlassian.com/software/jira
[5] https://www.atlassian.com/software/bitbucket
[6] https://www.atlassian.com/software/bamboo
[7] https://hockeyapp.net

have a stronger focus on problem-based and cooperative learning. For example, software engineering is a collaborative activity [Whi07]. Therefore, teamwork is an essential skill the students have to learn. The teams use Rugby [KABW14] as an agile and continuous process model with an initial warm-up phase called Sprint 0 and five development sprints. Table 6.5 shows the team project schedule.

#	Start	End	Phase
0	Week 1	Week 3	Sprint 0
1	Week 4	Week 5	Sprint 1
2	Week 6	Week 7	Sprint 2
3	Week 8	Week 9	Sprint 3
4	Week 10	Week 11	Sprint 4
5	Week 12	Week 13	Sprint 5

Table 6.5: Phases of the team exercises in POM

In addition, students only receive a vague description of the exercises that deliberately miss detailed instructions so that the teams have to think on their own about how to solve the exercise and incorporate their ideas [KBC+17]. Students first learn and experience concepts in individual exercises. Then, they apply the knowledge in team exercises based on one project context to understand the relationships between the taught concepts and improve their long-term memory. Finally, they have to tailor the concepts to their concrete team situation and agree on different decisions in their team, which facilitates communication, collaboration, and conflict handling. Later in the course, students reflect on their team experiences, e.g., using sprint review and sprint retrospective meetings.

6.1.3 Patterns in Software Engineering (PSE)

Up to 600 students participate actively in PSE. The course includes critical concepts of different types of patterns used during software development, particularly design patterns, architectural patterns, testing patterns, antipatterns, and organizational patterns. Students' learning goals are to understand patterns to describe reusable knowledge for analysis, system design, object design, and software project management activities. Given a problem, they can identify the applicability of a pattern that addresses the problem, describe the pattern in UML and map it to Java source code. Bachelor and master students mainly in the field of computer science attend the course. Table 6.6 shows the schedule and the content of the course.

During the 13 lectures, instructors conducted around 40 exercises: most were in-class exercises, and a few were homework. The course includes the exercises types interactive coding challenge and interactive modeling challenges. It also carries out quizzes and interactive tutorials. In the homework exercises, students further deepened their knowledge. Exercise participation was optional for the students.

Week	Content
1	Introduction and pattern definition
2	Basic concepts
3, 4, 5	Design patterns
6, 7	Architectural patterns
8, 9	Antipatterns
10, 11	Testing patterns
12	Pattern-based reengineering
13	Global software engineering

Table 6.6: Overview of the course content in PSE

Interactive Coding Challenge

The participants have to write new source code or adjust existing code, commit their changes to a version control system that automatically triggers test cases on a continuous integration server to verify the given solution. The instructor rewards the first three students who submit a correct solution with gummy bears or donuts to increase the extrinsic motivation. In addition, there is a wildcard winner, which is randomly picked and acknowledged without being among the first three correct submissions.

After introducing the theory and explaining the problem, the students start to work on the exercise. Artemis makes sure that all students start working on the exercise simultaneously and have the same timeframe for solving the problem. The instructors determine the timeframe to submit the exercise, and the elapsed time during the exercise is visualized on a big stop clock on one of the projectors. Once the deadline has passed, the instructors provide a sample solution and discuss it with the students. Winning students also have the opportunity to explain their solution and why they came up with this approach. Due to the use of Artemis, it is possible to track and validate the students' results in real-time.

One example of an exercise is the application of the state design pattern in lecture 4. After the introduction and explanation of the state design pattern, the exercise for the students is to implement a basic remote control for a TV with four states. Next, they need to apply the state pattern to implement the transition between the appropriate states. Then, the instructors provide a standard Java Eclipse project with existing source code unrelated to the state pattern. Finally, a UML state diagram visualizes the problem with the different state transitions and their limitations.

The instructor sets the timeframe to solve this exercise to 15 minutes, released the exercise on Artemis, and the students start to work on the exercises. During the exercise, tutors help the students if there are questions. Students can ask for help by raising their hands or by asking questions on Slack. After 5 minutes, the instructors give a hint using a class diagram representing the implementation of the state pattern. Each submission leads to a compilation and the execution of 20 test cases. When the deadline for working on the exercise has passed, the first three correct submissions and the wildcard winner

are honored. Finally, the instructors discuss a sample solution with the students to reflect on the learned concepts.

6.1.4 Software Engineering Essentials (SEECx)

The MOOC Software Engineering Essentials (SEECx) applies interactive learning on edX[8]. The goal in the creation of the course was to make it as interactive as possible. The MOOC was launched in May 2017 as a course over nine weeks and offered several times.

Learning Goals

It is an intensive online course with interactive exercises that go beyond the learning experience of existing software engineering MOOCs. It has the following learning goals: Students get to know methods and techniques to develop software for different domains and platforms using agile techniques in the context of change. First, starting from a problem statement, instructors teach the participants how to analyze requirements and transform them into models based on textual analysis. Next, participants model multiple representations of the system consistently, understand and identify patterns. Finally, they map models to source code, integrate it into an app, and deliver it using build and release management techniques.

Course Structure

Two instructors and three to five tutors organize the course. It includes eight sections (comparable to lectures) covering eight major topics: project organization and management, software configuration management, object-oriented programming, requirements analysis, system design, object design, testing, build and release management. All sections consist of three to five units, each covering a concrete learning goal. Thus, the whole course includes 34 units. Table 6.7 shows the eight sections of the course and their content.

Units include a video with the theory of the topic and an example, followed by an exercise and a summary to reflect on the learned concepts. The duration of the videos ranges from 3 to 15 minutes (mean: 8.2 min). The videos are kept short to enable the students to apply the newly acquired knowledge in practice in the exercises. In addition to slide-based lecture videos, instructors added short clips with animations in an explanation style and real-world scenes into the video to make them more entertaining and rich in variety. Such videos make the thinking process visible and support cognitive apprenticeship [CBH91]. After each unit, there is a quiz to assess whether the students can remember and explain the learned concepts (levels 1 and 2 in Bloom's taxonomy). Students get immediate feedback on their responses and test their newly acquired knowledge. They can try each quiz two times in the course, so even if they failed initially, they

[8] https://www1.in.tum.de/seecx or https://www.edx.org/course/software-engineering-essentials

Section	Course content	Units	MC	SA	DD	Model.	Progr.
S1	Project Organization and Management	6	14	0	1	1	0
S2	Software Configuration Management	4	14	0	1	0	1
S3	Object-Oriented Programming	4	10	2	0	0	2
S4	Requirements Analysis	5	6	0	1	1	1
S5	System Design	4	6	2	0	0	1
S6	Object Design	3	6	0	3	0	1
S7	Testing	5	5	2	2	0	2
S8	Build and Release Management	3	13	0	1	0	1
	Sum	34	74	6	9	2	9

Table 6.7: The course schedule of SEECx is eight weeks long, corresponding to 8 sections, and consists of 34 units. Each section has at least one interactive exercise (model. = modeling, progr. = programming) in addition to quizzes (MC = multiple-choice, SA = short-answer text input, DD = drag & drop).

can have another look at the video and then score the total points in the assessment. This keeps the students motivated.

Each section also includes programming and modeling exercises that focus on higher cognitive skills. Those exercises assess if students can apply the previously obtained knowledge, analyze a problem, evaluate different solution strategies and create new solutions to given problems (level 3 - 6 in Bloom's taxonomy).

In order to pass the course, students have to achieve at least 60 % of all available points (400). By participating in the interactive exercises, students can earn up to 60 % of the total points (240), 30 points for each section. At the end of the course, students can participate in a final assessment which accounts for the remaining 40 % of the total points (160).

The instructors use different exercises to make the course interactive and rich in variety, following the learning goals. For example, they use multiple-choice, text input, and drag & drop questions to support learning goals on levels 1 and 2 of Bloom's taxo. In addition, they integrate interactive programming and modeling exercises.

All programming and modeling exercises are based on a common problem statement about the *University App*, also used for examples in the videos. This allows the students to recognize relationships between the topics (e.g., between the requirements and system design) and makes it easier to understand the context of the problem.

Interactive Programming Exercises

Students submit their exercise solutions directly in Artemis and receive immediate feedback through structural and behavioral tests. They can use this feedback to improve their solution iteratively. Artemis uses the LTI (Learning Tool Interoperability) interface of edX.

The online editor of Artemis includes assignments using interactive tasks and interactive diagrams. After each submission through the **Commit & Run Tests** button,

students' code is assessed automatically. The result is shown immediately, and the interactive tasks and diagrams are updated accordingly. In addition, students can see detailed, individual feedback on why their solution is wrong by clicking on the result. This helps to identify which tasks the students have already solved and which parts of their program do not work as expected.

Modeling Exercises

The ability to understand and create models is an important learning goal for software engineers. Therefore, modeling is an essential part of the course. However, it is challenging to correct models automatically because there are different correct solutions. Modeling is a creative activity, and students should not be limited in the creative thinking processes [KBC+17]. One learning goal of the course is that participants can review models given a set of quality criteria. Therefore, the instructors use the concept of peer reviews consisting of the following steps: (1) upload response, (2) learn to assess responses, and (3) assess peers.

Students first create a solution to a given problem and upload it. Then, they review sample solutions towards a given set of criteria to learn how to assess other solutions. Finally, they assess multiple other students' solutions to evaluate each model by at least three reviewers (other students). The final score is the average of three reviews. As a result, students receive valuable feedback about their models and can improve their modeling skills in the future.

While peer reviews lead to additional effort for students, they stimulate higher cognitive skills: by assessing other solutions, students reflect on alternative solution approaches and evaluate if they are correct concerning the given problem statement. This activity is beneficial, but it should be used carefully not to overload the students. There are two peer review exercises in the course: creating low-fidelity mockups for the university app and creating an analysis object model.

Project Work

In addition, the course also offers project work which allows the students to experience the complete software engineering process from analysis over the design to implementation, testing, and delivery. Table 6.8 shows the project work exercises related to the sections of the course. For example, a second problem statement allows students to apply and transfer their knowledge to a different problem domain. Project work focuses on Bloom's taxonomy's upper two cognitive skill levels, where students should create and evaluate a new solution to a problem. The project work starts in the fourth section and allows the students to evaluate how their own decisions, e.g., the requirements analysis, influence the system design and implementation.

Examples of project work exercises are the analysis of the problem domain, the design of the software architecture, sketches of user interfaces up to the implementation, testing, and delivery of a small app. The system cannot assess such exercises automatically because it should not limit the creativity of students. It is essential to motivate the

Section	Project Work
1, 2, 3	n.a.
4	Requirements elicitation and analysis of the problem domain
5	Design of the systems with a focus on subsystem decomposition and hardware-software mapping
6	Implementation of the system with a focus on the usage of design pattern
7	Testing of the system with automated unit tests
8	Setup of continuous integration and delivery

Table 6.8: Project work exercises in the sections of the SEECx course. Sections 1, 2, and 3 did not include project work, as students first had to get familiar with basic concepts and theories.

students to discuss their solutions with us and each other. Project work is optional for students. They can pass the course without active participation. Nonetheless, it is highly recommended to participate and allow the students to deepen their knowledge and gain practical experience.

Communication

MOOCs should foster social interaction and frequent contact between the students [KS19]. Therefore, it makes sense to use a chat for instant and direct communication instead of discussion forums to further improve the interaction between course participants and instructors. Many existing MOOCs rely on discussion forums, which are limited in interactivity. Instead, SEECx promotes the exchange with and between students based on a chat system. Instructors and tutors are active in the chat. Students can provide feedback and ask for help.

SEECx uses Slack as an instant messaging service because it has a lower entry barrier than discussion forums. Students can get in touch with each other and write direct messages to instructors and tutors if they need help. They ask questions more easily without having to pay attention to the exact wording and phrasing. Instructors add repeating questions to a question and answer page. The channels #questions, #general, and #feedback were the most important ones. In the channel #questions, students asked questions. In the channel #feedback, they stated how to improve the learning material. Instructors and tutors answer questions within one working day to keep the interactivity high.

Clear communication of learning goals, expectations, and deadlines is essential. The course description clarifies what the students can expect and what they must accomplish to pass the course.

6.1.5 Dissemination of Artemis

Interactive learning and Artemis have been applied in the four courses EIST, POM, PSE, and SEECx, and many other courses. Table 6.9 shows all courses on the Artemis

platform of the Technical University of Munich with the respective semester, the number of exercises, students, tutors, and instructors. Some of the courses, e.g., Introduction to Software Engineering before summer 2019, also used Artemis for homework exercises, but not necessarily interactive learning.

#	Course	Module	Sem	Ex	Stu	Tu	In
1	Introduction to Software Engineering	IN0006	SS16	4	676	20	10
2	Patterns in Software Engineering	IN2081	WS17	34	388	12	10
3	Software Engineering Essentials	IN1504	WS17	16	30	4	10
4	Software Engineering Essentials	IN1504	SS17	14	299	3	7
5	Introduction to Software Engineering	IN0006	SS17	7	1,066	23	2
6	Project Organization and Management	IN2083	SS17	1	271	9	10
7	Patterns in Software Engineering	IN2081	WS18	36	418	13	4
8	Software Engineering Essentials	IN1504	WS18	14	176	3	7
9	Introduction to Software Engineering	IN0006	SS18	33	1,472	48	9
10	Protein Prediction 1	IN2322	SS18	9	275	0	6
11	Project Organization and Management	IN2083	SS18	33	296	13	2
12	Hochschule München Software Engineering I	HM	WS19	11	94	2	2
13	Hochschule München Softwareentwicklung 1	HM	WS19	3	49	2	2
14	Patterns in Software Engineering	IN2081	WS19	38	431	13	5
15	Hochschule München Software Architektur	HM	SS19	10	79	2	2
16	Hochschule München Softwareentwicklung 2	HM	SS19	6	52	0	2
17	Protein Prediction 1	IN2322	SS19	16	174	0	7
18	Introduction to Software Engineering	IN0006	SS19	76	1,682	43	6
19	Project Organization and Management	IN2083	SS19	46	323	2	6
20	Software Engineering Essentials	IN1504	SS18	14	680	3	7
21	Introduction Programming Digital Health	SG160445	WS20	1	83	0	4
22	Praktikum: Grundlagen der Programmierung	IN0002	WS20	90	1,813	95	12
23	Grundlagen: Betriebssysteme	IN0009	WS20	17	998	3	11
24	Patterns in Software Engineering	IN2081	WS20	32	540	8	13
25	Grundlagen der Künstlichen Intelligenz	IN2062	SS20	2	859	2	5
26	Introduction to Software Engineering	IN0006	SS20	88	1,779	50	15
27	Protein Prediction I	IN2322	SS20	16	114	0	5
28	Programmierung für Sozialwissenschaftler	POL20100	SS20	2	11	0	2
29	Grundlagen: Algorithmen & Datenstrukturen	IN0007	SS20	12	1,304	3	6
30	Seminar JavaScript Technology	IN4790	SS20	7	16	0	1

#	Course	Module	Sem	Ex	Stu	Tu	In
31	Grundlagen: Betriebssysteme	IN0009	WS21	22	980	16	8
32	Nachqualifikation Inf-GY ST	-	WS21	12	48	2	2
33	KIU Fundamentals of Programming	KIU	WS21	56	150	3	9
34	Praktikum: Grundlagen der Programmierung	IN0002	WS21	70	2,043	85	11
35	Techniques in Artificial Intelligence	IN2062	WS21	4	766	7	4
36	Einführung in die Rechnerarchitektur	IN0004	WS21	15	1,217	12	4
37	Functional Programming and Verification	IN0003	WS21	38	977	22	6
38	Patterns in Software Engineering	IN2081	WS21	52	776	21	21
39	Betriebssysteme & HW Programmierung	IN0034	WS21	17	125	0	3
40	Software Engineering Essentials	IN1504	WS21	11	46	0	6
41	Software Engineering Business Applications	IN2085	WS21	18	296	5	3
42	IoT Remote Lab	EI78049	WS21	7	41	0	2
43	Nachqualifikation Inf-GY AD	-	WS21	8	48	2	4
44	Introduction to Programming Comp. Eng.	-	WS21	42	184	10	7
45	Repetitorium Einführung in die Informatik	IN0001	WS21	14	209	0	2
46	Introduction to Software Engineering	IN0006	SS21	92	2,097	68	5
47	Protein Prediction I	IN2322	SS21	1	85	0	5
48	Grundlagen: Algorithmen & Datenstrukturen	IN0007	SS21	13	1,610	23	4
49	Security Engineering	IN2178	SS21	32	228	0	4
50	KIU Functional Programming & Verification	KIU	SS21	42	80	0	6
51	Praktikum Data Science u. Masch. Lernen	-	SS21	11	19	0	3
52	IoT Remote Lab	EI78049	SS21	6	37	1	2
			Sum	**1,271**	**28,510**	**653**	**311**

Table 6.9: Overview of 52 courses on the Artemis instance at the Technical University of Munich with the semester (**Sem**) number of exercises (**Ex**), number of students (**Stu**), number of tutors (**Tu**), and number of instructors (**In**). Some courses (KIU and HM) have been offered to students from other universities.

The courses range from smaller ones with only 11 students to huge ones with up to 2,097 students. Some courses use more extensive and fewer exercises, e.g., IN0007. Others have many small exercises, e.g., IN0002. The number of instructors varies because some courses allow tutors to create exercises and therefore promote them to instructors. Artemis recently introduced the new role editor to overcome this issue, which has access rights between tutors and instructors. Sometimes, new instructors of follow-up courses also get instructor rights for previous instances to import exercises into the new course. Thus, there are only two to five actual instructors in most courses, even if the numbers in Table 6.9 have other values.

Interactive learning and Artemis are also used in other universities. Table 6.10 shows the universities that also contribute to the Artemis community, e.g., by participating in the open-source development or providing feedback. Artemis is used in five universities in Germany and five universities in Austria as part of the Codeability project, led by Michael Breu. In addition, one company uses Artemis for industry-related education. The Austrian universities share the same Artemis instance, while the German universities all use and administer their own Artemis instance.

#	University	URL	Main Administrator
1	Technical University of Munich	https://artemis.ase.in.tum.de	Stephan Krusche
2	University of Innsbruck	https://artemis.codeability.uibk.ac.at	Michael Breu
3	University of Salzburg	https://artemis.codeability.uibk.ac.at	Michael Breu
4	Johannes Keppler University Linz	https://artemis.codeability.uibk.ac.at	Michael Breu
5	University of Klagenfurt	https://artemis.codeability.uibk.ac.at	Michael Breu
6	Technical University Wien	https://artemis.codeability.uibk.ac.at	Michael Breu
7	University of Stuttgart	https://artemis.sqa.ddnss.org	Steffen Becker
8	University of Bonn	https://alpro.besec.uni-bonn.de	Alexander Von Trostorff
9	University of Passau	https://artemis.se2.fim.uni-passau.de	Christian Bachmaier
10	Karlsruhe Institute of Technology	https://artemis.praktomat.cs.kit.edu	Dominik Fuchß
11	University 4 Industry (company)	https://artemis.university4industry.com	Deryk Hopley

Table 6.10: Use of Artemis in universities and companies

Only the Artemis instance at the Technical University of Munich uses the setup with JIRA, Bitbucket, and Bamboo. All other Artemis instances in Table 6.10 use the setup with Gitlab in Jenkins. Some of them use internal user management and allow students to register. Others use integrations using Shibboleth[9] to connect users with the central university user management.

Notably, in the Karlsruhe Institute of Technology, Artemis replaced the existing system Praktomat (see Chapter 3 for an explanation) used for several years because it was not actively maintained anymore and lacked critical new features.

Table 6.11 shows the courses that have been offered in external universities. While the number of students is generally lower than at TUM, most courses have more than 50 students. All courses focus on the use of programming exercises and their automatic assessment. Most instructors use Java to teach the respective concepts in software engineering and programming. For example, two courses in Innsbruck and Salzburg teach Python, one course in Innsbruck teaches C/C++. The exercises range from four larger ones in Stuttgart, Passau, and Innsbruck to 120 small ones in Klagenfurt. This shows that instructors apply interactive learning and Artemis quite differently.

Artemis has also been used for online exams. Starting with the repeat exam period of the winter semester 2019/20, which was delayed due to the Corona pandemic, three courses have offered online examinations on Artemis, including programming exercises.

[9]Shibboleth is a standard for single sign-on that enables people from multiple institutions to access services shared in a circle of trust known as a federation. Shibboleth is one possible implementation of SAML (Security Assertion Markup Language): https://www.shibboleth.net.

#	University	Course	Language	# Ex	# Stu	# Tu	# In
1	University of Bonn	Algorithmen und Programmierung	Java	30	483	7	1
2	University of Stuttgart	Programmierung und Softwareentwicklung	Java	4	617	37	1
3	University of Passau	Software Testing	Java	4	60	1	1
4	University of Innsbruck	Einführung in die Programmierung	C/C++	38	70	1	2
5	University of Innsbruck	Software Architektur	Java	3	120	0	4
6	University of Innsbruck	Einführung in die Programmierung: Programmieren mit Python	Python	4	60	0	1
7	University Linz	Einführung in die Softwareentwicklung	Java	70	50	3	2
8	University of Klagenfurt	Einführung in die strukturierte und objektbasierte Programmierung	Java	120	13	1	3
9	University of Salzburg	Python Kurs	Python	12	120	0	6
10	University of Salzburg	Einführung in die Programmierung	Java	11	120	3	9
11	Karlsruhe Institute of Technology	Programmieren	Java	20	192	14	3
			Sum	**316**	**1,905**	**67**	**33**

Table 6.11: Artemis courses in external universities with the number of exercises (# Ex), number of students (# Stu), number of tutors (# Tu), and number of instructors (# In).

Table 6.12 shows the number of registered participants, how many students have started, and how many have submitted the exam.

#	Course	Module	Exam	Date	Participants	Started	Submitted
1	Functional Programming and Verification	IN0003	Repeat	04.07.20	240	240	194
2	Einführung in die Informatik	IN0001	Repeat	06.07.20	413	413	352
3	Praktikum: Grundlagen der Programmierung	IN0002	Repeat	08.07.20	173	173	136
				Sum	**826**	**826**	**682**

Table 6.12: Overview of all exams on Artemis in the winter term 2019/20 (WS1920)

In the exam period of the summer semester 2020, two courses have offered the final and repeat exam on Artemis, as shown in Table 6.13. The number of students who submitted started and submitted their exam is relatively high compared to exams written on paper. 90 % of the registered participants started the exam. From these, 98 % completed and submitted the exam. The general feedback about the use of Artemis for online exams was positive and encouraged the development.

#	Course	Module	Exam	Date	Participants	Started	Submitted
1	Introduction to Software Engineering	IN0006	Final	27.07.20	1,377	1,297	1,276
2	Networks for Monetary Transactions	IN2161	Final	03.08.20	564	482	477
3	Introduction to Software Engineering	IN0006	Repeat	28.09.20	452	393	380
4	Networks for monetary transactions	IN2161	Repeat	29.09.20	278	236	236
				Sum	**2,671**	**2,408**	**2,369**

Table 6.13: Overview of all exams on Artemis in the summer term 2020 (SS20)

In the exam period of the winter semester 2020/21, even more courses have used Artemis for final and repeat exams. Table 6.14 shows the 20 exams with more than 5,000 submitting students that have been conducted between January and April 2021.

The participation was relatively high again. 85 % of the registered participants started the exam. From these, 96 % completed and submitted the exam. In the largest exam, the final exam of the course IN0001, 1,595 students submitted the exam simultaneously. This shows that Artemis is scalable to many students. In the summer semester of 2021, the instructors of the course IN0002 expect an even larger final exam with more than 2,000 students.

#	Course	Module	Exam	Date	Participants	Started	Submitted
1	Fundamentals of Programming	KIU-FOP	Final	20.01.21	91	84	65
2	Introduction to Computer Science	KIU-I2CS	Final	22.01.21	113	109	102
3	Fundamentals of Programming	KIU-FOP	Repeat	03.02.21	50	46	42
4	Introduction to Computer Science	KIU-I2CS	Repeat	05.02.21	51	50	50
5	Einführung in die Informatik 1	IN0001	Final	13.02.21	1,763	1,606	1,595
6	Scientific Data Processing	SG8100044	Final	19.02.21	58	44	44
7	Microprocessors	IN2075	Final	23.02.21	53	39	38
8	Advanced Topics of Software Engineering	IN2309	Final	24.02.21	220	167	153
9	Functional Programming and Verification	IN0003	Final	26.02.21	796	629	591
10	Virtualization Techniques	IN2125	Final	01.03.21	42	32	32
11	Program Optimization	IN2053	Final	02.03.21	97	67	66
12	Computer Architecture and Networks	IN2189	Final	03.03.21	56	47	47
13	Parallel Programming Systems	IN2365	Final	04.03.21	33	19	18
14	Patterns in Software Engineering	IN2081	Final	05.03.21	454	385	368
15	Einführung in die Rechnerarchitektur	IN0004	Final	06.03.21	1,001	865	819
16	Praktikum: Grundlagen der Programmierung	IN0002	Repeat	26.03.21	258	225	220
17	Einführung in die Informatik 1	IN0001	Repeat	29.03.21	395	326	321
18	Functional Programming and Verification	IN0003	Repeat	31.03.21	413	331	326
19	Advanced Topics of Software Engineering	IN2309	Repeat	06.04.21	81	51	47
20	Einführung in die Rechnerarchitektur	IN0004	Repeat	06.04.21	337	261	243
				Sum	**6,362**	**5,383**	**5,187**

Table 6.14: Overview of all exams on Artemis in the winter term 2020/21 (WS2021)

The administrators of one external university, the University of Bonn, have reported using Artemis for four online exams. Table 6.15 shows how many students have been registered and have submitted.

#	Exam	Type	Registered	Submitted
1	Exam 1.1	Final	227	202
2	Exam 1.2	Final	32	25
3	Exam 2.1	Repeat	99	76
4	Exam 2.2	Repeat	27	10
		Sum	385	313

Table 6.15: Exams in the University of Bonn

The administrators of external universities reported that Artemis was perceived well by the instructors and students. Therefore, they plan to keep using it in the subsequent semesters.

6.2 Results

We performed several evaluations in EIST, POM, PSE, and SEECx. We summarize the most important evaluations and results here. More detailed evaluations and results can be found in the individual publications in Chapter 8 which are cited in the relevant subsections.

Table 6.16 shows that the three university courses POM, EIST, and PSE received very good grades in the standardized evaluations performed by the student council. The average grades of other large university courses with traditional teaching approaches are typically lower between 2.5 and 3.0.

#	Course Title	Short	Semester	Course Grade	Instructor Grade
1	Project Organization and Management	POM	SS17	1.7	1.6
2	Introduction to Software Engineering	EIST	SS19	1.9	1.6
3	Patterns in Software Engineering	PSE	WS20/21	1.4	1.2

Table 6.16: Overview of the grades in the evaluations in the three courses based on interactive learning

Students appreciate the interactivity of the courses, the mix between theory and exercises, and the possibility of receiving a bonus for the final exam. In the free-text comments, many students stated the following or similar comments:

- "I think Artemis is a very good and helpful tool."
- "Really a great course. I think you are doing really well! Especially the interactive exercises on Artemis, where you can already collect points for the exam, encourage you to learn continuously! In my opinion, the best course at TUM by far."
- "Best course so far in master regarding practical relevance."
- "Best lecture in the Informatics faculty."

The following subsections are structured according to the hypotheses stated in Chapter 1. We describe the experiments and results for four of the five hypotheses. While the implementation of adaptive learning has started in Artemis, we did not use it in courses yet and have therefore not evaluated H5. It will be part of future work.

6.2.1 H1: Scalability

The use of Artemis in very large university courses with up to 2,000 students and very large exams with more than 1,500 participants has shown that interactive learning is scalable. For example, we have reported on multiple interactive modeling exercises during lectures in the course EIST 2019 with more than 1,000 participants in [KvFRB20]. In the course EIST 2021, there have been up to 1,673 simultaneous participants in the offered in-class exercises.

The deployment with multi-server nodes for horizontal scaling, as shown in Figure 5.28, allows to scale the approach technically. In addition, we have shown in multiple case studies that the teaching philosophy interactive learning is applicable to very large audiences and improves the interactivity of lecture-based courses significantly. By using

Artemis, instructors can interact with each student on an individual level, even in large-scale courses.

6.2.2 H2: Engagement

Figure 6.1 shows the participants per lecture in a traditional course compared to the participants per lecture in a subsequent instance of the same course one year later, when the course was based on interactive learning as published in [KSBB17]. The participation rate increased significantly by 165 %, from 17 % in the last four lectures of the traditional course to 46 % in course's last four lectures based on interactive learning [KSBB17].

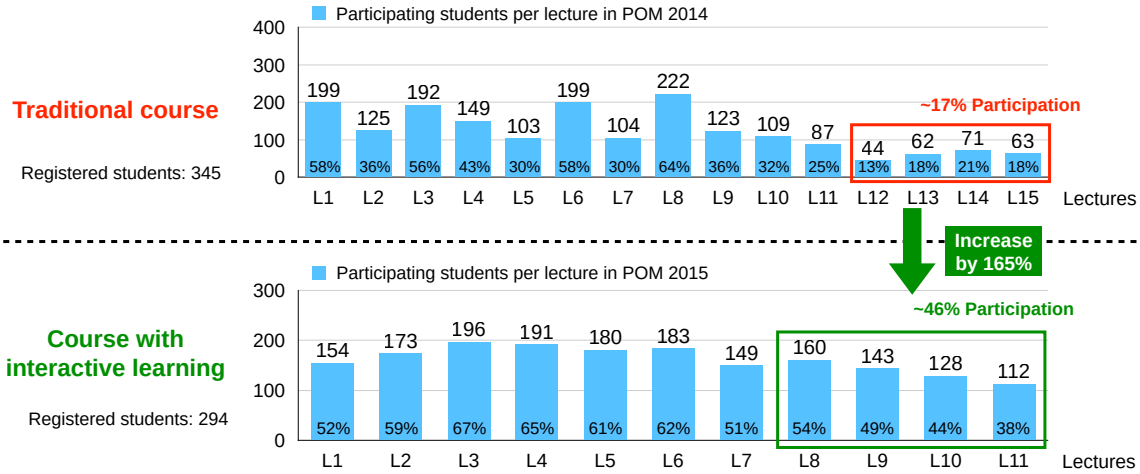


Figure 6.1: Participation in a traditional course (top) and a course based on interactive learning (bottom) with a 165 % increase in the last four lectures.

We observed similar changes in the other courses, but due to the lack of participation numbers before the introduction of interactive learning, we have not compared them. This means that the students are more motivated to participate in lectures and engage with the content and exercises in courses based on interactive learning.

We tried to measure the students' motivation in additional studies with pre-tests and online questionnaires in the course POM. We wanted to use regression analysis to find correlations between participation, motivation, and students' learning outcome. However, we faced significant challenges because we could not control the different variables that influence the students' engagement in the actual course (field study), and there have been too many different situations that led to outliers.

The sample size of students who participated in the pre-test, the online questionnaires, and the post-test (the final exam) was below 50 % and did not represent the whole course population. For example, some students, e.g., did not seriously participate in the final exam because the course was elective, which led to a negative learning gain (delta between post-test and pre-test). Other students were not satisfied with their performance during the exam and handed in early or crossed out their solutions. The variance in the measured motivation in the online questionnaires was very low (between 70 % and 100 %) because students typically only participate in voluntary questionnaires if they are motivated and tend to overestimate their motivation. The conclusion was

that the course was too heterogeneous. There were too many issues in the evaluation to find anything meaningful. Therefore, we used different evaluations with fewer variables.

6.2.3 H3: Learning Outcome

We conducted quantitative evaluations to investigate the learning outcome in three university courses POM, EIST, and PSE. The quantitative measurement focused on the relationship between exercise performance and the final exam score of the students. We calculated the exercise points in the courses for each participating student and correlated them with the final exam grade. We grouped the students after the relative exercise points into five categories (0 %-20 %, 20 %-40 %, 40 %-60 %, 60 %-80 %, 80 %-10 %), calculated the final exam score for each category and computed the correlation using a χ^2 test. Details of the concrete methodology for the POM and PSE course can be found in [KSBB17].

Figure 6.2 shows the correlation between exercise performance (x) and average exam score (y) in three different courses POM, EIST, and PSE. The average score in the exam without a bonus significantly increases from left to right.

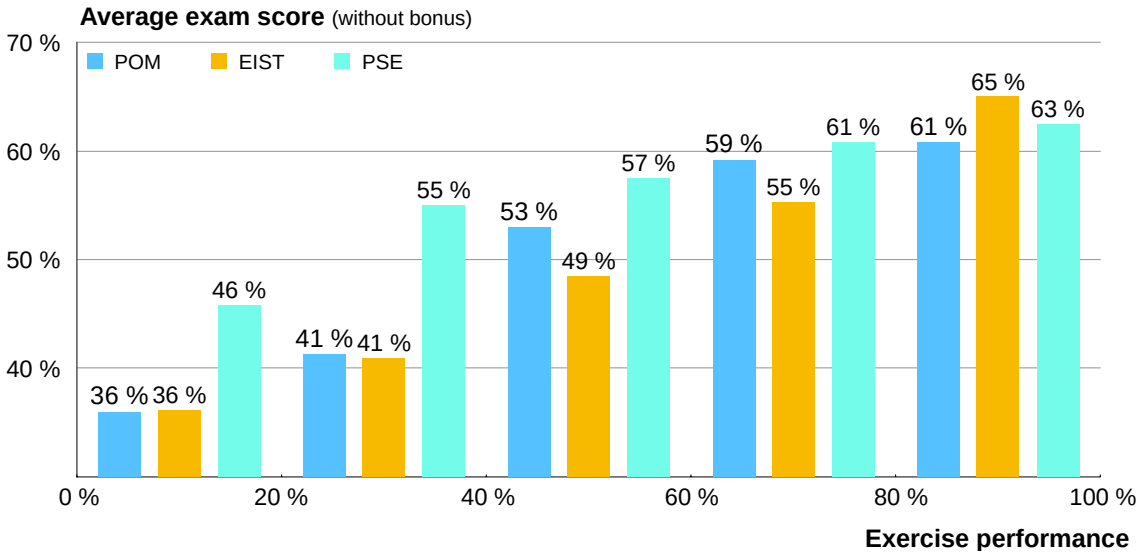


Figure 6.2: Correlation between exercise performance (x-axis) and average exam score (y-axis) in three different courses

The results show a correlation between two variables but do not state anything about a causal relationship because not all the variables such as motivation and students' existing knowledge have been measured. It could, e.g., be that motivation has a stronger influence on the exam score and that motivated students also participate more often in exercises. The results only indicate that higher exercise performance is related to a higher average exam score. However, online questionnaires in these courses confirm the findings [KSBB17].

Table 6.17 shows the details of the χ^2 test of the course POM with 294 students who completed the exam. 75 students had an exercise performance between 0 % and 20 %. They reached an average exam score of 36 %, corresponding to a grade point average of

3.9 (sufficient). Thus, 56 % of those students passed the exam, while 44 % failed. Nine students were in the highest category with an exercise performance between 80 % and 100 %, reaching a grade point average of 2.4 (good to satisfactory) and 1.4 (very good to good), including the bonus. All nine students passed the exam. The data shows a moderate correlation.

Exercise performance	Students	Average exam score	GPA	GPA with bonus	Passed	Failed
0 % - 20 %	75 (26 %)	36 %	3.9	3.9	56 %	44 %
20 % - 40 %	66 (22 %)	41 %	3.6	3.4	73 %	27 %
40 % - 60 %	88 (30 %)	53 %	2.9	2.3	90 %	10 %
60 % - 80 %	56 (19 %)	59 %	2.5	1.7	98 %	2 %
80 % - 100 %	9 (3 %)	61 %	2.4	1.4	100 %	0 %
Total	294	47 %	3.2	2.8	79 %	21 %

Table 6.17: Correlation details in the course POM (GPA = grade point average, in Germany, lower grades are better)

Table 6.18 shows the details of the χ^2 test of the course EIST with 1,128 students who completed the exam. 223 students had an exercise performance between 0 % and 20 %. They reached an average exam score of 36 %, which led to a grade point average of 4.2 (sufficient). Only 44 % of these students passed the exam, 56 % failed it. On the other range of the table, 379 students had a high exercise performance between 80 % and 100 %. They had an average exam score of 65 %, corresponding to a grade point average of 2.5 (good to satisfactory). The grade point average with a bonus was 1.6 (very good to good). With 98 %, almost all students passed the exam, and only 2 % did not pass. This course EIST showed the highest correlation between exercise performance and average exam score.

Exercise performance	Students	Average exam score	GPA	GPA with bonus	Passed	Failed
0 % - 20 %	223 (20 %)	36 %	4.2	4.2	44 %	56 %
20 % - 40 %	108 (10 %)	41 %	3.9	3.8	63 %	37 %
40 % - 60 %	111 (10 %)	49 %	3.5	3.2	75 %	25 %
60 % - 80 %	307 (27 %)	55 %	3.1	2.4	93 %	7 %
80 % - 100 %	379 (34 %)	65 %	2.5	1.6	98 %	2 %
Total	1,128	53 %	3.2	2.7	80 %	20 %

Table 6.18: Correlation details in the course EIST (GPA = grade point average, in Germany, lower grades are better)

Table 6.19 shows the details of the χ^2 test of the course PSE with 324 students who completed the exam. 146 students (45 %) had an exercise performance between 0 % and 20 %. We assume the following reasons for this: (1) PSE is an elective course mostly for

master students or advanced bachelor students who typically work as working students besides the company. Not all the students have enough time to work on exercises. (2) In the observed instance of PSE in WS2015/16, the instructors did not offer a bonus to improve the exam grade. (3) Most exercises in PSE are programming exercises. Students knew that such exercises could not be included in the paper-based exam. Therefore, the motivation to participate was low. (4) Students knew that the course offers a relatively easy exam focusing on knowledge instead of the application. In total, 90 % of the students passed the exam.

81 students had an exercise performance between 20 % and 40 % and scored significantly better than the students in the last category. They received an average exam score of 55 % or a grade point average of 2.5 (good to sufficient). All 38 students with an exercise performance between 0 % and 20 %, and all 16 students with more than 80 % passed the exam. However, the average score and grade point average did not significantly improve compared to the students in the lower exercise performance categories. PSE showed the weakest correlation between exercise performance and average exam scores among the analyzed courses. However, the correlation is still considered moderate.

Exercise performance	Students	Avgerage exam score	GPA	Passed	Failed
0 % - 20 %	146 (45 %)	46 %	3.1	81 %	19 %
20 % - 40 %	81 (25 %)	55 %	2.5	96 %	4 %
40 % - 60 %	43 (13 %)	58 %	2.3	95 %	5 %
60 % - 80 %	38 (12 %)	61 %	2.1	100 %	0 %
80 % - 100 %	16 (5 %)	63 %	2.0	100 %	0 %
Total	324	54 %	2.6	90 %	10 %

Table 6.19: Correlation details in the course PSE (GPA = grade point average, in Germany, lower grades are better)

Table 6.20 shows the statistical values for the correlation shown in Figure 6.2. The χ^2 are all highly significant with very small p-values. The computed values for $CramerV$ and the adjusted contingency coefficient show a moderate correlation between the two values for POM and PSE [Cra46]. They show a higher correlation for EIST, which also had the highest amount of participants.

Course	POM	EIST	PSE
Participants	294	1,128	324
$\chi^2(0.99; 16)$	83	547	48
p	$5.8e^{-11}$	$2.2e^{-16}$	$4.7e^{-05}$
$CramerV$	0.265	0.348	0.192
Adjusted contingency coefficient	0.523	0.639	0.402

Table 6.20: Statistical values of the correlation between exercise performance and average exam score

To further analyze the learning outcome, we carried out a quasi-experiment with post-testing of two student groups, i.e., students who took EIST in 2018 and students who took EIST in 2019, by comparing their scores in the modeling tasks of the final exam [KvFRB20]. Both course instances in 2018 and 2019 had the same learning goals, the same course schedule with the same content, and the same exercise structure except for modeling exercises: In 2018, EIST did not use the interactive learning method for modeling exercises and instead relied on practicing modeling only in homework. In 2019, EIST used the interactive learning method and introduced in-class modeling exercises. Thus, in terms of the quasi-experiment, the interactive learning method is the intervention. Apart from that, there were no substantial differences in other variables.

The control group is comprised of the 2018 students, the experimental group of the 2019 students. We did not execute a pre-test. We assumed that students from both groups had similar knowledge regarding modeling before taking part in the EIST course, mainly because most of both student groups were second-semester bachelor students, with both groups following the same curriculum. In both years, the course was attended by over 1000 students. Thus, a normal distribution of the results can be assumed. Both exams included five similar modeling tasks:

1. **Functional model**: Create a UML use case diagram based on a given problem statement (easy)
2. **Structural model**: Create an analysis object model using a UML class diagram based on a given problem statement (medium)
3. **Dynamic model**: Create an UML activity diagram (2018) / UML communication diagram (2019) based on a given problem statement (medium)
4. **Architecture**: Create a UML communication diagram of an architectural style (2018, medium) / model the architecture based on a given problem statement using a UML component diagram (2019, hard)
5. **Model refactoring**: Analyze an existing model, propose a model refactoring and explain the reasoning (easy)

In the post-test, we compared these five modeling assignments in two-sample one-tailed t-tests to evaluate whether the 2019 students performed significantly better than the 2018 students. For all model tasks, the null hypothesis is that the 2019 students performed less or equal than the 2018 group with a significance level of $\alpha = 0.01$. 1128 students completed the exam in 2018, 1225 completed the exam in 2019. We calculated the average score and standard deviation as relative values to make the results comparable (two tasks differed by one point).

Figure 6.3 shows the average scores per exercise and improvements between the exercises in the 2018 exam with 1,128 participants (orange, above) and the 2019 exam with 1,125 participants (blue, below). Each exam assignment included a UML modeling task. The 2018 average scores (orange, above) correspond to the control group without interventions, the 2019 average scores (blue, below) correspond to the experimental group with the interactive learning intervention.

Except for the exercise about refactoring, where students received 4 % fewer points on average in 2019 (78 %) than in 2018 (81 %), the students performed better in the SE1

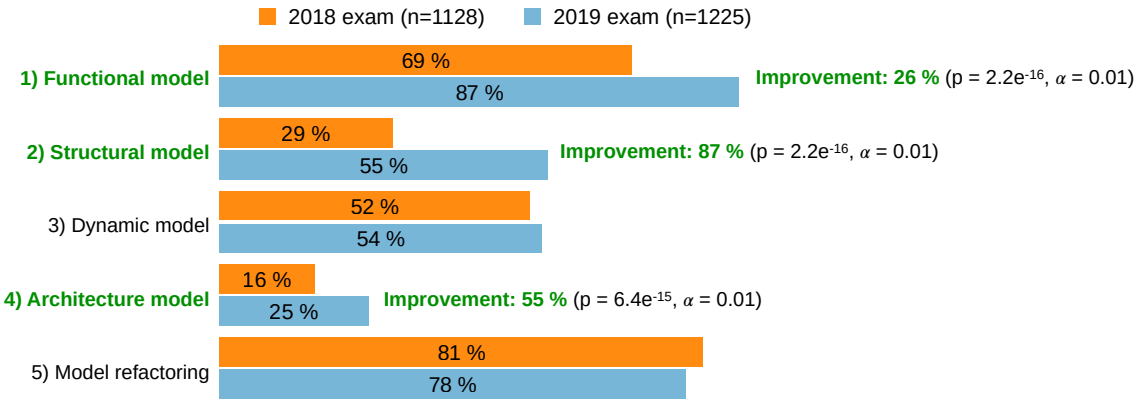


Figure 6.3: Average score of five similar modeling assignments in the 2018 exam vs. the 2019 exam. In three of the five assignments, the average score significantly increased by up to 87 %.

2019 exam than in the 2018 exam in the analyzed modeling tasks. For the functional model, the average score is 87 %, 26 % higher than in the 2018 exam, where students received 69 % of available points on average. In the structural model, students received 55 % of the points on average, 87 % more than in 2018, where they received 29 % on average. The results for the dynamic model are 4 % higher (54 %) than in 2018 (52 %). The average score of the architecture exercise is 55 % higher in 2019, with 25 % compared to 16 % in 2018.

Model	$\overline{x}_{\mathbf{2018}}$	$\overline{x}_{\mathbf{2019}}$	$\sigma_{\mathbf{2018}}$	$\sigma_{\mathbf{2019}}$	**p**
Functional	0,689	0,867	0,334	0,237	2,2e^{-16}
Structural	0,293	0,549	0,265	0,310	2,2e^{-16}
Dynamic	0,519	0,538	0,327	0,365	0,09
Architecture	0,161	0,249	0,273	0,278	6,4e^{-15}
Refactoring	0,812	0,776	0,281	0,208	0,99

$n_{2018} = 1128$, $n_{2019} = 1225$ $\alpha = 0,01$

Table 6.21: Statistical values of the t-tests of the EIST 2018 and EIST 2019 results underline that the 2019 students were significantly better for functional, structural, and architecture models.

To evaluate the significance of the results, we performed a two-sample one-tailed t-test with a significance level of $\alpha = 0.01$. The results of the t-tests per exercise are depicted in Table 6.21. If the p-value for the exercise was below alpha, we rejected the null hypothesis and considered the results from 2019 as significantly better than 2018. For the functional and structural models, we calculated a p-value of 2,2e^{-16} and for the architecture model a p-value of 6,4e^{-15}, which means that students were significantly better in the 2019 exam for the same model type in 2018. For the dynamic model, we calculated a p-value of 0,09. For the refactoring, the p-value was 0,99. This means that the 2019 results were not significantly better than in 2018.

Additionally, we performed an online questionnaire in EIST 2019 with 954 participants (response rate: 68 %). The findings of the questionnaire can be found in [KvFRB20].

They support the findings of the quasi-experiment and let us conclude that interactive learning improves students' learning outcomes. This is also based on the higher engagement caused by interactive learning, as found in H3.

6.2.4 H4: Grading Effort and Feedback Quality

In addition to the engagement and learning outcome, we also investigated how the grading effort and feedback quality change using Artemis. Due to the automatic assessment of programming and quiz exercises, the effort for those exercises was significantly reduced. The gained time can be spent to improve the wording of the quiz questions and to design test cases with valuable feedback.

For modeling and text exercises, the assessment cannot be fully automated. Artemis implements a semi-automatic assessment approach based on supervised machine learning. We wanted to know how much grading effort can be reduced and how this affects the quality and consistency of the feedback. Instructors activated the semi-automatic assessment approach in several exercises on Artemis. During the assessment process, Artemis tacked whether feedback was created manually or automatically proposed. A third possibility was that proposed feedback was automatically adapted, e.g., to fix typos, extend it with additional information, or change it because it was wrong.

We retrieved the classification of the reviews from the Artemis database using SQL queries. Multiple researchers verified the correctness. More details about the results for text exercises can be found in [BKB21]. Figure 6.4 shows six exemplary exercises: three modeling exercises and three text exercises together with the grading effort divided into manual effort (orange, right) and automatic effort (green, left). Feedback which was proposed automatically and then adjusted manually is shown in the middle in yellow (also compare the related activities "assess automatically", "assess manually", and "adjust assessment" in Figure 5.11).

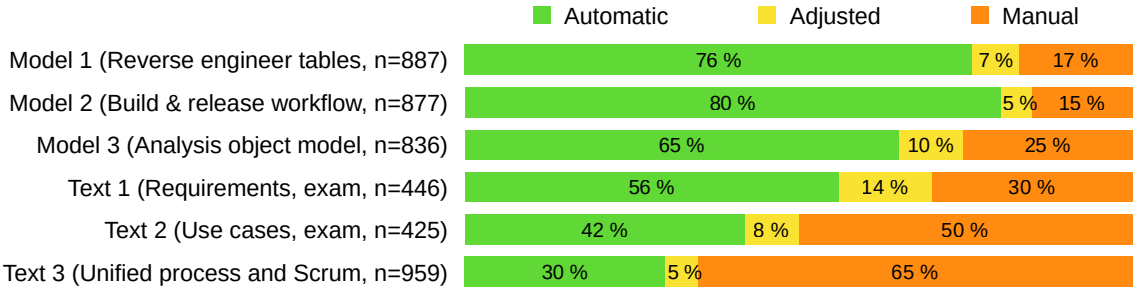


Figure 6.4: The automatic and manual assessment distribution for six exemplary text and modeling exercises that have used the semi-automatic assessment workflows based on supervised machine learning. The rate of automatic (green, left), adjusted (yellow, middle), and manual (orang, right) assessments is shown. Depending on the exercise type, the automatic rate varies between 30 % and 80 %. Adjusted feedback includes extensions with additional explanations and changes due to wrongly assigned feedback.

The results show that the semi-automatic assessment workflows leads to less effort manual effort. Up to 80 % of the feedback could be proposed automatically for modeling exercises, up to 65 % for text exercises. The adjustment rate is relatively low for most

exercises. A manual inspection revealed that many adjustments only included typo fixes and extensions.

To measure the quality of the feedback, we analyzed the number of complaints about these exercises and compared them with exercises that have been graded entirely manually. We found that, on average, the complaints have been lower. However, this result should be taken with caution, as several aspects may influence whether or not a student complains.

Artemis allows students to rate the feedback. The instructors asked students to rate their received feedback on a 5-star scale. The students rated a total of 396 reviews out of 10,240 total assessments [BKB21]. Artemis presents the rating input underneath the feedback and asks, "How useful is the feedback for you?" In the study, 85 % of the ratings were either 1-star or 5-star ratings. Students with semi-automatic feedback were more likely to give a 5-star rating (72 %) when compared to students who received manual feedback (62 %). On the same page, semi-automatic feedback received 1-star ratings less often (14 %) than manual feedback (25 %). On average, students giving a 5-star rating (92 % and 89 %, respectively) had better scores than students giving 1-star ratings (69 % and 66 %, respectively).

Comparing purely manual assessments vs. semi-automatic assessments shows anecdotal evidence that the perceived quality has at least the same quality or is even higher for semi-automatic assessments [BKB21]. However, this also has to be analyzed further because the number of ratings was relatively low and influenced by the student's satisfaction with the score.

In the end, the automatic assessments are based on the manual input of the reviewers. If the manual reviews have lousy quality, Artemis cannot magically produce high-quality automatic feedback proposals. Therefore, it is also essential to apply reviewer training and regularly assess the average rating and the number of complaints of individual reviewers.

From the results in the evaluations, we can conclude that semi-automatic assessment based on supervised machine learning can significantly reduce the grading effort while increasing the perceived quality, which supports H4.

6.3 Threats to Validity

The evaluations and findings shown in this habilitation and the accompanying publications have limitations that we want to summarize in the following.

Internal validity

The evaluations do not measure all variables that could lead to higher engagement and an improved learning outcome. For example, existing knowledge, intrinsic motivation, the exact wording of exercises and exam questions, and other external factors might influence how students perform a task in an exam and how much they learn. In addition,

exam results are only indications of the learning outcome of students. For example, a good exam result does not necessarily mean that the student has understood the concepts.

The internal validity of the results in the evaluations might therefore be limited [RHRR12]. However, the exercises and exam questions are constructively aligned and focus on higher cognitive skills, which are typically perceived as more challenging to receive high scores without the obtained skills. Online questionnaires and data analysis also support the findings.

The author of this habilitation and co-authors of the associated publications have been involved in organizing several courses and might have influenced the empirical evaluation. However, they tried to separate the research and instructor perspectives.

External validity

EIST, PSE, POM, and SEECx are specific examples of courses at the Technical University of Munich. The results might not be generalizable to other institutions' courses due to different study programs and study regulations. However, when discussing Artemis with instructors of other universities, we received much positive feedback. Thus, the first observations confirm these results in other environments. However, more thorough evaluations are necessary to confirm the results.

EIST is a mandatory course with UML modeling as a learning goal offered to many students, who may differ in their field of studies and their previous experiences. We assume that the interactive learning method can be successfully applied in other software engineering courses and is generalizable. However, other study programs and regulations might make it challenging to adopt the approach.

Construct validity

The validity of the used questionnaires might be affected by the wording of the questions or because students like getting feedback, which does not necessarily improve their learning outcome. We carefully designed the questions, used Likert scales as answer options, and asked multiple researchers to review the wording to limit the influence. The measures in the quantitative experiments and the data analysis of the experiments support the findings of the questionnaires.

The validity of ratings might be affected by the question's wording and by the score that the students received. Students with a higher score are more satisfied and less likely to complain about the quality of the feedback. However, a good rating does not necessarily mean that the feedback had a good quality. Another limitation could be the fact that students like the approach of getting feedback. Ratings only measure the perceived quality, which is subjective. We can only infer the quality based on the ratings. Therefore, we consider the findings on the quality of the feedback in H4 as anecdotal evidence. It has to be investigated in more detail in the future.

Chapter 7

Conclusion

"Learn continually - there's always 'one more thing' to learn!"

— Steve Jobs

With interactive learning and Artemis, we have shown that individual interaction between instructors and students is possible even in larger courses. We established a teaching philosophy that allows instructors to teach small pieces of content in short, iterative, and agile cycles. Students practice and reflect on the taught concepts in real-time based on individual feedback. In addition, interactive learning encourages creativity and does not limit students to sample solutions. In this chapter, we summarize the contributions of this habilitation and propose future work.

7.1 Contributions

This habilitation includes five significant contributions.

1. The teaching philosophy **interactive learning**
2. The teaching platform **Artemis**
3. **Application** of both in multiple courses and case studies
4. **Dissemination** of both in other universities
5. **Empirical evaluations** based on five hypotheses

Interactive learning is a scalable and adaptive teaching philosophy based on constructive alignment. Instructors can embed it into the course syllabus It puts the interaction with students into the core of the educational activities. It integrates aspects of team-based learning and creativity to stimulate problem-solving skills and soft skills. Instructors integrate exercises into lectures to activate students and to increase their engagement with the taught concepts. Individual feedback and reflection allow to prevent misconceptions and improve the competencies of the students.

Artemis is a teaching platform based on interactive learning. It integrates many modern features, particularly the automatic assessment of programming and quiz exercises that provide immediate feedback to students and allow them to iteratively and

incrementally improve their knowledge. It uses supervised machine learning to enable a semi-automatic assessment approach that reduces grading and increases consistency, quality, and fairness. An integrated training process for reviewers based on example submissions and example assessments ensures that reviewers have enough knowledge to properly assess submissions and provide feedback. Double-blind grading improves the overall fairness in larger courses and prevents unconscious bias due to personal relationships or other reasons, enhancing the equity, diversity, and inclusion of the whole teaching platform. Structured grading instructions that can be dragged and dropped include predefined and standardized feedback, making it easier and faster to provide feedback. Ratings and a leaderboard motivate the reviewers to participate in the grading process and provide high-quality feedback, valuable for the learning process.

Artemis includes an exam mode that allows instructors to use the same activities as in the course before (constructive alignment). Team-based exercises enable students to work with peers and to improve their soft skills. Lecture units can embed live streams, recordings, exercises, attachments, and text explanations next to the lecture content. Questions and answers directly next to an activity allow students to capture the context of their inquiries, help tutors or other students to answer these questions more efficiently, and prevent duplications.

Application: We applied interactive learning and Artemis in multiple instances of large courses as case studies in the context of computer science, in particular software engineering. Up to 2,100 active students participated in the 2nd-semester bachelor course *Introduction to Software Engineering*. The course *Project Organization and Management* included up to 400 active students in a heterogeneous distribution, including bachelor students who had to take the course and master students who participated voluntarily.

Patterns in Software Engineering attracted up to 600 active students as a purely elective course for bachelor and master students. We offered the massive open online course *Software Engineering Essentials* multiple times on the edX platform and reached many worldwide students. In the university evaluations, students consistently rated these courses with excellent grades. Many reported that these were the best courses in their studies and appreciate the interactive style with the individual interaction and feedback.

Dissemination: Another contribution is the use of Artemis by other instructors and universities. Multiple instructors at the Technical University of Munich have chosen to use Artemis after students have asked them. In addition, we helped educators from other German and Austrian universities to set up Artemis in their environment. As a result, 63 courses across ten universities used Artemis and interactive learning to benefit from automatic assessments and individual feedback. More than 30,000 students used Artemis in these courses. In addition, 8,500 students took their exams on Artemis in 31 online examinations with realistic exercises based on constructive alignment.

Empirical evaluations: We empirically evaluated interactive learning and Artemis in the courses mentioned above as case studies based on five hypotheses regarding scalability (H1), engagement (H2), learning outcome (H3), and grading effort and feedback quality (H4). We started with the implementation of adaptability (H5) but could not implement all objectives and evaluate it within this habilitation scientifically. The analysis of the results shows promising findings:

1. Interactive learning is scalable to courses with 1,500 participants submitting their exercise solutions simultaneously (H1).
2. It increased the student engagement in the last four lectures of one course by 165 % compared to a previous traditional instance of the same course (H2). More students are motivated to participate in exercises, and more students complete the exercises.
3. An analysis between the exercise performance and the exam results shows moderate to high correlations in the three analyzed courses (H3).

Combining these results means that more students will receive better results in the examination. Therefore, as the examinations followed the constructive alignment approach, we can conclude that interactive learning improves the learning outcome.

Another evaluation found that interactive learning improved students' learning outcomes in modeling exercises in one course by up to 87 % when comparing the exam results with the previous course without interactive learning (H3). In addition, the semi-automatic grading process in Artemis reduced the grading effort by up to 80 % in text and modeling exercises (H4).

7.2 Future Work

Future research directions include integrating learning analytics to let students reflect on their performance and allow instructors to identify whether students have understood specific topics. Instructors could use the information to identify knowledge gaps and misunderstandings of the students. They could clarify those issues in upcoming lectures or announcements. Students can use learning analytics to identify the competencies in which they still have deficiencies.

Artemis will provide instructors with the possibility to explicitly model the intended acquisition of competencies (learning goals) of a course and link it to the interventions of the learning platform (e.g., watching videos, reading texts, answering questions, solving tasks). Artemis can use this information to create learning paths. For example, students can voluntarily activate a learning assistant that analyzes their interactions with the learning platform (indirect conclusions through learning analytics and direct findings through analysis of the answers). Based on this, the learning assistant gives students feedback on their learning progress. The instructor also receives feedback on the progress of the students and the effectiveness of the interventions offered. Students and instructors can use this information to understand better which learning steps and activities are beneficial. This feedback creates control loops to improve and supplement teaching and learning, make the processes more adaptive, and individualize the learning experience.

Learning paths and exercises with different difficulty levels would build the first step towards adaptive learning. Artemis could create a student profile based on pre-tests (e.g., quizzes) and previously completed courses. User profiles could enable the automatic configuration of individual paths through the course content based on the unique skills of one student. It could also help students to deepen the content that is required as a prerequisite for a course. Artemis would then automatically propose the concepts

and exercises in which the students need to practice. Instructors could create multiple versions of exercises with different difficulty levels (e.g., easy, medium, difficult). Artemis could propose more straightforward exercises for weaker students and more challenging exercises for stronger students to improve their motivation. However, one issue would be the influence of the assessment on the students' grades because students could consider exercises with different difficulty levels unfair.

Further improvements in the semi-automatic assessment of modeling and textual exercises would include reusing knowledge of existing exercises from previous course instances or other instructors of other universities. This knowledge exchange would also involve a synchronization between multiple universities in an exercise repository. One challenge then would be to store the knowledge about correct and wrong solution aspects in exchangeable graphs. Another one would be how much knowledge would stay valid when instructors include minor variations, e.g., in the formulation or the used problem statement.

Artemis includes team-based exercises. However, we did not have the chance to study the effect of such exercises on the technical and soft skills of the students using Artemis. An empirical evaluation of the benefits of team-based exercises in larger courses would be interesting, particularly when it comes to the diverse background of students and the collaboration between team members. For example, would it make sense to force all team members to contribute to the team solution towards an exercise, or could the students circumvent such rules? How is the influence of pair programming or pair modeling in such cases?

From the technical perspective, it is desirable to migrate Artemis from a monolithic architecture to microservices and micro frontends, allowing a more lightweight and flexible configuration during the deployment. Microservices are also known as the microservice architectural style. They structure an application as a collection of services that are easier to maintain and test, loosely coupled, and independently deployable [New15]. Thus, the microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications.

Micro frontends deliver the same improved flexibility and maintainability to browser-based client applications that microservices provide for server applications [Gee20]. By adopting the micro frontends approach and designing the Artemis client application as a system of features, we can deliver faster feature development and easier upgrades. Figure 7.1 shows an overview of the possible decomposition of features into smaller client applications (i.e., micro frontends) and microservices on the server-side that all connect to the same database.

Kubernetes clusters could improve the automatic scaling and performance when many students simultaneously use Artemis's new microservice-based architecture [VSTK18]. These deployment improvements could further enhance the robustness in situations with multiple hundreds of submissions simultaneously (e.g., at the end of an online exam with 2,000 participants). They would also simplify the operation of multiple server nodes. Furthermore, by splitting the features of Artemis into microservices, only the services that are used often could be started numerous times. One challenge will be synchronizing

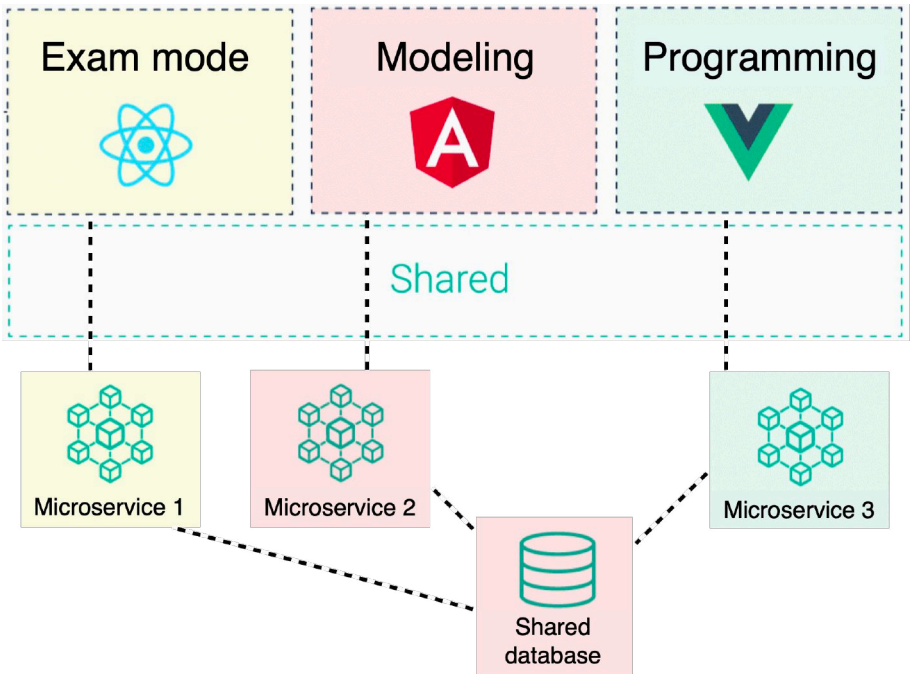


Figure 7.1: Overview of a potential future Artemis architecture based on microservices and micro frontends with exemplary features in the client (top) and the server (bottom).

services based on scheduled tasks in the emerging distributed systems architecture to avoid race conditions and illegal application state.

Chapter 8

Publications

"Teaching is more than imparting knowledge, it is inspiring change."

— William Arthur Ward

This chapter contains a copy of the ten publications of this habilitation in chronological order. It first summarizes the publication and then includes a reprint of it. Every publication has been peer-reviewed by at least two expert reviewers of the respective scientific community. The research presented has been used to refine interactive learning and Artemis. The four most significant publications are highlighted in blue in Table 8.1. The last column of the table includes the acceptance rate. Each section of this chapter shows more details on the respective publication, such as the exact paper type and the DOI.

#	Year	Type	Authors	Title	Venue	Publisher	Acc. Rate
1	**2017**	**Conf**	**S. Krusche**, A. Seitz, J. Börstler, and B. Bruegge	**Interactive Learning – Increasing Student Participation through Shorter Exercise Cycles**	ACE	ACM	39%
2	2017	WS	**S. Krusche**, N. von Frankenberg, and S. Afifi	Experiences of a Software Engineering Course based on Interactive Learning	SEUH	CEUR	75%
3	2017	Conf	**S. Krusche**, B. Bruegge, I. Camilleri, K. Krinkin, A. Seitz, and C. Wöbker	Chaordic Learning: A Case Study	ICSE	IEEE	24%
4	**2018**	**Conf**	**S. Krusche** and A. Seitz	**ArTEMiS - An Automatic Assessment Management System for Interactive Learning**	SIGCSE	ACM	29%
5	**2018**	**Jour**	**S. Krusche**, D. Dzvonyar, H. Xu and B. Bruegge	**Software Theater — Teaching Demo Oriented Prototyping**	TOCE	ACM	12%
6	2019	Conf	**S. Krusche** and A. Seitz	Increasing the Interactivity in Software Engineering MOOCs - A Case Study	CSEE&T	ScholarSpace	57%
7	2019	WS	C. Laß, **S. Krusche**, N. von Frankenberg, and B. Bruegge	Stager: Simplifying the Manual Assessment of Programming Exercises	SEUH	CEUR	75%
8	**2020**	**Conf**	**S. Krusche**, N. von Frankenberg, Lara Marie Reimer, and B. Bruegge	**An Interactive Learning Method to Engage Students in Modeling**	ICSE	ACM	24%
9	2020	Conf	J. Bernius, A. Kovaleva, **S. Krusche**, and B. Bruegge	Towards the Automation of Grading Textual Student Submissions to Open-ended Questions	ECSEE	ACM	63%
10	2021	Conf	J. Bernius, **S. Krusche**, and B. Bruegge	A Machine Learning Approach for Suggesting Feedback in Textual Exercises in Large Courses	L@S	ACM	30%

Table 8.1: Overview of the publications this habilitation is based on (Conf = Conference, Jour = Journal, WS = Workshop, Acc. Rate = Acceptance Rate). The four most significant publications are highlighted in blue.

8.1 Interactive Learning – Increasing Student Participation through Shorter Exercise Cycles

This conference paper was an essential milestone in the research for this habilitation. Interactive learning is based on active learning, experiential learning, and computer-based learning. It aims to decrease the time between content delivery and content deepening in large university courses to a few minutes, allowing for flexible and more efficient learning. In addition, shorter exercise cycles allow students to apply and practice multiple concepts per teaching unit directly after they first heard about them. An empirical evaluation in two large courses shows that students' learning experience and exam grades correlate with increased participation due to interactive learning. The publication formalized the learning approach and formed the basis for several subsequent research initiatives. We further analyzed the correlation to find causal effects between interactive learning as an intervention, the motivation of the students, and the learning outcome.

Authors	**S. Krusche**, A. Seitz, J. Börstler and B. Bruegge
Conference	19th Australasian Computing Education Conference
Publisher	ACM
Pages	10
Type	Conference: Full Research Paper
Review	Peer Reviewed (3 Reviewers)
Year	2017
Citation	[KSBB17]
DOI	https://doi.org/10.1145/3013499.3013513

Interactive Learning – Increasing Student Participation through Shorter Exercise Cycles

Stephan Krusche
Technische Universität München
Munich, Germany
krusche@in.tum.de

Andreas Seitz
Technische Universität München
Munich, Germany
seitz@in.tum.de

Jürgen Börstler
Blekinge Institute of Technology
Karlskrona, Sweden
jurgen.borstler@bth.se

Bernd Bruegge
Technische Universität München
Munich, Germany
bruegge@in.tum.de

ABSTRACT

In large classes, there is typically a clear separation between content delivery in lectures on the one hand and content deepening in practical exercises on the other hand. This temporal and spatial separation has several disadvantages. In particular, it separates students' hearing about a new concept from being able to actually practice and apply it, which may decrease knowledge retention.

To closely integrate lectures and practical exercises, we propose an approach which we call interactive learning: it is based on active, computer based and experiential learning, includes immediate feedback and learning from the reflection on experience. It decreases the time between content delivery and content deepening to a few minutes and allows for flexible and more efficient learning. Shorter exercise cycles allow students to apply and practice multiple concepts per teaching unit directly after they first heard about them.

We applied interactive learning in two large software engineering classes with 300 students each and evaluated its use qualitatively and quantitatively. The students' participation increases compared to traditional classes: until the end of the course, around 50% of the students attend class and participate in exercises. Our evaluations show that students' learning experience and exam grades correlate with the increased participation. While educators need more time to prepare the class and the exercises, they need less time to review exercise submissions. The overall teaching effort for instructors and teaching assistants does not increase.

Keywords

Active Learning, Experiential Learning, Feedback, Reflection, Computing Education, Software Engineering

ACE '17, January 31-February 03, 2017, Geelong, VIC, Australia

ISBN 978-1-4503-4823-2/17/01…$15.00

DOI: http://dx.doi.org/10.1145/3013499.3013513

1. INTRODUCTION

When teaching large classes with hundreds of students, there is typically a significant delay between delivery of content in lectures and deepening and practicing of that content in follow-up exercises. The delay between lectures and exercises is usually a few days, up to a week (see Figure 1).

Figure 1: Delay between lectures and exercises in traditional learning in large classrooms

During this time, learners forget content that was discussed in the lecture. Participation, learning and knowledge retention might be reduced in both quantity and quality, when students are not cognitively active during content delivery and when there is a large time span between the content's delivery and actively dealing with it. This might lead to unnecessary knowledge gaps for the learners, especially if the interaction between educators and learners is low.

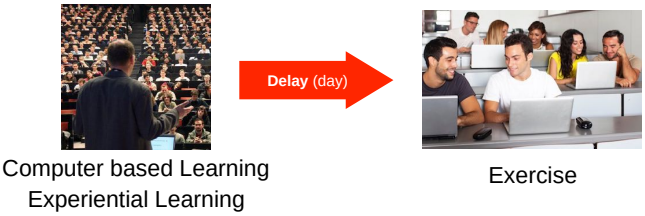


Figure 2: Reduced delay through the use of computer based and experiential learning

Computer based learning [8] and experiential learning [14] are approaches to reduce the delay between lectures and exercises as shown in Figure 2. Computer based learning supports students' learning in digital exercises and online media. Experiential learning creates opportunities to reflect

on experience, a methodology in which educators engage with students to increase knowledge and develop skills.

Active learning is an educational approach to increase student involvement and excitement with the subject being taught by engaging students in activities [3]. **Interactive learning** is based on active learning, integrates computer based learning and experiential learning, immediate feedback and reflection. It tightens the relationship between content delivery and problem solving in class by integrating multiple, small units of content delivery and content deepening through exercises. By combining lectures and exercises into interactive classes, it reduces the delay to a few minutes (see Figure 3). Students reflect on the learned content immediately and increase their knowledge incrementally through a couple of short cycles covering theory, example, exercise, solution and reflection. Our hypothesis is that reducing the delay between lectures and exercises as proposed in interactive learning increases student participation in exercises and thereby improves the learning experience.

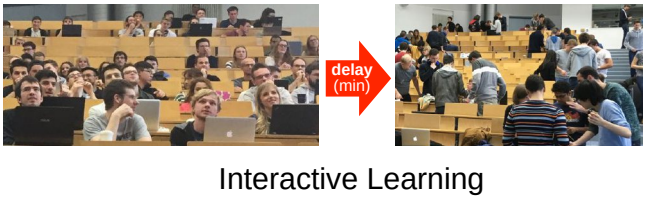


Figure 3: Interactive learning combines lectures and exercises into interactive classes and further reduces the delay to minutes

The paper is organized as follows: In Section 2, we describe the foundations in the areas of experiential and active learning. Section 3 presents the idea of interactive learning as an iterative process that combines lectures and exercises into short cycles. Section 4 shows a case study about two large software engineering courses, in which we applied interactive learning. In Section 5, we present the findings of qualitative and quantitative evaluations in these courses. Section 6 discusses related work and Section 7 summarizes the paper and presents the conclusion.

2. FOUNDATIONS

Exercises and examples are important elements in teaching and learning: "[E]xamples appear to play a central role in the early phases of cognitive skill acquisition" [27]. Just letting learners solve more problems is, however, not the most effective way to support learning. Carefully developed and integrated examples increase the learning outcome more than just letting learners solve more problems [25, 26]. In particular, in complex problem spaces, like software development, "[l]earners may learn more by solving problems with the guidance of some examples than solving more problems without the guidance of examples" [26].

Software engineering is an activity that requires collaboration and practical application of knowledge [29, 24]. Educators struggle when teaching it in traditional lecture based environments where activities take place in the front of the classroom. Lectures are usually similar to broadcasting, where essential education interactions take place initiated by the educator with only limited participation on the learners side. Self-guided learning, personal responsibility, practical relevance and individualization are important elements of a great learning experience. Several pedagogic theories have been developed that include these elements.

Problem based learning is a technique to learn about a subject through problem solving. Educators facilitate by supporting, guiding, and monitoring this process [4]. While working in groups, learners identify what they know, what they need to know, and how and where to access new information that leads to the resolution of the problem.

Cooperative learning is an educational approach which aims to organize classroom activities into social learning experiences: learners work in groups to complete tasks collectively towards a common goal [11]. The educators role changes from giving information to facilitating learners' learning. Everyone succeeds when the group succeeds.

Experiential learning is the process of learning from experience, a methodology in which educators engage with learners in direct experience to increase knowledge, develop skills, and clarify values [14]. Aristoteles said: "For the things we have to learn before we can do them, we learn by doing them". John Dewey followed this idea with his statement that "there is an intimate and necessary relation between the process of actual experience and education".

Experiential learning is considered to be more efficient than passive learning like reading or listening. It is in contrast to academic learning where students acquire information through the study of a subject without the necessity for direct experience. The main dimensions of experiential learning are analysis, initiative, and immersion. Academic learning promotes the dimensions of constructive learning [28] and reproductive learning [12]. Both methods instill new knowledge, though academic learning makes use of more abstract techniques, whereas experiential learning actively involves the learner in a concrete experience such as an exercise.

Active learning is an educational approach to increase student involvement and excitement with the subject being taught [3]. Instead of students acting as receivers of knowledge by passively listening, active learning puts the emphasis on developing student skills and engaging them in activities such as small group discussions or a class game. Grabinger and Dunlop emphasize that authentic contexts encourage students to take more responsibility and engage students in learning activities that promote high level thinking processes [9]. An authentic context in software engineering would e.g. be the management of a project where students experience typical activities such as meeting and task management. Their learning progress is supported and assessed through realistic tasks such as planning and conducting a meeting or distributing tasks within the team.

Michael Prince examined the evidence for the effectiveness of active learning and discussed its common forms [23]. He concluded that active learning positively influences knowledge transfer and student performance. Joel Michael reviewed the literature and found that there is evidence that active learning improves the learning outcome compared to more passive approaches [22]. However, certain active learning approaches are not feasible for large classrooms. It is not possible to have a group discussion with 300 students in the same lecture hall. In addition, it is important that instructors "place a strong emphasis on guidance of the student learning process" to prevent misconceptions [13].

While the combination of these learning techniques leads to a more complex experience and to more effort for educators,

it lowers their stress and leads to higher satisfaction [2]. A Chinese proverb, first mentioned by Confucius and adapted by Benjamin Franklin describes the underlying philosophy of experiential and active learning. In recent publications [15] an extended version of the proverb is mentioned: "Tell me and I will forget. Show me and I will remember. Involve me and I will understand. Step back and I will act."

"Tell me and I will forget" describes that explaining a concept only theoretically does not give learners the possibility to apply it. *"Show me and I will remember"* includes the idea of cognitive apprenticeship: an apprentice observes the skills of a master who shows how a concept works in practice, e.g. in a tutorial. Clarifying the thinking process behind the application of the concept makes it easier for the apprentice to imitate the behavior [7].

"Involve me and I will understand" includes aspects of active learning. Involving learners in the learning process allows them to apply a concept on their own, possibly in a different way that fits to their own techniques. It helps learners to understand a concept together with its application. *"Step back and I will act"* refers to self-guided learning, self improvement and problem based learning. Learners take the responsibility to solve a certain problem on their own using the concepts they learned before. This proverb is the foundation of our teaching approach: **interactive learning**.

3. INTERACTIVE LEARNING

Interactive learning aims to decrease the cycle time between teaching a concept and exercising it by combining lectures and exercises into interactive classes with multiple, short iterations. Thus, the typical separation between lectures and exercises disappears. We define interactive learning as follows: **Educators teach and exercise small chunks of content in short cycles and provide immediate feedback so that learners can reflect on the content and increase their knowledge incrementally.** Interactive learning expects active participation of learners and the use of computers (laptops, tablets, smartphones) in classes. Instructors provide guidance during the learning process to facilitate learning and to prevent misconceptions.

Figure 4 shows the iterative process of interactive learning, where each iteration consists of five phases:

1. **Theory:** The educator introduces a new concept and describes the theory behind it. Learners listen and try to understand it.
2. **Example:** The educator provides an example, so learners can refer the theory to a concrete situation.
3. **Exercise:** The educator asks the learners to apply the concept in a short exercise. The learners submit their solution of the exercise.
4. **Solution:** The educator provides a sample solution and explains it to the learners. The educator can also show some exemplary solutions, submitted by learners and discuss their strengths and weaknesses to provide immediate feedback.
5. **Reflection:** The educator facilitates a discussion about the theory and the exercise so that the learners reflect on their first experience with the new concept.

In large education environments with hundreds of learners who participate in a course at the same time, teaching assistants (TAs) help in the conduct of the exercises. TAs walk through the classroom, answer questions and provide help in case problems occur or exercise instructions are unclear. The evaluation of the submitted solutions can either be automated using tool support or manually done by the TAs reviewing the submitted solutions and providing immediate feedback to the learners. The degree of automation depends on the exercise type and the format of the solution. The evaluation of programming assignments can e.g. be automated using continuous integration and test cases.

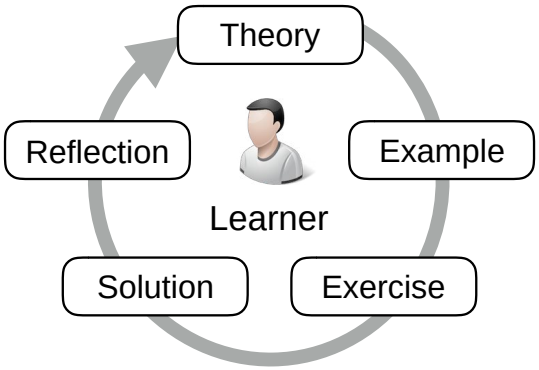


Figure 4: Interactive learning as iterative process

Interactive learning can be applied in **individual exercises** and in **team exercises**:

Individual exercises

- **E1** Quizzes with multiple choice questions (automatic evaluation through a quiz system)
- **E2** Interactive tutorials with step by step instructions (semi-automatic evaluation, degree depends on the exercise)
- **E3** Interactive coding challenges to solve programming assignments (automatic evaluation through test cases)
- **E4** Interactive modeling exercises (manual evaluation)

Interactive tutorials help students to directly experience a new concept. They are very detailed and include step-by-step instructions so that even beginners are able to conduct the exercise. The instructor conducts the tutorial live in-class so that students can follow the tutorial on their own computer. He asks the students several times during a tutorial how many can follow to synchronize the speed. TAs walk around and help students with problems. The instructor uploads slides with detailed screenshots before class, so that students can look up steps on the slides if they could not follow in the given time. More experienced students are kept motivated with optional challenges. Students who miss a class can catch up the exercise at home.

Team based exercises

- **E5** Project work that includes communication and collaboration aspects (semi-automatic evaluation, degree depends on the exercise)

Team based exercises also incorporate the concepts of peer learning and cooperative learning. They repeat topics of individual exercises to deepen and retain the knowledge by applying the learned concept in a different setting. Students have to transfer the knowledge they learned before to the

concrete team situation and have to tailor the concepts. This facilitates self-guided learning and promotes the idea of self organization.

4. CASE STUDY

In 2015 and 2016, we applied interactive learning in two software engineering university courses "Software Engineering II: **P**roject **O**rganization and **M**anagement" (POM) and "**P**atterns in **S**oftware **E**ngineering" (PSE). In previous instances of these courses, content delivery (theory) and content deepening (exercises) were separated, i.e. students learned a concept in the lecture and applied it a week later in a central exercise session. In POM 2015 and in PSE 2015/16, we applied interactive learning and combined theory and exercises into interactive classes. Students learned a concept theoretically, then immediately applied it in a short exercise and received feedback about their progress.

4.1 Project Organization and Management

We taught POM in summer 2015 with 294 students. Typically, between 100 and 200 students attended class and participated in the exercises. They formed a heterogeneous group because the course was offered in multiple programs. Two distinct groups participated: (1) bachelor students in information science, a few with experience in software engineering, and (2) master students in computer science, some with existing experience in the taught topics. The challenge of this heterogeneity was that students completed class exercises at different rates. To improve this situation, the exercises included optional tasks specifically for more experienced students. In addition, the students had the opportunity to solve exercises as homework if they were not able to finish them in class. Some tasks and exercises were also explicitly designed as homework.

The module description of POM describes the following intended learning outcomes. Participants understand the key concepts of software project management. They are able to deal with problems such as writing a software project management plan, initiating and managing a software project and tailoring a software lifecycle. They are familiar with risk management, scheduling, planning, quality management, build management and release management, and can apply these techniques to solve simple problems. Table 1 shows the schedule and the content of the lecture.

Week	Content
1	Team formation
2	Project organization
3	Software process models
4	Agile methods [18]
5	Prototyping
6	Proposal management
7	Branch & merge management [19]
8	Contracting
9	Continuous integration
10	Continuous delivery [17]
11	Risk and demo management
12	Global project management [21]
13	Project management antipatterns

Table 1: Overview of the course content in POM

Students can earn bonus points for completing exercises successfully. They can use these bonus points to improve their final exam mark. If they e.g. earn between 60% and 80% of the total points, their mark in the final exam is improved by one grade. This possibility motivates the students to participate in the individual and team based exercises. In POM, we used the exercise types: quizzes (E1), interactive tutorials (E2) and project work (E5), as described in Section 3.

Interactive Tutorial

Students had to solve individual tasks on their computer. They cooperated with the instructor, TAs and fellow students to solve particular problems. They learned from their experience in exercises and reflected on the concepts they had just learned. The instructor conducted four large interactive tutorials in POM using dedicated tools:

1. Agile Methods (Atlassian JIRA[1])
2. Branch and Merge management (Atlassian Bitbucket[2])
3. Continuous Integration (Atlassian Bamboo[3])
4. Continuous Delivery (HockeyApp[4])

In these interactive tutorials, the instructor introduced concepts and immediately applied them in short exercises. The students completed the exercises on their own computer using the mentioned tools in the browser. During each tutorial, the students either looked at the detailed slides that were handed out at the beginning of the exercise or watched how the instructor conducted the exercise on the presentation computer. In addition, TAs walked through the lecture hall, helping students by answering questions directly.

Each interactive tutorial consisted of three to five exercises which were decomposed into smaller tasks. In summary, the students had to solve between twelve and twenty tasks in one tutorial. The instructor synchronized the speed of the tutorial several times by asking students about their progress and by checking the number of participants and results in the tools. If more than about 90% were able to complete a particular tasks, the instructor proceeded to the next exercise.

As an example, we describe the execution of the two exercises about continuous integration and continuous delivery based on the release management workflow described by Krusche et al. [17]. The instructor mapped an exemplary delivery process for a mobile application to the continuous integration server that was used in class, Bamboo. To simplify the exercises, each student first forked a preconfigured source code repository and cloned a preconfigured build plan. Then, the students adapted and configured the build plan, fixed existing test cases and wrote additional test cases. A change in the requirements of the software led to a bug that was detected by Bamboo during a regression test and fixed by the students so that all tests passed again at the end of the exercise and the students could deliver the software to their fellow students who played the role of test users.

Project Work

In addition to the individual exercises, the students participated in a team project (exercise type E5) with five team

[1]http://www.atlassian.com/software/jira
[2]http://www.atlassian.com/software/bitbucket
[3]http://www.atlassian.com/software/bamboo
[4]http://hockeyapp.net

members, a simplified version of the team projects described by Bruegge et al. [5]. The goal of the project was that the students experience the learned concepts in a more realistic environment. The instructor played the role of the customer and provided three short problem statements about the development of mobile applications. The teams had to choose one of the problem statements and a development environment and target platform, either Android, Windows Phone or iOS.

The instructor arranged the students into different teams according to their self-assessment. The goal was to have balanced teams with respect to the skill level of the students. Team based exercises also built on experiential learning techniques. However, they had a stronger focus on problem based and cooperative learning. Software engineering is a collaborative activity [29], therefore team work is an important skill students have to learn. The teams used Rugby [18] as agile and continuous process model with an initial warm-up phase and five development sprints.

In addition, students only received a vague description of the exercises that deliberately missed detailed instructions so that the teams have to think on their own about how to solve the exercise. This approach follows the principle "Step back and I will act" of the Chinese Proverb in Section 2.

Students first learned and experienced concepts in individual exercises. Then, they applied the knowledge in team exercises to improve their long term memory. They had to tailor the concepts to their concrete team situation and had to agree on different decisions in their team which facilitates communication, collaboration and conflict handling. In later classes, students reflected on their team experiences.

4.2 Patterns in Software Engineering

We taught PSE in winter 2015/16 with 324 students. Typically, between 150 and 250 students attended class and participated in the exercises. The course included key concepts of different types of patterns that can be used during software development, in particular design patterns, architectural patterns, testing patterns, antipatterns, and organizational patterns. The learning goals are that students understand patterns as a way to describe reusable knowledge for analysis, system design, object design and software project management activities. Given a problem, they are able to identify the applicability of a pattern that addresses the problem, describe the pattern in UML and map it to Java source code. The course was attended by bachelor and master students mainly in the field of computer science. Table 2 shows the schedule and the content of the course.

Week	Content
1	Introduction and pattern definition
2	Basic concepts
3, 4, 5	Design patterns
6, 7	Architectural patterns
8, 9	Antipatterns
10, 11	Testing patterns
12	Pattern based reengineering
13	Global software engineering

Table 2: Overview of the course content in PSE

During the 13 classes, we conducted 39 exercises in total: 29 of them were in-class exercises, 10 of them were homework. In contrast to POM, we did not conduct any team exercises (E5) in PSE. We focused on the exercises types: interactive coding challenge (E3) and interactive modeling challenges (E4). We also carried out quizzes (E1) and interactive tutorials (E2). In the homework exercises, students further deepened their knowledge. Exercise participation was optional for the students.

Interactive Coding Challenge

The participants had to write new source code or adjust existing code, commit their changes to a version control system which then automatically triggered test cases on a continuous integration server to verify the given solution. To increase the extrinsic motivation, the first three students submitting a correct solution were rewarded by the lecturer in the form of gummy bears or donuts. In addition, there was a wildcard winner which was randomly picked and acknowledged without being among the first three correct submissions.

We applied continuous integration with unit tests to verify the submitted solutions of the students automatically and immediately. Coding exercises were distributed with a version control system. We used Atlassian Bitbucket as the repository server and Atlassian Bamboo as the integration server. The required material for the specific exercise was provided before the lectures in a repository accessible to all students. To synchronize the working time on the exercise, access to the material was secured by a password.

After introducing the theory, explaining the problem and providing the corresponding password, the students started to work on the exercise. With this approach, we made sure that all students started working on the exercise at the same time and had the same timeframe for solving the problem. The timeframe to submit the exercise was determined by the instructors and the elapsed time during the exercise was visualized on a big stop clock on one of the projectors. Once the deadline had passed, the instructors provided a sample solution and discussed it with the students. Winning students also had the opportunity to explain their solution and why they came up with this approach. Due to the fact that we used a version control system and continuous integration, it was possible to track the participation and validate the results of the students in real time.

As an example we describe the conduct of an exercise regarding the state design pattern that we conducted in lecture 4. After the introduction and explanation of the state design pattern, the exercise for the students was to implement a basic remote control for a TV with four states. The exercise was to apply the state pattern to implement the transition between the appropriate states. The instructors provided a standard Java Eclipse project with existing source code, unrelated to the state pattern itself. The problem was visualized in a UML state diagram describing the different state transitions and its limitations.

The instructor set the timeframe to solve this exercise to 15 minutes, released the password and the students started to work on the exercise. During the work on the exercise, the TAs helped the students in case there were any questions regarding the exercise. Students could ask for help by raising their hand. After 5 minutes, a hint was given to the students in the form of a class diagram representing the implementation of the state pattern. In this exercise, 145 students took part using 145 repositories and 145 automatically configured build plans. Each submission led to an execution of 20 test cases, resulting in 2900 test results within 15 minutes. As

the deadline for working on the exercise had lapsed, the first three correct submissions and the wildcard winner were honored. The instructors discussed a sample solution with the students to reflect on the concept of the state pattern.

5. EVALUATION

In both courses, we evaluated the effects of interactive learning by investigating whether there is a correlation between exercise participation and the students' grade in the final exam, which represents the competence and gained knowledge of students in the taught topic. In addition, we conducted an online questionnaire in POM, where we asked students about their personal opinion on their improvements in a specific technique and their confidence to apply the technique in a later situation.

5.1 Study Design

We state the following three hypotheses with respect to interactive learning:

- **H1 Participation**: Interactive learning increases the participation of students in classes.
- **H2 Improved Learning**: Interactive learning leads to an improved learning experience for students.
- **H3 Scalability**: Interactive learning is scalable to large classes with 300 students.

We validated the hypotheses with a qualitative evaluation in POM and quantitative evaluations in both courses. The qualitative evaluation was conducted as an online questionnaire. We investigated the students' improvements and confidence in the techniques that we applied in the individual and team based exercises. After the end of POM, we invited 294 students, who completed the final exam of the course, to participate in the questionnaire. The anonymous questionnaire consisted of six closed questions, took about five minutes and was not mandatory for the students.

The first two questions were about personal data, field of study and degree. The third question asked whether students participated in specific individual exercises, the fourth question asked whether they applied specific techniques in their team project. The last two questions used a five point Likert type scale with the answers *strongly disagree*, *disagree*, *neutral*, *agree*, *strongly agree* to measure either negative, neutral or positive responses. The fifth question measured if students were able to improve their skills in these techniques and the sixth question measured if students were confident to apply these six techniques in their next team project.

We conducted the survey in July 2015 and gave students two weeks to participate in it. We created personalized tokens and sent them to the students, who completed the exam. The open source survey tool LimeSurvey[5] guarantees that the answers are still anonymous by strictly separating token and answer tables in the database. We sent two reminders to the students who had not yet participate. We combined the responses of the results into a three point Likert type scale with positive responses (*strongly agree* and *agree*), neutral and negative responses (*strongly disagree* and *disagree*) to minimize positive and negative outliers.

We conducted two quantitative evaluations to investigate the learning experiences in both courses. The quantitative measurement focused on the relationship between exercise participation and the final grade of the students. We calculated the exercise points in POM and the exercise participation in PSE for each participating student and correlated it with the final exam grade. We grouped the students after the relative exercise points / participation into five categories, calculated the grade point average for each category and computed the correlation using a χ^2 test.

[5]http://www.limesurvey.org

5.2 Findings

We first present the qualitative findings of the online questionnaire. 223 out of 294 students (76%) participated in the questionnaire at the end of POM. As described in more detail in Section 4, the students first learned concepts in individual exercises conducted in class. Then, they applied these concepts in team exercises. The questionnaire asked the students whether they participated in four individual exercises and whether they applied these four techniques in their team: agile methods, branch and merge management, continuous integration and continuous delivery.

The questionnaire included the following two statements for each of these techniques:

1. **Improved**: In the exercises, I was able to improve my skills in the technique
2. **Confident**: I am confident to apply the technique in my next team project

In the following, we describe the answers to these statements for students who participated in individual exercises **and** who applied the technique in their team.

Figure 5 shows that 85% of the students perceived that they improved their skills in agile methods and that 88% of the students were confident to apply agile methods in their next team project. This result confirms the strong focus of the exercises on agile methods and shows that students feel prepared for the management of their next agile project.

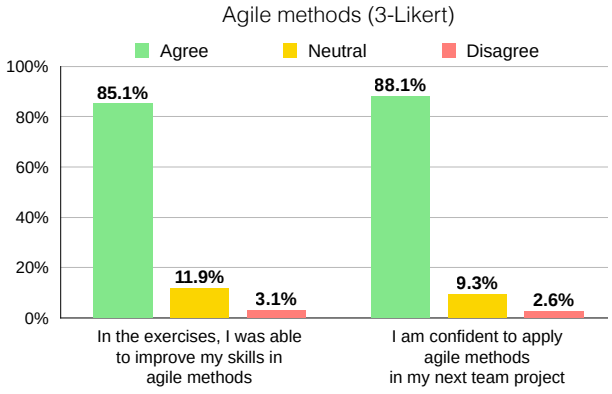


Figure 5: Perceived improvements and confidence in *agile methods* in POM

Figure 6 shows that 78% of the students perceived that they improved their skills in branch and merge management. 87% of the students were confident to apply it in their next team project. These results show that branch and merge management have become teachable. Students are able to handle multiple branches and can deal with merge conflicts.

Figure 7 shows that 76% of the students perceived that they improved their skills in continuous integration. The

Content	(1) Individual exercise			(2) Team based exercise			(3) Both exercise types		
	#	Agree improved	Agree confident	#	Agree improved	Agree confident	#	Agree improved	Agree confident
Agile methods	209	83.7%	87.1%	198	84.8%	88.4%	194	85.1%	88.1%
Branch & merge management	197	75.6%	81.7%	162	75.9%	85.8%	155	78.1%	86.5%
Continuous integration	166	71.7%	73.5%	138	70.3%	73.9%	119	75.6%	76.5%
Continuous delivery	149	71.8%	71.8%	118	73.7%	73.7%	98	77.6%	78.6%

Table 3: Percentage of students who participated in exercises and perceived that they *improved* their skills respectively perceived that they are *confident* to apply the technique in their next project

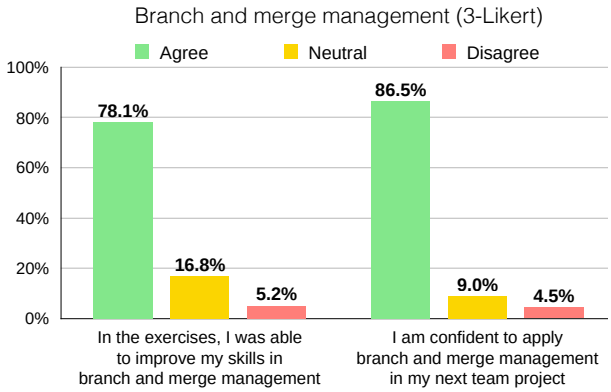


Figure 6: Perceived improvements and confidence in *branch and merge management* in POM

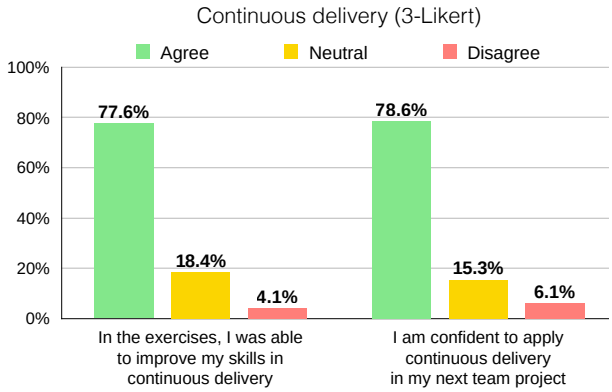


Figure 8: Perceived improvements and confidence in *continuous delivery* in POM

students could experience the benefits of immediate feedback about integration and test failures after they committed their changes to the source code repository. 77% feel confident to apply continuous integration in their next team project.

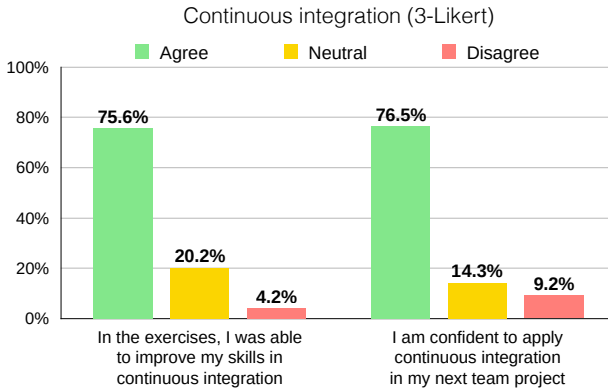


Figure 7: Perceived improvements and confidence in *continuous integration* in POM

Figure 8 shows that 78% of the students perceived that they improved their skills in continuous delivery. In the corresponding exercise, they configured continuous delivery for a mobile application and applied it in their team project as well. 79% of the students were confident to apply continuous delivery in their next team project.

In Table 3, we summarize the evaluation results of the questionnaires for the four techniques. The table shows that the participation in individual exercises was higher than in team based exercises. While 209 students (71%) participated in the individual exercise on agile methods, 198 students (67%) participated in the team exercise, and 194 students (66%) in both exercises.

In one of the last exercises about continuous delivery, there were still 149 students (51%) participating in the individual exercise, 118 students (40%) in the team exercises and 98 students (33%) in both exercises. The numbers were lower, because the exercise was more challenging in complexity and required more effort by the students. Table 3 includes the following three different filters:

(1) **Individual exercise:** We considered students who reported that they participated in the individual exercise of the corresponding technique.
(2) **Team based exercise:** We considered students who reported that they applied the concept in their team project.
(3) **Both exercise types:** We considered students who reported that they participated in the individual exercises **and** who applied the technique in their team. These are the same results as shown in Figure 5 - Figure 8.

In addition to the qualitative evaluation, we also looked at attendance rates. Figure 9 shows that the number of participants per lecture was relatively stable throughout the POM course in 2015. The number of students decreased from 61% on average in the first seven weeks of the course to 47% on average in the last five weeks of the course, although classes started at 8:15 am in the morning and the in-class quizzes were conducted in the beginning of the class[6].

[6]Some students missed these quizzes as they came late to the 8:15 am classes, so the actual attendance rate was higher.

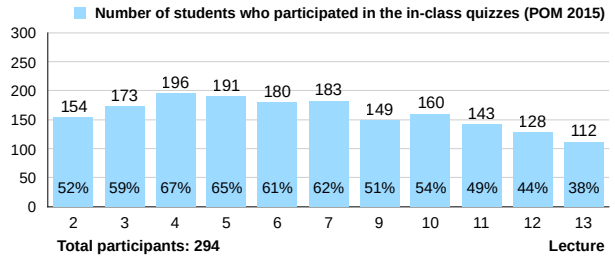


Figure 9: Number of participants per lecture in POM in summer term 2015

The attendance rate in 2015 was higher than in the previous instance of the course offered in 2014 (compare Figure 10), when the number of students who visited a lecture dropped to below 20% on average in the last five weeks of the course[7]. The attendance rates in 2014 are more in line with other courses at our faculty that are taught in a more traditional way. This indicates that interactive learning might help to increase the participation of students in classes (H1). The increase in participation might, however, be a result of other factors. Further investigations are therefore needed.

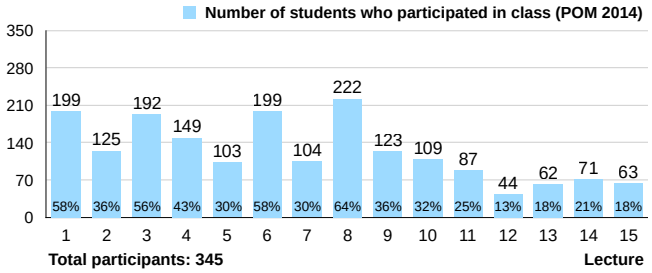


Figure 10: Number of participants per lecture in POM in summer term 2014

We evaluated whether there is a correlation between exercise participation and the average grade of the students in the final exam. The students could receive up to 600 bonus points through the participation in quizzes, individual exercises and team based exercises.

We grouped the 294 students, who completed the exam, into five categories with equal distances describing their participation in the exercises, see Table 4 and Table 5. For instance, the first category in POM contains 75 students who obtained less than 20% of the exercise points and the second category contains 66 students who obtained between 20% and 40% of the exercise points. Figure 11 show that students with lower exercise participation have worse grades (i.e. higher grades in our grading system) than students with a higher participation, who have better, i.e. lower, grades.

In fact, students who successfully participated in less than 20% of the exercises have a grade point average (GPA) of 3.9 in POM and a GPA of 3.1 in PSE. Students with more than 60% have a GPA of 2.5 or better in POM[8] and a GPA of 2.1 or better in PSE.

[7]Some lectures in 2014, e.g. 1, 3, 6 and 8, were between 12:15 and 13:45 and had higher attendance rates than the other ones starting at 8:15 am.

[8]The GPA in Figure 11 and Table 4 does not include bonus that students could obtain through exercise participation.

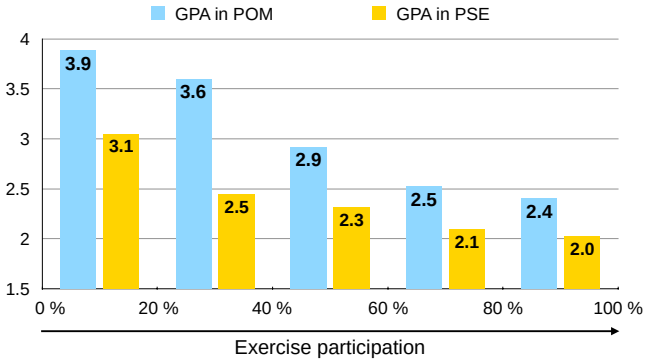


Figure 11: GPA (grade point average) of the final exam of POM (blue) and PSE (yellow) grouped by students' exercise participation. Final exam grades vary between 1.0 and 5.0 and do not include any bonuses; lower grades are better.

A χ^2 test [1] shows that there is a strong and highly significant correlation between exercise participation and the grade in the final exam in POM ($\chi^2 = 82.53; p < 0.0001$) as well as in PSE ($\chi^2 = 48.01; p < 0.0001$), see Table 4 and Table 5. Students who participate more in the exercises tend to achieve better grades in their final exams. This might indicate a positive effect of interactive learning (H2).

However, it could also mean that stronger students (with good grades) tend to participate more in the exercises compared to weaker students (with lower grades). We have no data from previous courses to evaluate the effects of other factors such as motivation or previous experience of students on exercise participation and exam results. Further studies are therefore needed. From the number of students in our two case studies, we also conclude that instructors can apply interactive learning (as implemented in our setting) in large courses with 300 students, which supports H3.

5.3 Limitations

In our qualitative evaluation in POM in summer 2015, one threat to the validity is that the personal opinion of students might not reflect on the real situation, because it is subjective. Most students were beginners in the taught concepts and reported about their perceived improvements. A student without previous knowledge in an area will improve his knowledge, even if he only learns a limited amount of concepts. Beginners might not be able to objectively estimate their improvements in a subject. The confidence to apply a concept does not necessarily mean that the student is really able to apply it.

Other variables of the course, such as an open atmosphere towards feedback, have a positive influence on the evaluation result. If a student likes interactive exercises, it does not necessarily mean that he improves his skills. We were not able to exclude these variables in the evaluations of the case study. To alleviate these threats, we additionally evaluated the participation in the exercises and the correlation of students exercise participation and exam results quantitatively.

In the quantitative measurements, we recognize the following threats to validity. The participation in interactive exercises might be one reason for the found correlation that leads to the improvement of the final exam grade, but the

Exercise points (relative)	0 - 20%	20 - 40%	40 - 60%	60 - 80%	80 - 100%	**All**
Number of students with very good grade (1.0 - 1.3)	0	0	8	10	2	20
Number of students with good grade (1.7 - 2.3)	5	11	22	21	3	62
Number of students with satisfactory grade (2.7 - 3.3)	23	21	35	12	2	93
Number of students with sufficient grade (3.7 - 4.0)	14	16	14	12	2	58
Number of students who failed (4.3 - 5.0)	33	18	9	1	0	61
Number of students who completed the exam (Sum)	75	66	88	56	9	294
Grade point average (GPA) without bonus (all students)	3.9	3.6	2.9	2.5	2.4	3.2

Table 4: Correlation between exercise participation and GPA in the final exam in POM: students who successfully completed more exercises received more exercise points and scored significantly better in the exam (grades vary between 1.0 and 5.0; a lower grade is better, a higher is worse).

Exercise participation (relative)	0 - 20%	20 - 40%	40 - 60%	60 - 80%	80 - 100%	**All**
Number of students with very good grade (1.0 - 1.3)	9	13	12	9	4	47
Number of students with good grade (1.7 - 2.3)	45	34	15	18	7	119
Number of students with satisfactory grade (2.7 - 3.3)	44	25	8	8	5	90
Number of students with sufficient grade (3.7 - 4.0)	20	6	6	3	0	35
Number of students who failed (4.3 - 5.0)	28	3	2	0	0	33
Number of students who completed the exam (Sum)	146	81	43	38	16	324
Grade point average (GPA, all students)	3.1	2.5	2.3	2.1	2.0	2.7

Table 5: Correlation between exercise participation and GPA in the final exam in PSE: students who participated in exercises scored significantly better in the exam (grades vary between 1.0 and 5.0; a lower grade is better, a higher is worse).

correlation does not necessarily show this causality. Other causes of the correlation might be the motivation or the previous knowledge of the students. Students who participate in exercises more frequently usually also have a higher motivation to learn the theory for the exam, or they might have more previous experience. We were not able to measure these variables in the presented courses or to exclude them and therefore more detailed studies are necessary to investigate the actual effects of them on the learning outcomes.

6. RELATED WORKD

There are several courses in computer science that apply active learning techniques. They report an increase in students' learning, engagement, and performance. We look at three approaches and compare them with our approach.

Kurtz et al. describe an active learning approach using microlabs [20]. Students perform short activities during a lecture, either individually or in groups, and submit their answers to an automated grading system. They receive constructive feedback, and can revise their answers. Kurtz et al. conclude that microlabs can increase the students learning experience. Their approach can be compared with in-class exercises in interactive learning. In our case, students also have a time limit for exercises and submit their solutions to an automated grading system, or upload them for manual assessment if no automated assessment is possible. Both approaches have one thing in common: that they are used in class during lectures. Heckman reports that there is "a large increase in student engagement" for the use of in-class labs [10].

Another popular approach is think pair share (TPS), where students work on a problem individually, then in small groups and finally reflect on it with the whole class. We propose a similar approach where students first experience a concept individually and then apply it within a team. Kothiyal et al. describe a large programming course which uses TPS [16]. The course includes programming labs and lectures with two TPS activities: students worked on questions individually first, and then in pairs, while the instructor helps in case of questions. Class wide discussions were facilitated concerning the former tasks. The study reports an average of 83% student engagement for TPS based activities. This approach shows parallels to our course setup, since we introduced individual and team exercises, similar to the think and pair phases. Interactive learning also covers the share phase, as students could join a discussion with the instructor.

Campbell et al. describe an approach based on the flipped classroom concept with video lectures, labs and assignments [6]. They also used quizzes, contributing to the course grade as we propose for our approach. However, the authors do not use in-class exercises and report a low lecture attendance rate. Our approach implements frequent homework assignments, as well as immediate feedback for exercises to keep students motivated. Campbell et. al. suggest the use of such an approach instead of labs for future improvement.

7. CONCLUSION

In this paper we described interactive learning, an approach based on active learning, computer based learning and experiential learning for large classrooms with guidance and immediate feedback. In interactive learning, the educator delivers small chunks of content and exercises incrementally, in short cycles, so that learners reflect on the content immediately. This can increase students' participation, can support knowledge deepening and can improve knowledge retention. By reducing the delay between theory, example, exercise, solution and reflection to a few minutes, we establish a tighter integration of lectures and exercises leading to interactive classes.

We applied and evaluated interactive learning in two large software engineering classes with 300 students each. In the quantitative evaluations, we found a strong and highly sig-

nificant correlation between exercise participation and the final exam grade. Students who participated more in the exercises also achieved better grades in their final exams. A qualitative evaluation shows that students perceive that they improve their skills and feel able to apply these skills in later projects. This indicates that interactive learning might improve students' learning experience and learning outcomes. More studies are, however, necessary to investigate the role of interactive learning in those results.

Our case studies show that interactive learning is scalable and applicable to large classes without increasing the teaching effort significantly. We will introduce interactive learning into more courses. We believe it can help to bring an interactive component to massive open online courses (MOOCs) where we want to mix videos and fully automated interactive exercises to improve the students' learning experience and the adaptability of the course with respect to heterogeneous student groups.

8. REFERENCES

[1] M. Abramowitz, I. A. Stegun, et al. Handbook of mathematical functions. *Applied mathematics series*, 55:62, 1966.

[2] R. Ben-Ari, R. Krole, and D. Har-Even. Differential effects of simple frontal versus complex teaching strategy on teachers' stress, burnout, and satisfaction. *International Journal of Stress Management*, 2003.

[3] C. Bonwell and J. Eison. *Active Learning: Creating Excitement in the Classroom*. ASHE-ERIC Higher Education Reports., 1991.

[4] D. Boud and G. Feletti. *The challenge of problem-based learning*. Psychology Press, 1998.

[5] B. Bruegge, S. Krusche, and L. Alperowitz. Software engineering project courses with industrial clients. *ACM Transactions on Computing Education*, 2015.

[6] J. Campbell, D. Horton, M. Craig, and P. Gries. Evaluating an inverted cs1. In *Proceedings of the 45th technical symposium on Computer science education*, pages 307–312. ACM, 2014.

[7] A. Collins, J. S. Brown, and A. Holum. Cognitive apprenticeship: Making thinking visible. *American educator*, 1991.

[8] R. Garrison and H. Kanuka. Blended learning: Uncovering its transformative potential in higher education. *The internet and higher education*, 2004.

[9] S. Grabinger and J. Dunlap. Rich environments for active learning: A definition. *Research in learning Technology*, 3(2), 1995.

[10] S. Heckman. An empirical study of in-class laboratories on student learning of linear data structures. In *Proceedings of the 11th annual International Conference on International Computing Education Research*, pages 217–225. ACM, 2015.

[11] D. Johnson, K. Smith, and R. Johnson. Cooperative learning: increasing college faculty instructional productivity. *ASHE-ERIC higher education reports.*, 1991.

[12] S. Jong, A. Jan, R. Wierstra, and J. Hermanussen. An exploration of the relationship between academic and experiential learning approaches in vocational education. *British Journal of Educational Psychology*, 76(1):155–169, 2006.

[13] P. Kirschner, J. Sweller, and R. Clark. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist*, 41(2):75–86, 2006.

[14] D. Kolb. *Experiential learning: Experience as the source of learning and development*, volume 1. Prentice Hall, 1984.

[15] J. Korthagen, Fredand Kessels, B. Koster, B. Lagerwerf, and T. Wubbels. *Linking practice and theory: The pedagogy of realistic teacher education*. Routledge, 2001.

[16] A. Kothiyal, R. Majumdar, S. Murthy, and S. Iyer. Effect of think-pair-share in a large cs1 class: 83% sustained engagement. In *Proceedings of the 9th annual international conference on International computing education research*, pages 137–144. ACM, 2013.

[17] S. Krusche and L. Alperowitz. Introduction of Continuous Delivery in Multi-Customer Project Courses. In *Proceedings of the 36th International Conference on Software Engineering*, pages 335–343. IEEE, 2014.

[18] S. Krusche, L. Alperowitz, B. Bruegge, and M. Wagner. Rugby: An agile process model based on continuous delivery. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pages 42–50. ACM, 2014.

[19] S. Krusche, M. Berisha, and B. Bruegge. Teaching Code Review Management using Branch Based Workflows. In *Companion Proceedings of the 38th International Conference on Software Engineering*. IEEE, 2016.

[20] B. Kurtz, J. Fenwick, R. Tashakkori, A. Esmail, and S. Tate. Active learning during lecture using tablets. In *Proceedings of the 45th technical symposium on computer science education*, pages 121–126. ACM, 2014.

[21] Y. Li, S. Krusche, C. Lescher, and B. Bruegge. Teaching global software engineering by simulating a global project in the classroom. In *Proceedings of the 47th SIGCSE*, pages 187–192. ACM, 2016.

[22] J. Michael. Where's the evidence that active learning works? *Advances in Physiology Education*, 30(4):159–167, 2006.

[23] M. Prince. Does active learning work? a review of the research. *Journal of Engineering Education*, 93(4):223–231, 2004.

[24] D. Shaffer. Pedagogical praxis: The professions as models for postindustrial education. *Teachers College Record*, 106(7):1401–1421, 2004.

[25] J. Sweller and G. A. Cooper. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2(1):59–89, 1985.

[26] J. G. Trafton and B. J. Reiser. Studying examples and solving problems: Contributions to skill acquisition. Technical report, Naval HCI Research Lab, Washington, DC, USA, 1993.

[27] K. VanLehn. Cognitive skill acquisition. *Annual Review of Psychology*, 47:513–539, 1996.

[28] J. D. Vermunt. The regulation of constructive learning processes. *British journal of educational psychology*, 68(2):149–171, 1998.

[29] J. Whitehead. Collaboration in software engineering: A roadmap. *FOSE*, 7(2007):214–225, 2007.

8.2 Experiences of a Software Engineering Course based on Interactive Learning

This workshop paper describes experiences in a software engineering course on project management and organization (POM). It was one of the first courses taught with the idea of interactive learning. The paper describes POM in detail and includes an evaluation of the teaching philosophy. The main result was that interactive learning increases the participation of students. The findings show that students are more engaged and motivated if they apply and exercise the previously learned theory in class. Furthermore, by providing students with theoretical foundations and practical exercises, their learning experience improves.

Authors	**S. Krusche**, N. von Frankenberg and S. Afifi
Conference	15. Workshop Software Engineering im Unterricht der Hochschulen
Publisher	CEUR
Pages	9
Type	Workshop Paper
Review	Peer Reviewed (3 Reviewers)
Year	2017
Citation	[KvFA17]
Link	http://ceur-ws.org/Vol-1790

Experiences of a Software Engineering Course based on Interactive Learning

Stephan Krusche, Nadine von Frankenberg, Sami Afifi
Technische Universität München, Munich, Germany
krusche@in.tum.de, nadine.frankenberg@tum.de, afifi@mytum.de

Abstract

Learning to apply software engineering requires practical experience, and can not be taught through traditional theory-based lectures. Interactive learning is an approach that combines lectures and exercises into multiple iterations of theory, example, exercise, solution and reflection. It is based on active, computer based and experiential learning and on immediate feedback to improve the learning experience in large classes. It includes hands-on activities with the goal to increase students' motivation and engagement.

This paper describes an interactive learning course design that includes multiple choice quizzes and interactive tutorials as in-class exercises and a team project in which students apply their knowledge in a different setting. Based on this course design, we present a case study with 300 students in 2016. An evaluation shows that students are more engaged and motivated, if they practically apply and exercise the previously learned theory in the classroom. By providing students with both, theoretical foundations and practical exercises, their learning experience improves.

1 Introduction

Software engineering (SE) requires practical application of knowledge (Connolly et al., 2007; Shaffer, 2004), because it is an interactive and collaborative activity (Whitehead, 2007). In particular, project management in SEering is an activity that requires practical experience. The learning experience of students is low when educators disregard the practical relevance of SE and do not handle real problems in a course (Cunliffe, 2002). Interaction with students is limited if the learning activities focus on the educator in front of the classroom. Then, students' participation and motivation are low and the learning outcome decreases.

Educators can apply self-guided learning, personal responsibility, practical relevance and individualization to overcome this problem. Several pedagogic theories have been developed that include these elements: Problem-based learning teaches a subject through the experience of problem solving. Educators support, guide, and monitor this process (Boud and Feletti, 1998). Cooperative learning organizes classroom activities into social learning experiences: Students complete exercises in groups towards a common goal (Johnson et al., 1991). Computer based learning allows students to learn through computer-mediated activities (Garrison and Kanuka, 2004). Experiential learning is the process of learning from experience and reflecting about it (Kolb, 1984). Active learning promotes that students actively participate in the learning process (Bonwell and Eison, 1991). Instead of only passively listening, they are involved in exercises and engaged in solving problems.

We developed a course that includes a mix of these approaches and teaches SE concepts through **interactive learning** (Krusche et al., 2017). The course includes interactive tutorials and quizzes as activating in-class exercises where students immediately receive feedback to reflect about their performance. It also integrates team based exercises in which students apply the knowledge in a different situation to deepen their understanding and to increase their knowledge retention.

While the integration of multiple learning theories and exercise types increases the effort for educators, it can lower their stress and it can lead to higher satisfaction for educators and learners (Ben-Ari et al., 2003). We base our teaching methodology on a Chinese proverb: *"Tell me and I will forget. Show me and I will remember. Involve me and I will understand. Step back and I will act"* (Korthagen et al., 2001). It emphasizes that involving students into the learning process, activating them in the classroom, is the key for their understanding. Self-guided and problem-based learning let students take responsibility to solve a problem on their own, using concepts they learned before.

The paper is organized as follows: Section 2 describes active learning and the Revised Bloom's Taxonomy as foundations of the learning theories in our course. In Section 3, we present the course design that follows an interactive learning approach with an iterative process combining lectures and exercises into short cycles. Section 4 presents a case study about a large software engineering course with 300 students

in which we applied interactive learning. In Section 5, we present the findings of an evaluation of this case study. Section 6 discusses related work and Section 7 concludes the paper.

2 Foundations

Active learning is an educational approach to increase student involvement with the subject being taught. Instead of students acting as receivers of knowledge by passively listening to lectures, active learning puts the emphasis on developing student skills and engaging them in activities. Bonwell and Eison define active learning as "anything that involves students in doing things and thinking about the things they are doing" (Bonwell and Eison, 1991). The active learning approach draws from constructivist learning theories and can be summarized in four main premises (Brophy and Good, 1994):

1. Learners construct their own meanings
2. New learning builds on prior knowledge
3. Learning is enhanced by social interaction
4. Meaningful learning develops through "authentic" tasks

Grabinger and Dunlap emphasize that authentic contexts encourage students to take more responsibility and engage them in learning activities that promote high level thinking processes (Grabinger and Dunlap, 1995). In SE education, an authentic context would be a software project where students have to develop an application: they experience typical development workflows and tools such as software configuration management.

Students experience collaborative learning through learning communities that involve both peer students and instructors. Their learning progress is supported and assessed through realistic tasks such as planning and conducting a meeting. There is broad support for the benefits of active learning on knowledge transfer and student performance (Prince, 2004). Active learning has an improved learning outcome compared to more passive approaches (Michael, 2006).

Bonwell and Eison propose a set of activities that align with the principles of active learning (Bonwell and Eison, 1991). Examples are:

- **Think-Pair-Share:** Students think and discuss about a topic in pairs.
- **Simulation:** Classroom activities resemble real-life situations.
- **Working in group:** Collaborative or cooperative group work requires high involvement of students.
- **Case studies:** Practical examples encourage students to integrate knowledge from class with real-life.

While keeping active learning principles in mind, instructors can use the Revised Bloom's Taxonomy (RBT) to classify curricular objectives and exercises (Krathwohl, 2002). The RBT identifies six cognitive process categories (remember, understand, apply, analyze, evaluation, create) and four knowledge categories, ordered from concrete to abstract knowledge:

- **Factual:** Basic knowledge to acquaint with a discipline and to solve problems.
- **Conceptual:** Connection of basic knowledge in a larger context.
- **Procedural:** Methodology of knowledge application using skills, techniques, and methods.
- **Metacognitive:** Knowledge about the use of particular strategies for learning or problem solving.

The cognitive process dimension together with the knowledge dimension help to formulate learning objectives. Learning activities that require higher order cognitive processes and that lead to acquisition and construction of more abstract knowledge can be classified as active learning. Figure 1 shows the RBT matrix classifying more active and more passive learning approaches.

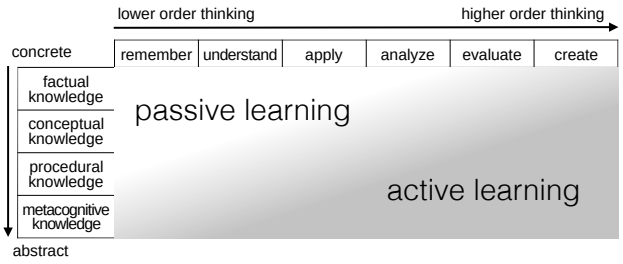


Figure 1: Classification of more active and more passive learning approaches in the matrix of the Revised Bloom's Taxonomy (adapted from (Krathwohl, 2002))

3 Course Design

With the aforementioned pedagogical foundations in mind, we designed a course to teach software project management by mixing lectures with engaging in-class and homework activities, such as hands-on tutorials, multiple choice quizzes, team exercises and team projects. To activate students, we put emphasis on the interactivity of the course.

3.1 Learning Objectives

The course has the following intended learning outcomes: Participants understand the key concepts of software project management. They learn and apply the basic techniques and methods of project organization that are used when complex software systems are developed such as task, issue and meeting management. The course focuses on agile models as preferred software lifecycle, in particular Scrum (Schwaber, 1995) and Kanban (Anderson, 2010).

Students communicate and collaborate in team projects, learn to estimate tasks and to schedule a

project. They learn how to model software life-cycles and how to write an agile contract. They design user interfaces, create prototypes and evaluate these using typical usability heuristics. Student apply software configuration management including change, branch, merge and review management using git (Chacon, 2009) and pull requests (Krusche et al., 2016). They apply build and release management by implementing typical continuous integration and continuous delivery workflows (Krusche and Alperowitz, 2014).

Students do not only get familiar with the theory of each topic, but also get practical experience. They get to know specific cases and learn how to use each concept in different settings.

3.2 Organization

The course is designed for large audiences with more than 100 students. One instructor teaches the course with the help of teaching assistants (TAs). Besides helping the students, the TAs also act as intermediaries between the students and the instructor. To maintain a high level of interactivity, informal communication channels encourage students to interact with other course participants, with their team members and with the instructor. A learning management system is used for formal information sharing.

During class, TAs monitor a question channel of a chat tool, and respond when necessary. This gives students the opportunity to clarify questions through informal communication. People respond and communicate more frequently when using an informal communication tool (Kraut et al., 1990), such as a chat application. During class, TAs can inform the instructor about issues that are of interest for other students. Then, the instructor can clarify issues and answer questions in front of all students. In addition, the instructor encourages students to ask questions in the lecture hall as well.

Lectures and exercises are combined into interactive classes to encourage students to attend. Students are expected to actively participate: they must bring their own laptop, tablet or smartphone and use it in class for computer based exercises. To motivate students to participate in these exercises, students can earn bonus points to improve their grade in the final exam. In addition, students can participate in a team project to apply the learned knowledge in another setting. This team project includes with five team members and is a simplified version of the team projects described by Bruegge and his colleagues (Bruegge et al., 2015): there is no real customer and students have less deliverables, but the applied process model is the same. While the team projects are not mandatory to the students, the instructor encourages them to take part, because students learn important communication and negotiation skills when it e.g. comes to task distribution and meeting management.

3.3 Interactive Learning

Figure 2 shows the iterative process of interactive learning. Each lecture has multiple iterations of the following five phases (Krusche et al., 2017):

1. **Theory:** The instructor introduces a new concept and describes the theory behind it. Students listen, try to understand it and ask questions.
2. **Example:** The instructor provides an example so that students can refer to a concrete situation.
3. **Exercise:** The instructor asks the students to apply the concept in a small exercise. The students submit their solution to the exercise.
4. **Solution:** The instructor provides a sample solution, explains it to the students and discusses exemplary student submissions to provide immediate feedback and guidance.
5. **Reflection:** The instructor facilitates a discussion about the theory and the exercise so that students reflect about the concept.

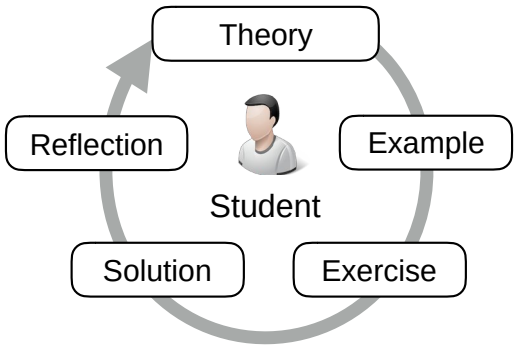


Figure 2: Iterative process of interactive learning performed multiple times during lectures (Krusche et al., 2017)

While the theory helps to build factual knowledge and conceptual knowledge, exercises can build procedural and metacognitive knowledge.

3.4 Exercises

TAs help in the conduction of the exercises: they walk through the classroom, answer questions and provide help in case problems occur or exercise instructions are unclear. The assessment of the submitted solutions can either be automated using tool support, or manually done by the TAs who review the submitted solutions and provide immediate feedback to the students. The degree of automation depends on the exercise type and the solution's format. The course includes individual exercises and team based exercises:

Individual exercises

E1 Quizzes with drag and drop questions or multiple choice questions (automatic evaluation through a quiz system).

E2 Interactive tutorials with step-by-step instructions (automation degree depends on the exercise).

In each class, theory is followed by short in-class quizzes, serving as self-assessment so that students can instantly check whether they understood the main concepts or not. Therefore, quizzes help to increase factual and conceptual knowledge by repeating and connecting the learned theory.

Interactive tutorials include detailed, step-by-step instructions so that even beginners are able to conduct the exercises. They are helpful to experience a concept for the first time. The instructor performs these tutorials live in class so that students can follow on their own laptops. He uploads the presentation slides with detailed screenshots before class, so that students can look up the steps of the exercise on the slides if they cannot follow in the given time.

He asks the students several times during a tutorial how many of them can still follow. If not enough students raise their hand, he waits and explains the current step again in more detail. If more than around 80 % raise their hand, the instructor continues. More experienced students are kept motivated with optional challenges. Interactive tutorials help to build procedural knowledge by involving the students into the methodology of knowledge application and by building skills, techniques and methods to solve particular tasks.

All tutorials are self-contained and do not depend on previous exercises. They are based on the same common problem statement provided in the beginning of the class, so students know the context and can follow exercises more easily. The instructor does not have to introduce a new problem statement in each class and saves time. If students miss a class, they can also catch up with the exercise at home.

Team based exercises

E3 Project teamwork that includes communication and collaboration aspects (automation degree depends on the exercise).

Team based exercises incorporate the concepts of peer learning and cooperative learning. They repeat the topic of individual exercises to deepen and retain the knowledge by applying the learned concepts in a different setting. Students transfer the previously learned knowledge to the concrete team situation and tailor the concepts. This facilitates self-guided learning and promotes the idea of self-organization, an important management aspect.

To create a context for their project, the teams choose a problem statement and a development environment in the beginning of the course. In team based exercises, students need to transfer the previously-learned factual, conceptual and procedural knowledge into a concrete situation. They need to adapt the learned skills, techniques and methods or find new ones to solve the problem collaboratively while taking responsibility because the instructor steps aside. This helps to build metacognitive knowledge.

4 Case Study

The following case study describes the university course "Software Engineering II: **P**roject **O**rganization and **M**anagement" (POM) that implements interactive learning. We evaluated the course in summer 2016, amongst others by means of a questionnaire that included free text fields in which the students stated their thoughts. The detailed evaluation is discussed in Section 5.

4.1 Course Format

The course had a heterogeneous distribution of 300 students with two groups standing out: around half of the students are bachelor students with major in information system who have to take this course in their studies. The other half are master students in computer science and take the course by their own choice. The challenge is to keep the lecture content easy enough for less experienced students, but also stimulating enough for more experienced ones.

The course took place in one semester over 13 weeks in the summer 2016, with a three hour time slot for lectures and exercises between 8:15 am and 11:30 am, including a 15min break. A single instructor taught the course with the help of 9 teaching assistants. Table 1 shows the schedule and the content of each lecture.

Week	Class content
1	Team Formation
2	Project Organization
3	Software Lifecycle Models
4	Agile Methods (Krusche et al., 2014)
5	Prototyping & Usability Management (Bruegge et al., 2012)
6	Proposal Management
7	Branch, Merge & Review Management (Krusche et al., 2016)
8	Contracting & Estimation
9	Continuous Integration
10	Continuous Delivery (Krusche and Alperowitz, 2014) Feedback Management (Krusche and Bruegge, 2014)
11	Risk and Demo Management
12	Global Project Management
13	Project Management Antipattern

Table 1: Overview of the course content

In large courses, students get easily distracted, may no longer pay attention to the lecture, or may engage in off-topic conversations with each other. Therefore, our main goal was to design and structure each class so that students are engaged and motivated using interactive learning.

As additional motivation, students were able to earn bonus points (BP) for participating in exercises. If they earned enough BPs, their grade in the final exam was improved accordingly. Students reported in a survey

that the bonus was a “strong motivation to be active in the course”. In the following, we illustrate the course concept in more detail.

4.2 Quizzes

During each class, multiple choice quizzes gave students the opportunity to revise the covered theory, and to earn BPs. To motivate students to attend class, we performed the quizzes dynamically during the class after certain lecture content was completed. Thus, only students present in class could participate in the quizzes. On average, 181 students participated in the quizzes per class.

We optimized the creation of quiz questions during the course using an iterative feedback approach to minimize misunderstandings and ambiguities. Two TAs created the quizzes based on the lecture content, then all TAs reviewed the questions to find errors and misunderstandings. On average, there were three quizzes per class. A quiz consisted of three questions with three to four answer choices each, an example of a question is shown in Figure 3.

Question: What are key characteristics of the Waterfall Model?

- ✓ 1. Progress is measured by the number of tasks that have been completed.
- ✓ 2. At the end of each activity, a verification step prevents the deletion or unwanted introduction of requirements.
- X 3. The Waterfall Model allows the repetition of activities if requirements change unexpectedly.
- X 4. Traditional managers do not like waterfall-based models, since fixed milestones are bad for progress measurement.

Figure 3: Example question with four answer choices

To prevent cheating during the quiz, students could see their results only after the quiz was closed. Questions and answer choices were randomly interchanged in the learning management system which included an automated quiz grading component that showed the overall quiz performance. As a result, there was no correction overhead.

Most students liked the concept of using quizzes directly after the lecture content, and stated in a survey that it was ”good to use quizzes to see right away what I learned”. The quiz performance of all participating students was available instantly to the instructor, and provided information on how well the students could follow the lecture. After each quiz, the instructor shortly reviewed and discussed questions and answers with the students. This offered the opportunity to repeat key points of the taught theories. Students also had the chance to give feedback about the quiz.

In the beginning of the course, some students reported “too little time to read and answer some questions”. We considered this when designing subsequent quizzes. Other students’ reported inconsistencies in the quizzes and helped to improve the questions. There was controversial feedback concerning the quizzes, as some students felt that they “put too much pressure” on them. Most students appreciated the quizzes as a direct control of their learning outcome. One student reported: “I like the quizzes a lot. They really help me to understand the topics faster and better”. Another one stated in the Slack[1] channel that was open during class: “quizzes wake me up better than coffee” as shown in Figure 4.

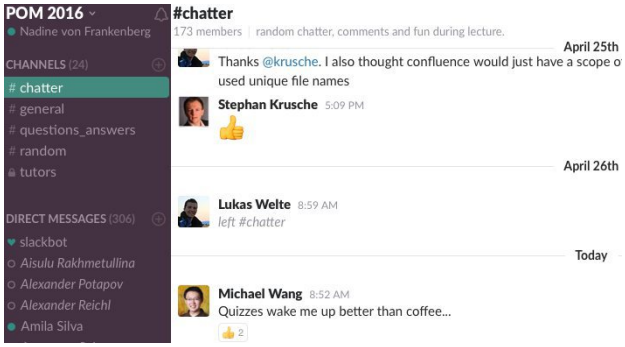


Figure 4: Slack channel with students’ comments during the lecture

4.3 Exercises and Homework

In the first class, students organized themselves into 51 teams, with 5 students per team, to participate in a team project. Each team chose one of three distinct problem statements, and had the task to implement a small mobile app until the end of the course, while applying project management methodologies. The students had to include at least one experienced and one unexperienced team member to give unexperienced students the opportunity to learn from more experienced ones, following the master apprentice approach (Collins et al., 1991). Experienced students could also benefit, since they deepened their knowledge, and were challenged by optional harder tasks.

The first team exercise was an icebreaker in the first class where all teams participated in a small competition, the marshmallow challenge (Wujec, 2010) in the lecture hall as part of the team formation class. The teams had 18 minutes to build the tallest structure using spaghetti sticks, tape, rope and a marshmallow. Figure 5 shows how the teams built the spaghetti towers in the classroom. After the exercise, each team had to measure its own tower according to specific rules and upload the picture to a shared space. The TAs evaluated the tower and the measurement and awarded the best team with small prices.

The instructor performed in-class exercises live on a computer shown on a projector. A second projector showed the corresponding lecture slides with detailed screenshots. The TAs walked through the lecture hall and helped if necessary. However, TAs did not explicitly tell students the solutions to the exercises, but

[1] Slack is a popular free team chat service that we used in class to improve the communication between instructor, TAs and students: `https://slack.com`.

Figure 5: Icebreaker with about 200 students in 51 teams in the lecture hall

rather pointed them into the right direction, so that they worked out the solutions themselves.

Immediate feedback was an important factor for the exercises. Students could review sample solutions on the projector, and were assisted by TAs. The exercise tools and the screenshots in the lecture slides provided additional feedback. If the student's screen looked identical to the screenshot on the slide after a task, or the tool reported a success message, the students knew that they performed the exercise correctly. We used several workflows and tools that are used in industry in order to demonstrate practical usage (Klepper et al., 2015). Most students found this approach helpful, and liked that the "exercises were practical and relevant".

To deepen the students' understanding, each lecture included a team project exercise to learn and apply management concepts, following a learning by doing approach. Students e.g. learned agile methods, and had to apply them throughout the team project. They performed self-organized meetings and documented their meetings, including a meeting-selfie of all participants to add a fun factor to the exercise. Another example is that they formed pairs in class, implemented a small feature, and then reviewed their partner's pull requests to understand the code review workflow (Krusche et al., 2016). Students then followed this pull request workflow when implementing the mobile app in the team projects. While the team projects were not mandatory, students could earn 50 % of the bonus points. Therefore, many students were motivated to participate in the team projects.

Multiple TAs reviewed the exercises that could not be assessed automatically. The students received feedback at the latest two weeks after the submission deadline. As Kothiyal and his colleagues point out, "prompt and descriptive feedback on their [the students] understanding" enables both, students and instructor, to "use this feedback to modify their learning and teaching respectively" (Kothiyal et al., 2013). Students found fast feedback motivating and helpful. For each exercise, students could see the current grading status and deadline so that they had an overview which exercises were due in the current week. They could see which TA graded their exercise to directly communicate with the TA to clarify questions.

5 Evaluation

This section describes the study design, the findings and the limitations of an evaluation of the course POM in summer 2016.

5.1 Study Design

We state the following hypotheses:

H1 Participation: Interactive in-class exercises increase the participation of students.

H2 Improved Learning: The mix of theory, quizzes and exercises in class leads to an improved learning experience for students.

We validate the hypotheses with a quantitative and a qualitative evaluation. In the quantitative evaluation, we measured the participation of students by counting how many students attended class and completed specific exercises.

In the qualitative evaluation, we investigated the students' improvements in topics that we applied in individual and team based exercises using an online survey. We asked about their opinion on the exercise concept. The questionnaire consisted of 14 questions, took about 10 minutes and was not mandatory for the students. It included questions about personal data, the participation in individual exercises, and application of techniques in the team project. We also wanted to know if students improved their skills in techniques and if they felt confident to apply techniques in their next team project. Finally, we asked in an open question how the course can be improved.

We conducted the survey in July 2016 and gave the students two weeks to complete it. We created personalized tokens and asked the 272 students, who completed the final exam of the course, to participate in the anonymous survey. The open source survey tool LimeSurvey[2] guarantees that the answers are anonymous by strictly separating token and answer tables in the database. We received 190 responses, which corresponds to a response rate of 70 %.

5.2 Findings

The quantitative evaluation shows that more students participated in POM in summer 2016 than in a previous instance of the course without interactive learning in summer 2014 or in other courses of the same faculty. Figure 6 shows that the number of participants per class in 2016 was around 80 % in the beginning and around 60 % in the end of the course, although the class started early at 8:15 am in the morning. Figure 7 shows that in the same course in 2014, the attendance rate steadily decreases to less than 20 % until the end

[2] http://www.limesurvey.org

of the course. On average, 71 % of all participants of the course completed the team exercises, whereas 72 % completed the individual exercises. From these numbers, we have first anecdotal evidence that H1 is supported: interactive in-class exercises increase the participation of students.

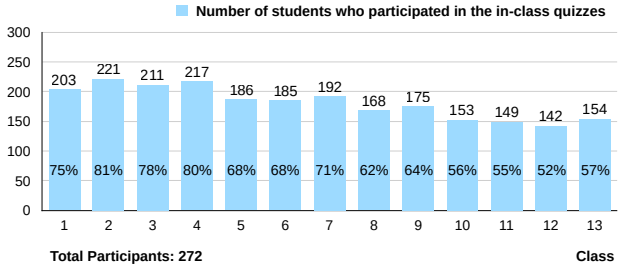


Figure 6: Number of participants per class in POM in summer 2016 **with** interactive learning

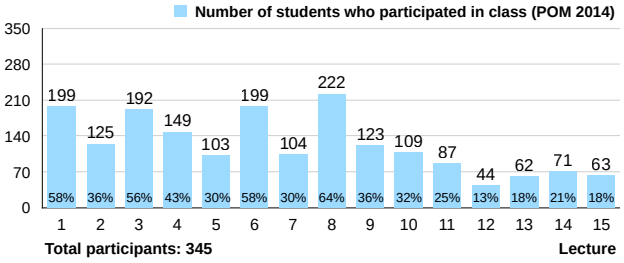


Figure 7: Number of participants per lecture in POM in summer 2014 **without** interactive learning

The qualitative evaluation showed that on average 80 % of the students, who participated in an individual and team exercise, agree (or strongly agree) that they improved their knowledge and that they are confident to apply the knowledge in their next team project. Table 2 shows results of the qualitative evaluation, i.e. whether students agreed to given statements in the qualitative evaluation. 80 % of the students agreed that the use of interactive learning increased their learning success (S1), and 71 % agreed that it improved their understanding of the theory during class (S2).

In-class exercises motivated 76 % of the students to attend the lecture (S3) and quizzes motivated 54 % of the students to listen to the lecturer (S4). Interactive tutorials helped 65 % of the students to learn new concepts and also 65 % were able to deepen their knowledge in team exercises. These answers can be considered as anecdotal evidence that H2 is supported: the mix of theory, quizzes and exercises in class leads to an improved learning experience of the students.

Formulating multiple choice quiz questions and their respective answer possibilities unambiguously proved to be challenging, especially when asking question that are beyond simple definitions. At the beginning of the course, we varied the number of questions per class, the number of answer choices per question, and the time per question. After the quizzes, we presented the correct solutions and discussed them quickly with the students. In the first half of the course the average number of quizzes per lecture session was higher: on average, we had five quizzes with 3 questions each, leading to 15 questions per class.

In an intermediate evaluation, we found that the number of quizzes per class was too high, so we reduced it to three quizzes and nine questions on average per class. The available time per questions was between 60 and 120 seconds depending on its difficulty and length. Additionally, we evaluated each quiz regarding the students' response rate. This helped us to extract weak and misleading factors of the questions and answers, e.g. when many students selected a wrong answer choice due to misinterpretation.

In total, many students reported that this course was their favorite course in the semester and that they wish that more courses would be rich in variety and activation during class.

5.3 Limitations

A limitation of the qualitative evaluation is that personal opinions of students might not reflect the real situation, because they could be subjective. Beginners cannot estimate objectively about their real improvement, and the confidence to apply a concept does not necessarily mean that the student is in fact able to apply it.

Other positive effects of the course, such as the open atmosphere towards feedback, might have a positive influence on the evaluation result. Only if students like interactive exercises, this does not necessarily mean that their skills improve. To alleviate these threats, we additionally evaluated the participation in the lectures and exercises quantitatively in a more objective manner.

6 Related Work

Active learning techniques applied in computer science show an increase in students' learning, engagement, and overall performance. A popular approach is Think-Pair-Share (TPS), where students first work on a problem individually, then in small groups and finally with the whole class.

Kothiyal and his colleagues describe a setting that uses TPS in a large level-1 programming course (Kothiyal et al., 2013). The course included lectures and programming labs. The lectures had two TPS activities, where students first worked on questions individually, and then with a subsequent task in pairs, while an instructor could be asked for help. Finally, class-wide discussions were facilitated concerning the former tasks. The study reports an average of 83 % student engagement for TPS-based courses. This approach shows some parallels to our course setup, since we introduced individual and team exercises, similar

	Statement	Strong Agree	Agree	Neutral	Disagree	Strong Disagree
S1	The mix of theory, quizzes and exercises in class contributed to my learning success	36 %	44 %	13 %	4 %	3 %
S2	The mix of theory, quizzes and exercises improved my understanding of the theory during class	33 %	38 %	16 %	10 %	3 %
S3	In-class exercises motivated me to attend the lecture	35 %	41 %	14 %	6 %	4 %
S4	Quizzes motivated me during class to actively listen to the lecturer	19 %	34 %	23 %	15 %	9 %
S5	Interactive tutorials were particularly helpful to understand concepts that I did not know before	24 %	41 %	24 %	9 %	2 %
S6	Team exercises helped me to apply the concept in a different setting to deepen my knowledge and understanding	16 %	49 %	26 %	6 %	3 %

Table 2: Evaluation results with Likert scale typed responses whether students agree to given statements

to the think and pair phases. The share phase is also present, as students could join discussions with the instructor.

Kurtz and his colleagues describe an active learning approach using microlabs (Kurtz et al., 2014). Students perform 5-10min activities during lectures, either individually or in groups, and submit their answers to an automated grading system, using tablets as delivery mechanism. Students receive constructive feedback, and can revise their answers. The study concludes that microlabs can increase the students' learning gains. This approach can be compared with the in-class exercises we performed. Students had a pre-defined time limit for the exercises and submitted their solutions to an automated grading system, or to an online documentation tool. The key point is, that both approaches are used during lectures.

Campbell and his colleagues describe a flipped classroom approach with video lectures, labs and assignments (Campbell et al., 2014). Similar to our approach, quizzes were used and contributed to the course grade. However, the authors do not give credit for in-class exercises, and report a low lecture attendance rate. Our course design includes homework assignments as team exercises, as well as immediate feedback for in-class exercises to keep students motivated.

Heckman reports, there is "a large increase in student engagement" for the use of in-class laboratories (Heckman, 2015). His approach is similar to our in-class exercises, but not used in large classes.

7 Conclusion

In this paper we described our experiences with an interactive learning course in project management in software engineering. Interactive learning is based on active, computer based and experiential learning: the instructor combines lectures and exercises into multiple iterations of theory, example, exercise, solution and feedback. The course includes multiple choice quizzes and interactive tutorials as in-class exercises and an additional team project where students apply their knowledge in a different setting. This mix supports different knowledge dimensions.

We applied and evaluated interactive learning in a large course with 300 students. We found that interactive learning increases the participation of students. Our findings show that students are more engaged and motivated, if they practically apply and exercise the previously learned theory in class. By providing students with theoretical foundations and practical exercises, their learning experience improves.

References

Anderson, D. (2010). *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press.

Ben-Ari, R., Krole, R., and Har-Even, D. (2003). Differential effects of simple frontal versus complex teaching strategy on teachers' stress, burnout, and satisfaction. *International Journal of Stress Management*.

Bonwell, C. and Eison, J. (1991). *Active Learning: Creating Excitement in the Classroom.* ASHE-ERIC Higher Education Reports.

Boud, D. and Feletti, G. (1998). *The challenge of problem-based learning*. Psychology Press.

Brophy, J. and Good, T. (1994). *Looking in Classrooms*. HarperCollins College Publishers.

Bruegge, B., Krusche, S., and Alperowitz, L. (2015). Software engineering project courses with industrial clients. *ACM Transactions on Computing Education*.

Bruegge, B., Krusche, S., and Wagner, M. (2012). Teaching Tornado: from communication models to releases. In *Proceedings of the 8th edition of the Educators' Symposium*, pages 5–12. ACM.

Campbell, J., Horton, D., Craig, M., and Gries, P. (2014). Evaluating an inverted cs1. In *Proceedings of the 45th technical symposium on Computer science education*, pages 307–312. ACM.

Chacon, S. (2009). *Pro git*. Apress.

Collins, A., Brown, J., and Holum, A. (1991). Cognitive apprenticeship: Making thinking visible. *American educator*.

Connolly, T., Stansfield, M., and Hainey, T. (2007). An application of games-based learning within software engineering. *British Journal of Educational Technology*, 38(3):416–428.

Cunliffe, A. (2002). Reflexive dialogical practice in management learning. *Management learning*, 33(1):35–61.

Garrison, R. and Kanuka, H. (2004). Blended learning: Uncovering its transformative potential in higher education. *The internet and higher education*.

Grabinger, R. and Dunlap, J. (1995). Rich environments for active learning: A definition. *Research in Learning Technology*, 3(2):5–34.

Heckman, S. (2015). An empirical study of in-class laboratories on student learning of linear data structures. In *Proceedings of the 11th annual conference on International Computing Education Research*, pages 217–225. ACM.

Johnson, D. et al. (1991). *Cooperative Learning: Increasing College Faculty Instructional Productivity. ASHE-ERIC Higher Education Report*. ERIC.

Klepper, S., Krusche, S., Peters, S., Bruegge, B., and Alperowitz, L. (2015). Introducing continuous delivery of mobile apps in a corporate environment: A case study. In *Proceedings of the 2nd International Workshop on Rapid Continuous Software Engineering*, pages 5–11. IEEE/ACM.

Kolb, D. (1984). *Experiential learning: Experience as the source of learning and development*. Prentice Hall.

Korthagen, F., Kessels, J., Koster, B., Lagerwerf, B., and Wubbels, T. (2001). *Linking practice and theory: The pedagogy of realistic teacher education*. Routledge.

Kothiyal, A., Majumdar, R., Murthy, S., and Iyer, S. (2013). Effect of think-pair-share in a large cs1 class: 83% sustained engagement. In *Proceedings of the 9th annual conference on International computing education research*, pages 137–144. ACM.

Krathwohl, D. (2002). A revision of bloom's taxonomy: An overview. *Theory into Practice*, 41(4):212–218.

Kraut, R., Fish, R., Root, R., and Chalfonte, B. (1990). Informal communication in organizations: Form, function, and technology. In *Human reactions to technology: Claremont symposium on applied social psychology*, pages 145–199. Citeseer.

Krusche, S. and Alperowitz, L. (2014). Introduction of Continuous Delivery in Multi-Customer Project Courses. In *Proceedings of the 36th International Conference on Software Engineering*, pages 335–343. IEEE.

Krusche, S., Alperowitz, L., Bruegge, B., and Wagner, M. (2014). Rugby: An agile process model based on continuous delivery. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pages 42–50. ACM.

Krusche, S., Berisha, M., and Bruegge, B. (2016). Teaching Code Review Management using Branch Based Workflows. In *Companion Proceedings of the 38th International Conference on Software Engineering*. IEEE.

Krusche, S. and Bruegge, B. (2014). User feedback in mobile development. In *Proceedings of the 2nd International Workshop on Mobile Development Lifecycle*, pages 25–26. ACM.

Krusche, S., Seitz, A., Börstler, J., and Bruegge, B. (2017). Interactive learning: Increasing student participation through shorter exercise cycles. In *Proceedings of the 19th Australasian Computing Education Conference*. ACM.

Kurtz, B., Fenwick, J., Tashakkori, R., Esmail, A., and Tate, S. (2014). Active learning during lecture using tablets. In *Proceedings of the 45th technical symposium on computer science education*, pages 121–126. ACM.

Michael, J. (2006). Where's the evidence that active learning works? *Advances in Physiology Education*, 30(4):159–167.

Prince, M. (2004). Does active learning work? a review of the research. *Journal of Engineering Education*, 93(4):223–231.

Schwaber, K. (1995). Scrum development process. In *Proceedings of the OOPSLA Workshop on Business Object Design and Information*.

Shaffer, D. (2004). Pedagogical praxis: The professions as models for postindustrial education. *Teachers College Record*, 106(7):1401–1421.

Whitehead, J. (2007). Collaboration in software engineering: A roadmap. *FOSE*, 7(2007):214–225.

Wujec, T. (2010). The Marshmallow Challenge - TED Talk. Retrieved January 08, 2016 from `http://marshmallowchallenge.com`.

8.3 Chaordic Learning: A Case Study

This conference paper describes chaordic learning, a self-organizing, adaptive, and non-linear learning approach, stimulating students' creative thinking. Instructors provide structure and guidance and integrate freedom for self-organization and self-guided learning and embrace innovation and creativity. Deviations are opportunities, and failures are possibilities for students to learn and improve. The paper presents two case studies on courses that use chaordic learning: a games development course and a joint advanced student school. Students in these courses report increased intrinsic motivation, a higher level of self-organization, and more room for creativity leading to an improved learning experience and more fun. Chaordic learning complements interactive learning and emphasizes the creative aspects of learning.

Authors	**S. Krusche**, B. Bruegge, I. Camilleri, K. Krinkin, A. Seitz and C. Wöbker
Conference	39th International Conference on Software Engineering
Publisher	IEEE
Pages	10
Type	Conference: Full Research Paper
Review	Peer Reviewed (2 Reviewers)
Year	2017
Citation	[KBC+17]
DOI	https://doi.org/10.1109/ICSE-SEET.2017.21

2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track

Chaordic Learning: A Case Study

Stephan Krusche
Technische Universität München
Munich, Germany
krusche@in.tum.de

Bernd Bruegge
Technische Universität München
Munich, Germany
bruegge@in.tum.de

Irina Camilleri
Technische Universität München
Munich, Germany
irina.camilleri@tum.de

Kirill Krinkin
Saint Petersburg Electrotechnical University
St. Petersburg, Russia
kirill.krinkin@fruct.org

Andreas Seitz
Technische Universität München
Munich, Germany
seitz@in.tum.de

Cecil Wöbker
Technische Universität München
Munich, Germany
woebker@in.tum.de

***Abstract*—Software engineering is an interactive, collaborative and creative activity that cannot be entirely planned. Inspection and adaption are required to cope with changes during the development process. Software engineering education requires practical application of knowledge, but it is challenging and time consuming for instructors to evaluate the creation of innovative solutions to problems. Current higher education practices lead to a multitude of rules, guidelines and order. Instructors see deviations of students as failures and limit the creative thinking processes of students.**

In this paper we describe chaordic learning, a self-organizing, adaptive and nonlinear learning approach, to stimulate the creative thinking of students. Instructors provide structure and guidance, but also integrate freedom for self-organization and self-guided learning and embrace innovation and creativity. Deviations are seen as opportunities and failures as possibilities for students to learn and improve. We introduced chaordic learning into a games development course and a joint advanced student school and describe the chaordic process of these courses as case studies. Students in these courses report about an increased intrinsic motivation, a higher level of self-organization and more room for creativity leading to an improved learning experience and more fun.

***Keywords*-Experiential Learning, Agile Methods, Creativity, Chaos, Order, Self-guided Learning, Self-organization**

I. Introduction

Software engineering (SE) has undergone several fundamental shifts since the term was first proposed in the 1960s when software systems became larger and more complex to develop [1]. Initially, the focus was on defined process control and ordered processes to mimic established manufacturing processes [1]. In recent years, the emphasis changed towards empirical process control that embraces frequent inspection and adaption to react to changing environments [2]. Software development consists of experimental knowledge work where creativity is important [3]. SE is the process of creating software solutions that have - in their entirety - not been developed before.

SE education requires practical application of knowledge [4], [5], in particular interaction and collaboration [6]. Current higher education practices lead to a multitude of rules, guidelines and order, which in turn "makes creative teachers and students feel less and less in place" [7]. University courses focus too much on analytical and logical thinking, because this is easier to teach and to grade.

Instructors follow a defined teaching approach and see deviations of students as failures. They prematurely set up structures (e.g. assignment requirements, syllabus, instructions) without taking time to determine the desired learning outcomes and how the process of learning should be conducted. When all educational activities are well defined, students only follow instructions, but do not use their creative and intuitive thinking processes and do not learn to act themselves.

Some instructors even sanction creativity if students' solutions to exercises do not follow correctness criteria, even if they include innovative aspects. In such cases, there is no room for self-organization, as instructors fear this might lead to chaos, in particular if students are unexperienced. This leads to a gap between skills that current SE education provides and skills that are required in industry and research.

Chaordic learning is an approach to overcome these problems by balancing education between chaos and order. Instructors define the learning environment and high-level learning goals to provide structure and order. Students have the freedom to choose the concrete learning activities and the solution approaches to given problems to allow creativity, innovation and self-guided learning. Instead of defining all learning steps, instructors provide guidance in this approach and give feedback to the students' learning progress. Examples of courses following this idea are capstone courses as described in [8], [9], and [10]. Capstone courses provide students with a real-life experience to prepare them for their future career [11].

The paper is structured as follows. Section II provides background information, defines the term chaordic, describes the learning organization that focuses on individuals, and presents agile methods as concrete practice for a chaordic development process. In Section III, we define the chaordic learning approach, which is based on a design process, and we present its properties and its benefits. Section IV shows two case studies, in which we applied chaordic learning, a games

978-1-5386-2671-9/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-SEET.2017.21

development course and a joint advanced student school. In Section V, we discuss other chaordic learning courses, constructivism and design thinking as related work. Section VI concludes the paper.

II. Foundations

In the 1960s, software systems became larger and more complex to develop [1]. Projects failed because development concepts and methods were missing and teams worked together in rather chaotic and unstructured ways. The term software engineering (SE) "was deliberately chosen as being provocative, in implying the need for software manufacture to be [based] on the types of theoretical foundations and practical disciplines[,] that are traditional in the established branches of engineering" [12], [1]. Detailed process models emerged that describe the process of engineering software, resulting in a structured software process. The aim was to bring **order and control** into the development approach following strict rules and avoiding deviations, which were seen as errors that need to be corrected.

In the following years, software developers increasingly recognized that too much order and control is counterproductive and that the essence of SE is to deal with changes: defined process models are not capable of addressing this need [13]. SE consists of experimental knowledge work where creativity is important, including unexpected events, incidents and uncertainty [3]. Software development is a complex process with random variables, that cannot be defined completely deterministic: "It is typical to adopt the defined (theoretical) modeling approach when the underlying mechanisms by which a process operates are reasonably well understood. When the process is too complicated for the defined approach, the empirical approach is the appropriate choice" [2].

As response, agile methods emerged with the philosophy that software development should follow an empirical approach: still structured, but not entirely planned, and thus providing more freedom to adapt to changes. Deviations, errors and failures are seen as opportunities to inspect, adapt and improve the methodology. If random variables such as uncertainty and change are allowed, the process control is stochastic (nondeterministic) and allows **chaos**. Stochastic control is not solved analytically and deals with the existence of uncertainty and chaos [14]. Empirical process control balances between **chaos** (stochastic control) and **order** (defined control).

A. Software Engineering Education

Several pedagogic theories have been developed and integrated into education. Educators recognized that SE education requires practical application of knowledge [4], [5], in particular interaction and collaboration [6]. Experiential learning is a methodology in which educators engage with learners in direct experience to increase knowledge, develop skills, and clarify values [15].

Problem-based orientation allows students to identify what they know, what they need to know, and how and where to access new information that leads to the resolution of a particular problem [16]. It is one important aspect in capstone courses [17].

The introduction of computers in education enables students to learn through the delivery of content and instructions via computer mediated activities, digital and online media [18]. It promotes simultaneous, independent and collaborative learning experiences.

In cooperative learning approaches, students work in groups to complete tasks collectively towards a common goal and the educator's role changes from giving information to facilitating learning [19].

Active learning is an educational approach to increase involvement and excitement with the subject being taught. Instead of learners acting as receivers of knowledge by passively listening to lectures, active learning puts the emphasis on developing learner skills and engaging them in activities.

B. Chaordic

In 1995, Hock first coined the term *chaord*[1] [20]. This neologism is the combination of the words **cha**os and **ord**er, meaning a state in between that adapts the principles and properties of both. Hock shares his experiences in managing organizations and concludes that a **chaordic** approach is required in complex situations in order to facilitate innovation and creativity.

> "By chaord, I mean any self-organizing, adaptive, non-linear complex system, wether physical, biological, or social, the behavior of which exhibits characteristics of both order and chaos or loosely translated to business terminology, cooperation and competition." [20]

Complex systems arise and thrive on the edge of chaos with just enough order to give them pattern [21]. Chaord is a universal concept that can be applied to different systems and environments. The core principles of chaos and order are essential for a chaordic system. There is an aversion to disorder in today's world [22], because disorder means a loss of control. Since chaos can lead to disorder, it is important to overcome the fear against disorder to make a positive use of chaos, which is central to the idea of chaord. Chaos is used to increase responsiveness and adaptivity to the environment and order is used to keep the system stable and to make sure that its boundaries are not violated.

Chaordic means organizing and shaping a system in a way to be able to adapt to a changing environment. It can be applied to different types of contexts [21]:

- *Organization:* working environment that is governed by both structure and chaos leading to more innovation
- *Business:* organization that harmoniously blends characteristics of competition and cooperation
- *Leadership:* combining induced and compelled behavior in the relationship between a leader and a follower
- *Education:* approach that seamlessly blends theoretical and experiential learning

[1] "Chaord" is a substantive, "chaordic" is the corresponding adjective.

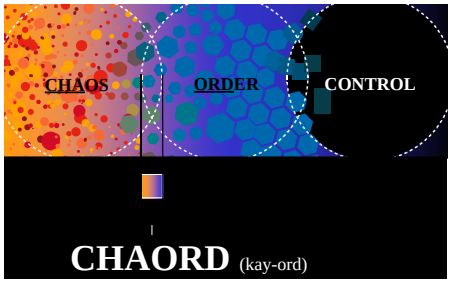


Fig. 1. Chaord is a state between chaos and order (adapted from http://www.chaordic.org)

Organizational forms are still following industrial principles defined more than a century ago [23]. Structure and order are essential in these traditional organizational forms and are often seen as more important than individuals which are treated like gearwheels that either function or get replaced [20]. A chaordic organization however includes characteristics of chaos and order at the same time. It focuses on the individual people in the organization and allows them to innovate and to think out of the box [24]. How decisions are managed plays an important role in governing the organization effectively [21]. Governance in chaordic organizations is distributed to many different people and stakeholders. This is contrary to the traditional, more hierarchical view of organizational structure.

A chaordic system, or a chaordic organization in particular, is defined by the following characteristics [20]: power and function of the system are distributed. There is no single person controlling the whole system or governing all organizational functions. The system is self-organizing: there is no external system or individual required to make the original system last and function correctly. Collaboration and competition are both integrated into the system. The system is malleable and durable: it can adapt while staying successful in a changing environment. The system is owned cooperatively and equitably: everyone is involved in the system's continuing success.

C. Learning Organizations

The learning organization is "a form of organization that enables the learning of its members in such a way, that it creates positively valued outcomes, such as innovation, efficiency, better alignment with the environment and competitive advantage" [25]. While there used to be a divide between individuals and their associated organizations, there is now a consensus that being successful requires collaboration with one another [26]. Individuals play a more important role in today's organizations and especially in how those organizations learn and adapt. This has become important since workers increasingly look for more flexible working arrangement which in turn can increase their satisfaction and commitment [27].

By adapting to the needs of individuals and helping them produce knowledge, organizations can profit themselves. They need to continuously listen to the wishes of their people and transform accordingly. The individual's knowledge can be used to improve organizations or the products and services they provide. The success of the individual is directly linked to the success of the organization as a whole, because knowledge is created when people interact with each other [26].

Learning is an activity of interdependent people working collaboratively with each other [28]. A learning organization is a place where people are "continually discovering how they create their reality. And how they can change it" [29]. Companies applying the principles of learning organizations act between chaos and order, so they are in a chaordic state [20]. Enabling people to learn in these organizations depends on two key factors. Organizations have to provide the possibility and the options for their people to advance themselves, while also providing the culture to learn in the first place [29]. When organizations follow these principles, both the organization itself and its individuals benefit from the chaordic approach. By ensuring that the individual can learn in a chaordic environment, the organization itself can become a learning organization and can improve its efficiency. This is especially important to be successful in the long-run.

D. Agile Methods

In the 1990's, agile development methods emerged with the philosophy that software developement should follow an empirical process model. The empirical model for Scrum described by Schwaber [30] is based on Ogunnaike's definition of a stochastic model [2]. It handles changes and failures as opportunities: a quick reaction to these can lead to advantages compared to competitors. The agile manifesto (visualized between chaos and order in Figure 2) identified new ways of developing software by moving the focus from complete order in traditional development processes towards a more lightweight methodology in between chaos and order [31].

Agile methods play an increasing role in SE [32] and in its education [33]. They allow developers to stay creative while working on complex problems and projects. A team that applies agile methods is self-organizing: the team decides about the concrete procedures, and learns from problems, risks, and failures allowing a more chaotic way of thinking. There are also members in the team, such as the scrum master, who have control over the process by moderating and negotiating with the rest of the team, leading to an ordered course of action. The acceptance of failure is an important aspect of agile methods. Realizing that a mistake has been made, learning from it and iterating on the solution to fix the mistake is of importance to the success in a complex software project.

Due to the application of empirical process control which could also be called chaordic process control, agile methods shifted the focus in SE from order (defined process control) towards chaos (stochastic process control). They include a mix of order and chaos and balance between both. On the one hand, they provide a structured framework with principles such as daily meetings with specific questions or procedures how to organize iterations. On the other hand, they are open to concrete methods, tools and workflows suggested by devel-

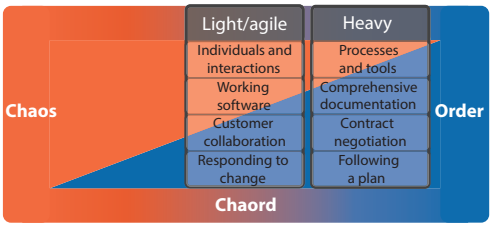


Fig. 2. Agile methods are an instance of a chaordic setup in between order and chaos.

opers, allow changes and facilitate creativity. Therefore, agile methods are a concrete instance of a chaordic setup that has a direct application and impact on the software development process. The goal is to give freedom to the development team while setting the necessary boundaries to control it and to lead it into the right direction.

III. Chaordic Learning

Chaordic learning is an educational approach that "seamlessly blends theoretical and experimental learning" [20] and includes aspects of order and chaos. In this context, structured courses with detailed instructions represent order, while experimental learning and educational innovation represent chaos.

While chaordic learning moves control to students, instructors still provide guidance[2] to ensure that students understand theory in the right way and to avoid misconceptions. Chaordic learning puts the instructor and the students on the same level to remove hierarchies and to improve collaboration. It includes the idea of cognitive apprenticeship [35]: an apprentice (the learner) observes the skills of a master (the educator) who shows how a concept works in practice. Clarifying the thinking process behind the application of the concept makes it easier for the apprentice to imitate the behavior.

Chaordic learning does not require the abandonment of structured learning, but a shift from traditional educational approaches would require creating room for chaos. This can be achieved by promoting creative problem solving experiments, where students apply principles learnt in theory or propose own new ideas. Experimentation and collaboration should be rewarded, while instructors focus on personalization, cooperation and informal learning.

Software development is the process of creating software solutions that have never been created before in the same concrete configuration. Within the context of SE education, development of creative problem solving skills is not addressed properly. We believe chaordic learning can bridge the gap between the skills current education provides and those the SE discipline requires. The encouragement of experimental learning (chaos) teaches the creative skills required in SE. A chaordic balance can be introduced in the form of project courses, where instructors provide a rough structure for the learning environment and the learning goals and students work in a self-organizing, adapting way to achieve them.

[2]Chaordic learning is not to be confused with unguided or minimal guided learning approaches that fail to improve the learning experience [34].

A. The Chaordic Design Process

We propose the following chaordic design process including six steps to help instructors facilitate the creation of chaordic learning environments. These steps were adapted, to an educational context, based on the work of Hock on creating a chaordic organization. It is important to point out that such environments cannot be created by instructors alone, but rather through collaboration between students and instructors, where students take charge of their own learning.

1) Purpose: Define the learning objectives of the course and a common understanding of what students aim to gain upon completion. Questions, such as "Why are you here?" often help illuminate purpose. This component is the key to motivating students - educational efforts must feel meaningful. Flexibility in course structure should allow for student input. Depending on the content, students can be asked to narrow down a set of requirements or choose technologies they feel are suitable or wish to explore.

2) Principles: Discuss the fundamental beliefs of how students and instructors shall conduct themselves in pursuit of new knowledge. An example would be emphasis on the positive outcome of learning from failure. Thus, experimentation can be seen as more valuable than a perfect solution. The grading scheme should also reflect these principles whereby students are not penalized for failure.

3) People: Identify people and institutions necessary to achieve the defined purpose. Chaordic principles encourage collaboration across program and institutional boundaries, with government and industry, both national and international. Students should consider engaging the broader community. This mindset helps students appreciate the full context of their learning experience, e.g. the context their software is being developed in. A concrete example taken from agile practices is the inclusion of the customer as participant of the team.

4) Concept: Create key guidance for interaction among the participants in a generally flat chaordic hierarchy. Being the chaordic equivalent of an organizational chart, it includes forming teams and introducing student support mechanisms (e.g. teaching assistants). Students can conceive a new organizational structure that is effective with respect to all team members and that defines how work is distributed internally. Teaching staff should be seen as facilitators and should explain who, how and when students can ask for help. Individual roles - such as those borrowed from agile practices, (e.g. scrum master) could rotate.

5) Structure: Embed the learning goals, roles and responsibilities in the course's official structure. In a higher-education context, it is the course description, assignment requirements, due dates and credits awarded for completion. Assignment requirements should be flexible to allow room for incorporating new ideas or specifying detailed requirements. The grading scheme should reflect and encourage experimentation.

6) Practices: Decide on the activities participants are required to undertake. There should be an emphasis to promote activities inspiring innovation. Mechanisms whereby instructors negotiate with students about their work should be established (e.g. by discussing status, impediments, promises). Work itself e.g. could be organized by using a software configuration management component, such as version control, with changes being reviewed through pull requests [36]. A series of meeting practices helps disseminate information. A course-wide meeting could be used for teams to share their progress, while regular team meetings allow internal discussions of problems or ideas. Teaching staff can guide students to employ best practices, but individual students or teams are allowed to choose own internal team practices.

The chaordic design process is iterative and adaptive, so a decision in each step should be reflected in all other steps. Modifications or refinements of elements may shed light on changes required in other elements. Over time, the environment is established when all elements are defined. Figure 3 shows the six steps of the chaordic design process combined in a creative figure that exemplifies its ideation aspect.

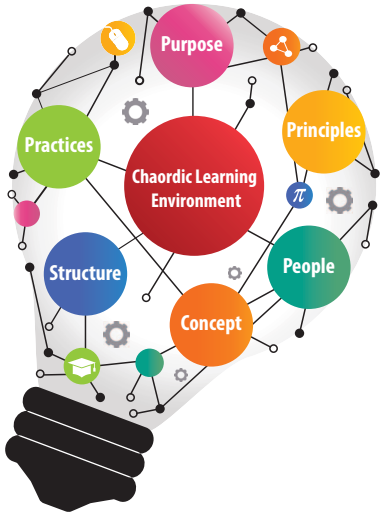


Fig. 3. The chaordic design process includes six steps to facilitate the creation of the chaordic learning environment.

The key to applying chaordic principles in an educational setting is negotiating each of these steps with the students. Students can choose to not use provided infrastructure, adopt own practices or team roles. That way each of the choices being made follows a deeper understanding of the options and the rationale behind them. Instructors act as enablers and moderators, who help students explore, rather than showing them a well-trodden path. A quote by Hock also reflects this well: "In the chaordic age success will depend less on rote and more on reason; less on authority of the few and more on the judgement of many; less on compulsion and more on motivation; less on external control of people and more on internal discipline" [21].

B. Properties

While these six steps in the chaordic design process help to create chaordic courses, they do not necessarily include chaotic properties and instructors might still focus too much on structure and order. It is important to understand the chaotic nature and to include it into the course. There are five principles of chaos [37] that are prevalent in chaordic learning:

1) Consciousness: The essential ground state of learning is mind, more than matter. Ideas are primary and drive the learning experience of students.

2) Connectivity: The learning experience is related to its environment and its context and cannot be seen as independent element. Educators and learners interact and collaborate with each other towards a common goal.

3) Indeterminacy: Learning cause and effect are intertwined. There is no single entity that can completely plan or control what exactly is learned. While the learning process is framed, no one is able to predict the exact next action in the learning process. Guidance and feedback by instructors determine the right direction.

4) Emergence: The learning experience is constantly growing more complex, more coherent and more differentiated, but at the same time not descending into chaos. Students create coherent networks and relationships between knowledge through self-organization.

5) Dissipation: Small learning peer groups are being formed and dissolved based on purpose creating dialogues between students and instructors and generating new knowledge.

These chaos properties provide a mindset that should be included in the design of chaordic courses. They make sure that students evolve their creative skills. However, instructors can not only focus on chaos properties, they also need to incorporate structure into the courses, e.g. with milestones, intermediate deadlines and control points where students obtain feedback and where they can evaluate their learning progress. Examples would be daily or weekly meetings where students report about their progress, their challenges and their promises, while instructors listen and provide feedback and guidance.

C. Benefits

In our experience, chaordic learning leads to the following benefits when it is applied properly in SE education:

1) Increased motivation: Students who can influence the learning subject, methods and process, have a higher intrinsic motivation because they have control over their learning experience and over the learning outcome.

2) Increased self-organization: As the instructor steps aside, students have to act, organize themselves and are responsible for the progress. They have to find the right learning activities and actively ask for feedback.

3) Less hierarchies: Students are treated on the same level as instructors, leading to an improved discussion between students and instructors. Students then come up with new ideas and suggestions to form the learning process.

4) Improved learning from failures: Instructors create a culture of successful failures: they emphasize that students are

allowed to make failures and to learn from these failures. This lowers the pressure and allows students to experiment new approaches and to think out of the box.

5) More room for creativity: Decreased pressure to succeed in every step facilitates out of the box thinking and the application of new, innovative approaches.

It is important that instructors facilitate the chaordic learning approach and provide guidance. In particular, unexperienced students will not be able to immediately self-organize themselves. Instructors need to clearly communicate the chaordic learning environment and the focus towards self-organization, innovation and creativity. They ask students in the beginning of the course to actively organize themselves and make the self-organization part of the assessment. In the first weeks, they help unexperienced students to organize themselves without taking over the organization. Frequent inspection and adaption, as in empirical process control, is required to control the learning outcome and to prevent misconceptions.

IV. Case Study

In this section, we present two courses in which we applied chaordic learning to improve the learning experience of students. The first course is on mobile games development, the second course is an international student school on software development for mobile platforms and the internet of things.

A. Games Development Course

The games development course is a two-week block course with up to 40 students. We described the structure of this course in [38]. In this paper, we summarize the course and focus on its chaordic learning characteristics. We describe course design and learning objectives using the six steps of the chaordic design process and the outcome of the course.

1) Chaordic Design Process: While the course is structured in the first week where students learn programming and games development, it provides a lot freedom and facilitates self-organization in the second week.

Purpose: The course is focused on learning games design and iOS app development using the Swift programming language. It includes various topics students get to explore: programming, use of platform specific frameworks, distributed version control, game design, application of design and architectural patterns, and user interface design for games.

Depending on previous experience and interest, students participate in the beginner or advanced modes. The beginner mode targets students who have no experience with Swift and iOS development. However, the course assumes that beginners have basic knowledge of an object-oriented programming language and UML. During the first week, the focus for these students is on learning Swift, the use of iOS frameworks and the Xcode IDE. In the second week, students apply their knowledge in small team projects.

In advanced mode, students participate as teaching assistants (TAs) who have already acquired knowledge of the development environment, including programming language and tool chain. The objectives for TAs include specializing in a technical topic, such as sprite animation within games, and helping beginners during the course. To achieve this, TAs prepare an interactive tutorial on their chosen topic under the supervision of an instructor and present this during the first week of the course. They also guide teams and individuals to apply games development practices in their projects to learn from known methods that achieve best results.

All students practice social and non-technical skills such as working in a team, presenting, communicating and being proactive. Learning objectives and assessment structure of the course are made clear on the first day. During the introduction round, students share "Why are they here?" and what they wish to gain from attending the course. This helps choose topics of interest, set personal goals and illuminate the purpose for the students.

Principles: Students can choose their own game idea and select the programming concepts, frameworks and technologies they want to apply for their game. This increases the motivation of the students because they can try their own ideas without limiting their creativity. Ambitious students might go for more complex 3D technologies, while others prefer the development of simple jump and run games in 2D.

In addition, they have the ability to adapt the game idea during the development. Instructors and TAs provide guidance: students should implement a simple game with easy to apply technologies that is extensible with new features later on. This guidance should prevent them from starting with overly complex games they would not be able to implement in a week. It facilitates the prototyping idea that is important in today's SE practices. In the intermediate milestone in the second week, the teams present the game idea and justify the chosen technologies. They obtain feedback and decide on their own whether and how to include it.

While the first week provides basic knowledge about game development techniques, students need to obtain further knowledge on their own for details of a specific technique by reading references or by using further tutorials. Students also need to organize their work in teams on their own.

People: Students have to interact with instructors, TAs and their peer students. Two instructors organize the course, one is responsible for introducing Swift, the other one is familiar with games development. Instructors select students, prepare presentations and set up infrastructure. Four to six TAs help in the organization. Game experts from industry provide help for the design of graphics and animations.

Concept: The course promotes a flat hierarchy between instructors, TAs and students and facilitates feedback in both directions, from students to teaching staff and from teaching staff to students. This increases the commitment and the motivation of students and decreases communication barriers.

Structure: The structure of the course is defined by its schedule in the first week and its milestones in the second week. In the first week, instructors and TAs give three tutorials per day based on interactive learning [39] that are required for games development. These tutorials are 90 min long and include a mix of theory, examples, exercises, sample solutions

and reflections about the taught concept and its usefulness in different programming situations. During a tutorial, TAs walk around and help students if they face problems. After a tutorial, students get 30 min to solve an exercise on their own that is based on the tutorial content.

In the second week, students build teams to develop their own game idea. They face several milestones: start of development, intermediate feedback and final presentation. These milestones are made clear on the first day. Otherwise, the students can freely choose when, how and where to develop their game providing freedom and a necessity to self-organization. The intermediate milestone is two days after the start of the second week, where the teams informally present their ideas and current progress.

The course phase ends with the final presentations of all teams, which are recorded. Students receive the videos and obtain feedback on content and delivery from the TAs and the instructors. Students are assessed on the following criteria in a descending emphasis: originality and complexity of the game, how well it is implemented, how it was designed and on an additional documentation where the students show UML diagrams and explain those. Students can receive the best grade with both - an unfinished game using complex technologies and a simple game with completed functionality. This assessment scheme is made clear to the students in the beginning of the course, so students are encouraged to experiment knowing that the difficulty of their choices is taken into account during assessment.

Practices: The course does not formally present project management techniques such as the agile methodology Scrum [30] to avoid an increase of its complexity. Instead, only some of the practices, such as stand-up meetings, are introduced to students. These are used to stay organized by presenting the minimum viable products throughout the day. Students work in pairs for one week which encourages the cooperative and self-guided learning process: they work towards a common goal and have the freedom to organize themselves however they want. This gives them the confidence in their programming abilities, required to build larger applications, while at the same time not limiting them to mere implementation of small programming assignments.

2) Outcome: At the end of the course, each team presents their game in a short Pecha Kucha presentation [40] showing 20 slides, each in 20 seconds with automatic transitions. During the presentation, they focus on the game idea, the proposed game features, the software architecture, the object design, the used technologies and frameworks, status and outlook. In addition, they include a demo of the game where they show the most interesting features.

After the course, instructors encourage and further accompany students to finalize and submit their game to the iOS AppStore. Until now, 18 teams published their games.

Our evaluations show that students appreciate a great learning experience with practical aspects that they can further use in their career, as well as the possibility to work self-organized and to integrate their own ideas. They improve their SE abilities, in particular object-oriented programming, and their soft skills with the help of games development and they have a lot of fun.

B. International Student School

The second course is a one week course with 20 students and four instructors. The course on SE in the *Joint Advanced Student School* (JASS) has been conducted for the third time in March 2016, in St. Petersburg, Russia in the office of JetBrains[3], following two previous courses in 2008 and 2012. The idea is to bring together students with different cultural background (Germany and Russia), to work together on innovative topics in SE in a chaordic manner. 10 German students from Munich and 10 Russian students from St. Petersburg participated in 2016. We describe course design and learning objectives using the six steps of the chaordic design process and the outcome of the course.

1) Chaordic Design Process: The topic of JASS 2016 was "software development for mobile platforms and the internet of things (IoT)". Students are organized into balanced teams, each with four students (two Russian and two German students), to work on small projects relevant to this topic for one week while also spending time on sightseeing and team formation. While the project topics are roughly defined by the instructors, the teams have the freedom to adapt the topics and to include their own ideas.

Purpose: Students learn to develop innovative apps for mobile and IoT platforms and get in contact with technologies, such as iOS, Android, micro-controllers, drones, and sensors. This combination enables new scenarios and requires the ability to connect sensors of IoT devices with visualization components and sensors on mobile devices. Students acquire soft skills, gain experiences with international teams and become familiar with latest innovations and technologies. Participants collect experiences in global SE by working in international teams.

Principles: At the core of the educational offering is a more engaging teaching method that comes with less control and moderation and facilitates creativity. The freedom of choice with regards to tools, processes and technologies forces students to organize themselves as a team and to overcome the problems on their own. Students can choose the project topic and influence the problem statement and the requirements. This allows them to include their own ideas and to try out new, innovative approaches.

People: Four instructors organize the course. They arrange the travel, housing, working environment and catering. Although the project environment is established, no strict rules, nor processes of how the projects should evolve are defined. Students apply for the school by providing personal details and a short motivational letter. The instructors review these applications and select a heterogenous group of students with different levels of experience.

[3]JetBrains Research: https://research.jetbrains.org

Concept: The course promotes a flat hierarchy between instructors and students and facilitates feedback in both directions. This increases commitment and motivation and decreases communication barriers: No hierarchies within the team and flat hierarchies between instructors and students.

Structure: The course is scheduled for six days. Students spend about four days for development and two days for sightseeing and self-organized team formation events. The course starts with a short introduction round of the participants, followed by the presentations of the project ideas. The projects are framed upfront by the instructors in a short problem statement that is handed out to the students.

Instructors emphasize that the given problem statements are only starting points and that the students can integrate their own ideas. Relevant hardware and software needed to work on the different projects (e.g. drones, sensors, beacons and mobile devices) are provided to the students. After the team assignment, instructors conduct an icebreaker to lower barriers between team members. Teams organize themselves, choosing which tools and processes to use, and distributing responsibilities. Teams receive regular feedback in a daily standup meeting where they report on progress and impediments, and promise what they will achieve until the next standup meeting. These meetings are similar to standup meetings in Scrum [30].

In a rotating manner each team member reports on its team's progress. The discussion with instructors and other team members is considered as valuable feedback. On the last day of the school, each team presents their project including the design of the system, problem description, motivation, introduction of team members and a live demo. Assessment is based on the final presentation and demonstration of the app functionality. As with the games development course's assessment structure, difficulty reflected in students' choices with respect to technology and functionality is rewarded. Aspects like motivation, personal progress, team skills and quality of the final presentation are also considered in addition to the project functionality. The course is not part of the curriculum: all students participate voluntarily.

Practices: In line with the agile manifesto, the school focuses more on individuals and interactions, than processes and tools and aims for a working prototype instead of documentation. Daily standup meetings structure the communication between instructors and students. Instructors are available for additional questions and guidance. Social activities are an important aspect of the course. All participants have breakfast, lunch and dinner together and go out in the evenings. This motivates the students and strengthens the international teams. The school does not focus on formal management processes, so it is up to the students to organize themselves and work as team to present a viable product at the end of the school.

2) Outcome: After a week of design, development and integration, all teams presented their projects. Students also produce a short, creative trailer to visualize the project idea. The following five projects were presented in 2016:

Multimodel iNTeraction (MiNT): A modeling tool which facilitates real-time collaboration on models during early stage requirements engineering. MiNT allows to collaborate on the creation and editing of informal models, and enables collaborative modeling across iOS and Android devices.

Quadcopter Autopilot: The goal of this project was the creation of an app that allows to capture dynamic scenes using a quadcopter autopilot in several operation modes.

KneeHapp - Rehabilitation Monitoring on the Wrist: KneeHapp is a smart knee bandage that aims to support patients suffering from ligament ruptures. Patients perform required exercises correctly and provide relevant metrics to a doctor over a smartphone app to enable better treatment decisions [41]. KneeHapp Watch extends the smartphone app with a component for the Apple Watch. The smart watch enables the display of real time exercise data directly on a patient's wrist improving the user experience during exercises.

Octopus - Mobile First Responder for Emergencies: Octopus is a sensor system for optimizing the treatment in emergency situations. Octopus consists of a set of hardware sensors (e.g. heart rate, breathing, brain activity sensor) and mobile devices that are used to collect patient data inside the ambulance while the patient is transported to the hospital.

Geo Quest Travel Game: The smartphone app helps travelers to plan city trips with regard to points of interest and food recommendations. It creates individual quests to make city trips more informative and interesting.

C. Discussion

Chaordic courses benefit from flat hierarchies, but also need trust to function properly. Instructors need to be able to move control over the teaching and learning process to students, while still providing enough guidance to keep the learning outcome on a qualitatively high level. It is essential for instructors to provide guidances and feedback to students to control the learning outcome [34]. Instructors need to identify situations where students are learning incorrect approaches. Then, they need to interrupt the process and guide the students back to the correct path. Students need to be comfortable with openness and self-organization to benefit from the chaordic education approach. If students prefer structured courses where all instructions are provided, the chaordic approach will fail.

To summarize, it is essential for students to actively participate in the process and to contribute their own opinions and ideas to the course. Shy students may have an issue with the chaordic approach and should be especially treated to get out of their comfort zone to make sure that they also succeed. We evaluated the case studies in informal discussions and have received many positive comments about the chaordic learning approach. Students have reported an increased motivation, higher level of self-organization and more room for creativity. They appreciate the openness of the courses and the low hierarchies between instructors and students. In particular, they like that they can try out new approaches and have the freedom to make failures and to learn from successful failures.

V. Related Work

In this section, we relate our chaordic learning approach to another chaordic learning context, describe the relation

between chaordic learning and constructivism and show how design thinking relates to chaordic learning.

A. Chaordic Learning Context

Leigh argues that "structured activities have the potential to create unpredictable learning contexts" [22]. Simulations and games can be used in cooperation with chaordic approaches inside of a traditional classroom setting. Like in our case study, instructors shift from an orchestrating role to a supporting one.

Leigh uses open simulations, which come with a high uncertainty regarding the possible outcome. She focuses how the chaos framework can be useful in understanding complex interactions. These complex interactions can occur when giving up control over the learning process. Furthermore, Leigh describes what it means to give control away as an instructor and how trying to reestablish order can actually hinder the process of chaordic learning itself. Depending on how instructors react to chaotic circumstances within the chaordic learning experience, the approach can either fail or succeed. It might be difficult to create a positive experience for instructors since they have to adapt their teaching style to a new approach and since transferring control to students can be intimidating.

Our paper focuses on SE education in particular and the different organizational structures that surround it. Furthermore, we directly apply the chaordic organization principles by Hock to the learning contexts and focus on how to give control to students. One of the key aspects is how instructors can steer this process and which decisions they allow the students to make on their own. The students in our chaordic courses have freedom to choose different aspects in a software project and can also choose their own responsibilities in this process. The distribution of the decisions, between instructors and students, play a major role in a chaordic learning approach.

B. Chaordic Learning and Constructivism

Desouza identified the role of 'Radical Engineers' in [42], who are dealing with chaotic requirements, opportunities and problems; they allow projects to incorporate innovative ideas and tools. In this work, the authors do not use the term chaordic, but incorporate the idea behind it. Their conclusion is that working in a chaordic manner is essential for software development projects to provide flexibility to students and to develop a corresponding set of skills. The learning process itself has a chaordic nature. Vygotskii in his works [43] pointed that chaos (and chaos ordering) is a base for learning and acquiring new knowledge.

The growing trend in SE is that development happens under conditions of notable uncertainty. There are no standard rules or methods for setting up the development process itself because it is hard to define the concrete roles which are distributed in the team. Veli-Pekka [44] suggested the following organizational pattern: "develop the development culture before process" and after that let people be self organized.

Uncertainty in requirements and environment requires to experiment not only before the development process was set up, but even during the development. According to Thomke [45], an innovative process (and we believe, that software development is about innovation) encompasses success and failure; it is an iterative process of understanding what does work and what does not. The obvious consequence is that the learning process should have a similar chaordic structure, i.e. incorporate failures and successes and learn from experimentation to create successful failures.

Learning by experimentation is a philosophical viewpoint covered in the constructive approach that tries to describe the nature of learning [46], [47]. It focuses on hands-on approaches where students experiment with concrete problems, try out methods and techniques and learn from the reflection about their usage. Students then make their own inferences, discoveries and conclusions and adapt the behavior. As such, constructivism promotes the chaos side of the chaordic learning process and influences the idea of the chaordic learning approach described in this paper.

C. Design Thinking

Design thinking was originally explored and developed as a human-centric methodology to solve complex creative problems closely associated with conceptual design [48]. Different versions of the design thinking process exist. All describe iterative phases that are not necessarily ordered and can occur simultaneously. One of the latest views of the process proposed by Meinel and Leifer has five phases: (re)defining the problem, need-finding and benchmarking, ideating, building, testing. In the past decade, design thinking has gained attention as a meta-disciplinary methodology relevant in a wide range of contexts beyond the traditional preoccupation of designers [49]. It has been proposed as a "team-based learning process [which] offers teachers support towards practice-oriented and holistic modes of constructivist learning in projects" [50].

Design thinking is an iterative process that guides the teacher to "realize what is recommended theoretically in constructivist theory" [50]. In this sense, design thinking is similar to chaordic learning and can also be applied to SE courses to include creativity and innovation. In order to reconcile the relation between chaordic learning and design thinking, as applied in education, we consider chaordic learning as an overarching educational approach. Design thinking is a process that could be employed as one of the practices (described in Section III-A) when the chaordic learning environment is being established. In our courses, we use agile methods within the practices of the chaordic design process.

VI. Conclusion

Chaordic learning is an approach to balance education between order and chaos to stimulate analytical and creative thinking processes. Instructors provide structure and guidance, but also integrate freedom for self-organization and self-guided learning and embrace innovation and creativity, so that students learn important management and communication skills. The chaordic approach is similar to agile methods following empirical process control. It allows instructors to

react to specific situations, while providing an overall plan in their teaching approach. When integrating chaordic learning into their courses, instructors still control the learning outcome and avoid misconceptions.

Chaordic learning is particularly helpful in SE education, where practical application of knowledge, interaction and collaboration is required. Instructors view deviations as opportunities and failures as possibilities for students to learn and improve. We introduced chaordic learning into two courses, a games development course and a joint advanced student school. Students report an increased intrinsic motivation, a higher level of self-organization and more room for creativity which led to an improved learning experience and more fun.

REFERENCES

[1] M. Mahoney, "The roots of software engineering," *CWI Quarterly*, vol. 3, no. 4, pp. 325–334, 1990.

[2] B. A. Ogunnaike and W. H. Ray, *Process Dynamics, Modeling, and Control*. Oxford University Press, 1994, vol. 1.

[3] V. Basili, "The role of experimentation in software engineering: past, current, and future," in *Proceedings of the 18th international conference on Software engineering*. IEEE, 1996, pp. 442–449.

[4] T. Connolly, M. Stansfield, and T. Hainey, "An application of games-based learning within software engineering," *British Journal of Educational Technology*, vol. 38, no. 3, pp. 416–428, 2007.

[5] D. Shaffer, "Pedagogical praxis: The professions as models for postindustrial education," *Teachers College Record*, vol. 106, no. 7, pp. 1401–1421, 2004.

[6] J. Whitehead, "Collaboration in software engineering: A roadmap," *Future of Software Engineering*, vol. 7, pp. 214–225, 2007.

[7] I. Mulder, "A pedagogical framework and a transdisciplinary design approach to innovate hci education," *Interaction Design and Architecture(s) Journal*, no. 27, pp. 115–128, 2015.

[8] J. Tomayko, "Teaching a project-intensive introduction to software engineering," DTIC Document, Tech. Rep. CMU/SEI-87-TR-20, 1987.

[9] B. Bruegge, J. Cheng, and M. Shaw, "A software engineering project course with a real client," Carnegie Mellon University, Software Engineering Institute, Tech. Rep. CMU/SEI-91-EM-4, 1991.

[10] B. Bruegge, S. Krusche, and L. Alperowitz, "Software engineering project courses with industrial clients," *ACM Transactions on Computing Education*, vol. 15, no. 4, pp. 17:1–17:31, 2015.

[11] A. Dutson, R. Todd, S. Magleby, and C. Sorensen, "A review of literature on teaching engineering design through project-oriented capstone courses," *Journal of Engineering Education*, vol. 86, no. 1, pp. 17–28, 1997.

[12] P. Naur, B. Randell, and J. Buxton, *Software engineering: concepts and techniques: proceedings of the NATO conferences*. Petrocelli/Charter, 1976.

[13] M. Lehman and L. Belady, *Program evolution: processes of software change*. Academic Press, 1985.

[14] M. Kellner, "Software process modeling support for management planning and control," in *Proceedings of the 1st International Conference on the Software Process*. IEEE, 1991, pp. 8–28.

[15] D. Kolb, *Experiential learning: Experience as the source of learning and development*. Prentice Hall, 1984, vol. 1.

[16] D. Boud and G. Feletti, *The challenge of problem-based learning*. Psychology Press, 1998.

[17] J. Dunlap, "Problem-based learning and self-efficacy: How a capstone course prepares students for a profession," *Educational Technology Research and Development*, vol. 53, no. 1, pp. 65–83, 2005.

[18] R. Garrison and H. Kanuka, "Blended learning: Uncovering its transformative potential in higher education," *The internet and higher education*, 2004.

[19] D. Johnson *et al.*, *Cooperative Learning: Increasing College Faculty Instructional Productivity. Higher Education Report*. ERIC, 1991.

[20] D. Hock, "The chaordic organization: Out of control and into order," *World Business Academy Perspectives*, vol. 9, no. 1, pp. 5–18, 1995.

[21] D. Hock, "The art of chaordic leadership," *Leader to leader*, vol. 15, no. Winter, pp. 20–6, 2000.

[22] E. Leigh and L. Spindler, "Simulations and games as chaordic learning contexts," *Simulation & Gaming*, vol. 35, no. 1, pp. 53–69, 2004.

[23] F. W. Taylor, *The principles of scientific management*. Harper, 1914.

[24] D. Hock, *Birth of the chaordic age*. Berrett-Koehler Publishers, 1999.

[25] M. Huysman, "Balancing biases: A critical review of the literature on organizational learning," *SAGE Publications*, 1999.

[26] J. Pfeffer and J. Veiga, "Putting people first for organizational success," *The Academy of Management Executive*, vol. 13, no. 2, pp. 37–48, 1999.

[27] C. Kelliher and D. Anderson, "Doing more with less? flexible working practices and the intensification of work," *Human Relations*, vol. 63, no. 1, pp. 83–106, 2010.

[28] R. Stacey, "Learning as an activity of interdependent people," *The Learning Organization*, vol. 10, no. 6, pp. 325–331, 2003.

[29] F. van Eijnatten and G. Putnik, "Chaos, complexity, learning, and the learning organization: towards a chaordic enterprise," *The Learning Organization*, vol. 11, no. 6, pp. 418–429, 2004.

[30] K. Schwaber, "Scrum development process," in *Proceedings of the OOPSLA Workshop on Business Object Design and Information*, 1995.

[31] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries *et al.*, "Manifesto for agile software development," *The Agile Alliance*, 2001.

[32] VersionOne, "9th annual state of agile development survey," 2015, retrieved January 08, 2016 from https://www.versionone.com/pdf/state-of-agile-development-survey-ninth.pdf.

[33] D. Rico and H. Sayani, "Use of agile methods in software engineering education," in *Agile Conference*, 2009, pp. 174–179.

[34] P. Kirschner, J. Sweller, and R. Clark, "Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching," *Educational Psychologist*, vol. 41, no. 2, pp. 75–86, 2006.

[35] A. Collins, J. S. Brown, and A. Holum, "Cognitive apprenticeship: Making thinking visible," *American educator*, 1991.

[36] S. Krusche, M. Berisha, and B. Bruegge, "Teaching Code Review Management using Branch Based Workflows," in *Proceedings of the 38th International Conference on Software Engineering*. IEEE, 2016.

[37] L. Fitzgerald, "Chaos: The lens that transcends," *Journal of Organizational Change Management*, vol. 15, no. 4, pp. 339–358, 08 2002.

[38] S. Krusche, B. Reichart, P. Tolstoi, and B. Bruegge, "Experiences from an experiential learning course on games development," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE)*. ACM, 2016, pp. 582–587.

[39] S. Krusche, A. Seitz, J. Börstler, and B. Bruegge, "Interactive learning: Increasing student participation through shorter exercise cycles," in *Proceedings of the 19th Australasian Computing Education Conference*. ACM, 2017, pp. 17–26.

[40] A. Beyer, "Improving student presentations pecha kucha and just plain powerpoint," *Teaching of Psychology*, 2011.

[41] J. Haladjian, Z. Hodaie, H. Xu, M. Yigin, B. Bruegge, M. Fink, and J. Hoeher, "Kneehapp: A bandage for rehabilitation of knee injuries," in *Proceedings of the International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2015, pp. 181–184.

[42] K. Desouza and Y. Awazu, "Managing radical software engineers: Between order and chaos," in *Human and Social Factors of Software Engineering*. ACM, 2005, pp. 1 – 5.

[43] L. S. Vygotsky and R. W. Rieber, *The collected works of LS Vygotsky: Volume 1: Problems of general psychology, including the volume Thinking and Speech*. Springer Science & Business Media, 1988, vol. 1.

[44] V.-P. Eloranta, "Patterns for controlling chaos in a startup," in *Proceedings of the 8th Nordic Conference on Pattern Languages of Programs*. ACM, 2014, pp. 1:1–1:8.

[45] S. H. Thomke, *Experimentation Matters: Unlocking the Potential of New Technologies for Innovation*. Boston, MA, USA: Harvard Business School Press, 2003.

[46] D. Jonassen, K. Peck, and B. Wilson, *Learning with technology: A constructivist perspective*. Prentice Hall, 1999.

[47] T. Duffy and D. Jonassen, *Constructivism and the Technology of Instruction: A Conversation*. Psychology Press, 1992.

[48] P. Rowe, "Design thinking," 1987.

[49] L. Kimbell, "Rethinking design thinking: Part I," *Design and Culture*, vol. 3, no. 3, pp. 285–306, 2011.

[50] A. Scheer, C. Noweski, and C. Meinel, "Transforming constructivist learning into action: Design thinking in education," *Design and Technology Education: An International Journal*, vol. 17, no. 3, 2012.

8.4 ArTEMiS - An Automatic Assessment Management System for Interactive Learning

This conference paper describes Artemis, an automated assessment management system for interactive learning. It automatically assesses solutions to programming exercises and provides instant feedback so that students can iteratively solve the exercise. It is open-source and highly scalable based on version control, regression testing, and continuous integration. The paper describes the early use of Artemis in three university courses and one online course and reports about the experiences. Artemis is suitable for beginners. It helps students to realize their progress and to improve their solutions gradually. It reduces the effort of instructors and enhances the learning experience of students. Artemis is one integral part of the research in this habilitation because the learning platform enables further research in team-based learning, gamification, learning analytics, and semi-automated assessment using machine learning.

Authors	**S. Krusche** and A. Seitz
Conference	49th Technical Symposium on Computer Science Education
Publisher	ACM
Pages	6
Type	Conference: Full Research Paper
Review	Peer Reviewed (6 Reviewers)
Year	2018
Citation	[KS18]
DOI	https://doi.org/10.1145/3159450.3159602

ArTEMiS - An Automatic Assessment Management System for Interactive Learning

Stephan Krusche
Technische Universität München
Munich, Germany
krusche@in.tum.de

Andreas Seitz
Technische Universität München
Munich, Germany
seitz@in.tum.de

ABSTRACT

The increasing number of students in computer science courses leads to high efforts in manual assessment of exercises. Existing assessment systems are not designed for exercises with immediate feedback in large classes. In this paper, we present an **A**u**T**omated ass**E**ssment **M**anagement **S**ystem for interactive learning.

ArTEMiS assesses solutions to programming exercises automatically and provides instant feedback so that students can iteratively solve the exercise. It is open source and highly scalable based on version control, regression testing and continuous integration. ArTEMiS offers an online code editor with interactive exercise instructions, is programming language independent and applicable to a variety of computer science courses. By using it, students gain experiences in version control, dependency management and continuous integration.

We used ArTEMiS in 3 university and 1 online courses and report about our experiences. We figured out that ArTEMiS is suitable for beginners, helps students to realize their progress and to gradually improve their solutions. It reduces the effort of instructors and enhances the learning experience of students.

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**; *Computer science education*; • **Applied computing** → **Interactive learning environments**; **Learning management systems**;

KEYWORDS

Automated Assessment, Programming Exercises, Continuous Integration, Version Control, Instant Feedback, Online Editor, Interactive Exercise Instructions, Online Courses, In-class Exercises.

ACM Reference Format:
Stephan Krusche and Andreas Seitz. 2018. ArTEMiS - An Automatic Assessment Management System for Interactive Learning. In *SIGCSE '18: SIGCSE '18: The 49th ACM Technical Symposium on Computing Science Education, February 21–24, 2018, Baltimore, MD, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3159450.3159602

SIGCSE '18, February 21–24, 2018, Baltimore, MD, USA

ACM ISBN 978-1-4503-5103-4/18/02…$15.00
https://doi.org/10.1145/3159450.3159602

1 INTRODUCTION

The amount of students in university classes and online courses is increasing. The number of freshmen at our computer science department increased by 67 % between 2013 (1110) and 2016 (1840). In addition, the number of enrollments in online courses such as MOOCs is increasing as well. Class Central recently reported 35 million students which have signed up for at least one MOOC. With such large numbers of students, manual assessment of programming exercises in computer science and software engineering courses is no longer feasible. Yet, programming exercises are essential in computer science education [17].

While automatic assessment is an established concept, large courses raise new challenges for instructors and tools. Automated assessment must be scalable to handle a large number of students and provide immediate feedback. The effort for an instructor conducting an programming exercise has to be independent of the number of participating students. Instant feedback to students is required to allow resubmissions and learning from failures. This is particularly important in courses based on active learning [3], where the programming exercises happen in-class in specific time frames. During the exercise, instructors need to be able to get an overview of submitted solutions and typical problems to react and guide the students.

There are many existing tools for automatic assessment and grading but most of them are custom tailored solutions for specific programming languages and requirements (compare Section 5). This makes it particularly hard to integrate them into existing infrastructure and use them in large scale courses.

Continuous integration (CI) is an approach allowing to detect defects and failures in programs [7]. This idea can be used for programming exercises to validate the correctness and completeness of source code with test cases. Students upload their solution to a version control system (VCS) and receive instant feedback about the result of their submissions from the CI server. While doing the exercise, they get used to VC and CI, which are important skills in software development.

In this paper, we describe our experiences with ArTEMiS, an open source **A**u**T**omated ass**E**ssment **M**anagement **S**ystem for interactive learning. In Section 2, we cover the basic background behind the methodology and tools for our approach. Section 3 presents the approach behind and the use of ArTEMiS in more detail. We describe its scalability and show the applicability of the system within large university courses and an online course in a multi case study in Section 4. We analyze the participation of students in the case study and provide insights. Section 5 relates and differentiates ArTEMiS to other automated assessment tools. In Section 6, we conclude the paper and provide directions for future work.

2 FOUNDATIONS

ArTEMiS uses version control (VC) and continuous integration (CI) to automatically assess programming exercises in interactive learning environments. This sections describes its foundations.

2.1 Interactive Learning

Interactive learning combines lectures and exercises into interactive classes with multiple iterations of theory, example, exercise, solution and reflection [12]. Educators teach and exercise small chunks of knowledge in short cycles. They focus on immediate feedback to exercises to improve the learning experience in large classes so that students reflect and increase their knowledge incrementally.

Hands-on activities in class increase students' motivation and engagement and allow continuous assessment over the course [13]. This approach expects active participation of learners and the use of computers (laptops, tablets, smartphones) in the classroom. Instructors provide guidance during the learning process to prevent misconceptions and to facilitate the learning process.

2.2 Continuous Integration

CI was first described by Grady Booch as concept to avoid risky late integrations [4]. Martin Fowler defines CI as follows: "Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily – leading to multiple integrations per day" [7].

Each developer works on a local copy of a shared code base. After implementing source code, the developer integrates the changes into the shared code base. An automated build (including test cases) verifies each integration to detect compile, test, and integration errors as soon as possible [7]. The build is triggered on the developer's machine or on a central CI server. While CI does not require special tools, developers use dedicated servers to perform it.

Figure 1 illustrates a common CI workflow. (1) The process starts with a developer committing code changes to a VCS. (2) The CI server regularly checks the VCS for code changes and (3) automatically triggers the build and test process on every commit. Tests can range from small unit tests over larger integration tests to complete system tests. (4) After the build has either succeeded or failed, the developer is notified about the build result. Every commit triggers a build, so the developer responsible for the failure is notified immediately to fix the problem.

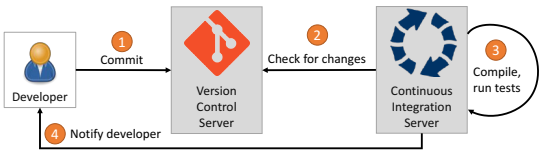


Figure 1: Typical continuous integration workflow

2.3 Automatic Assessment

"The [manual] assessment of [programming] assignments places significant demands on the instructor's time and other resources" [5]. To solve this problem and due to the logical character of programming, the task of assessing programming exercises can be automated. A first example was a grading program for punch card programs by Hollingsworth [9]. Since then, many automatic assessment tools (e.g. [6], [11], [15]) have been developed, together with guidelines on how programming exercises should be assessed.

Automatic assessment systems provide feedback on students' solutions for programming exercises [19]. Compared to manual assessment, they are able to provide consistent feedback for every student without bias while significantly reducing the effort for instructors and TAs. Ala-Mutka describes different types of assessments [1]. The historically most common type is dynamic assessment. Its main aspects are to assess functionality, efficiency, and testing skills by executing a program with test input data and checking the output for correctness. Another type is static analysis, which provides feedback on style, programming errors or software metrics by analyzing the code without executing it.

Students using automatic assessment tools achieve important learning goals: they develop a clean and reusable code style, reflect critically on errors and establish a testing culture. These goals are achieved by multiple methods, e.g. varying the amount of feedback provided or allowing to work incrementally on a solution. They can be effectively implemented using different assessment techniques such as black-box testing, white-box testing or peer reviews.

3 APPROACH

We identified the following goals for ArTEMiS by analyzing the applicability of existing tools for large interactive courses:

Independence of programming language: different programming languages are taught in university courses [17]. The system should work independently of a specific programming language.

Scalability: the system should be scalable usable in university classes and online courses. It should work with hundreds of participating students at the same time, e.g. during an interactive class. The workload for preparing and grading exercises must be independent of the number of participating students.

Instant feedback: the system should provide instant feedback for submitted solutions and describe why the particular solution is correct or wrong so that students can improve it.

Learning from failures: students can learn, reflect and iteratively submit new solutions even if they fail initially. Students should have the chance to resubmit their solutions as often as they want (potentially limited to a specific time frame).

Different exercise types: the system allows programming exercises in different areas of the software engineering process including programming basics, system design, object design, testing and build and release management.

Different assessment ways: the system should allow different ways to assess submissions. For programming exercises the system has to support different test types (e.g. structural, behavioral, runtime, performance or functional tests).

Traceability: the system must ensure traceability for instructors and for students. With regards to team exercises, the contribution of each team member should be accountable. Traceability enables early detection of difficulties for the students in the assignment and lets the instructor react accordingly.

Immediate evaluation: solutions and results are easily accessible and evaluable for instructors enabling them to remove ambiguity, answer open questions, or extend the given working time.

Interactive exercise instructions: dynamic tasks and UML diagrams visualize the current progress of students. They update their color from red (incomplete) to green (complete) when students submit their solution and when associated test cases pass.

Easy to use online editor: to simplify the participation and improve the learnability, programming beginners can work on programming exercises in an interactive and lightweight online editor. They can submit their solutions with just one button click.

ArTEMiS fulfills these goals by using the concepts of VC and CI. Each student works with a given template code in his own repository and has a build plan which executes test cases after each commit. Students can solve the programming exercise in an online editor, on the local computer using an IDE or with a mix of both. When using the online editor, they don't need to setup a VC client and an IDE. If they work on their own computer, they need to apply version control and install an IDE (e.g. Eclipse) and have more functionalities in the code editor (e.g. auto completion, error highlighting, etc.). Conducting a programming exercise consists of 7 steps distributed among instructor, ArTEMiS and students:

1. **Instructor prepares exercise:** set up a base repository containing the exercise code and test cases. Set up a base build plan on the CI server, and create the exercise on ArTEMiS.
2. **Student starts exercise:** click on *Start Exercise* on ArTEMiS. This automatically generates a copy (fork) of the base repository with the exercise code and a copy of the base build plan. ArTEMiS sets the permissions so that students can only see their personal repository.
3. **Student clones repository:** optionally clone the personalized repository from the remote VCS to the local machine.
4. **Student solves exercise:** Solve the exercise with an IDE of choice on the local computer **or** in the online editor.
5. **Student submits solution:** upload source code changes to the VCS by committing and pushing them to the remote server **or** by clicking *Commit & Run Tests* in the online editor.
6. **CI server verifies solution:** verify the student's submission by executing the test cases (see step 1) and provide feedback which parts are correct or wrong.

7a. **Instructor reviews course results:** review overall results of all students, and react to common errors and problems.

7b. **Student reviews personal result:** review build result and feedback using ArTEMiS. In case of a failed build, reattempt to solve the exercise (step 4.).

Figure 2 shows this approach as UML activity diagram. CI server, VCS, and ArTEMiS are combined in the *System* actor. The approach consists of 2 phases, exercise preparation and exercise execution.

Exercise preparation: an instructor sets up a VC repository containing the exercise code (template) handed out to students and test cases to verify students' submissions (*base repository*). This repository typically includes a small sample project including some predefined classes, dependencies to external libraries, e.g. a testing framework, and test cases. A combination of behavioral (black-box) and structural (white-box) tests allows to check for both functionality and implementation details of the submitted code. In addition, the instructor stores the tests separately in a test repository, which is not accessible to students, to prevent that they adapt the test cases. It can make sense to completely hide the tests from the students to prevent reverse engineering the solution.

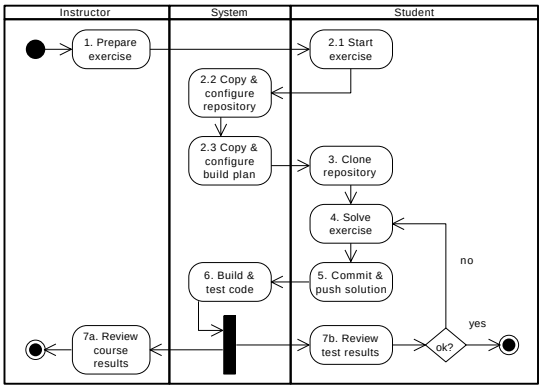


Figure 2: Process for automated assessment in ArTEMiS

After setting up the base and test repositories, the instructor configures the build plan on the CI server which compiles and tests the exercise code using the previously defined test cases (*base build plan*). This build plan includes a task to pull the source code from the base repository and the test repository whenever changes occur, and to combine them so that the tests can be executed in the second step. A final task, which is also executed when compilation or test execution fails, notifies ArTEMiS about the new result. The instructor finally creates an exercise on ArTEMiS by selecting the preconfigured base repository of the VCS and the preconfigured base build plan of the CI server.

Exercise execution: a student starts an exercise with a single click, triggering the setup process: ArTEMiS creates a personal copy of the base repository, the *student repository*, and provides access only to this student. It creates a personal copy of the base build plan, the *student build plan*, and configures it to be triggered when the particular student uploads changes to this personal student repository. The student can usually not access the build plan to hide its complexity. Personalized means that each student gets one repository and one build plan. When 200 students participate in an exercise, ArTEMiS creates 200 student repositories and 200 student build plans. Students only have access to their personal repository, they cannot access other student repositories. This prevents cheating, because students cannot access code of each other.

After the setup is complete, ArTEMiS displays the clone URL and/or allows the student to open the exercise in the online editor. The student clones the repository to the local computer and starts working on the exercise. When the students uploads a new solution to the personalized repository, the personalized build plan of the student assesses the solution. Students upload solutions by committing and pushing changes in their source code to their personal repository or by submitting their changes in the online editor (which triggers a commit and a push operation in the background). The new commit on the personal repository triggers the personal build plan to assess the solution on a build agent. The build agent pulls the submitted code from the personal repository and the tests from the test repository, and combines them in a working directory. It compiles the code, executes the tests and uploads the results to ArTEMiS in a few seconds, so that the student can immediately review the feedback and iteratively improve their solution.

In case of an incorrect solution, the feedback includes how many tests failed and the corresponding failure message for each failed

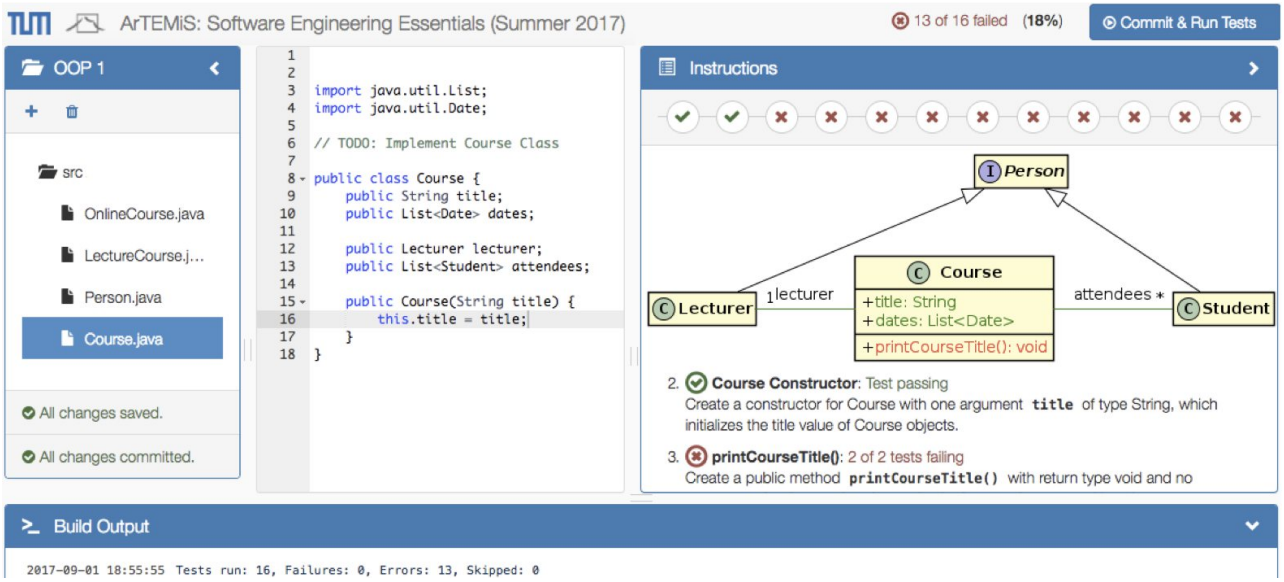


Figure 3: Screenshot of the ArTEMiS online editor with interactive exercise instructions on the right

test. The student can now reattempt to solve the exercise and submit a new solution. The instructor can review the results, gain insights on the exercise progress and react immediately to errors and problems during the exercise. ArTEMiS acts as facade to the CI process and hides complex details, which enables less experienced students to participate in the exercise. Every student has a personalized build plan, so the approach can be used to teach the concepts of CI. Then, students get access to their build plan and have to configure it on their own. Students can only see, edit and adjust their personalized build plan. They cannot inspect build plans of their fellow students.

ArTEMiS includes an online editor that allows unexperienced students to participate in exercises without dealing with the complex setup of VC and IDEs. Figure 3 shows the online editor with interactive and dynamic exercise instructions on the right side. Interactive instructions change their color depending on the progress of students. Already completed tasks are marked with a green tick, incomplete tasks are marked with a red cross. This helps students to identify which parts of the exercise they have already solved correctly. When they submit their current solution with the *Commit & Run Tests* button in the upper right corner, the interactive instructions dynamically update. The exercise tasks and the UML diagram elements are referenced by the predefined test cases. They change their color from red to green when all test cases associated with the task or diagram element pass. This allows the students to immediately recognize which tasks are already fulfilled and is particularly helpful for beginners.

Figure 4 shows the system architecture of ArTEMiS. A student uses the *ArTEMiS Application Client* (a browser) and a *VC Client* of his choice to obtain the exercise code and to submit solutions. *VC Server*, *CI Server*, *Local Build Agents*, and the *ArTEMiS Application Server* run on the university's infrastructure. The *CI Server* delegates the builds to local and/or remote agents, e.g. on Amazon Web Services, depending on how much capacity is required for the number of participating students. This makes it easy to scale the approach by adding additional build agents. ArTEMiS uses the university's *User Management System*. The components *VC Server* and *CI Server* are exchangeable, resulting in a flexible system which can be adapted to the specific requirements of instructors.

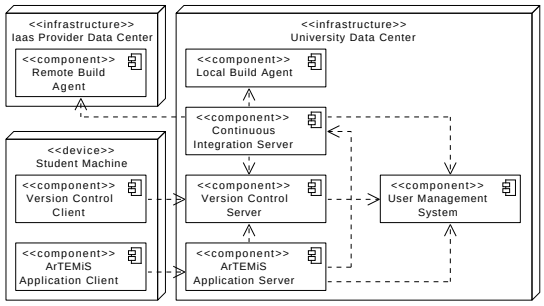


Figure 4: System Architecture of ArTEMiS

The use of CI leads to several benefits: the programming language for programming exercises is freely configurable. Compilation and testing of student solutions is not a matter of ArTEMiS, it only depends on the configuration of the build system. Students learn the concepts and workflows of VC, CI and testing, all important skills in software development. The system is scalable and can react to the number of assessments to be handled with a corresponding number of build agents. Adding more build agents allows more students to submit their solution at the same time and still receiving instant feedback. This is crucial for interactive learning based exercises that are integrated into a class.

4 CASE STUDY

We use ArTEMiS in 3 large university course (UC) and in a MOOC:

(1) **UC - Introduction to Software Engineering**: mandatory subject, 1400 bachelor students (2nd semester), 6 exercises, 814 students participated

(2) **UC - Patterns in Software Engineering**: elective subject, 400 master students, 34 exercises, 334 students participated

(3) **UC - Project Organization and Management**: mandatory subject for business informatics, elective subject for computer science, about 300 bachelor and master students, 1 exercise, 224 students participated
(4) **MOOC - Software Engineering Essentials**: 300 active students, 10 exercises, 257 students participated

In the beginning of these courses, we go through a short interactive tutorial together with the students to show them how to use ArTEMiS. In all these courses, we recommend to use Eclipse as IDE and SourceTree as VC client (git). Students could individually decide which VC client to use to retrieve the assignment, and hand in their solution. We describe 4 typical exercises from our courses:

(1) **Programming:** write source code for a problem statement or UML model. Typical examples are the implementation of the quick sort algorithm or the strategy pattern. Test cases assess the correctness of the solution.
(2) **Testing:** write test cases for the given program code and mock parts of the program. The instructor creates test cases that assess the test cases of the students with the given correct program code and by injecting wrong program code to review if the tests of the students are correct.
(3) **Merge conflict:** experience a merge conflict in git and have to resolve it.
(4) **Release management:** learn how commits trigger the CI server, which produces new build artifacts. Students get direct access to their personalized build plan to be able to view its configuration in detail.

By automizing all setup steps, we decreased the effort for students and instructors. The instructor can dynamically adapt the working time for different assignments based on the students' progress in class. This allowed students to actively think about the exercises instead of just following along with the solution.

4.1 Results

In simulations with 800 students and 2000 submissions over a period of 10 minutes, ArTEMiS assessed solutions in an average of 10 seconds. In our courses, we measure the average assessment time. This allows us to constantly evaluate whether ArTEMiS fulfills its scalability goals under real classroom conditions. In a quantitative analysis, we found the following results:

(1) Scalability: ArTEMiS can handle 200 submissions per minute.
(2) Feedback: ArTEMiS provides feedback within 10 seconds.
(3) Usability: Programming beginners are able to use ArTEMiS.

Table 1 displays an overview of the number of students and submissions for each exercise together with the average assessment time. The number of submissions per student varies from 1.6 to 4.3 on average. In one exercise, where students could only use the online editor, this number was higher: 7.2. If students participate in the exercise and submit solutions, most of them also successfully solve the exercise. The assessment time, i.e. the time for students to wait from the moment they submit their solution until they see the test results and the feedback, varies from 5.1 to 10.3 seconds on average depending on the complexity of the exercise, the number of tests and the number of external dependencies. In exercises with more complex tests, e.g. asynchronous client-server tests with timeouts, this number can increase. In such cases, it makes sense to distribute tests to students so that they can also execute them on their computer. ArTEMiS uses a separate test repository so that students who try to cheat and change tests locally, e.g. to let all tests pass immediately without solving the exercise, can be detected.

We asked the students for feedback regarding the use of ArTEMiS. Most students had less experience in the areas of distributed VC and CI, nonetheless they had no issues working with ArTEMiS. They stated, that the test results and the feedback was helpful to solve the exercises, they enjoyed working with it and preferred the usage of ArTEMiS over the previous process where there was no automatic and instant feedback. In an online questionnaire, we found that more than 90 % of the students consider the interactive exercise instructions helpful in solving the exercise. They are particularly valuable in online courses, where students are distributed and instructors cannot guide them directly in case of problems.

	(1) Programming	(2) Testing	(3) Merge conflict	(4) Release management
Participating students	317	167	224	248
Submitting students	209 (66 %)	109 (65 %)	211 (94 %)	149 (60 %)
Successful students	200 (96 %)	108 (99 %)	183 (87 %)	135 (91 %)
Overall submissions	340	340	904	285
Correct submissions	236	236	291	198
Test cases	12	12	2	0
Assessment time*	10.3 s	8.3 s	5.1 s	9.6 s
Submissions per student*	1.6	3.1	4.3	1.9

Table 1: Numbers in typical exercises (* on average)

4.2 Discussion

ArTEMiS provides flexibility in how instructors conduct exercises. It allows them to distribute assessment tests to students so that they can find their own errors during debugging easier. However, this might facilitate that students only work on getting the tests passed and do not take the time to understand the actual problem and solve it on their own. In such cases, instructors can hide the tests and only show the test results on ArTEMiS.

Another choice is the use of the online editor vs. the use of an IDE on the local computer. While the online editor lowers the entrance barrier, it has limited features. In a comparison, we found that students prefer the local IDE if they are already familiar with it, as it offers more features such as syntax and error highlighting, auto completion and debugging. In an experiment, we forced them to use the online editor, however some students copied the code file by file into their local IDE, solved the exercise there, and copied the code back. An open question at this point is whether providing features such as auto completion is beneficial to the learning experience. Novice programmers often heavily depend on such features [18]. This may make students reliant on them and may prevent learning the correct syntax of the programming language. It could be a viable strategy to provide only minimal features in a code editor.

ArTEMiS supports in-class exercises with hundreds of students and can additionally be used for homework exercises. We use it in online courses where students are distributed and rely on the exercise instructions and in university classes where instructors can guide the students in addition. The costs of providing the system are not negligible. We could use already existing, self-hosted CI and VC systems. These are hosted at our institution, but we have to take care of the maintenance. Alternatively, cloud-based solutions such as GitHub Education, GitLab or Bitbucket Cloud can be used. These usually offer attractive opportunities for educational institutions.

5 RELATED WORK

A variety of systems for automatic assessment exists. Multiple surveys have been published, summarizing and categorizing these systems. The first extensive surveys on this topic were done by Ala-Mutka [1] and Douce et al. [5]. They describe multiple auto assessment tools and categorize them into dynamic and static assessment and differentiate between local and web based systems.

The survey by Ihantola et al. focuses on identifying key features of automatic assessment tools, such as supporting different programming languages, allowing resubmissions, or providing a sandbox environment to handle malicious submissions [10]. The authors state most systems are not open source or available otherwise, even if a publication describes the development of a prototype. A survey by Queiros states interoperability and compatibility to other services is a key factor for automatic assessment systems [14]. He concludes that this factor is not considered for many existing assessment tools and that future solutions have to improve this. Our approach fulfills many of the stated features by connecting to university user management or providing interfaces to VCS.

From those surveys, we identified multiple publications related to our system. WebCAT was first created in 2003 and is arguably one of the most complete automatic assessment tools [6]. It has been developed as open-source software and allows extensibility by plugins. In terms of assessment, it supports student written tests, test coverage, static code analysis and a combination of both automatic and manual grading. Our approach covers most of the features of WebCAT, while removing the dependence on a single software product. Instead, our approach consists of multiple independent software systems that are connected using common interfaces. This leads to a higher flexibility as individual parts of the architecture can be replaced, for example in favor of lower costs, superior support, larger communities or general management decisions.

Marmoset focuses on information collection during the development process of students [16]. The system takes regular snapshots of the students' progress. It allows the instructor to study the development process of the students and to identify common bug patterns. By using VCS and teaching its application, we achieve the same outcome. Students commit multiple iterations of their solution, resulting in a commit history that can be evaluated. This allows to identify common mistakes and study problem solving behavior. Amelung et al. propose a system thats splits e-learning and e-assessment platforms into separate systems, allowing independent deployment and easier adoption [2]. Our approach targets the same idea, but our goal is not to implement another closed source assessment system. Instead, we reuse workflows provided by existing CI tools to achieve similar results.

Gruenewald et al. focus on the challenge of conducting programming lectures as MOOCs [8] integrating active experimentation and relating to concrete experience. Those aspects are considered in ArTEMiS as well. In recent years, commercial products have become available for automatic grading, including Vocareum, Turing Craft, etc. An open source and free alternative is Codeboard, developed by ETH Zurich. These online tools are cloud based and use regression testing, a technique used for quality control in software development. They allow students to edit and submit source code in the browser to simplify the participation.

6 CONCLUSION

ArTEMiS combines VC and CI with automated assessment of programming exercises and immediate feedback. This enables high flexibility and scalability in large classes. Our experiences in 3 university courses and 1 online course show that programming beginners are able to use the system, improve their solutions iteratively with immediate feedback and increase their learning experience.

Dynamic and interactive exercise instructions are particularly helpful for beginners to immediately recognize which tasks are resolved. The effort for instructors and TAs is reduced. They can evaluate student results immediately during the exercise to help students when problems occur. ArTEMiS is free and open source on https://github.com/ls1intum/ArTEMiS, so that other instructors can use it in their courses. We will support additional interactive exercises in the future, in particular quizzes and modeling.

ACKNOWLEDGMENTS

We want to thank Dominik Münch, Andreas Greimel and Josias Montag who participated in the development of ArTEMiS.

REFERENCES

[1] K. Ala-Mutka. 2005. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education* (2005), 83–102.
[2] M Amelung, P. Forbrig, and D. Rösner. 2008. Towards Generic and Flexible Web Services for E-Assessment. *SIGCSE Bulletin* (June 2008), 219–224.
[3] C. Bonwell and J. Eison. 1991. *Active Learning: Creating Excitement in the Classroom.* ASHE-ERIC Higher Education Reports.
[4] G. Booch. 1991. *Object Oriented Design with Applications.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.
[5] C. Douce, D. Livingstone, and J. Orwell. 2005. Automatic Test-Based Assessment of Programming: A Review. *Journal on Educ. Resources in Computing* (2005).
[6] S. Edwards. 2003. Improving student performance by evaluating how well Students test their own programs. *Journal on Educ. Resources in Computing* (2003).
[7] M. Fowler. 2006. Continuous Integration. http://www.martinfowler.com/articles/continuousIntegration.html. (2006).
[8] F. Grünewald, C. Meinel, M. Totschnig, and C. Willems. 2013. Designing MOOCs for the Support of Multiple Learning Styles. In *European Conference on Technology Enhanced Learning*. Springer, 371–382.
[9] J. Hollingsworth. 1960. Automatic Graders for Programming Classes. *Commun. ACM* (1960), 528–529.
[10] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. 2010. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Koli Calling Conference on Computing Education Research.* ACM, 86–93.
[11] M. Joy, N. Griffiths, and R. Boyatt. 2005. The BOSS Online Submission and Assessment System. *Journal on Educational Resources in Computing* (2005).
[12] S. Krusche, A. Seitz, J. Börstler, and B. Bruegge. 2017. Interactive Learning: Increasing Student Participation through Shorter Exercise Cycles. In *Proceedings of the 19th Australasian Computing Education Conference.* ACM, 17–26.
[13] S. Krusche, N. von Frankenberg, and S. Afifi. 2017. Experiences of a Software Engineering Course based on Interactive Learning. In *Proceedings of the 19th Workshop on Software Engineering Education in Universities.* 32–40.
[14] R. Queirós and J. Leal. 2012. Programming Exercises Evaluation Systems – An Interoperability Survey. In *Conference on Computer Supported Education.* 83–90.
[15] R. Singh, S. Gulwani, and A. Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. *SIGPLAN Notices* (2013), 15–26.
[16] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, . Hollingsworth, and N. Padua-Perez. 2006. Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. *SIGCSE Bulletin* (2006), 13–17.
[17] T. Staubitz et al. 2015. Towards Practical Programming Exercises and Automated Assessment in Massive Open Online Courses. In *International Conference on Teaching, Assessment, and Learning for Engineering.* 23–30.
[18] A. Vihavainen, J. Helminen, and P. Ihantola. 2014. How novices tackle their first lines of code in an IDE: Analysis of programming session traces. In *Koli Calling Conference on Computing Education Research.* ACM, 109–116.
[19] M. Vujošević-Janičić, M. Nikolić, D. Tošić, and V. Kuncak. 2013. Software Verification and Graph Similarity for Automated Evaluation of Students' Assignments. *Inf. Softw. Technol.* 55, 6 (2013), 1004–1016.

8.5 Software Theater—Teaching Demo-Oriented Prototyping

This journal paper describes one example of a more extensive team exercise called software theater based on interactive learning. The paper shows the usage in two courses: the capstone course iPraktikum with 100 students and the lecture-based course Project Organization and Management (POM) with 400 students. The creation and conduction of software theater focus on students' creativity. The workflow consists of multiple steps that combine teaching the concepts and immediately practicing them in a larger project-based context with realistic problem statements. An evaluation shows that software theater is more creative, memorable, dynamic, and engaging. Moreover, it brings fun into education.

Authors	**S. Krusche**, D. Dzvonyar, H. Xu and B. Bruegge
Conference	Transactions on Computing Education
Publisher	ACM
Pages	30
Type	Journal Article
Review	Peer Reviewed (4 Reviewers)
Year	2018
Citation	[KDXB18]
DOI	https://doi.org/10.1145/3145454

Software Theater—Teaching Demo-Oriented Prototyping

STEPHAN KRUSCHE, DORA DZVONYAR, HAN XU, and BERND BRUEGGE,
Technische Universität München

Modern capstone courses use agile methods to deliver and demonstrate software early in the project. However, a simple demonstration of functional and static aspects does not provide real-world software usage context, although this is integral to understand software requirements. Software engineering involves capabilities such as creativity, imagination, and interaction, which are typically not emphasized in software engineering courses. A more engaging, dynamic way of presenting software prototypes is needed to demonstrate the context in which the software is used. We combine agile methods, scenario-based design, and theatrical aspects into software theater, an approach to present visionary scenarios using techniques borrowed from theater and film, including props and humor.

We describe the software theater workflow, provide examples, and explain patterns to demonstrate its potential. We illustrate two large case studies in which we teach students with varying levels of experience to apply software theater: a capstone course involving industrial customers with 100 students and an interactive lecture-based course with 400 students. We empirically evaluated the use of software theater in both courses. Our evaluations show that students can understand and apply software theater within one semester and that this technique increases their motivation to prepare demonstrations even early in the project. Software theater is more creative, memorable, dynamic, and engaging than normal demonstration techniques and brings fun into education.

CCS Concepts: • **Social and professional topics** → **Software engineering education**; *Information technology education*; • **Software and its engineering** → **Agile software development**; **Software prototyping**; *Rapid application development*; *Object oriented development*;

Additional Key Words and Phrases: Agile methods, visionary scenarios, scenario-based design, collaborative learning

ACM Reference format:
Stephan Krusche, Dora Dzvonyar, Han Xu, and Bernd Bruegge. 2018. Software Theater—Teaching Demo-Oriented Prototyping. *ACM Trans. Comput. Educ.* 18, 2, Article 10 (July 2018), 30 pages.
https://doi.org/10.1145/3145454

1 INTRODUCTION

Software engineers have to cope with uncertainties and constantly changing requirements (Lehman and Belady 1985). Many educators teach software engineering in a setting close to the real world with industrial customers, which enables students to experience such challenges and prepares them for their later career in industry (Tomayko 1987; Shaw et al. 1991; Bruegge et al.

Authors' addresses: S. Krusche (corresponding author), D. Dzvonyar, H. Xu, and B. Bruegge, Chair of Applied Software Engineering, Department of Computer Science, Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany; emails: {krusche, dzvonyar, xuh, bruegge}@in.tum.de.

ACM 1946-6226/2018/07-ART10 $15.00
https://doi.org/10.1145/3145454

2015). Modern capstone courses use agile methods such as Scrum (Schwaber and Beedle 2002) to address uncertainty through an incremental process: software is delivered and demonstrated early in the project in the form of potential product increments or prototypes.

However, a simple demonstration of the functional and static aspects of a prototype fails to provide enough context of real-world software usage, although this is integral to fully understand the software requirements (Shaw et al. 2006). Clarifying requirements is particularly important in exploratory projects with emerging technologies where the real requirements are often unclear until they are met (Carroll 2000). Software engineering involves experimental knowledge work (Basili 1996) such as creativity, imagination, and interaction, but these are typically not emphasized in education. A more engaging, dynamic way of presenting software prototypes is needed to demonstrate the context in which the software is used.

To address this problem, we combine agile methods, scenario-based design (Carroll 1995), and theatrical concepts into a technique called **software theater** (Xu et al. 2015). Software theater is a way to present and evaluate scenarios with a demo of the software under development using techniques borrowed from theater and film, including props[1] and humor. The creation of the demo is based on the Tornado model (Bruegge et al. 2012): developers demonstrate and evaluate requirements, user experience, and the design of the system even if the software is not yet fully realized. Model-based demonstration and the mock object pattern (Mackinnon et al. 2000) allow the simulation of subsystems and objects that collaborate in the demo, but are not yet implemented.

The use of scenarios enables developers to reflect and reason about the requirements before they are realized, by focusing on concrete usage situations in the problem domain and by inspecting them from different perspectives (Carroll 2000). With regular demonstrations, developers can iteratively validate their development progress and technical decisions with the customer and avoid having to change their software at a later stage, which would require higher effort (Hazzan 2002; Shaw et al. 2006).

To examine the benefits of software theater, we investigated the following two hypotheses in two university courses, a capstone course and a lecture-based course:

- **H1 Increased motivation:** software theater increases the motivation of students to prepare a demonstration early in the project.
- **H2 Higher creativity:** demonstrations using software theater are more creative than normal[2] demonstrations (without software theater).

The remainder of this article is structured as follows: Section 2 describes innovation management, interaction design, informal modeling, prototyping, scenario-based design, and the Tornado model as the foundations of software theater. Section 3 illustrates the software theater workflow including all steps from the customer's visionary scenario to the developers' actual demonstration. It introduces the idea of model-based demonstration using the mock object pattern. Section 4 shows an example including all the artifacts created during the workflow. Section 5 presents three recurring patterns: students use a narrator to tell the story behind the system, they use software theater to explain complex technical details of a system, and they make an abstract concept more relatable using a metaphor from a different domain.

[1]The term "prop" is taken from the theater world and describes a theatrical property, an object used on stage or on screen by actors during a performance. It is anything movable or portable on a stage, distinct from the actors, scenery, costumes, and electrical equipment and can be seen as proxy for non-digital objects.

[2]In a normal or traditional demonstration (without software theater), a developer goes through the user interface of an application explaining the functionality of the different user interface elements and the implemented logic (e.g., client server communication) behind it without providing real-world usage context.

We illustrate two case studies in which we taught students with varying levels of experience how to apply software theater. (1) Section 6 describes a large capstone course involving industrial customers with up to 100 students and 10–12 teams per semester (Bruegge et al. 2015). In this course, 60 teams have performed software theater demos in their intermediate and final presentations since 2011. (2) Section 7 presents an interactive lecture-based course about software project management with up to 300 students (Krusche et al. 2017), in which 40 teams have performed software theater demos since 2015. We describe how we teach software theater in both course formats to enable other educators to adopt it in their own courses, and show the results of an empirical evaluation on the use of software theater in both case studies. Section 8 discusses these results with respect to the two hypotheses and provides best practices for instructors who consider to adopt the approach. Section 9 describes related work. Section 10 concludes the article.

2 BACKGROUND

In this section, we describe innovation management, interaction design, informal modeling, prototyping, scenario-based design, and the Tornado model as the foundation of software theater.

2.1 Innovation Management

Software development deals with the improvement of changing and uncertain environments and can be considered as an enabler to innovation management. Innovation is considered as "complex, uncertain, and subject to changes of many sorts" (Kline and Rosenberg 1986). Software projects deal with innovation because the user[3] needs, the technology, and the project environment can change due to external factors. Iterative and incremental approaches allow to start with a manageable set of requirements and to keep improving through validation and refinement cycles.

Validation is a crucial step in the iterative development of software systems; it ensures that innovation goes in the right direction and addresses the correct user needs, integrates the right technology, and provides the best project environment. It reviews whether the user understands the system as it is designed, or vice versa, whether the system is designed as expected by the users (Nielsen 1994). During validation, developers/designers and users need to communicate with the appropriate medium (text, picture or video, story-telling) and pattern (scenario based or not; from the system's viewpoint or from the user's).

2.2 Interaction Design

This communication is necessary so that designers understand how users interact with software systems. Sharp defines interaction design as "designing interactive products to support people in their everyday and working lives" (Sharp 2003). Norman introduced the terms `Design model` and `User model` to describe the mental understanding of designers and users and their interaction with the system (Norman and Draper 1986).

The `Design model` describes the designer's mental model of how the user will use the system through its user interface. A mental model is based on tacit (as opposed to explicit) knowledge and is difficult to transfer to other persons by means of writing or verbalization. Since the user only interacts with the user interface of the system, the design model focuses on the interface and interaction between user and system.

The `User model` presents the user's mental model of how the system will work from the usage perspective. While a user can have existing knowledge about the problem domain, this model is also formed by interacting with the system and by reading the documentation (e.g., user manuals).

[3]A user is an external stakeholder using the system. Different roles and types of users can be distinguished.

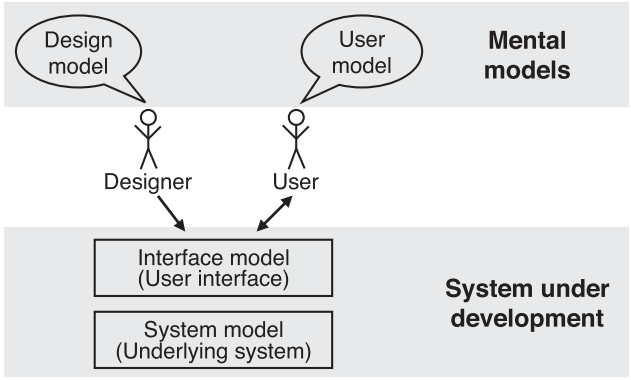


Fig. 1. The relationship between mental models and system under development (adapted from Norman and Draper (1986)).

Figure 1 shows these two mental models and their relation with the `Interface model` and the underlying `System model`.

The `Interface model` helps to emphasize the system components that are relevant to the end user and is separated from the system model. It hides the underlying details that are invisible to the user and is about the user-visible aspects of the system.

The `System model` provides an abstraction of the implementation and is used as a convention for communication among the developers. System modeling allows to focus only on the interesting aspects of a complex system and ignore irrelevant details (Bruegge and Dutoit 2009). System models are typically documented and conveyed in the form of Unified Modeling Language (UML) diagrams.

Design model and user model are mental models, i.e., "conceptual models formed through experience, training, and instruction" (Norman 2013). In a well-usable software system, the user model is consistent with the design model and is addressed by the interface model and the system model. User involvement helps to achieve usable systems in the software design process. It aims to evaluate the design and obtain feedback from users about the consistency of the design model with the user model.

2.3 Informal Modeling

Models are used for different purposes such as specification and documentation, but also facilitate the communication between developers. Formal models, such as UML diagrams, are used to specify and document the software system completely, consistently, and correctly. However, the creation of such formal models is usually time-consuming. Developers resist to change the model if they have invested a lot of time to create it. Small changes can lead to huge effort if all formal models have to be updated to ensure consistency. If changes occur, the time spent to create the formal model might have been worthless. Formal models are difficult for users without a technical background to understand (Pressman 2009). They cannot be used to validate the user model and the design model.

Informal models can be created faster and changed easier because they do not focus on completeness, consistency, correctness, and formality. They might not follow formal notations and include inconsistencies or vagueness for parts of the design that have not yet been realized. They are adapted incrementally and iteratively and are often used in agile methodologies to reduce the effort for comprehensive documentation as they focus more on communication (Beck et al. 2001).

They typically describe the look-and-feel and the interaction with the system from the user's point of view (Bruegge et al. 2012). Examples of informal models include sketches, paper prototypes, low-fidelity user interfaces, storyboards, and text-based and video-based scenarios (Xu et al. 2013).

2.4 Prototyping

Prototypes are examples of informal models and can be used to validate if the requirements are understood correctly. There are different prototyping techniques and tools to "involve an early practical demonstration of relevant parts of the desired software" (Floyd 1984). Demonstrating the design to stakeholders and expecting feedback from them provides validation at different levels: requirements, user experience design, and technical architecture design. The demonstration can be conducted based on prototypes of different fidelity levels depending on which is appropriate in the given situation. Ideally, the design should be demonstrated and evaluated when there is a change that may cause significant consequences.

Compared to fully implemented systems, prototypes allow to evaluate design ideas more quickly and at lower costs. This is achieved by defining appropriate focus and choosing the right form of prototype for the given situation (Arnowitz et al. 2010). As prototypes alone do not provide enough context of the usage, scenarios can be used as a complement (Weidenhaupt et al. 1998). Scenarios are concrete descriptions of the system usage and are helpful in making sound design decisions by focusing on both the problem space and the solution space (Jarke and Pohl 1993; Jarke et al. 2010).

2.5 Scenario-based Design

Another example for informal models are scenarios, which are used as examples for illustrating common use cases and requirements of the system. Their focus is on understandability and communication. Different types of scenarios exist. Visionary scenarios describe a future system: they are used by developers to refine their ideas and as a communication medium to elicit requirements from users. They can be viewed as an inexpensive prototype. As-is scenarios describe an existing system or a current situation. They can be validated for correctness and accuracy with the users of the corresponding system/approach. Demo scenarios focus on the demonstration of a system/approach: they describe a future system that is currently implemented and needs to be validated.

Scenario-based design is a development approach that uses scenarios to design the future system. It focuses on the users of the system and their interaction with the system. Rolland and his colleagues state that "people react to 'real things' and ... this helps in clarifying requirements" (Rolland et al. 1998). Scenarios are intuitive and suitable for communication and validation. The story-like description with context information makes it easier for stakeholders to understand abstract concepts in the system design. Scenarios are cheap to create and enable quick iterations. In a changing environment, the design of the system often takes several revisions to reach a "stable" state. Scenarios are also open-ended and stimulate the user's imagination. They enable the users to come up with more specific requirements and help "the analysts to consider contingencies they might otherwise overlook" (Carroll 2000). There are different ways to use prototypes with scenarios depending on who actually demonstrates the scenario. It can be user-performed, where users are provided with scenarios as description of tasks and are told how to use the prototype to perform the task (Pohl 2010), or developer-performed, where scenarios provide a context in which the prototype demonstrates how to achieve specific tasks (Sutcliffe 1997). Weidenhaupt and his colleagues report that combining the development of scenarios and prototypes enables stakeholders to check, discuss, and update scenarios and prototypes at the ground level, and can lead to better customer satisfaction (Weidenhaupt et al. 1998). This is particularly important in innovative applications based on emerging technologies (such as wearable computing, Internet

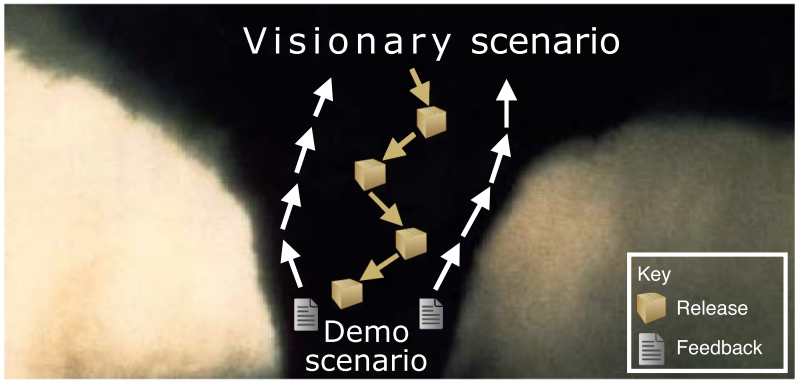


Fig. 2. Tornado model: wide in analysis, narrow in implementation with multiple releases funneling down and feedback as updrafts (Bruegge et al. 2015).

of Things, augmented reality, virtual reality) because the desired applications have not been seen or even imagined by users before. The IKIWISI (I'll know it when I see it) principle (Boehm 2000; Cao and Ramesh 2008) must be considered.

2.6 Tornado Model

The Tornado model, shown in Figure 2, describes how visionary scenarios can be transformed into demo scenarios in order to show an executable system that represents the vision under evaluation (Bruegge et al. 2012). It is a demo-oriented development process aiming to deliver "touchpoints," a metaphor for creating executable prototypes that evaluate design ideas and obtain feedback from the stakeholder. Informal models are good at closing the communication gap between developers and users because they are easier to create, understand, and revise. The Tornado model highlights the importance of informal models in addition to formal models. This is also expressed with the idea of yin-yang balance in software development (Xu et al. 2013).

Visionary scenarios represent design ideas of systems, and are used for requirements brainstorming. They require several rounds of iterations to reach a stable version. The main task at this stage is exploring the problem space, so low-fidelity prototypes are sufficient. Demo scenarios are refinements of visionary scenarios for reviews and presentations. They provide a demonstration how the core of the problem is realized when using the system and can be played out in a demo. Demo scenarios are based on a (partially) working system and take advantage of mockups for cost-efficiency reasons.

The Tornado model describes an evolutionary scenario-based design process. The initial version of the design is depicted using low-fidelity prototypes, used in the early stages to obtain user feedback about the user interaction design. This enables the user to explore possible design alternatives and reformulate the initial requirements. In the middle of the project, when more stable design alternatives have been chosen, interactive prototypes (e.g., created with Balsamiq[4]) are used for a more tangible and reliable evaluation. At the end of the project, the finally adopted design is implemented and delivered using the tornado metaphor: A tornado is wide in the clouds (vision), but only a part of it funnels down and hits the ground at its touchpoint (demo). The touchpoint is where an executable demo system is created and presented. Feedback, obtained through the demo, influences the visionary scenario as updrafts of the tornado.

[4]Balsamiq is a digital low-fidelity mockup tool with support for executable prototypes: http://balsamiq.com.

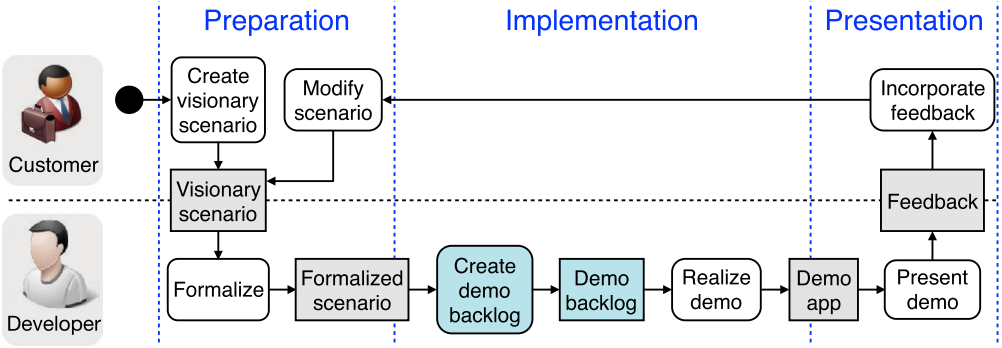


Fig. 3. Main activities and artifacts in the software theater workflow divided into three phases represented with swim lanes (UML activity diagram). The blue elements are explained in further detail in Figure 4.

3 SOFTWARE THEATER WORKFLOW

Similarly to performing a prototype based on predefined scenarios, software theater is performed based on a theater screenplay. The screenplay describes the event flow of the demo, the cast (participating actors), and the props required for the demo. The purpose of software theater is to demonstrate how end users would benefit from the new product in a real-world context. Figure 3 shows the full workflow for creating a demo using software theater. The workflow is divided into three phases represented with UML swim lanes: preparation, implementation, and presentation.

3.1 Preparation

In the beginning of the project, the customer creates the problem statement, which includes `Visionary scenarios`. The development team prioritizes these visionary scenarios with the customer. Each scenario is formalized using a template with six components: scenario name, participating actors, flow of events, entry conditions, exit conditions, and quality requirements. A `Formalized scenario` is the basis for the demonstration. It describes the same content as the visionary scenario, but in a structured way (Bruegge and Dutoit 2009): "Formalization helps to identify areas of ambiguity as well as inconsistencies and omissions in a requirements specification."

3.2 Implementation

After the preparation, the developers start to implement the demonstration. This phase consists of the activities `Create demo backlog`, which involves multiple sub-activities and `Realize demo`. Figure 4 shows the detailed actions and artifacts for the activity `Create demo backlog`.

The team writes the `Screenplay` (called demo script) based on the event flow as well as the participating actors of the formalized scenario, and identifies the props and stage directions needed for the scene. The developers select the subsystems and services that are required to realize the demo based on the system architecture. The focus of the demo is on subsystems that require technical validation (e.g., performance-critical or features related to user experience).[5] Less critical subsystems can be mocked to save development time for the demo.

The team identifies participating methods and participating objects in the selected subsystems by inspecting the flow of events in the formalized scenario. It uses textual analysis (e.g., Abbott's technique) to identify nouns as candidates for classes (participating objects) and verbs as

[5]The selected subsystems are highlighted in the subsystem decomposition (UML component diagram).

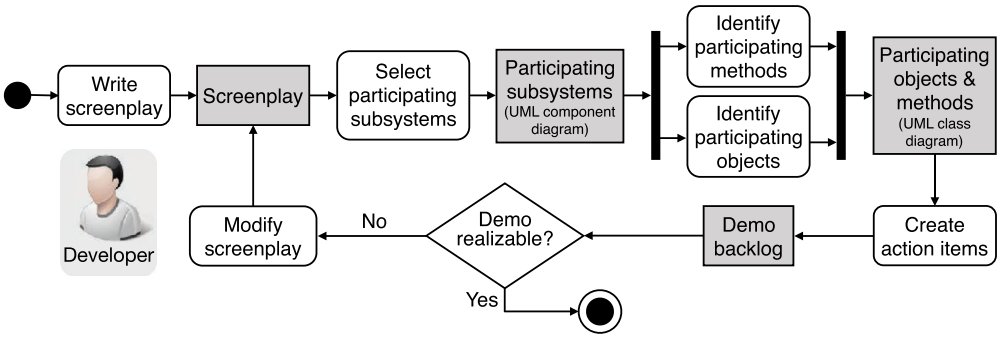


Fig. 4. Detailed actions and artifacts for the `Create demo backlog` activity (UML activity diagram).

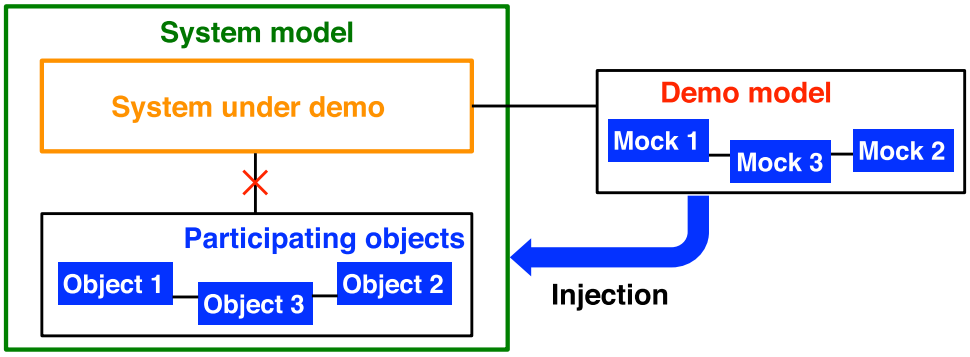


Fig. 5. Model-based demonstration using the mock object pattern: the demo model with mocks replaces participating objects that are not implemented yet.

candidates for operations (participating methods) (Abbott 1983). This identification can be performed in parallel and includes the creation of a UML class diagram that includes these objects and methods. Mocked subsystems, objects, and methods are highlighted in the class diagram so that all developers are aware of the decision.

Developers apply the mock object pattern (Mackinnon et al. 2000) in which real collaborators (i.e., participating objects) are replaced by mock collaborators as shown in Figure 5. Based on the concept of model-based testing (Apfelbaum and Doyle 1997), we call this technique **model-based demonstration**. It allows to focus on the implementation of the identified objects and methods (`System under demo`) and to mock other parts of the system that are not relevant. The dependency injection pattern (Martin 1996) allows to switch between mock and real implementations during development.

The resulting `Demo backlog` contains all the action items for the implementation of participating objects and methods and represents the task model and management aspects behind the workflow. The demo backlog also includes the generation of collaborating mock objects and the preparation of props. After the creation of the demo backlog, the developers assign the action items and estimate if they are able to realize all action items before the demonstration.

If they answer this question with "Yes," they start with realizing the demo. Otherwise, they need to modify the screenplay and leave out certain aspects in the demonstration or mock additional subsystems, objects, and methods. The realization of the demo app can follow a continuous integration and continuous delivery process. During the realization, the developers tick off

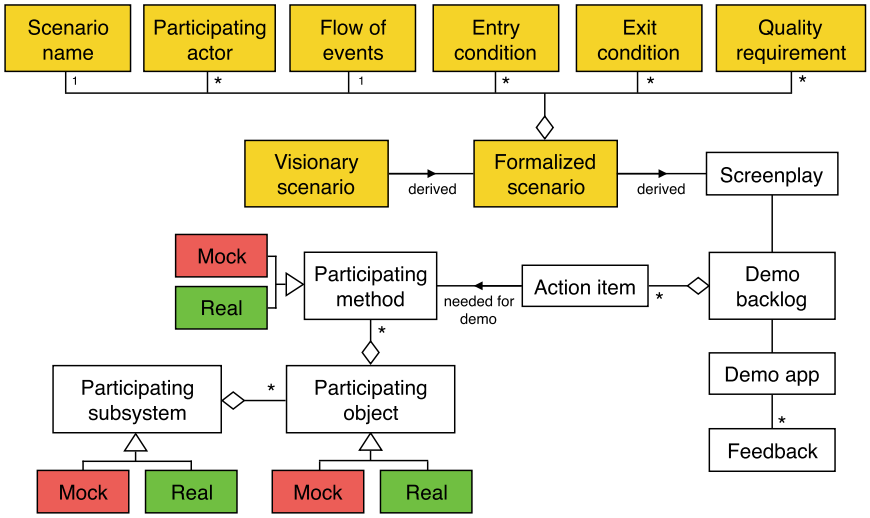


Fig. 6. Software theater's main concepts and relationships in an analysis object model (UML class diagram).

finished action items or further refine the demo backlog. After the realization and the delivery, the workflow proceeds to the presentation phase.

3.3 Presentation

During the presentation, team members assume the roles of the actors of the screenplay. After the demo, the team collects feedback and incorporates it by modifying the visionary scenarios. Typical feedback includes user interface issues and conceptual aspects of the system; it usually leads to changes in the prioritization of the visionary scenarios. Sometimes, it even requires a change of the used technologies.

Figure 6 shows all artifacts of the software theater workflow as objects in an analysis object model. This illustrates how these artifacts are related to each other. For instance, the action items are connected to participating methods that need to be realized for the demo. The action items therefore connect the task model (demo backlog) with the object model of the developed system that is visualized in a UML component diagram (participating subsystems) and in a UML class diagram (participating object and participating methods).

4 SOFTWARE THEATER EXAMPLE

To exemplify this workflow, we show an example taken from our capstone course (cf. Section 6). It includes all artifacts of the software theater workflow for one concrete project of the course.

The project was done in collaboration with Zeiss Meditec,[6] an industry partner in the field of medical technology. Its goal was to develop the Zeyes app to digitize the process of stock taking and re-ordering of lenses for cataract surgeries at hospitals around the world. One of the visionary scenarios received from the customer was as follows:

> *The sales representative for Mexico is in charge of managing several customers. The sales process of consumables involves much more operational activities for the sales representative than the sales of devices. She needs to visit each customer regularly to initiate replenishment orders for the customer and to conduct stock checks.*

[6] Zeiss Meditec is a department of Zeiss, specializing in medical technologies: https://www.zeiss.com/meditec.

Scenario name	Perform stock taking	
Participating actors	Christina: Sales Representative, Matthias: Sales Representative	
Flow of events	**User steps**	**System steps**
	1) Tap on 'Mexican Medical Company' in the favorite customer list	
		2) Show customer locations
	3) Tap on the customer location 'Mexican Hospital 1'	
		4) Show the current stock item list
	5) Tap on 'Scan barcodes'	
		6) Activate camera
	7) Point camera at the barcode on each box	
		8) Identify the corresponding item and insert it into the scanned list. If an item is scanned twice, show an alert.
	9) Tap on 'Show Report'	
		10) Show the list of identified and missing items
	11) Uncheck 'Reorder items with report' and tap on 'Send report'	
		12) Show loading indicator and notification of sent report
Entry conditions	• The app is connected to the customer service hub and opened • The sales representative is logged in	
Exit conditions	• The correct report with all scanned items is sent to customer service hub	

Fig. 7. Formalized scenario for the stock taking demo with the flow of events as basis for the screenplay.

> *Both processes are quite manual today and reduce the time she can talk to the customer about new products and application related questions during her visit. The manual process introduces errors in stock taking.*
>
> *She already works with her smartphone for the customer account management, so she would appreciate if she could also use it for ordering and stock management. This would be desirable because mistakes in stock taking introduce organizational issues and costs, and every minute that she saves in this process can be used to have value added discussions with the customer.*

The team selected parts of this scenario for their demonstration and formalized it. Figure 7 shows the formalized scenario with name, participating actors, flow of events with user and system steps, as well as entry and exit conditions (Bruegge and Dutoit 2009). This concludes the artifacts of the "preparation" phase of the workflow.

The team wrote a corresponding screenplay for their demonstration. It chose to have a narrator who leads the audience through the scene and two employees who perform the stock taking process. During the theater, Christina manually takes the stock and gets frustrated by its inefficient and confusing nature, while Matthias uses the Zeyes application, which enables him to scan the products by barcode, see which products are missing in the inventory, and send a report with the results. Figure 8 shows this story as screenplay. After writing the screenplay, the students continued with the selection of participating subsystems, and the identification of participating objects and methods. They serve as basis for creating the demo backlog. The team did not document these steps, as they only went through them informally. Therefore, we used their system models to create possible representations of the artifacts in this phase of the workflow to complete this example.

Figure 9 shows a simplified version of the subsystem decomposition that focuses on the components included in the demo. The `Smartphone App` runs on the customer representative's smartphone at the customer location (e.g., a hospital). It uses the `Smartphone Camera` to identify items by barcode and an `NFC Reader` (near field communication) to scan items by NFC tag. It communicates with the `Customer Service Hub` to retrieve necessary customer information, to submit reports, and to place orders as a result of stock taking.

(A storage room with a big shelf with boxes with barcodes on them. CHRISTINA walks in with a stack of binders and lots of paper, almost tripping over her big pile. She puts it down on a desk and begins manual stock taking. MATTHIAS walks in with a smartphone in his hand.)

NARRATOR
These two are Christina and Matthias. They both are Sales representatives. They just entered a hospital to perform a cycle count in order to check which items are still there and which items need to be re-ordered. This is a customer service provided by sales representatives.

CHRISTINA
(fumbles around with the paper and notes thing down, having a hard time and getting confused and frustrated)

NARRATOR
On the one hand, Christina has to look up all the information she needs in her stack of paper. Matthias on the other hand, is using the newly developed App. Now, Matthias, can you please show us how the app works?

MATTHIAS
Yes, of course, I'll just hop right in. I open up the app and I can already see my favorite customers. These customers I visit regularly, and now I choose Mexican Medical Company (select Mexican Medical Company). Then I can see all the locations where they keep stock. We are in Mexican Hospital 1, let's select that (select the location).

Down here (point at the list), I can get a quick overview over every item that's supposed to be here. Let's check if this is true. I point the camera at the item (scan the barcodes of the boxes one by one) and it starts scanning.

CHRISTINA
(is obviously pretty stressed out and keeps dropping boxes. The storage room is a complete mess.)

MATTHIAS
One barcode is not here, we found a missing item! Let's klick on the report icon (click on the icon) and we can see which item is missing. The customer support should know that. We don't want to order the missing item (uncheck the option to order) so I just send the report.

CHRISTINA
I hate this, this is a complete disaster. I will never finish it in time.

MATTHIAS
(Walks over to Christina.) This is a horrible mess you have created.

CHRISTINA
It's such a mess. I will have to start all over again.

MATTHIAS
Well, I can help you with the new feature in the app. Check this out, we now have scanning via RFID in the app. So I can just press the button (press 'scan via RFID' button) and keep the device close to all items (wait for all items to appear)… and we're done. Every item was there, so I just send the report and your work is done.

CHRISTINA
Oh my God, this is exactly what I needed, then I'm not always the one to clean up the storage rooms of all the hospitals.

MATTHIAS
So, we're finished. Let's have dinner!
(both of them walk out together, smiling)

Fig. 8. Excerpt of the screenplay for the stock taking demo with the narrator, Christina, and Matthias as actors. The screenplay starts on the left side and continues on the right side.

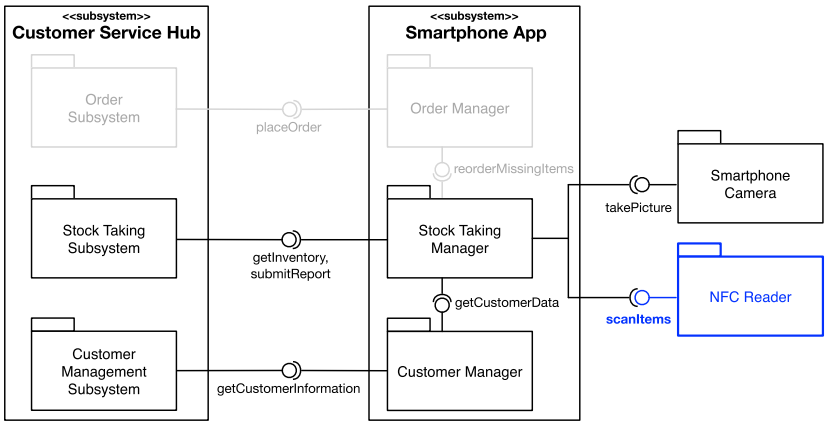


Fig. 9. Selected participating subsystems for the stock taking demo (UML component diagram). Not participating subsystems are shown in grey. The blue subsystem NFC Reader is mocked.

The demo scenario at hand did not include submitting an order, which means that the `Order Manager` and the `Order Subsystem` are not relevant for preparing the demo and are thus greyed out in the UML model in Figure 9. The team knew already that they would mock the `NFC Reader` because they would not have the necessary accessory to read NFC tags on the iPhone. By realizing early which parts of their system are most relevant for the demo, the team could concentrate on implementing the critical functionality. The next step of the workflow involved the identification

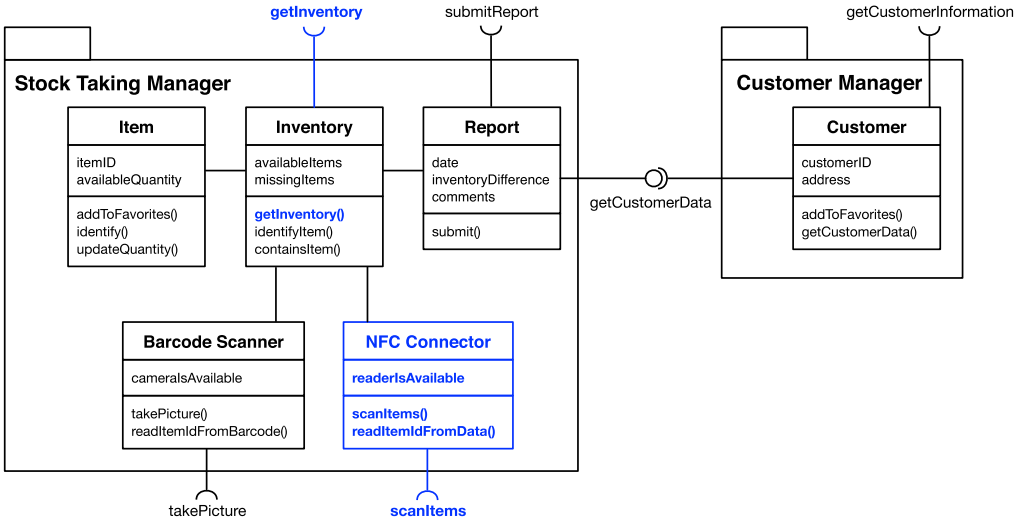


Fig. 10. Identified participating objects and methods for the stock taking demo (UML class diagram). Blue elements are mocked.

of participating objects and methods. Starting from the selected demo-critical subsystems, the developers identified the corresponding objects in these subsystems that they needed to implement for the demo. Figure 10 shows the result of this step, simplified to include only the two components of the Zeyes app that are required for the demo.

The developers decided which objects and methods can be mocked for demonstration purposes. For instance, they might need to mock a component because they do not have the time to fully implement it or because there is a technological constraint which keeps them from implementing it. Another reason could be that they want to reduce the risk of demo failures, e.g., in complex setups with distributed components. The team mocked the component `NFC Reader` because it did not have a reader to connect to the iPhone at the time. It implemented a mock object `NFC Connector`, which always indicates that the reader is available and returns the same, pre-defined data when scanning items.

The developers also mocked the interface `getInventory` provided by the `Stock Taking Subsystem` in the `Customer Service Hub` (cf. Figure 9) because the data of real server responses contained cryptic item names. They changed the `getInventory` method in the `Inventory` object to return more understandable item names for their live demo. The mocked elements are marked in blue in Figure 10.

After identifying the participating methods and objects, the team created the demo backlog including implementation tasks and organizational tasks. These tasks should be small enough so that they can be easily distributed to individual team members. Figure 11 shows an excerpt of the demo backlog for this demonstration, including action items for implementation, for the creation of mocks, and for the preparation of props.

The team performed their demonstration on February 2, 2017. Figures 12 and 13 show scenes of the live demonstration where two sales representatives interacted with the developed software and with each other.[7]

[7] A recording of the demo is available at http://youtu.be/MTayd0kyY6Y (the demo starts at 06:30).

- [ ] Implement Barcode Scanner
- [ ] Implement getCustomerData interface
- [x] Fix error when scanning multiple items
- [ ] Mock NFC Reader
- [x] Print barcodes (props)
- [x] Get binders and stack of paper (props)
- [ ] Set up Mexican customer data
- [ ] Put stock in customer inventory
- [ ] Put boxes on shelf (props)
- [ ] …

Fig. 11. Demo backlog with tasks to realize for the stock taking demo.

Fig. 12. Software theater demo: actors Christina and Matthias perform the scenarios for manual and digital stock taking (Zeiss Meditec project).

Fig. 13. Software theater demo: Matthias explains to Christina the advantages of the Zeyes app that enables digital stock taking (Zeiss Meditec project).

5 SOFTWARE THEATER PATTERNS

Based on the patterns idea (Alexander et al. 1977), we identified three software theater patterns that highlight the creativity and flexibility of software theater demonstrations.

5.1 Narrative Pattern

The narrative pattern describes the role of a narrator in the software theater scene. Teams typically use this pattern to leverage its capability for detailed explanation: for instance, the narrator can set the scene and explain how and why the system solves a problem in the particular situation.

An example of the application of the narrative pattern was the Zeiss Industrial Measuring Technology (ZIMT) project. It addressed the problem of monitoring expensive machines shipped to the customer around the world using multiple transport methods such as trucks, container ships, and sometimes even elephants. To know whether the shipping company is liable for a damage of the machine upon arrival, the customer accompanies each machine with a sensor, measuring shocks in all three axes, as well as deviations in temperature and humidity. The team developed a corresponding smartphone application. A Zeiss service mechanic can use it to read the sensor data and get a first clue on where to start inspecting the damaged machine.

The demonstration showed the machine in a closed box and the collection of sensor data was not an instantly visible process. Therefore, the team made the problem more obvious by involving a narrator who explained what was happening from the perspective of the machine. They even used the first person to make the situation more relatable. For instance, the scene started with the machine introducing herself and arguing: "Because of my high value, I have a sensor with me. It

tracks shocks and extreme temperature values. Zeiss uses it to keep an eye on my status." As a result, the actors could play their parts without explaining the rationale behind their actions.[8]

5.2 Explanation Pattern

The explanation pattern shows a technical detail of the system within a software theater scene. Some teams illustrate an algorithm, while others explore the details of a particular technology. An example for this is the Allianz project in the winter semester 2014/15. The application involved the identification of intrusions at home with the help of multiple sensor readings. The team used the blackboard architectural pattern (Buschmann et al. 1996) to detect an intrusion such as a burglary from the sensor readings. They brought in a superhero named AllianzMan symbolizing the intelligence of the system to explain how this pattern works.

AllianzMan coordinated several experts, illustrating the specialized knowledge sources of the blackboard pattern, such as a sensor change expert or an occupant identification expert. Each of these experts was played by a team member. In their demo, an intruder accessed the building while the home owner was away, leading to unusual sensor readings such as opening doors, lifting valuable objects, and raising noise levels. The experts took turns passing the values back and forth, analyzing and discussing them to combine their knowledge, and getting closer and closer to a solution under uncertain conditions. When they decided that this was likely an intrusion into the home, they alerted the home owner. Figure 14 shows a scene of the demo.[9]

5.3 Metaphor Pattern

Another software theater pattern is the use of a metaphor (e.g., a concept of a different domain) to explain the purpose of the system in a relatable way. This pattern is typically used when the system addresses a problem that only occurs in a certain field or for a particular kind of user, or if the concept behind the system is abstract and not easily understandable.

The BSH project in the winter semester 2016/17 used this pattern in their demonstration of their system called "Scentdipity." The app was built to help people design perfumes from home, introducing personalization and new opportunities for creativity. To illustrate the concept of creating a perfume out of solvents and scents coming from different olfactive families, the team chose to demonstrate their system with a blender mixing smoothies.

They adapted their application to show only scents based on fruits and explained why it was important that the system hides bad combinations of ingredients, or helps to select the right amount of solvent (symbolized by a carton of milk) for the perfume. This made it easier for end users to create a favorable result and to feel like the product is their own creation at the same time. The actress also showed how the Scentdipity machine (illustrated by the blender) could mix small amounts of the desired perfume for testing purposes. Figure 15 shows an impression of the demo.[10]

6 CASE STUDY 1: CAPSTONE COURSE

We teach software theater in our regular capstone course "iPraktikum," which takes place twice per year at the Technical University of Munich. In this course, 80–100 students develop systems involving a mobile component for a real customer from industry. The course is based on a multi-customer organization (Bruegge et al. 2015) with 10–12 projects running at the same time, and a continuous process model, Rugby (Krusche 2016). Rugby extends Scrum with continuous delivery workflows and was adapted for university courses to account for part-time developers.

[8] A recording of the demo is available at http://youtu.be/enpEO6bGaZQ (the demo starts at 01:55).
[9] A recording of the demo is available at https://youtu.be/efHEQQaVp6U (the demo starts at 04:20).
[10] A recording of the demo is available at http://youtu.be/85goFfxUkuk (the demo starts at 03:50).

Fig. 14. Illustrating the knowledge sources of the blackboard pattern with software theater using the explanation pattern (Allianz project).

Fig. 15. Borrowing concepts from other domains in software theater using the metaphor pattern (BSH project).

6.1 Course Description

Each project team consists of a project leader, who is responsible for the project outcome as well as grading the participants. The project leader works together with a team coach; this is a student who has previously participated in the course as developer, assumes the role of a scrum master, and takes care of day-to-day problems and agile processes. Each team consists of six to eight developers, some of which take an additional functional role, involving tasks or workflows such as modeling, release management, or merge management that are important across all projects. Functional teams meet regularly to discuss their respective topic and bring the knowledge back into their project team. In course-wide lectures, we cover topics concerning all participants, e.g., software theater.

We describe the structure and workflows of the course in more detail in Bruegge et al. (2015). Setting and structure of the course are very close to industry and address the majority of the gaps identified by Nurkkala and Brandle (Nurkkala and Brandle 2011). We have held 12 iterations of the course, including over 110 distinct projects with different customers from industry,[11] since its introduction in 2008.

We studied the use of software theater during a capstone course running from October 2016 to February 2017. In this instance, 91 students (80 developers and 11 coaches) participated in the course, and were distributed into 11 teams, each between 8 and 9 participants. We started the course with a kickoff meeting on October 20, 2016, in which the industry partners presented their problem to all participants. We then allocated the students to teams based on their project priorities and prior experience, taking into account the importance of balanced teams for learning (Bruegge et al. 2015). An excerpt of the organizational chart with five teams of the course is shown in Figure 16. In the following 2–3 weeks, each team went through a Sprint 0, an iteration where the focus is not on development, but on understanding and analyzing the problem (Bruegge et al. 2012; Dzvonyar et al. 2014).

Eight weeks after the beginning of the course, all teams presented the results of their requirements analysis and system design activities in the design review, an intermediate milestone. While we emphasized the importance of showing the architecture and technical details of the system, we encouraged the participants to also incorporate a demonstration using software theater in their presentation. We believe that the demonstration of an executable prototype early in the project leads to a better understanding of the project status and more realistic feedback from clients and

[11] Descriptions and videos for all projects: https://www1.in.tum.de/lehrstuhl_1/projects/all-projects#iOS.

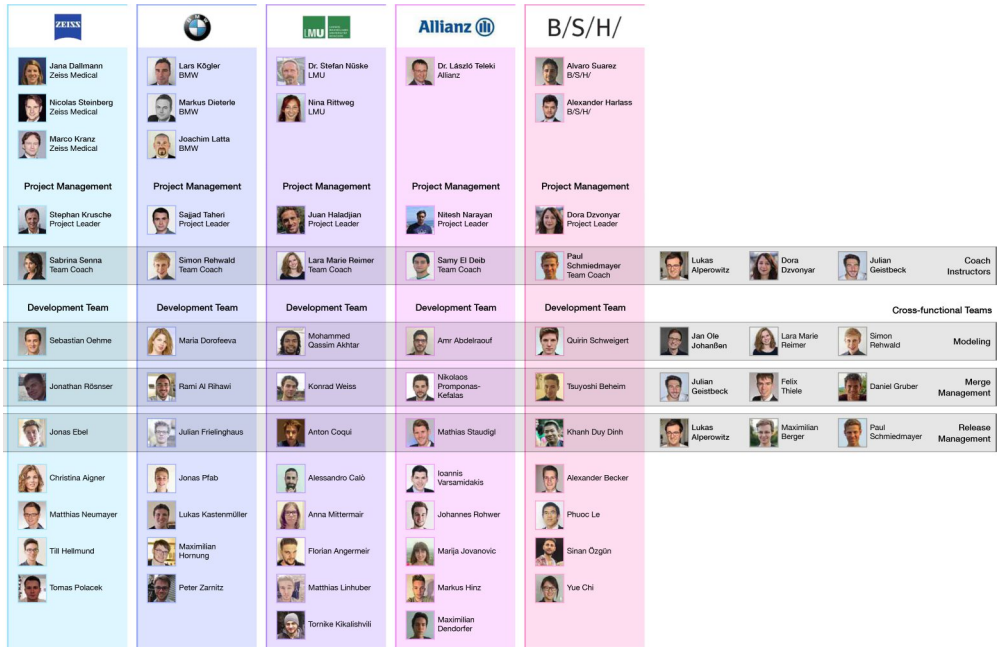


Fig. 16. Excerpt of the organizational chart of the capstone course in the winter semester 2016-17.

other stakeholders. The teams can use this feedback to refine the requirements for the following iterations (Bruegge et al. 2015).

To teach the participants the necessary knowledge about software theater, we held a course-wide lecture in which we introduced the software theater workflow, showed them examples, and provided recommendations and best practices on how to prepare their demo. This lecture took place 3 weeks before the design review so that the students had enough time to apply the workflow and to iterate over their demo and presentation. The teams rehearsed the theater play multiple times and iteratively adapted the interaction between the users and the developed demo app. We used Prototyper (Alperowitz 2017), which allows the repeated delivery of executable prototypes to the target environment, in our case the demo apps to the theater stage. We provided feedback on their presentation in a dry run one week before the design review.

After the design review, the teams continued development for another 7 weeks and presented the final outcome of their project in the client acceptance test on February 2, 2017. In these presentations, we asked the teams to focus on a meaningful demonstration of their system that is understandable to a broad audience. We also encouraged them to perform regular demonstrations to their customers using software theater in-between the course-wide events. The presentation recordings of the design review and client acceptance test are available on our web site.[12]

6.2 Evaluation Design

We closely observed the teams during our capstone course in the winter semester 2016/17 to analyze the impact of software theater. We measured the following variables at our main events, the design review, and client acceptance test:

[12] http://www1.in.tum.de/ios1617.

Table 1. Overview of Software Theater Usage for Each Team in the Design Review and the Client Acceptance Test

	Team name	Allianz	BMW	BSH	iHaus	LMU	McKin	Quart	T-Sys	ZIMT	Zeyes	ZF
	Number of developers	8	7	7	8	8	7	7	7	7	7	7
Design review	Application	yes	yes	no	yes	yes	yes	yes	no	yes	yes	yes
	Demo length [min]	2.6	3.6	3.6	2.1	2.4	2.6	2.3	0	3.4	2.8	1.1
	Demo participants	2	3	1	4	3	3	4	0	2	2	5
	Integration	no	yes	no	yes	no	no	no	no	no	no	no
Client acceptance test	Application	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
	Demo length [min]	4.4	2.6	5.4	2.5	3.3	3.5	3.7	9.1	6.2	3.5	3.2
	Demo participants	3	3	3	4	6	3	4	4	4	2	3
	Integration	no	no	no	yes	no	no	no	yes	yes	no	yes

(1) **Application:** whether teams used software theater for their demonstration
(2) **Demo length:** the length of the demonstration in minutes
(3) **Demo participants:** the number of participants in the demonstration
(4) **Artifacts:** which artifacts of the software theater workflow the team produced and documented
(5) **Integration:** the integration of the demo into the presentation

We regularly verified our observations in the development tools the teams used for communication and documentation. For instance, we checked whether the teams documented software theater artifacts in their team wiki. In addition to these observations, we evaluated the usage of software theater and the personal opinions of the students in an online questionnaire. After the client acceptance test, we sent out a survey to 80 developers and 11 team coaches who participated in the course. The survey included questions about the demonstrations in both major presentations. In particular, we had the following areas of interest:

(6) **Role:** the respondent's role at each event (demo participant, presenter, or none)
(7) **Preparation time:** the amount of time the respondent spent on the preparation of each demonstration
(8) **Mocks:** whether parts of the system were mocked and, if yes, why they were mocked
(9) **Software theater value:** a comparison of a demo using software theater to a normal demonstration
(10) **Software theater characteristics:** opinion on software theater as an approach to realistic demonstrations

We combined the observation and the questionnaire to get both a first-hand account of the students' experience using software theater, and a more reliable and complete overview of the usage of the approach in the case study.

6.3 Evaluation Results

We analyzed the demonstrations and measured the above variables for each of the 11 teams. Table 1 summarizes the results of the evaluation for both events: `Design review` and `Client acceptance test`.

In the design review, the teams dedicated on average 2.4 minutes out of the 10 minutes of presentation time to perform a live demonstration of their early software prototype. Nine out of the

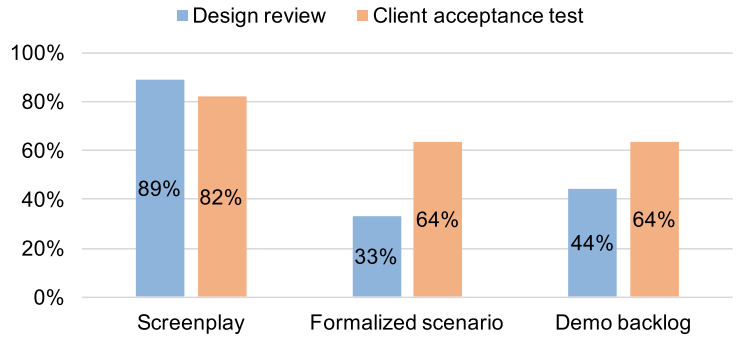


Fig. 17. Percentage of teams who produced software theater artifacts in the capstone course (grouped by event).

eleven project teams used software theater. Team BSH chose to narratively walk through their current state with only one demo participant. Team T-Systems skipped the demonstration because they felt they had not made enough progress with the software due to hardware challenges they needed to overcome in the first weeks of their project. Two of the teams designed their presentation from ground up with the demonstration in mind, integrating it into their story in multiple parts. For instance, team iHaus performed the first part of the demo and then interrupted it to explain a technical detail of their application before carrying on with their software theater. This allowed them to use the practical demonstration of their system to support their arguments and explanations in the presentation.

In the client acceptance test, all teams used software theater, with an average demo length of 4.3 minutes, which accounted for almost half of their presentation time. Team T-Systems, who had not included a demonstration in the design review, now chose to perform their whole presentation with software theater, explaining their requirements and project state through their role play. The number of teams that interweaved demo and presentation doubled in the client acceptance test, with teams using their live demo to explain technical details of their system in depth as described with the explanation pattern in Section 5.2. Team ZF performed their software theater in three parts, with interruptions explaining their algorithms for the detection of dangers while driving (detection and warning of intersections, gaze detection, and alert of distracting audio levels in the car). All teams had at least two demo participants; one team had six (3.5 on average).

Figure 17 shows how many teams produced software theater artifacts throughout the project. The majority of the teams (89%) wrote a screenplay and up to 64% created a formalized scenario as well as a demo backlog.

In the survey, we received 80 responses (70 developers and 10 team coaches, response rate: 88%). We were interested in the amount of preparation that went into the demonstration at each event, including producing the artifacts of the software theater workflow, preparing mock data, practicing the demonstration, preparing props, and iterating over the demo or giving feedback to fellow team members. Figure 18 shows the responses to this question. For the design review, 36% of respondents indicated that they spent 3 hours or less to prepare for the demo, while 24% invested 10 or more hours. The preparation time increased for the client acceptance test: 37% of participants prepared more than 10 hours, while 29% spent less than 3 hours on the demo.

We asked the developers and coaches to compare a demonstration with software theater to a normal software demo using a pre-defined set of adjectives. As illustrated in Figure 19, over 80% of respondents think that a demonstration using software theater is more creative, fun, and memorable than a demo without the technique, and two-thirds think that it is more dynamic

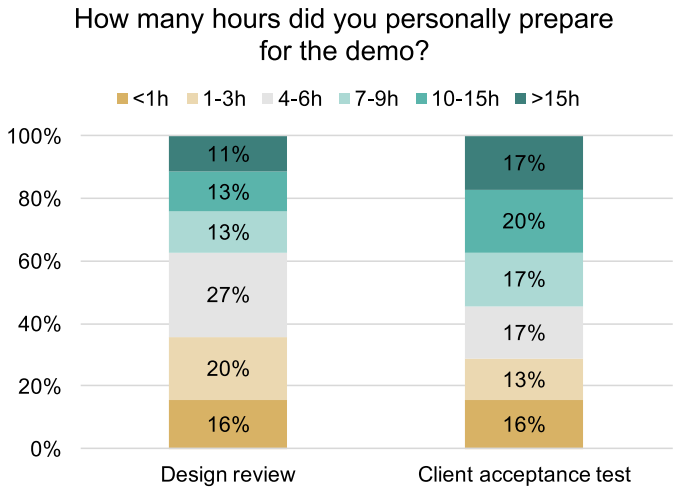


Fig. 18. Survey responses: preparation time for the client acceptance test was higher than for the design review.

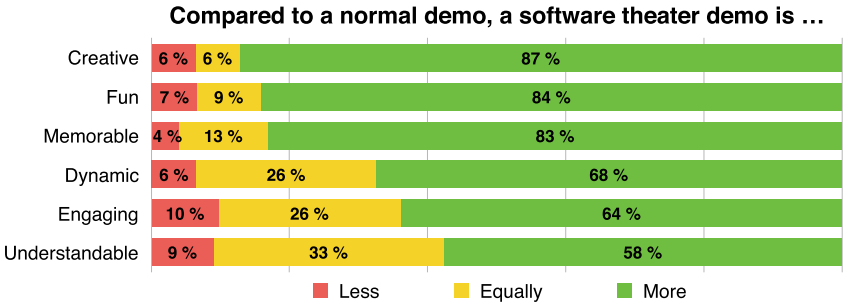


Fig. 19. Survey responses: the majority of students thinks that software theater is more creative, fun, memorable, dynamic, engaging, and understandable compared to a normal demo.

and engaging. Less students, 58%, think that software theater makes a demonstration more understandable.

This is also visible in the responses to the set of 3-point Likert-scale questions shown in Figure 20: while most participants agree with having an improved understanding of *other teams'* systems through software theater, their opinions are divided concerning *their own team.* 34% of respondents agreed that software theater helped them to understand their own project's requirements, with roughly the same amount of neutral responses and disagreeing participants. More participants disagree than agree that software theater helped them to communicate with their client. However, 55% agree that their demonstration using software theater gave them confidence about their system's usefulness, and the majority of respondents will use the approach to create future demonstrations.

71% of the respondents indicated that they mocked parts of their system using the mock object pattern. Among the most frequently stated reasons were the following:

(1) No access to components necessary to realize a certain functionality, e.g., NFC reader, sensors, or an external machine to communicate with.

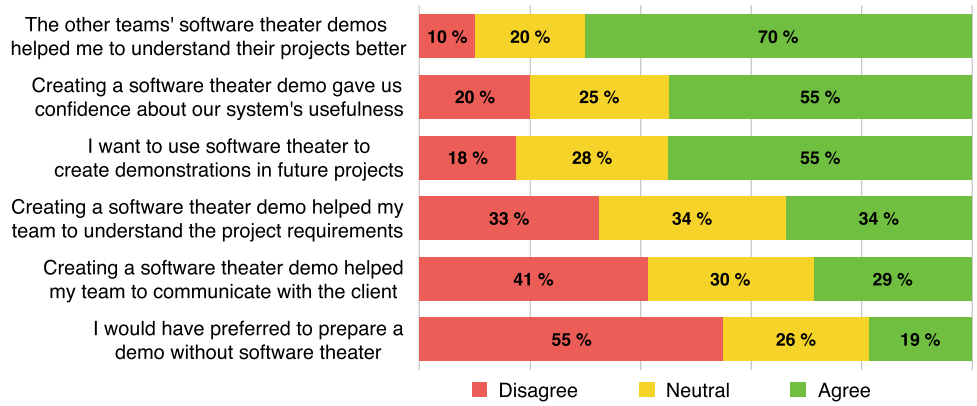


Fig. 20. Survey responses: disagreement (red) and agreement (green) with software theater characteristics on a 3-point Likert scale.

(2) Fallbacks for connectivity issues, e.g., mocking server responses in case the internet connection fails during the demo.
(3) Fallbacks for complex computation, e.g., complex sensor data analysis, which would have taken too long for the demo.
(4) Access to data that is not possible to get during the demo, e.g., historical GPS data, which would otherwise not be present on the demonstration device.

We discuss these findings in Section 8.

6.4 Threats to Validity

One limitation of our evaluation is that we did not have a control group within the same course, because all teams in our capstone course use software theater. Another threat is that we used Likert scales, which may be subject to distortion (Garland 1991). For instance, respondents may have avoided using extreme response categories (central tendency bias) and they may have agreed with statements as presented (acquiescence bias). They also may have tried to portray themselves or our course in a more favorable light (social desirability bias) because they were afraid that this would influence their grades and were thus trying to please the instructors. We addressed this threat by collecting the responses anonymously and by preventing multiple responses from the same student.

Novelty bias is an additional validity threat. The fact that most students interact with software theater the first time in their studies could cause an increased interest. We think that this threat is low because students who participate in the capstone course twice are equally motivated about software theater during their second time. To alleviate the threat of selection bias, we asked students in which team they worked: in each team, at least half of the team members responded in the online survey. We also analyzed the results on a team basis. While there are small deviations between the different teams, the general positive opinion about software theater is present in all teams. In addition, we found similar results in informal, personal interviews that support the statements we found in the questionnaire. Therefore, we think that the threat of selection bias is low.

Another threat to the validity in the evaluation is that most students were beginners in the taught concepts and mostly reported about their perceived experience within the course. Beginners

Table 2. Overview of the Schedule and Content in the Course POM

Week	Content
1	Team formation
2	Project organization
3	Software process models
4	Agile methods
5	Prototyping
6	Proposal management
7	Branch & merge management (Krusche et al. 2016)
8	Contracting
9	Continuous integration
10	Continuous delivery (Krusche and Alperowitz 2014)
11	Software theater
12	Global project management (Li et al. 2016)
13	Project management antipatterns

might not be able to objectively generalize their experience to other situations. Other variables of the course, such as an open atmosphere toward feedback, can have a positive influence on the evaluation result. If a student likes the capstone course, it does not necessarily mean that software theater was helpful. We were not able to exclude these variables in the evaluations of the case study. To alleviate these threats, we also analyzed how the students used software theater and these results support the findings of our study.

Our findings apply to a multi-customer software engineering course that was set up at our university. In other universities with different curricula and environments, it might not be possible to instantiate our course format and apply the software theater workflow easily.

7 CASE STUDY 2: INTERACTIVE LECTURE-BASED COURSE

In 2015 and 2016, we applied software theater in the university course "Software Engineering II: **P**roject **O**rganization and **M**anagement" (POM). We taught the course POM in the summer semester of 2015 with 294 students and in the summer semester of 2016 with 272 students who completed the final exam. 200 students regularly attended class and participated in exercises. Two distinct groups participated in the course: (1) bachelor students in information science, a few with experience in software engineering, and (2) master students in computer science, some with existing experience in the taught topics. The challenge of this heterogeneity was that students had different prior knowledge and completed in-class exercises at different speeds.

7.1 Course Description

The course POM has the following learning goals: participants understand the key concepts of software project management, in particular agile methodologies. They are able to deal with problems such as writing a software project management plan, initiating and managing a software project, and tailoring a software lifecycle. They are familiar with risk management, scheduling, planning, quality management, and build and release management. They can apply different techniques to solve development and management problems. Table 2 shows the schedule and the content of the lecture.

The course is based on active learning to increase student involvement and excitement with the subject being taught (Bonwell and Eison 1991). It integrates individual and team exercises into

the lectures (Krusche et al. 2017). Students participate in a team project with five team members, a simplified version of the projects in our capstone course (cf. Section 6). The goal of the project is to experience and apply the learned concepts in a more realistic environment. The instructor played the role of the customer and provided three short problem statements about the development of mobile applications. The teams had to choose one of the problem statements and one mobile app development environment: Android, iOS, or Xamarin.

Students formed teams with five team members on their own in the first lecture, following requirements to create balanced teams with respect to experience, nationality, and gender. In 2015, the students formed 58 teams, and in 2016, the students formed 51 teams. Software engineering is a collaborative activity (Whitehead 2007); therefore, team work is an important development skill students have to learn in the course. The teams used Rugby (Krusche et al. 2014) as an agile and continuous process model with an initial warm-up phase and five development sprints of 2 weeks each.

While individual in-class exercises are described in detail so that students can follow step by step, students receive a more vague description of the exercises in their team project that deliberately misses detailed instructions. The teams have to bring in their own ideas on how to solve the team project and apply the techniques they learned in individual exercises earlier in the course. For instance, we show them how to create low-fidelity prototypes with the tool Balsamiq in a detailed step-by-step tutorial in the individual in-class exercise. The corresponding exercise in the team project intentionally only states that the team should create prototypes: they can then choose their own preferred tool and tailor the prototype creation to their specific problem statement.

We introduced software theater in the 11th week of the course. In the class, we introduced students to the workflow and they applied the first steps directly in class within the context of their team project. They wrote the initial version of the demo scenario and the screenplay. As homework, they refined these artifacts, filmed the demo using their smartphones, and then uploaded a recording to the learning management system so that instructors and teaching assistants could review it.

7.2 Evaluation Design

We investigated the students' improvements in the exercises using an optional online questionnaire. We also asked them about their opinion on the exercise concept. It included questions about personal data, the participation in individual exercises, and application of techniques in the team project. We wanted to know if students improved their skills in software theater and if they felt confident to apply software theater in their next team project. Students could also comment on how the course can be improved.

We conducted the survey in July 2016 and gave the students of the 2016 course 2 weeks to complete it. Personalized tokens allowed the 272 students who completed the final exam of the course to participate exactly once in the anonymous survey.[13] We received 190 responses, which amounts to a response rate of 70%. In addition, we evaluated the use of software theater in 2015 and 2016 and analyzed the software theater artifacts of all teams who participated in the exercise. We counted how many teams delivered certain artifacts and how the teams scored in the exercise.

7.3 Evaluation Results

We analyzed the results of the online questionnaire and the uploaded artifacts of all teams who participated in the exercises. In the online questionnaire, 48% of the survey respondents stated

[13]The open source survey tool LimeSurvey (http://www.limesurvey.org) guarantees that the answers are anonymous by strictly separating token and answer tables in the database.

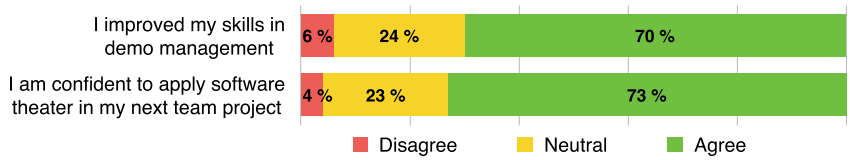


Fig. 21. Survey responses: disagreement (red) and agreement (green) with perceived improvements and confidence in software theater on a 3-point Likert scale.

Table 3. Number of Software Theater Artifacts Uploaded by the Participating Teams in the Lecture-based Course

Year	Teams	Formalized scenario	Screenplay	UML component diagram	UML class diagram	Demo backlog	Video	Feedback
2015	58	45 (78%)	39 (67%)	18 (31%)	24 (41%)	31 (53%)	20 (34%)	-[14]
2016	51	44 (86%)	43 (84%)	16 (31%)	23 (45%)	27 (53%)	20 (39%)	12 (24%)

that they participated in the in-class exercise about software theater **and** used it in their team project. Figure 21 shows that from these exercise participants, 70% agree that they were able to improve their skills in demo management (including software theater) and 73% agree that they were confident to apply software theater in their next team project.

In the statements about the exercise concept, students stated that software theater integrates creativity, interactivity, and fun into a lecture-based course, which is not common to most other, usually more static, lectures. The creation of a software theater demo improved their personal learning experience about software development and project management.

We evaluated how many teams participated in the software theater exercise and created the artifacts of the software theater workflow in both course instances in 2015 and 2016. The results are shown in Table 3.

The numbers in both courses in 2015 and 2016 were similar with a slightly higher participation in 2016. More than 80 teams participated in the software theater exercise: 78% of the teams in 2015 and 86% of the teams in 2016 created a formalized scenario; 67% (2015) and 84% (2016) created a screenplay. 53% of the teams created a demo backlog. Less than half of the teams created and uploaded UML diagrams. In both years, more teams created UML class diagrams than UML component diagrams. In total, 40 teams (34% in 2015 and 39% in 2015) performed the software theater demo, recorded a video, and uploaded it; 12 teams (24%) obtained feedback from other students in 2016. In 2015, we did not ask the students to collect feedback. Eleven teams (22%) in 2015 and seven teams (14%) in 2016 were able to complete all artifacts. We discuss these findings in Section 8.

7.4 Threats to Validity

Most of the limitations mentioned in Section 6.4 also apply for the lecture-based course, because we used similar evaluation techniques. We addressed social desirability bias by collecting the responses anonymously and by preventing multiple responses from the same student. Other variables of the course, such as the high rate of interactivity or an open atmosphere toward feedback, can have a positive influence on the evaluation result. If a student likes the lecture-based course, it does not necessarily mean that software theater was helpful. We were not able to control these

[14]We did not ask the teams in 2015 to collect feedback.

Table 4. Findings of the Evaluations in Relation to the Stated Hypotheses

ID	Hypothesis	Result
H1	**Increased motivation**: software theater increases the motivation of students to prepare a demonstration already early in the project.	Supported
H2	**Higher creativity**: demonstrations using software theater are more creative than normal demonstrations (without software theater).	Supported

variables in the evaluations of the case study. However, the evaluation results indicate that software theater can also be taught successfully in a lecture-based course.

8 DISCUSSION

In this section, we discuss the results of the two case studies with regard to our hypotheses mentioned in Section 1. We also provide best practices for other instructors who want to teach software theater.

8.1 Findings

Our case studies revealed that software theater can be taught to students in different types of courses. We found anecdotal evidence that software theater increases fun, motivation, and creativity in education, and is a technique to make software demos more relatable. Table 4 summarizes that our findings support the two proposed hypotheses: software theater increases the motivation of students to prepare a demonstration already early in the project (H1) and demonstrations using software theater are more creative than demonstrations without the technique (H2).

The hypothesis H1 is supported by our measures, both in the capstone course and the lecture-based course. While we encourage live demonstration in the design review of the capstone course and required it for the client acceptance test, we did not demand the usage of software theater for the demo. Nevertheless, 9 out of 11 teams used it to demonstrate a very early software prototype at the first event, and all teams used it in their final presentation. Although most teams could only show a rough first version of their system, their demo took up almost 25% of their maximum presentation time of only 10 minutes, and 24% of survey respondents spent 10 hours or more preparing it. This time investment shows recognition for the importance of a live demonstration using software theater at an early stage. However, as we do not have a control group that never learned about software theater, a further case study would be needed to fully examine this hypothesis.

In our lecture-based course, more than 70% of the students who applied software theater reported in the online questionnaire that they improved their skills in demo management and they are confident to apply software theater in their next team project. The number of participating students in the software theater exercise was relatively high, considering that it was one of the last exercises in the semester and required high effort. In both lecture-based courses in 2015 and 2016, more than a third of all teams performed the software theater and uploaded a video of their performance.

Hypothesis H2—software theater makes demonstrations more creative—is supported both by the survey and our observations during the case studies. 87% of respondents think that a demonstration with software theater is more creative than without, and the majority of them would employ the approach in future projects. We also observed a wide variety of creative techniques used in the demonstrations, such as the explanation of complex technological concepts with a narrator, through the theater play itself, and the use of metaphors from other domains to illustrate

the requirements of a system. Some teams also interweaved their demo with the presentation to support their arguments with a representation of their system in the usage context. While we do not have sufficient empirical data to back up this hypothesis, discussions with coaches, developers, and clients led us to the conclusions that software theater is a creative technique to demonstrate software systems, including early prototypes.

In addition, we observed that software theater demos help us in finding new customers for the capstone courses in the following semesters. We show potential customers the most creative parts and provide examples of what they can expect from the students. Customers can use the videos in their company to promote the collaboration with the university. The videos convince the industry partners about the application-oriented teaching and research at our department.

Another interesting aspect is that students share the software theater videos with family and friends to explain to them in an easy way what they achieved in their field of study. The students report that their relatives and friends without information technology background also understand how the developed software systems can help to improve specific problems. Software theater then generates a shared understanding between people with and without an information technology background and has a social relevance to facilitate digitalization and innovation. However, we do not have empirical evidence about this aspect yet.

Developers and team coaches agreed that software theater facilitates understanding of other student projects and increases confidence about their own projects in front of external parties. However, the opinions were divided about improved understanding concerning team-internal stakeholders, especially in communication with the client. We suspect that this is due to the setup of the capstone course: clients specify the project requirements in a problem statement and are usually strongly involved in the project: they regularly attend team meetings and review executable prototypes. With an already good knowledge of the requirements and up-to-date information about the current project state, a demonstration with software theater becomes less necessary because the added usage context does not provide further value. Thus, it is logical that teams do not prepare a demonstration using software theater for typical client meetings, but resort to a normal demonstration, which requires less preparation. In order to explore the improvement of understanding further, more data is needed on stakeholders who are less involved in the project, e.g., clients from other departments of the organization.

If a team chose to use software theater for their demo, we did not require them to go through every step of the workflow, but merely provided them with examples for each step and gave them the freedom to deviate. A majority of teams produced a screenplay for both events of the capstone course and the proportion of teams to produce a formalized demo scenario and a demo backlog increased between design review and client acceptance test. This implies that the students realized the usefulness of these artifacts for the software theater workflow and created them by choice because they helped them in their demo preparation. Over 70% of respondents indicated that they mocked parts of their system for the demonstration and the reasons stated for using mocks correspond to the recommendations we communicated during the lectures. Therefore, we can conclude that our teaching approach combining lectures, team coaching, and regular feedback in team sessions and dry runs leads to an improved understanding of the software theater workflow.

8.2 Best Practices

Having taught software theater in capstone courses since 2012 and in a lecture-based course since 2015, we have iteratively refined both our workflow and the approach of teaching it. We want to share the most relevant learnings for educators who want to adopt the method in their own courses.

Communicate knowledge multiple times: Our courses run over 3 months and students have to absorb a large amount of information in a relatively short amount of time. We aim to communicate the most important topics several times and through varying channels to ensure that everyone understands it correctly. In our capstone course, for instance, we give a general overview over software theater in a lecture, in which we combine theory with hands-on examples. The team coaches reiterate the workflow again, concentrating on the aspects that are most important for their specific team. We also revisit parts of the workflow in the cross-project teams and give feedback to each team how they apply software theater in their dry run before the real presentation.

Emphasize iteration: Students need multiple iterations to make sure that their demonstration represents their system and integrates nicely with the rest of their story in their presentation. Therefore, we introduce software theater at least three weeks before they first need to perform a demonstration, giving the teams enough time to prepare. We encourage them to go through multiple iterations of their screenplay. In our capstone course, we also hold a dry run in the week before each main event to give feedback to demo and presentation. To reduce the workload of the instructors, the team coaches help in the preparations and give feedback multiple times. The teams also help each other by commenting on others' presentations and demonstrations.

When it comes to the dry runs for the design review, our first event in the capstone course, it is common that teams receive challenging feedback from the instructors leading to fundamental changes in their demo. Our experience shows that the dry runs for the client acceptance test, our second event, require less input from instructor side. We believe that our course participants learn most about software theater by experiencing it in the first presentation. Therefore, we would recommend to give students at least two opportunities to perform a live demonstration during a course and to introduce mechanisms of peer feedback to keep instructor workload manageable.

Encourage creativity: It is helpful to show several examples of the application of software theater and to include cases in which a technical detail is explained (cf. Section 5.2), or where a team uses concepts from other fields to explain an abstract concept (cf. Section 5.3). We found that this not only shows students what is possible, but also motivates them to get creative and imagine how software theater would benefit the demonstration of their own system.

Stress the importance of preparation: In a demonstration, a lot of things can go wrong. We make sure to show common points of failure (e.g., a failed network connection) and to stress the importance of having a fallback. Most of the teams decide to mock parts of their system and prepare backup slides showing a video recording of the demo in case something goes wrong.

Document learnings: According to our experience, delivering a demo with software theater leads to an improved understanding of the technique itself. We encourage our student teams to perform a retrospective after their first presentation, discussing and documenting how it went and what they would like to improve. Changes in the adoption of software theater between the design review and the client acceptance test (e.g., more preparation time, more integrated demonstrations) shown in Section 6.3 are based on these learnings.

9 RELATED WORK

In the beginning of the 1990s, Brenda Laurel presented the idea that usability alone would not be enough in the development of successful computer systems (Laurel 1993). She described a theory of interaction, combining her experience in human computer interaction with knowledge of theater. "The real issue," she claims, is "How can people participate as agents within representational contexts? Actors know a lot about that, and so do children playing make-believe." Her hypothesis is that the vocabulary of traditional theater is similar to the vocabulary in human computer interaction. Laurel's work is based on Aristotle's poetics for dramatic theory, and explains how concepts such as catharsis, engagement, and agency can be applied in digital contexts.

The users' enjoyment must be a design consideration, which requires an awareness of dramatic theory and technique. She claims that effective design of interactive systems, like effective drama, must engage the users directly in an experience involving both thought and emotion: "Even in task-oriented applications, there is more to the experience than getting something done in the real world, and this is the heart of the dramatic theory of human-computer interaction." Our approach of software theater is based on Laurel's ideas and introduces theater into software engineering education to teach students the importance of interaction and emotion.

Mahaux and Maiden (2008) and Mahaux et al. (2010) proposed improvisational theater to support team-based innovation in the requirements engineering process. The commonality of their improvisational theater and software theater is that they both employ the form of theater as an effort to improve stakeholder communication and increase mutual understanding. But they differ in several aspects.

First, the purpose of improvisational theater is to generate creative ideas in the requirements engineering process, while the purpose of software theater is to demonstrate and evaluate design ideas for innovative software projects in education. Second, improvisational theater, as its name suggests, takes advantage of unplanned improvisational performance to stimulate the creativity of team members, while software theater emphasizes a pre-defined screenplay to set a framework for the demonstration. Third, software theater presents not only the applicability of user requirements, but also the feasibility of system requirements such as architecture design and hardware performance. Software theater is used in combination with specific software process and prototyping techniques, in our case the Tornado model. The Class, Responsibilities, and Collaborators (CRC) cards were introduced by XP practitioners Kent Beck and Ward Cunningham in 1989 as a teaching tool for object-oriented programming (Beck and Cunningham 1989). Instructors have used CRC cards for role-playing: each student plays the role of a specific class. They interact with other students representing collaborating classes to understand their responsibilities. While software theater is based on actors taking one or more roles, it is not designed to teach role-oriented programming (Kristensen 1995). Its purpose is to teach the demonstration of a system developed during a course. Rice and his colleagues used forum theater (a kind of interactive theater) to elicit requirements in the development of new technologies (Rice et al. 2007). They discovered the usefulness of storytelling through theater and video in promoting user involvement because it is easier for users (e.g., elderly people) that are not familiar with the state-of-the-art technologies to understand the system. They conclude that the technique "can increase designer empathy towards end users."

10 CONCLUSION

Software engineering is creative, imaginative, and interactive. However, simple demonstrations of the functional and static aspects of software do not provide the real-world usage context that is integral to understand the software requirements. We therefore advocate for a more dynamic way of presenting software prototypes. In this article, we described software theater, a combination of agile methods, scenario-based design, and theatrical aspects, as a way to demonstrate visionary scenarios in a more relatable and creative way, even if the software is not yet fully realized. The creation of the demo is based on the Tornado model and allows the validation of requirements, design, and technology decisions. We explained the software theater workflow in detail, including all steps from the customer's visionary scenario to the developers' actual demonstration.

We illustrated the artifacts of one example and showed three common software theater patterns. We taught software theater in a capstone course with industrial customers and in an interactive lecture-based course, described both courses as case studies, and explained our teaching approach. In empirical evaluations in these two courses, we found that software theater motivates

the students to prepare demonstrations even early in the project. Students acquire the ability to apply software theater within one semester and perceive software theater demonstrations as more creative, more memorable, more dynamic, and more engaging than normal techniques. While we found first anecdotal evidence that software theater improves the understanding of external stakeholders about the developed system, we further need to investigate this aspect.

Rise and his colleagues pointed out that theater "may not be equally well suited for very early requirements gathering, or later, more specific prototype evaluation" (Rice et al. 2007). We hypothesize that software theater can also be used as requirements elicitation technique and want to evaluate this in the future. We started to use it in other course formats such as short summer schools (1–2 weeks), but have not yet collected data about its usage and benefits in these shorter courses. Another research area is the further formalization of the described software theater patterns as well as the identification of further patterns. We want to use a common scheme such as the one proposed by the Gang of Four (Gamma et al. 1995) to describe different educational software theater patterns including their motivation, applicability, context, forces, and consequences.

ACKNOWLEDGMENTS

We want to thank all students of the capstone courses and the interactive lecture-based courses. We also thank our colleagues for management support, filming, and technical support and our industry partners for providing interesting problem statements.

REFERENCES

Russell Abbott. 1983. Program design by informal English descriptions. *Commun. ACM* 26, 11 (1983), 882–894.

Christopher Alexander, Sara Ishikawa, Murray Silverstein, Joaquim Romaguera i Ramió, Max Jacobson, and Ingrid Fiksdahl-King. 1977. *A Pattern Language.* Gustavo Gili.

Lukas Alperowitz. 2017. *ProCeeD—A Framework for Continuous Prototyping.* Ph.D. Dissertation. Technical University Munich, Germany.

Larry Apfelbaum and John Doyle. 1997. Model based testing. In *Proceedings of the Software Quality Week Conference.* 296–300.

Jonathan Arnowitz, Michael Arent, and Nevin Berger. 2010. *Effective Prototyping for Software Makers.* Morgan Kaufmann.

Victor Basili. 1996. The role of experimentation in software engineering: Past, current, and future. In *Proceedings of the 18th International Conference on Software Engineering.* IEEE, 442–449.

K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, and others. 2001. Manifesto for agile software development. *The Agile Alliance* (2001). http://agilemanifesto.org.

Kent Beck and Ward Cunningham. 1989. A laboratory for teaching object oriented thinking. In *Sigplan Notices*, Vol. 24. ACM, 1–6.

Barry Boehm. 2000. Requirements that handle IKIWISI, COTS, and rapid change. *Computer* 33, 7 (2000), 99–102.

Charles Bonwell and James Eison. 1991. *Active Learning: Creating Excitement in the Classroom.* ASHE-ERIC Higher Education Reports.

Bernd Bruegge and Allen Dutoit. 2009. *Object Oriented Software Engineering Using UML, Patterns, and Java* (3rd ed.). Prentice Hall.

Bernd Bruegge, Stephan Krusche, and Lukas Alperowitz. 2015. Software engineering project courses with industrial clients. *ACM Transactions on Computing Education* 15, 4 (2015), 17:1–17:31.

Bernd Bruegge, Stephan Krusche, and Martin Wagner. 2012. Teaching tornado: From communication models to releases. In *Proceedings of the 8th Edition of the Educators' Symposium.* ACM, 5–12.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. A system of patterns: Pattern-oriented software architecture. Addison Wesley.

Lan Cao and Balasubramaniam Ramesh. 2008. Agile requirements engineering practices: An empirical study. *IEEE Software* 25, 1 (2008), 60–67.

John Carroll. 1995. *Scenario-based Design: Envisioning Work and Technology in System Development.* John Wiley & Sons, Inc.

John Carroll. 2000. *Making Use: Scenario-based Design of Human-computer Interactions.* MIT Press.

Dora Dzvonyar, Stephan Krusche, and Lukas Alperowitz. 2014. Real projects with informal models. In *Proceedings of the 10th Edition of the Educators' Symposium.*

Christiane Floyd. 1984. A systematic look at prototyping. In *Approaches to Prototyping.* Springer, 1–18.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley.

Ron Garland. 1991. The mid-point on a rating scale: Is it desirable. *Marketing Bulletin* 2, 1 (1991), 66–70.

Orit Hazzan. 2002. The reflective practitioner perspective in software engineering education. *Journal of Systems and Software* 63, 3 (2002), 161–171.

Matthias Jarke, Ralf Klamma, Klaus Pohl, and Ernst Sikora. 2010. Requirements engineering in complex domains. *Graph Transformations and Model-driven Engineering* (2010), 602–620. http://dblp.uni-trier.de/rec/bibtex/conf/birthday/JarkeKPS10.

Matthias Jarke and Klaus Pohl. 1993. Establishing visions in context: Towards a model of requirements processes. In *Proceedings of the 14th ICIS*.

Stephen Kline and Nathan Rosenberg. 1986. An overview of innovation. In *The Positive Sum Strategy: Harnessing Technology for Economic Growth*. National Academy Press, 275–305.

Bent Bruun Kristensen. 1995. Object-oriented modelling with roles. In *OOIS*. 57–71.

Stephan Krusche. 2016. *Rugby - A Process Model for Continuous Software Engineering*. Ph.D. Dissertation. Technical University Munich, Germany.

Stephan Krusche and Lukas Alperowitz. 2014. Introduction of continuous delivery in multi-customer project courses. In *Proceedings of the 36th International Conference on Software Engineering*. IEEE, 335–343.

Stephan Krusche, Lukas Alperowitz, Bernd Bruegge, and Martin Wagner. 2014. Rugby: An agile process model based on continuous delivery. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. ACM, 42–50.

Stephan Krusche, Mjellma Berisha, and Bernd Bruegge. 2016. Teaching code review management using branch based workflows. In *Companion Proceedings of the 38th International Conference on Software Engineering*. IEEE, 384–393.

Stephan Krusche, Andreas Seitz, Jürgen Börstler, and Bernd Bruegge. 2017. Interactive learning: Increasing student participation through shorter exercise cycles. In *Proceedings of the 19th Australasian Computing Education Conference*. ACM, 17–26.

Brenda Laurel. 1993. *Computers as Theatre* (2nd ed.). Addison-Wesley.

Meir Lehman and Laszlo Belady. 1985. *Program Evolution: Processes of Software Change*. Academic Press.

Yang Li, Stephan Krusche, Christian Lescher, and Bernd Bruegge. 2016. Teaching global software engineering by simulating a global project in the classroom. In *Proceedings of the 47th SIGCSE*. ACM, 187–192.

Tim Mackinnon, Steve Freeman, and Philip Craig. 2000. Endo-testing: Unit testing with mock objects. In *Extreme Programming Examined*. ACM, 287–301.

Martin Mahaux, Patrick Heymans, and Neil Maiden. 2010. Making it all up: Getting in on the act to improvise creative requirements. In *Proceedings of the 18th International Requirements Engineering Conference*. IEEE, 375–376.

Martin Mahaux and Neil Maiden. 2008. Theater improvisers know the requirements game. *IEEE Software* 25, 5 (2008), 68.

Robert Martin. 1996. The dependency inversion principle. *C++ Report* 8, 6 (1996), 61–66.

Jakob Nielsen. 1994. *Usability Engineering*. Elsevier.

Donald Norman. 2013. *The Design of Everyday Things (Revised and Expanded Edition)*. Basic Books.

Donald Norman and Stephen Draper. 1986. User centered system design: New perspectives on human-computer interaction. CRC Press.

Tom Nurkkala and Stefan Brandle. 2011. Software studio: Teaching professional software engineering. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. ACM, 153–158.

Klaus Pohl. 2010. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer.

Roger Pressman. 2009. *Software Engineering: A Practitioner's Approach*. McGraw-Hill.

Mark Rice, Alan Newell, and Maggie Morgan. 2007. Forum theatre as a requirements gathering methodology in the design of a home telecommunication system for older adults. *Behaviour & Information Technology* 26, 4 (2007), 323–331.

Colette Rolland, C. Ben Achour, Corine Cauvet, Jolita Ralyté, Alistair Sutcliffe, Neil Maiden, Matthias Jarke, Peter Haumer, Klaus Pohl, Eric Dubois, and P. Heymans. 1998. A proposal for a scenario classification framework. *Requirements Engineering* 3, 1 (1998), 23–47.

Ken Schwaber and Mike Beedle. 2002. *Agile Software Development with Scrum*. Prentice Hall.

Helen Sharp. 2003. *Interaction Design*. John Wiley & Sons.

Mary Shaw, Bernd Bruegge, and John Cheng. 1991. A software engineering project course with a real client. Carnegie Mellon University Pittsburgh Software Engineering Institute.

Mary Shaw, Jim Herbsleb, Ipek Ozkaya, and Dave Root. 2006. Deciding what to design: Closing a gap in software engineering education. 28–58.

Alistair Sutcliffe. 1997. A technique combination approach to requirements engineering. In *Proceedings of the 3rd International Symposium on Requirements Engineering*. IEEE.

James Tomayko. 1987. *Teaching a Project-intensive Introduction to Software Engineering*. Technical Report CMU/SEI-87-TR-20. DTIC Document.

Klaus Weidenhaupt, Klaus Pohl, Matthias Jarke, and Peter Haumer. 1998. Scenarios in system development: Current practice. *IEEE Software* 15, 2 (1998), 34–45.

Jim Whitehead. 2007. Collaboration in software engineering: A roadmap. *FOSE* 7, 214–225.

Han Xu, Oliver Creighton, Naoufel Boulila, and Bernd Bruegge. 2013. User model and system model: The yin and yang in user-centered software development. In *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*. ACM, 91–100.

Han Xu, Stephan Krusche, and Bernd Bruegge. 2015. Using software theater for the demonstration of innovative ubiquitous applications. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 894–897.

Received April 2017; accepted August 2017

8.6 Increasing the Interactivity in Software Engineering MOOCs - A Case Study

This conference paper describes the application of interactive learning and Artemis in a software engineering online course, an important milestone in the habilitation. It shows that instructors can also integrate the teaching philosophy into massive open online courses (MOOC) while keeping the course interactive and maintaining an interactive atmosphere with social aspects. The paper defines an interactivity model and describes best practices for online education. It shows how the different aspects of online courses contribute to learning success and highlights the helpfulness of feedback and chat-based communication. The conference nominated the manuscript for the best paper award.

Authors	**S. Krusche** and A. Seitz
Conference	31st Conference on Software Engineering Education and Training (52nd Hawaii International Conference on System Sciences)
Publisher	ScholarSpace
Pages	10
Type	Conference: Full Research Paper
Review	Peer Reviewed (5 Reviewers)
Year	2019
Citation	[KS19]
DOI	https://doi.org/10.24251/HICSS.2019.915

Increasing the Interactivity in Software Engineering MOOCs - A Case Study

Stephan Krusche
Technische Universität München
krusche@in.tum.de

Andreas Seitz
Technische Universität München
seitz@in.tum.de

Abstract

MOOCs differ from traditional university courses: instructors do not know the learners who have a diverse background and cannot talk to them in person due to the worldwide distribution. This has a decisive influence on the interactivity of teaching and the learning success in online courses. While typical online exercises such as multiple choice quizzes are interactive, they only stimulate basic cognitive skills and do not reflect software engineering working practices such as programming or testing. However, the application of knowledge in practical and realistic exercises is especially important in software engineering education.

In this paper, we present an approach to increase the interactivity in software engineering MOOCs. Our interactive learning approach focuses on a variety of practical and realistic exercises, such as analyzing, designing, modeling, programming, testing, and delivering software stimulating all cognitive skills. Semi-automatic feedback provides guidance and allows reflection on the learned theory. We applied this approach in the MOOC software engineering essentials SEECx on the edX platform. Since the beginning of the course, more than 15,000 learners from more than 160 countries have enrolled. We describe the design of the course and explain how its interactivity affects the learning success.

1. Introduction

Massive open online courses (MOOCs) offer new possibilities. Learners can participate in courses of interest with higher flexibility and are not bound to schedules, locations, or university costs. Universities can reach larger audiences outside of organizational boundaries [1]. With the help of the internet, education can be brought to countries with lower social and educational standards. MOOCs are becoming increasingly popular and more and more universities offer them in professional programs and micro masters.

However, it is not sufficient to replicate standard university lecture courses to design a MOOC [2]. It is not possible to promote an active learning process in online courses, only by broadcasting video recordings of lectures [1]. This poses the risk of students becoming passive and unmotivated. While embedding discussions and offering multiple choice quizzes can increase the interactivity, they typically do not stimulate higher cognitive skills [3, 4]. Most activities in existing online courses focus on lower cognitive skills, testing only the degree of understanding of the main concepts and forcing the learner to face recurrent mistakes [4].

Due to these limitations, simple software engineering MOOCs do not yet support the acquisition of higher cognitive skills such as applying, analyzing, evaluating and creating. However, software engineering is an activity that requires collaboration and practical application of knowledge [5, 6], in particular interaction and collaboration [7]. Video lectures, simple quizzes and reading material are not sufficient [8] because they do not reflect working practices such as programming, modeling and testing. The creation of new software depends on higher cognitive skills including application of knowledge, analysis, evaluation and creation.

Active learning engages course participants in the learning process by involving them into learning activities, e.g. into problem solving. It is used in more and more university courses [9], positively influences knowledge transfer and learners' performance, and leads to an improved learning experience [10]. However, instructors need to guide learners in these activities to facilitate learning and prevent misconceptions [11]. With several thousand learners in MOOCs, it is challenging to guide all learners.

In summary, MOOCs face the following three problems:

(P1) Lecture recordings are too static
(P2) Quizzes only stimulate basic cognitive skills
(P3) Guidance is challenging to scale

We present an interactive learning approach for software engineering MOOCs that addresses these

URI: https://hdl.handle.net/10125/60197
ISBN: 978-0-9981331-2-6

problems by using a variety of practical exercises. These interactive exercises go beyond multiple choice questions and stimulate cognitive skill acquisition on all levels. They provide individual guidance to learners through feedback and scale to a large number of students. The focus of the paper is **not** a discussion whether MOOCs can or will replace university courses.

The remainder of the paper is structured as follows: Section 2 describes active learning, Bloom's taxonomy for cognitive skills and existing definitions for interactivity as the foundations of this paper. Section 3 shows our interactive learning approach with different exercise types that stimulate all cognitive skills. In Section 4, we present a case study, in which we applied this approach in a software engineering online course. Section 5 discusses our learnings in this course and describes best practices for other instructors who want to adopt our approach. Section 6 presents related work and Section 7 concludes the paper.

2. Foundations

"MOOCs mostly replicated the standard lecture, an uninspiring teaching style but one with which the computer scientists were most familiar" [2].

2.1. Bloom's Taxonomy

Bloom developed a framework to classify expectations of what students should learn as the result of an instruction [12]. It serves as common language about learning goals. An example would be: "learners are able to describe the waterfall model". Bloom classified six major categories of cognitive processes ordered by their complexity from lowest to highest: knowledge, comprehension, application, analysis, synthesis and evaluation.

Constructive alignment proposes to align learning goals with activities and assessment. It was introduced by John Biggs [13] and is derived from constructivism and curriculum theory [14]. Biggs refers to the basic idea of constructivism that learners construct their own learning through learning activities, instead of passively receiving knowledge from the instructor. All components in the learning system - the learning goals, the learning activities, and the assessment tasks - are aligned to each other.

2.2. Active Learning

Active learning led to improved learning experiences on different cognitive skill levels of Bloom's taxonomy in university courses. It emphasizes on developing skills through active participation and engagement in activities. It moves away from teacher-centered approaches, where teachers instruct and learners listen passively, to a more learner-centered approach, where learners play an active role. Bonwell and Eison define active learning as "anything that involves students in doing things and thinking about the things they are doing" [15]. It requires learners to regularly assess their own problem-solving skills and their understanding of the taught concepts [16]. Brophy and Good identify four main premises of active learning [17]:

1. Learners construct their own meanings
2. New learning builds on prior knowledge
3. Learning is enhanced by social interaction
4. Learning develops through 'authentic' tasks

Prince [10] and Michael [16] found support for all forms of examined active learning in their studies. They concluded that active learning improves learning outcomes compared to passive learning approaches.

2.3. Interactivity

When an instructor says 'I am trying to make my classes more interactive', the meaning of interactive seems clearly intuitive, however an agreed-upon definition of interactivity is hard to find. The term is used in the context of various fields, such as communication, advertising, websites, internet and education to name a few [18]. Since Rafaeli's statement "Interactivity is an underdefined concept" [19], a number of attempts have been made to define the concept of interactivity in its different contexts leading to the inconsistent use of the term [20].

The term interactivity is rooted in the term interaction. The Cambridge dictionary defines interaction as "an occasion when two or more people or things communicate with or react to each other". Steffensen differentiates between interaction and interactivity: "[...] interaction captures a relation of dependence between separable systems, interactivity explores their bidirectional coupling" [21].

Jones and Gerard propose that all social interaction is goal-oriented [22]. They distinguish four different types of interactions according to their influence on the interaction partners:

1. **Pseudo interaction:** A sequence of actions that follow predefined patterns. The actions of an involved participant are not intended to be interpreted by the other participant.
2. **Asymmetrical interaction:** One participant follows his or her intentions while another party reacts complementarily to the previous actions.

3. **Reactive interaction:** The involved parties do not interpret the intentions of the actions of the other party and react in an isolated form.
4. **Interdependent symmetrical interaction:** Aligning one's action to the own intentions while considering the intentions of the others in a reciprocal fashion.

Similarly, Rafaeli argues that interactivity is best defined by considering the degree of responsiveness [19]. He recognizes three levels of communication. Two-way (non-interactive), reactive (quasi-interactive) and fully interactive communication. For an interaction to be classified as two-way communication, messages must flow bilaterally. If the messages cohere with previous messages, the interaction is at least reactive or quasi-interactive. The third level, full interactivity, adds a reference to the content, nature or presence of earlier references. Rafaeli defines interactivity as "an expression of the extent that in a given series of communication exchanges, any third (or later) transmission (or message) is related to the degree to which previous exchanges referred to even earlier transmissions." Domagk, Schwartz and Plass [23] and Johnson et al. [24] identified two fundamental conditions common in interactivity research: (1) At least two participants interact with each other. (2) Actions of these participants are reciprocal[1] and responsive[2].

Yacci examined interactivity in the context of distance learning and computer-based teaching and identified major interactivity attributes [25]. Interactivity is a message loop, whose messages must be mutually coherent. Instructional interactivity occurs from the learner's perspective. Its outputs are content learning and affective benefits.

3. Interactive Learning Approach

Interactive learning combines theory and practice into interactive classes with multiple, small iterations of theory, example, exercise, solution and reflection [26]. It is based on active, computer based and experiential learning [27] and focuses on immediate feedback to provide guidance and improve the learning experience in large university classes. Hands-on activities in class increase motivation and engagement and allow continuous assessment.

Instructors teach and exercise small chunks of knowledge in short cycles. Learners reflect and increase their knowledge incrementally. This approach expects active participation and the use of computers. Instructors provide guidance to prevent misconceptions and to facilitate the learning process. Considering the existing definitions of interactivity in literature, we define interactivity in terms of MOOCs as follows:

> **Interactivity** means a reciprocal and responsive communication which is addressed in a context-sensitive way by the learning management system as a whole, so that learners can construct meaningful knowledge increments.

This definition focuses on automatic feedback using a learning management system, but also allows instructors, teaching assistants (TAs) and peer learners to respond to communication initiated by learners. The purpose of semi-automatic, context-sensitive feedback is guidance and reflection to prevent misconceptions. Communication in MOOCs can be initiated by instructors, e.g. when motivating students using course announcements or emails. It can also be initiated by learners when they ask questions or face problems.

Figure 1 shows the idea of continuous interactive learning[3] that we adapted from Scrum [28] and experiential learning [27]. The course syllabus consists of high-level learning goals that are typically structured into lectures giving them meaningful boundaries in the learning activities. Each lecture consists of more detailed learning goals. The instructor teaches each learning goal in a learning sprint, a cycle that starts with theory and examples. Learners then work on an exercise and receive immediate feedback building a second small cycle that allows them to iteratively improve their solution to the exercise. After the exercise, the instructor stimulates reflection so that students relate their experience in the exercise with the taught theory. This closes the cycle of the learning sprint and leads to a learning gain, which we call knowledge increment, with respect to the taught learning goal.

Examples and exercises are important elements and play a central role in the early phases of cognitive skill acquisition [30]. Carefully developed and integrated examples increase the learning outcome [31, 32]. Dynamic exercises with context-sensitive feedback solve P1 and enable a richer learning experience. Continuous interactive learning focuses on the application of knowledge in a variety of exercise types, e.g. programming and modeling exercises with instant feedback. This supports the cognitive skill acquisition [30] on all levels of Bloom's taxonomy shown in Figure 2 and solves P2. Multiple choice

[1] Reciprocal means that actions of one participant trigger responses from the other and lead to change in the first.

[2] Responsive means that actions and reactions are related and sustain the continuity of the interaction.

[3] We first integrated continuous interactive learning into a classroom course on games development [29] in 2016.

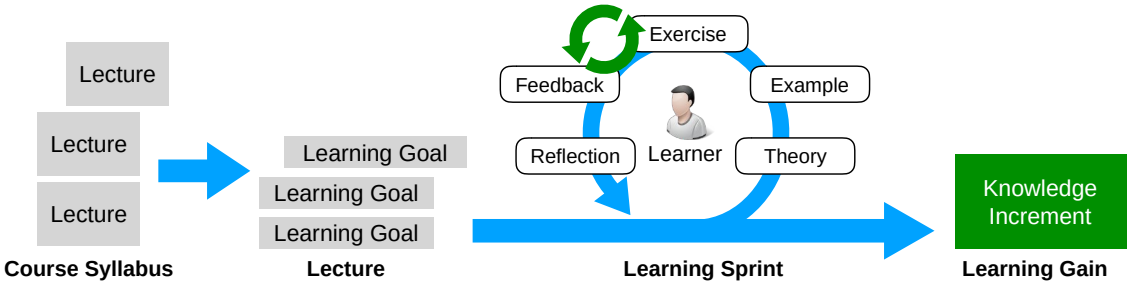


Figure 1. Continuous interactive learning embedded into a course consisting of lectures, each with a number of learning goals. Each learning goal is taught in a learning sprint through theory, example, exercise, feedback and reflection and leads to a new knowledge increment (adapted from Scrum [28] and experiential learning [27]).

quizzes focus on the first two levels, programming and modeling address the four higher and more complex levels[4]. The sample solution and the instant and context-sensitive feedback in the end of the cycle provide guidance. If feedback can be generated automatically (e.g. through test cases in programming exercises) or by other learners (e.g. through peer review in modeling exercises), it is scalable to a large number of learners and solves P3. This might require a higher effort for the creation of exercises, but reduces the effort during the conduction of the course.

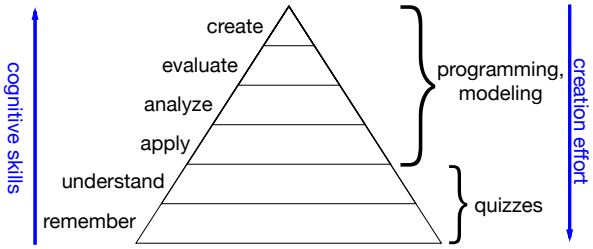


Figure 2. Mapping of exercises to cognitive skills

Depending on the type of the exercise, the learner's submission is automatically assessed or a manual review of the solution is carried out, involving other learners (peer review). The assessment leads to manual or automatic feedback which needs to be context-sensitive to be meaningful. Learners can use it to improve and submit another solution. Feedback motivates learners and allows them to reflect their learning progress.

We developed the concept of interactive instructions that visually explain the problem to be solved. Such instructions are dynamic and provide continuous and granular feedback with self-updating elements, e.g. tasks and UML diagrams with respect to the structure of the exercise. These elements respond to the interaction of learners by changing their color from red to green to indicate that the solution is correct as shown in Figure 3.

An interactive task is dynamically updated based on the learner's progress. It is associated with the assessment, e.g. a test or a peer review. An interactive task in a programming exercise is completed when all associated tests are passing. This association allows to refer the learner to the problem in the source code when the user clicks on the unfulfilled, red task. After completion, the task is displayed in green and ticked off.

An interactive diagram is dynamically created and updated based on the learner's progress. It consists of multiple elements, such as classes, attributes, or methods in a UML class diagram. A diagram element can be associated with an assessment and a source file. The implementation of a method is e.g. associated with its method name in the class diagram. Based on the test results, the color of this diagram element changes to green, if all associated tests succeed, or to red, if at least one test fails. Learners can immediately identify which parts of their exercise are correctly solved and which are still incorrect. In addition, the associated feedback includes context-sensitive information, why a test failed and refers to the theory learned in videos and handouts.

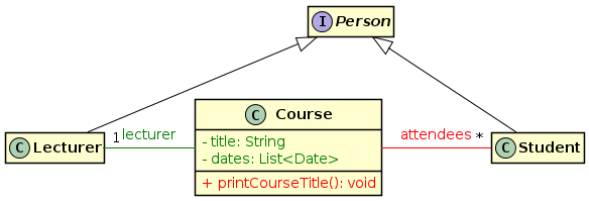


Figure 3. Interactive assignments (with UML diagrams) provide immediate feedback to learners about the correctness (red, green) of their solution

[4]While it might be possible to create multiple choice tests for higher cognitive skills, it is difficult and does not reflect software engineering working practices: software engineers do not answer multiple choice questions in their daily work when applying, analyzing, evaluating or creating something. Williams and Haladyna recommend to limit multiple choice tests to lower cognitive skills [3].

4. Case Study

We describe the application of interactive learning for the design, creation, implementation and execution of the MOOC Software Engineering Essentials (SEECx) that we offer on edX[5]. Our goal was to make the course as interactive as possible. The MOOC was launched in May 2017 as instructor-paced course over 9 weeks. We repeated the course in October 2017. Since May 2018, the course is available as self-paced course. In all three instances, 15,276 students enrolled in total until now.

It is an intensive course with interactive exercises that go beyond the learning experience of existing software engineering MOOCs. It has the following learning goals: Learners get to know methods and techniques to develop software for different domains and platforms using agile techniques in the context of change. Starting from a problem statement, we teach the participants how to analyze requirements and transform them into models using textual analysis. They model multiple representations of the system consistently, understand and identify patterns. They map models to source code, integrate it into an app and deliver this app using build and release management techniques.

4.1. Course Structure

4 instructors and 7 TAs organized the course. It includes 8 sections (comparable to lectures) covering 8 major topics: project organization and management, software configuration management, object oriented programming, requirements analysis, system design, object design, testing, build and release management. All sections are decomposed into smaller topics and consists of 3 to 5 units, each covering a concrete learning goal. The whole course includes 34 units.

A unit includes a video in which the theory of the topic is taught and an example is shown, followed by a small exercise with feedback and a summary to reflect on the learned concepts. The duration of the videos ranges from 3 to 15 minutes (mean: 8.2 min). The videos are kept short in order to enable the learners to apply the newly acquired knowledge in practice in the exercises. In addition to slide-based lecture videos, we added short clips with animations in an explanation style and real world scenes into the video to make them more entertaining and rich in variety. Such videos make the thinking process visible and support cognitive apprenticeship [33]. After each unit, there is a quiz to assess whether the learners can remember and explain the learned concepts (level 1 and 2 in Bloom's taxonomy). Learners get immediate feedback on their response and test their newly acquired knowledge. Learners can try each quiz two times in the course, so even if they failed initially, they can have another look at the video and then score the full points in the assessment. This keeps the learners motivated.

Each section also includes programming and modeling exercises which focus on higher cognitive skills. They assess if learners can apply the previously obtained knowledge, analyze a problem, evaluate different solution strategies and create new solutions to given problems (level 3 - 6 in Bloom's taxonomy).

In order to pass the course, learners have to achieve at least 60 % of all available points (400). By participating in the interactive exercises, learners can earn up to 60 % of the total points (240), 30 points for each section. At the end of the course, students can participate in a final assessment which accounts the remaining 40 % of the total points (160).

4.2. Participation

In the following, we want to show how learners participated in the first instance (instructor-paced) of the course between May and July 2017. Figure 4 shows that in the beginning, our course had 786 active learners[6] and 620 learners who scored in at least one exercise (in section 1). In the last section 8, 47 % of the learners were still active and 15 % scored in exercises. Between section 1 and 3, there was a drop of 33 % of the active learners and 68 % of the learners who scored. We attribute this to the increased complexity of the exercises. In addition, multiple instructors of other software engineering courses initially participated in our course due to advertisement on typical software engineering mailing lists such as SE World and SIGCSE. They tried out some videos and exercises, but were not interested in completing the course. Towards the end of the course, the dropout rate decreased.

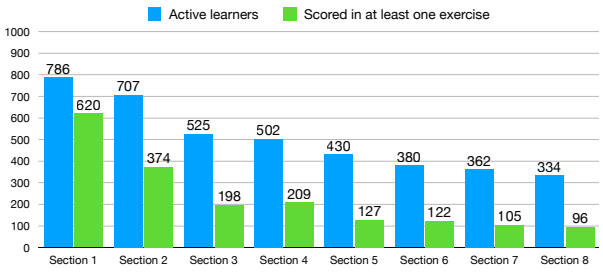


Figure 4. Number of active learners (blue) and learners who scored in at least one exercise (green) for each section in the first instructor-paced instance.

[5] www1.in.tum.de/seecx or www.edx.org/course/software-engineering-essentials

[6] Learners who visited at least one page in the course content.

4.3. Interactive Exercises

We use different types of exercises to make the course interactive and rich in variety following the learning goals. We used multiple choice, text input and drag & drop questions to support learning goals on level 1 and 2 of Bloom's taxonomy. In addition, we integrated interactive programming and modeling exercises.

We based all programming and modeling exercises on a common problem statement about the "University App", which we also used for examples in the videos. This allows the learners to recognize relationships between the topics (e.g. between the requirements and system design) and makes it easier to understand the context of the problem. In the following, we explain the different exercise types in more details.

4.3.1. Interactive Programming Exercises

We use an automated assessment system ArTEMiS for programming exercises based on version control and continuous integration [34]. Learners submit their exercise solution and receive immediate feedback through structural and behavioral tests. Learners can use this feedback to iteratively improve their solution. ArTEMiS automatically assesses the learners' submissions and provides context-sensitive feedback on their submissions.

The online editor of ArTEMiS includes assignments using interactive tasks and interactive diagrams. After each submission through the **Commit & Run Tests** button, the code of the learner is assessed. The result is shown immediately and the interactive tasks and diagrams are updated accordingly. In addition, learners can see detailed, individual feedback why their solution is wrong by clicking on the result. This helps to identify, which tasks the learners have already solved and which parts of their program does not work as expected.

4.3.2. Modeling Exercises

The ability to understand and create models is an important learning goal for software engineers. Therefore, modeling is an essential part of our course. However, it is difficult to automatically correct models because there are different correct solutions. Modeling is a creative activity and we do not want to limit the creative thinking processes of students [35]. One learning goal of the course is that participants can review models given a set of quality criteria. Therefore, we used the concept of peer reviews consisting of the following steps: (1) upload response, (2) learn to assess responses, and (3) assess peers.

Learners first create a solution to a given problem and upload it, then they review sample solutions towards a given set of criteria to learn how to assess other solutions. Finally, they assess multiple other learners' solution, so that each model is evaluated by at least three reviewers (other learners). The final score is the average of three reviews. Learners receive valuable feedback about their models and can improve their modeling skills in the future. While edX's peer review system does not allow to improve the model according to the feedback, we are working on an interactive system that allows learners to iterate on their model solution.

While peer reviews lead to additional effort for learners, they stimulate the acquisition of higher cognitive skills: by assessing other solutions, students reflect about alternative solution approaches and evaluate if they are correct with respect to the given problem statement. This is particularly helpful, but it should be used carefully to not overload the learners. We include two peer review exercises in the course, one on creating low-fidelity mockups for the university app, and another one on creating an analysis object model.

4.3.3. Project Work

In addition, we also offer project work which allows the students to experience the full software engineering process from analysis over design to implementation, testing and delivery. A second problem statements allows learners to apply and transfer their knowledge to a different problem domain. The exercises in the project work focus on the upper two cognitive skill levels in Bloom's taxonomy where learners should create and evaluate a new solution to a problem. The project work starts in the fourth section and allows the learners to evaluate how their own decisions, e.g. in the requirements analysis, influence the system design and the implementation.

Examples of project work exercises are the analysis of the problem domain, the design of the software architecture, sketches of user interfaces up to the implementation, testing and delivery of a small app. We cannot assess such exercises automatically because we do not want to limit the creativity of learners. We motivated the learners to discuss their solutions with us and each other to get feedback on their solutions and the TAs provided timely feedback. Project work is optional for learners, they can pass the course without active participation. Nonetheless, we highly recommend to participate and give the learners the opportunity to deepen their knowledge and gain practical experience.

4.4. Communication

Guàrdia describes that designing a MOOC is to "[s]et up a space to foster social interaction and frequent contact between the learners." This results in our approach using a chat for instant and direct communication instead of discussion forums to further improve interactivity between course participants and instructors. Many existing MOOCs rely on discussion forums, which are limited in interactivity. We promote the exchange with and between learners. Both instructors and TAs can be approached directly in the chat, learners can provide feedback and ask for help.

We use Slack as instant messaging service because it has a lower entry barrier than discussion forums. Learners can get in touch with each other and write direct messages to instructors and TAs in case they need help. They ask questions more easily without having to pay attention to the exact wording and phrasing. We add repeating questions to a question and answer page. In total, 754 learners regularly used Slack to get in touch and already sent 14,282 messages. The #questions, #general and #feedback channels were the most important ones. In the #questions channel, learners asked question, in #feedback, they stated how to improve the learning material. Instructors and TAs answer questions within one working day to keep the interactivity high.

Clear communication of learning goals, expectations and deadlines is important. The course description clarifies what the learners can expect and what they have to accomplish to pass the course.

4.5. Survey Results

We evaluated our approach using two surveys, an entry and an exit survey. The entry survey covered the background and motivation of learners. 83 % of the participants are male, 17 % are female. The median learner age is 28 and most learners are between 20 and 40 years old. 3 % are pupils at school, 31 % are students in university or college, 51 % are employees in a company and 15 % are unemployed or searching for a job. 51 % have already participated in another software engineering course before. 86 % have already participated in another programming course before. Regarding motivation, 27 % need the certificate of the course, 96 % are interested in the topic, 84 % need to know the concepts taught in the course for their career and 66 % assume the course is fun.

The exit survey asked about the opinion on the interactivity of the course. 67 learners took part in both surveys and allowed us to compare their existing knowledge and motivation with their results.

We asked the learners how the different components of the course contributed to their learning. Figure 5 shows that all components of the contribute. Videos play an essential role in the transfer of knowledge, as they impart the theoretical content to learners. Learners indicate that the contribution of programming exercises is higher (82 %) to their learning than in quizzes (62 %). Modeling exercises also have a high contribution with 65 %. We attribute the smaller numbers of modeling exercises to the higher complexity that peer reviews entail.

We also evaluated the helpfulness of immediate feedback in programming exercises, as well as the usefulness of Slack. 83 % of the respondents agree that feedback in programming exercises is helpful (left diagram in Figure 6). 57 % of the participants agree to prefer Slack over traditional discussion forums (right diagram in Figure 6).

These results represent first anecdotal evidence. Further studies are required to evaluate the approach. Due to the small amount of participants in the exit survey, selection bias is a threat to validity. The participants opinion might not be generalizable. Nevertheless, the first survey results show that our approach improves the learning experience: there are ways to make MOOCs more interactive and to reduce the gap to interactive classroom courses.

5. Discussion

This section discusses both the approach and the experience we have gained in developing and carrying out a software engineering MOOC. We first discuss the learnings before we derive best practices that can be useful for other MOOC instructors.

We faced a trade-off between too detailed and superficial feedback. Too detailed feedback has the risk of including the actual solution which might prevent learning. However, if the feedback is too superficial, it does not help learners and demotivates them. Especially at the end of the course, when the exercises became more demanding, less learners participated. As the difficulty increases, so does the amount of time spent by the learners. Many learners are not willing to invest this extra time in solving difficult exercises.

We experienced that learners who have completed the course, look back positively on the varied exercises, even if they were sometimes more demanding. One learner stated: "The lectures include both practical and theoretical videos and explanatory test cases. The final exam, quizzes, peer review projects and programming

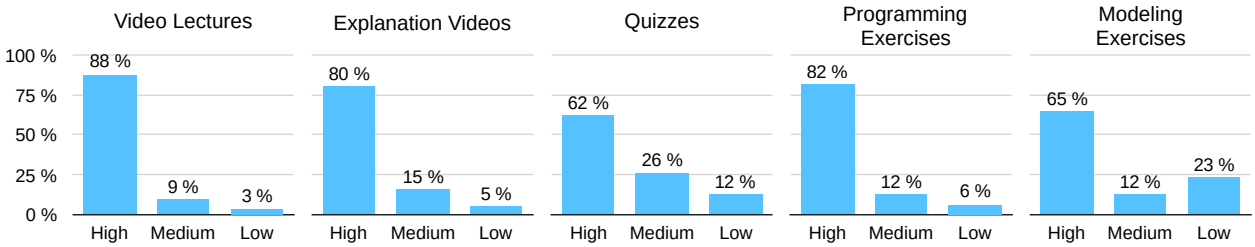


Figure 5. Answer distribution of learners about the contribution to their learning in the course.

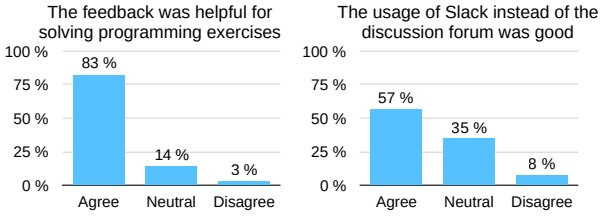


Figure 6. Helpfulness of feedback (left), Slack over discussion forums (right)

exercises are well thought and help in understanding and reviewing the core concepts of each week.“ This statement confirms the right mix of theory and practice contributing to learning success.

5.1. Learnings

We initially used a rather serious tone in the communication with the learners, but during the course we have moved further away from it. We approached the learners personally and asked them about their experiences via mail and Slack. This removes the barrier between instructors and learners and facilitates interactivity. We made clear why it is important for learners to participate in the interactive exercises. The typical MOOC learner is not accustomed to this multitude of varied exercises. Therefore, we communicate clearly from the beginning that our MOOC is an intensive course in which we expect active participation in exercises and discussions on Slack.

As a consequence of the learners' feedback, we have changed several aspects already during the first run of the MOOC. We extended the time for all exercises from one to two weeks to allow all participants to complete the exercises. This alleviated the stress factor, in particular for learners who worked in a full-time job and had families. In a further step, we increased the number of attempts for all quizzes from 1 to 2. This gives learners the opportunity to study the theory again after a wrong answer and motivates them. They can improve their knowledge and try the quiz a second time. It is important that learners receive feedback on their given answers, regardless of whether they are correct or incorrect. Only then, they can reflect on the theory again. We want learners to internalize the acquired knowledge. The feedback must be motivating and can include a sense of humor.

We also introduced a question and answers page and summary pages for each section. Many questions reached us multiple times via Slack, so we decided to collect the most frequently asked questions on a separate FAQ page. The section summary ensures that learners fully grasp the learning objectives of each section and connect the units with each other. This allows learners to reflect on the contents of the whole section again.

5.2. Best Practices

Make sure that all exercises are aligned with the learning goals in terms of constructive alignment. Double check the consistency of all exercises, especially with regard to difficulty and comprehensibility. Plan the learning goals before the production of the videos and adapt the exercises accordingly. Use the same working example throughout the course, ideally by using a problem statement that relates to the personal experience of the learners. Explain learners what they did wrong and why they did it wrong in the exercises using context-sensitive feedback. However, the feedback should not directly contain the solution, instead it should explain aspects of the theory related to the exercise. Learners have to come up with the correct solution themselves.

Be open to change and listen to the wishes and preferences of your learners. With small iteration cycles in interactive MOOCs, changes and wishes can be addressed easily. Small iteration cycles are only possible if the length of videos is limited. Address learners personally in videos to overcome the barriers in the beginning of the unit. Include animations and real world scenes in the videos and do not only rely on lecture style slides with too many text and bullet points. In programming exercises, write test cases to assess the behavior of the learners' solution and make sure the feedback in the assertions of the tests is understandable.

6. Related Work

In [36] and [37], we describe the application of interactive learning in the classroom. In the following, we focus on online courses.

Alario-Hoyos et al. describe their MOOC for introduction to programming with Java [4]. The authors state that they designed the course to enhance the learners' activity with learning contents. They found that traditional multiple choice quizzes only assess the two lower levels of Bloom's taxonomy. In addition to quizzes, they also rely on peer review and programming exercises, which they carried out with the help of the external tools *Blockly*, *Codeboard* and *Greenfoot*.

Daun et al. integrate conceptional modeling into their MOOC [38] by using ambiguous exercises and sketching multiple solutions in brief whiteboard-style videos. They state that it enables the students to assess their own solutions and fulfills their educational needs. In contrast, we use peer reviews to allow the students to receive feedback on their model solutions.

Kloos et al. use MOOCs as out-of-class-activities in addition to normal interactive classes following the inverted classroom approach [39]. They argue that more time can then be spent in interactive participation and on-site interaction leading to more effective learning. This confirms that MOOCs need to become more interactive in order to achieve a better learning experience for learners.

Krugel et al. describe their experiences on designing an interactive MOOC about object-oriented programming [40]. In addition to traditional quizzes, the authors rely on interactive programming exercises in order to put the theoretical knowledge into practice. They use various external tools, such as *SVG-edit*, *trinket*, *Java-Tutor*, and *Codeboard*. Kolas et al. introduce interactive modules [41], which are either videos or presentations to motivate and activate learners. In contrast to our approach, the authors do not focus on exercises.

Gruenewald et al. focus on the challenge of conducting interactive programming lectures as MOOCs [42]. They integrate active experimentation and relate to concrete experience. Existing literature shows that interactivity plays a role in online courses. Nevertheless, as far as we know, no one has yet defined what interactivity means in the context of MOOCs.

7. Conclusion

MOOCs can complement university courses and provide education to places that would otherwise have no access. However, they also face challenges in terms of interactivity, the stimulation of all cognitive skills and the provision of context-sensitive guidance to a large number of learners. We introduced an interactivity model for MOOCs that addresses these challenges. It includes a variety of practical exercises, in particular programming and modeling, which are typical learning goals in software engineering.

Learners can participate multiple times in exercises and learn from their failures and the context-sensitive feedback. We found first evidence that this improves the learning success. The different exercise types, the division into small learning sprints, direct communication and immediate feedback increase the interactivity and improve the learning experience.

In the future, we want to integrate semi-automatic assessment of modeling exercises using machine learning. It allows multiple solutions to be assessed as correct and does not limit the creative thinking of students. Using this approach, we can integrate more modeling exercises. In addition, we plan to integrate code reviews as described in [43]. It is important that students do not only learn to write correct programs, the code also needs to be understandable.

References

[1] T. Daradoumis, R. Bassi, F. Xhafa, and S. Caballé, "A review on massive e-learning (MOOC) design, delivery and assessment," in *8th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pp. 208–213, IEEE, 2013.

[2] R. Ubell, "How the pioneers of the mooc got it wrong," *IEEE Spectrum*, 2017.

[3] R. Williams and T. Haladyna, "Logical operations for generating intended questions (logiq): A typology for higher level test items," *A technology for test-item writing*, pp. 161–186, 1982.

[4] C. Alario-Hoyos, C. Kloos, I. Estévez-Ayres, C. Fernández-Panadero, J. Blasco, S. Pastrana, and J. Villena-Román, "Interactive activities: the key to learning programming with MOOCs," *Proceedings of the European Stakeholder Summit on Experiences and Best Practices in and Around MOOCs*, 2016.

[5] T. Connolly, M. Stansfield, and T. Hainey, "An application of games-based learning within software engineering," *British Journal of Educational Technology*, vol. 38, no. 3, pp. 416–428, 2007.

[6] D. Shaffer, "Pedagogical praxis: The professions as models for postindustrial education," *Teachers College Record*, vol. 106, no. 7, pp. 1401–1421, 2004.

[7] J. Whitehead, "Collaboration in software engineering: A roadmap," *FOSE*, vol. 7, no. 2007, pp. 214–225, 2007.

[8] T. Staubitz *et al.*, "Towards Practical Programming Exercises and Automated Assessment in Massive Open Online Courses," in *International Conference on Teaching, Assessment, and Learning for Engineering*, pp. 23–30, 2015.

[9] S. Freeman, S. Eddy, M. McDonough, M. Smith, N. Okoroafor, H. Jordt, and M. Wenderoth, "Active

learning increases student performance in science, engineering, and mathematics," *Proceedings of the National Academy of Sciences*, vol. 111, no. 23, pp. 8410–8415, 2014.

[10] M. Prince, "Does active learning work? a review of the research," *Journal of Engineering Education*, vol. 93, no. 4, pp. 223–231, 2004.

[11] P. Kirschner, J. Sweller, and R. Clark, "Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching," *Educational psychologist*, vol. 41, no. 2, pp. 75–86, 2006.

[12] B. Bloom, M. Engelhart, E. Furst, W. Hill, and D. Krathwohl, "Taxonomy of educational objectives: The classification of educational goals," 1956.

[13] J. Biggs, "Aligning teaching and assessing to course objectives," *Teaching and learning in higher education: New trends and innovations*, vol. 2, pp. 13–17, 2003.

[14] F. Marton and S. Booth, *Learning and awareness*. Psychology, 1997.

[15] C. Bonwell and J. Eison, *Active Learning: Creating Excitement in the Classroom*. ASHE-ERIC Higher Education Reports, 1991.

[16] J. Michael, "Where's the evidence that active learning works?," *Advances in physiology education*, vol. 30, no. 4, pp. 159–167, 2006.

[17] T. Good and J. Brophy, *Looking in classrooms*. Harper & Row, 1987.

[18] P. Lowry, N. Romano, J. Jenkins, and R. Guthrie, "The CMC interactivity model: How interactivity enhances communication quality and process satisfaction in lean-media groups," *Journal of Management Information Systems*, vol. 26, no. 1, pp. 155–196, 2009.

[19] S. Rafaeli, "From new media to communication," *Sage annual review of communication research: Advancing communication science*, vol. 16, pp. 110–134, 1988.

[20] R. Mayer, *The Cambridge handbook of multimedia learning*. Cambridge university press, 2005.

[21] S. Steffensen, "Human interactivity: problem-solving, solution-probing and verbal patterns in the wild," in *Cognition beyond the brain*, pp. 195–221, 2013.

[22] E. Jones and H. Gerard, "Foundations of social psychology," 1967.

[23] S. Domagk, R. Schwartz, and J. Plass, "Interactivity in multimedia learning: An integrated model," *Computers in Human Behavior*, vol. 26, no. 5, 2010.

[24] G. Johnson, G. Bruner, and A. Kumar, "Interactivity and its facets revisited: Theory and empirical test," *Journal of Advertising*, vol. 35, no. 4, 2006.

[25] M. Yacci, "Interactivity demystified: A structural definition for distance education and intelligent CBT," *Educational Technology*, vol. 40, no. 4, pp. 5–16, 2000.

[26] S. Krusche, A. Seitz, J. Börstler, and B. Bruegge, "Interactive learning: Increasing student participation through shorter exercise cycles," in *Proceedings of the 19th Australasian Computing Education Conference*, pp. 17–26, ACM, 2017.

[27] D. Kolb, *Experiential learning: Experience as the source of learning and development*, vol. 1. Prentice Hall, 1984.

[28] K. Schwaber, "Scrum development process," in *Proceedings of the OOPSLA Workshop on Business Object Design and Information*, 1995.

[29] S. Krusche, B. Reichart, P. Tolstoi, and B. Bruegge, "Experiences from an experiential learning course on games development," in *Proceedings of the 47th SIGCSE*, pp. 582–587, 2016.

[30] K. VanLehn, "Cognitive skill acquisition," *Annual Review of Psychology*, vol. 47, pp. 513–539, 1996.

[31] J. Sweller and G. Cooper, "The use of worked examples as a substitute for problem solving in learning algebra," *Cognition and Instruction*, vol. 2, no. 1, pp. 59–89, 1985.

[32] J. Trafton and B. Reiser, "Studying examples and solving problems: Contributions to skill acquisition," tech. rep., Naval HCI Research Lab, 1993.

[33] A. Collins, J. Brown, and A. Holum, "Cognitive apprenticeship: Making thinking visible," *American educator*, 1991.

[34] S. Krusche and A. Seitz, "ArTEMiS - An Automatic Assessment Management System for Interactive Learning," in *Proceedings of the 49th SIGCSE*, ACM, 2018.

[35] S. Krusche, B. Brügge, I. Camilleri, K. Krinkin, A. Seitz, and C. Wöbker, "Chaordic learning: A case study," in *Proceedings of the 39th ICSE*, pp. 87–96, 2017.

[36] A. Seitz and B. Bruegge, "Teaching pattern-based development," in *Proceedings of the 1st Workshop on Innovative Software Engineering Education*, pp. 20–23, 2018.

[37] S. Krusche, N. von Frankenberg, and S. Afifi, "Experiences of a software engineering course based on interactive learning.," in *Proceedings of the 15th SEUH Workshop*, pp. 32–40, 2017.

[38] M. Daun, J. Brings, P. Obe, K. Pohl, S. Moser, H. Schumacher, and M. Rieß, "Teaching conceptual modeling in online courses: Coping with the need for individual feedback to modeling exercises," in *30th Conference on Software Engineering Education and Training*, pp. 134–143, 2017.

[39] C. Kloos, C. Alario-Hoyos, I. Estévez-Ayres, P. Muñoz-Merino, M. Ibáñez, and R. Crespo-García, "Boosting interaction with educational technology," in *Global Engineering Education Conference*, pp. 1763–1767, April 2017.

[40] J. Krugel and P. Hubwieser, "Computational thinking as springboard for learning object-oriented programming in an interactive mooc," in *IEEE Global Engineering Education Conference*, pp. 1709–1712, 2017.

[41] L. Kolås, H. Nordseth, and J. Hoem, "Interactive modules in a mooc," in *15th International Conference on Information Technology Based Higher Education and Training*, pp. 1–8, Sept 2016.

[42] F. Grünewald, C. Meinel, M. Totschnig, and C. Willems, "Designing MOOCs for the Support of Multiple Learning Styles," in *European Conference on Technology Enhanced Learning*, pp. 371–382, 2013.

[43] S. Krusche, M. Berisha, and B. Bruegge, "Teaching code review management using branch based workflows," in *Proceedings of the 38th ICSE*, pp. 384–393, 2016.

8.7 Stager: Simplifying the Manual Assessment of Programming Exercises

This workshop paper describes an approach to simplify the manual assessment of programming exercise submissions. The approach increases the consistency, removes late submissions, and combines the changes of students in the version control system to one changeset. It was first developed independently of Artemis in an external tool called Stager. It was later integrated into Artemis to make it easier to use.

Christopher Laß proposed the approach and implemented it. The author of this habilitation supervised Christopher Laß throughout the development of the approach and writing the paper.

Authors	C. Laß, **S. Krusche**, N. von Frankenberg and Bernd Bruegge
Conference	16. Workshop Software Engineering im Unterricht der Hochschulen
Publisher	CEUR
Pages	10
Type	Workshop Paper
Review	Peer Reviewed (2 Reviewers)
Year	2019
Citation	[LKvFB19]
Link	http://ceur-ws.org/Vol-2358

Stager: Simplifying the Manual Assessment of Programming Exercises

Christopher Laß, Stephan Krusche, Nadine von Frankenberg, Bernd Brügge
Technische Universität München
christopher.lass@tum.de, krusche@in.tum.de, nadine.frankenberg@in.tum.de, bruegge@in.tum.de

Abstract

Assessing programming exercises requires time and effort from instructors, especially in large courses with many students. Automated assessment systems reduce the effort, but impose a certain solution through test cases. This can limit the creativity of students and lead to a reduced learning experience. To verify code quality or evaluate creative programming tasks, the manual review of code submissions is necessary. However, the process of downloading the students' code, identifying their contributions, and assessing their solution can require many repetitive manual steps.

In this paper, we present Stager, a tool designed to support code reviewers by reducing the time to prepare and conduct manual assessments. Stager downloads multiple submissions and adds the student's name to the corresponding folder and project, so that reviewers can better distinguish between different submissions. It filters out late submissions and applies coding style standards to prevent white space related issues. Stager combines all changes of one student into a single commit, so that reviewers can identify the student's solution more quickly.

Stager is an open source, programming language agnostic tool with an automated build pipeline for cross-platform executables. It can be used for a variety of computer science courses. We used Stager in a software engineering undergraduate course with 1600 students and 45 teaching assistants in three separate programming exercises. We found that Stager improves the code correction experience and reduces the overall assessment effort.

1 Introduction

The number of students in university courses is increasing. The number of new undergraduate students at our computer science department increased by 81 % between 2013 (1110 students) and 2017 (2005 students)[1]. Practical programming exercises are essential in computer science education and help students acquire important skills in software development [Staubitz et al., 2015]. However, a manual assessment of programming exercises in large courses can take a considerable amount of time and effort. Automatic assessment systems (also called auto-graders) aim at flexibility and scalability in large courses, and allow to integrate exercises into lectures [Krusche et al., 2017b]. These systems utilize, among others, version control systems (VCS) to store the code solutions of students in repositories and test cases that are executed on a continuous integration server to assess the solution to a programming exercise automatically [Heckman and King, 2018; Krusche and Seitz, 2018].

While automated assessment systems significantly reduce manual assessment effort, they have drawbacks. Predefined test cases cannot cover all possible solutions and therefore impose a certain solution on the students. Some students are limited in their programming skills, while other students can exploit the test cases by repetitive trial-and-error submissions. Especially first year students who are new to programming often experience problems when trying to formulate their solution and thoughts as an executable computer program [Robins et al., 2003]. Such submissions can be overly complicated, and assessment systems cannot (yet) provide enough useful feedback in that regard. Furthermore, some programming exercises cannot be assessed automatically. The automated grading of creative assignments with open problem statements is hardly possible because different solutions exist [Knobelsdorf and Romeike, 2008; Krusche et al., 2017a]. An example for such an assignment is to implement a creative collision strategy in a 2D racing game. Automated test cases could be able to validate a collision, but are incapable of assessing the creativity or code quality of the solution. As a result, manual assessment can be beneficial, even in large courses that have fully implemented automated grading solutions.

However, the process of manually assessing multiple students' solutions requires repeated manual steps. Tasks such as finding the next student's repository, downloading the source code, and renaming the folders and projects names for standardization can be time-consuming and error-prone. Determining a student's contribution is challenging when the exercise builds upon a provided code template and when the

[1] https://www.tum.de/die-tum/die-universitaet/die-tum-in-zahlen/studium

students use multiple commits in their code repository. Then it becomes difficult to separate the provided template and the final solution.

In this paper, we present Stager, a tool that is designed to support the manual assessment of programming exercises. Reviewers, e.g. teaching assistants or instructors, can automate the manual steps that are necessary to prepare the students' code repositories, for instance download all repositories at once, and thereby reduce the manual assessment time. The idea for Stager evolved during an undergraduate university course with 1600 students and 45 teaching assistants. An initial implementation was used for three separate programming assignments.

The remainder of the paper is organized as follows. We describe related work focusing on existing automated assessment solutions and the limitations of automated assessment approaches in Section 2. In Section 3, we cover Stagers' approach to automating the recurring manual steps during the correction of programming exercises. We describe design decisions, the exercise workflow with Stager, the configuration possibilities of the tool, and the concrete tasks of Stager, e.g. the *Download repositories* task. We analyze the improved code assessment experience of the teaching assistants by means of an experience report in Section 4, where we also present the results of a quantitative analysis of Stager's use in three programming exercises. Section 5 concludes the paper and provides directions for future work.

2 Related Work

Several automated assessment system approaches for programming assignments exist [Heckman and King, 2018; Knobelsdorf and Romeike, 2008; Krusche and Seitz, 2018; Pieterse, 2013]. Advantages include a decrease in the workload of course instructors and timely feedback for students [Pieterse, 2013]. Automated systems work well to grade programming assignments consistently and evaluate specific aspects, e.g. the functionality [McCracken et al., 2001] or efficiency of a system [Jackson and Usher, 1997]. However, they are missing the benefit of personal feedback which a manual grading approach could provide. The test cases used by such systems cannot assess the code quality and "elegance" of the solution [Poženel et al., 2015].

Building a robust automated assessment system amounts to a heavy workload, whereby the definition of the test cases is (usually) the most time consuming activity [Cerioli and Cinelli, 2008]. This workload is amplified when designing tasks with some degree of freedom of solutions [Chen, 2004]. The degree of freedom of solutions indicates the difficulty of the exercise [Striewe and Goedicke, 2013], meaning that a difficult exercise has more possible solutions and therefore has an increased workload to design the automated assessment system. Depending on the class size, it can therefore be less time consuming to manually assess solutions rather than to design the automated assessment system [Ala-Mutka, 2005].

Further, students can become distracted by automated feedback. For instance, students may be tempted to fix only the failing tests instead of focusing on the assignment [Heckman and King, 2018]. Automated assessment systems circumvent the detection of frequent mistakes or misunderstandings among students. The understanding and resolution of common errors is an essential learning experience for students. Semi-automated systems combine the mentioned aspects by providing automated grading, as well as manual feedback. Such systems offer personalized feedback to some extent, for instance the instructor can annotate a static assessment [Gerdes et al., 2017]. Other systems give the student instant feedback if the student's solution is correct. If it is not, the instructor reviews each solution and can give additional feedback if required [Insa and Silva, 2015]. Many systems focus on the grading itself, but not on the process the instructor has to follow to obtain the students' solutions.

Some commercially available systems and tools that are used in computer science (CS) courses offer features that aim at simplifying this process. In 2000, Jackson proposed an approach that pre-processes student submissions (sent via e-mail) by removing irrelevant information or unpacking files [Jackson, 2000]. For submissions via repositories, pull requests (also called merge requests) in GitHub[2], GitLab[3], or Bitbucket[4] allow students to commit their changes into separate branches. After requesting the code to be merged into the main branch, i.e. a submission, reviewers can highlight the student's contribution as difference to the template code and provide feedback by requesting changes. While pull requests can also be integrated with continuous integration systems, e.g. using TravisCI[5] to detect compile errors and to run automated tests, reviewers might still need to download the source code and execute it to verify if all requirements of the problem statement have been solved.

GitLab introduced a "Squash and Merge" option which "applies all of the changes in a merge request as a single commit, and then merges that commit using the merge method set for the project"[6]. This cleans up the commit history and can make it easier to identify the contribution of one particular student. Tools and services, such as Gerrit[7], support code reviews that enable the reviewer to see the code difference, and provide the option to leave in-line comments.

[2] https://github.com
[3] https://gitlab.com
[4] https://bitbucket.org
[5] https://travis-ci.org
[6] https://docs.gitlab.com/ee/user/project/merge_requests/squash_and_merge.html
[7] https://www.gerritcodereview.com

However, such tools primarily focus on continuous feedback rather than assessing a student's solution.

3 Stager's Approach

This section presents an approach that automates manual steps during the correction of programming exercises in order to prepare student repositories for easier assessment. We show how code reviewers can use Stager. Furthermore, we explain the different tasks that are automatically executed by Stager.

Figure 1 illustrates the exercise workflow including the manual assessment with the help of Stager as a UML activity diagram. As precondition for this workflow, every student must have their own repository with the code template for the exercise in a VCS[8]. After the students complete the exercise, they commit and push their solutions to the VCS (action 1.3).

Before reviewers start to work, they need to configure Stager (action 2.1). Then, they trigger Stager to process different tasks (actions 3.1 ... 3.6), such as *Download repositories* or *Normalize code style*. Finally, the reviewer can manually assess the pre-processed submissions and give qualitative feedback (action 4.2 and 5.) to the students in any arbitrary form (e.g. uploading the feedback into an exercise management system such as Moodle[9]).

The action *2.1 Configure Stager* of the *Reviewer* is described in Section 3.1. *Stager*'s actions are described as tasks in Section 3.2. The numbering in Section 3.2 aligns with the corresponding action in Figure 1.

3.1 Stager's Setup

Stager is free, open source, and available under the MIT license[10]. It is platform independent and programming language agnostic, making Stager universally applicable. It is written in the Go programming language[11] and makes use of the distributed version control system git[12]. Cross-platform executables can be downloaded from the automatic build pipeline or compiled from the source code.

Stager's configuration is separated into two files *students.csv* and *config.json*, based on how frequently the settings change. The list of students in *students.csv* might not change during the course duration, while *config.json* changes for every exercise. The configuration procedure must be completed after the code template was finished and before Stager is executed. Stager or its configuration does not add any preconditions or constraints on the students. The following settings can be edited:

1. Credentials: Remote git repositories can be accessed via the SSH or HTTP protocols [Lawrance et al., 2013]. For HTTP, the JSON keys *username* and *password* have to be set with valid credentials and access rights to the VCS. For SSH, Stager uses the operating system's global SSH settings and therefore does not require further configuration.

2. Latest commit hash of a programming exercise template: The programming exercises that are distributed to the students build upon a given code template. The SHA hash of the latest commit for the code template, meaning the latest code changes the reviewer included, must be set for the JSON key *squash_after*. This setting is required for Stager to distinguish between the given code by the reviewer and code written by the student. This configuration option is used by the task *Combine commits* and is further elaborated in Section 3.2.

3. Deadline for homework submission: Students have to submit their homework in a given time-frame.

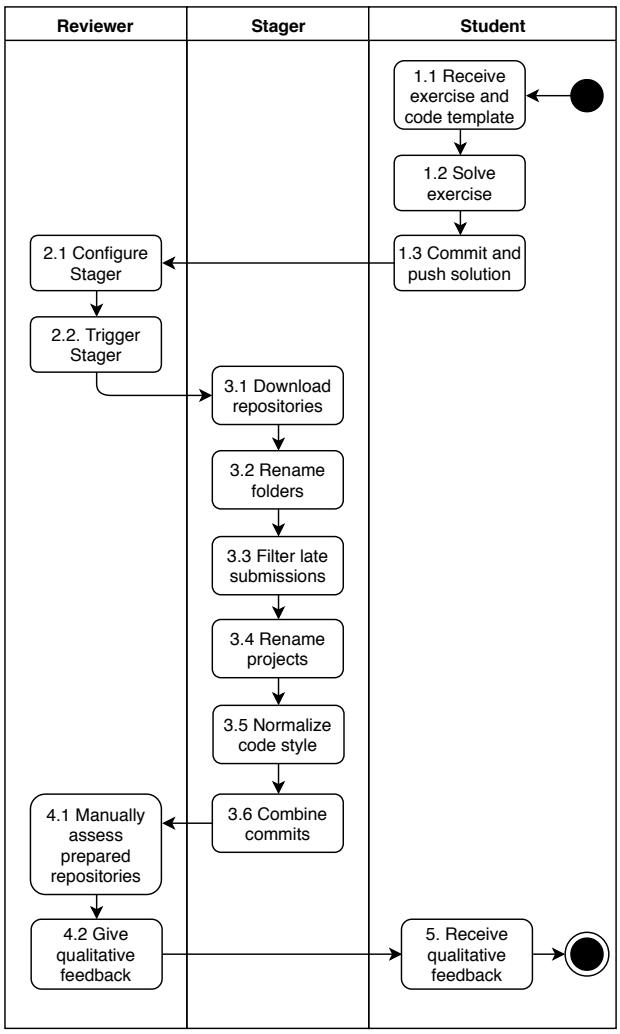


Figure 1: Exercise workflow with Stager: students complete the exercise and upload their solutions to a VCS. The reviewer configures and triggers Stager to process different tasks, e.g. *3.1 Download repositories*. Afterwards, the reviewer manually assesses the prepared repositories and gives qualitative feedback to the students.

[8] There are multiple tools available that automate this step, e.g. ArTEMiS, Github Classroom, etc.

[9] `https://moodle.org`

[10] `https://github.com/arubacao/stager`

[11] `https://golang.org`

[12] `https://git-scm.com`

For example, the homework must be submitted by Sunday midnight because the programming exercises will be discussed in class on Monday morning. However, VCSs have limitations when it comes to time-based repository access. As described in more detail in Section 3.2, the task *Filter late submissions* allows to overcome these VCSs limitations. The deadline for students submitting their homework is set with the JSON key *deadline*. The standard datetime format `YYYY-MM-DD HH:MM:SS` must be used. For example, `2018-08-31 23:59:59` is valid.

4. Remote repository URL schema: Each student has a personal repository that can be accessed with a unique URL. A general URL schema can be derived from these unique URLs, where the students' identifiers are substituted by a placeholder. For example, for the repository URL (1) of student *10001*, the derived general URL schema is (2). If the repositories are accessed using HTTP as in the example, two additional placeholders must be set for the reviewer's credentials (3). The resulting schema is set for the key *url*.

*https://repo.uni/cs101/exercise01-**10001**.git* (1)

*https://repo.uni/cs101/exercise01-**%s**.git* (2)

*https://**%s:%s**@repo.uni/cs101/exercise01-**%s**.git* (3)

5. List of students: In addition to the mentioned settings, Stager requires a list of students the reviewer wants to assess. The students' names and identifiers are defined in the *students.csv* file with the format shown in Listing 1. All mentioned people and courses in this paper are placeholder names and do not exist in reality.

Listing 1: Sample students.csv

```
name,id
John Doe,10001
Jane Roe,10002
```

After configuration, the Stager executable, *config.json*, and *students.csv* are placed in a dedicated and empty folder. Stager can then be executed via a double click or from the terminal. Listing 2 illustrates this workflow. After Stager terminates, the students' repositories are locally available and prepared by the tasks described in the following Section 3.2.

Listing 2: Folder setup and execution of Stager

```
$ cd ~/cs101/assessment3
$ ls
config.json   stager   students.csv
$ ./stager
```

3.2 Stager's Tasks

Stager provides an extendable framework which makes it easy to add or remove tasks according to the reviewer's requirements. Tasks are functions that modify the repository or its contents and have a single purpose. For example, the *Rename folders* task appends the student's name to the corresponding folder. Stager is composed of multiple tasks (shown in the Stager swimlane in Figure 1) that adhere to certain rules and are sequentially performed during the tool's execution. The implementation allows a clear distinction of tasks, such that each task addresses a separate purpose. Therefore, it is easy to add new tasks or remove existing ones conceptually and implementation-wise in the future. For example, when the reviewer does not need a certain task, only one line of code within the array of tasks has to be removed. Furthermore, tasks must be idempotent, meaning that multiple executions of the task lead to the same output. Even though tasks are independent, they are processed sequentially, i.e. the order of the tasks is relevant. For instance, repositories first have to be downloaded before other tasks have local file access.

The goal of Stager is to simplify the manual assessment of programming exercises by modifying source code, files, and repositories. Repetitive manual steps that are required for the reviewer to start the assessment should be reduced or eliminated by Stager. We identified the following relevant tasks (listed according to the order of execution) and describe each of them in detail in the following:

1. Download repositories
2. Filter late submissions
3. Rename folders
4. Rename projects
5. Normalize code style
6. Combine commits

1. Download repositories: In order to better determine the software quality and verify if all requirements of the problem statement have been solved by the students' submissions, it is necessary for the reviewer to compile and execute their homework source code locally. Hence the repositories must be available on the reviewer's computer. The initial task clones all repositories of the predefined students *as-is* and *all at once* to a given folder on the reviewer's computer. This first task takes potential existing local repositories into account and overwrites them. It ensures that each local repository is in sync with the remote repository and in a clean state.

The following tasks modify files and therefore require write access to the repositories. These modifications can only be performed when the repositories are locally available. Consequently, the *Download repositories* task must be first.

2. Filter late submissions: Homework submissions are tied to a hard deadline. With web-based VCSs like Bitbucket or GitLab, it is hardly possible to block student commits after a given deadline. Students could exploit this situation and extend their

time to finish the exercise as shown in Figure 2. The *Filter late submissions* task analyzes the commit timestamps and sets the repository to the state of the pre-configured deadline in *config.json*. Commits after the deadline are not considered anymore. This way time-based limitations of web-based VCSs are bypassed. However, this procedure is not fully forgery-proof, since commit timestamps can be manipulated.

File changes made prior to this task would be striped out, since the repository is set to the state of the pre-configured deadline. Therefore, the *Filter late submissions* task must be executed before any other task can modify files.

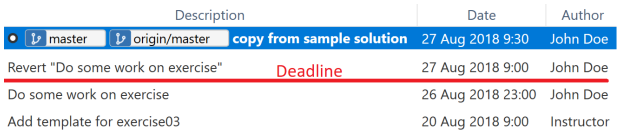


Figure 2: Filter late homework submissions by excluding commits after the homework submission deadline. The two commits above the red line are after the deadline while the two commits below the red line are before the deadline.

3. Rename folders: Depending on the naming convention, only the student's identifier is used for the repository name. The resulting folders can be hard to keep separate and to associate with the correct student. For obfuscation and identity protection this is reasonable, but counterproductive on the reviewer's local system since it is easier to identify a student by their name and not through their id. Once the repositories are locally available, the *Rename folders* task appends the student's name to the corresponding folder as illustrated in Figure 3.

```
1    - cs101-exercise03-10001
2    - cs101-exercise03-10002
1    + cs101-exercise03-10001_john_doe
2    + cs101-exercise03-10002_jane_roe
```

Figure 3: Append names to folders to better distinguish between students. Without the names john_doe and jane_roe, it would be difficult to identify which folder belongs to which student.

4. Rename projects: As precondition of Stager, each student must have their own repository for each published exercise. The content of these repositories is always identical. As a result, the project names are also identical for all students. This leads to the problem that reviewers could only import one project at the same time into Eclipse in order to review and execute the code. Renaming all projects manually is time-consuming and error-prone. Analogue to the *Rename folders* task, a student's name is prepended to the corresponding project name. This makes it possible to distinguish between students within source code editors or integrated development environments (IDEs), e.g. Eclipse[13] (Figure 4), and allows to import multiple projects at the same time. Eclipse, for instance, does not allow to import multiple projects with identical names, which makes it impossible to compare multiple solutions without renaming the projects.

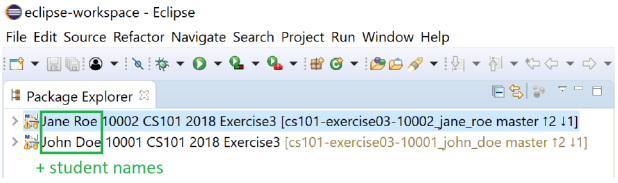


Figure 4: Prepend student names to projects so that the submissions of multiple students can be imported into Eclipse and reviewed at the same time. Jane Roe and John Doe are prepended to the project name. Otherwise the reviewer could only import one Eclipse project at the same time.

5. Normalize code style: The encoding and code style of the provided code template and the final student's contribution should be consistent. Windows and Unix-based systems use different line breaks for code files by default. Windows uses carriage return and line feed ''\r\n'' as a line ending, whereas Unix based systems use just line feed ''\n''. Also, IDEs might automatically enforce a different code style standard than desired. As illustrated in Figure 5, this could lead to non-relevant changes and obscured code differences in commits, thereby making it harder to assess the submission. To avoid these non-relevant file changes by the student, Stager invokes a *linter* that automatically normalizes the code to the same standards as the initial template. This means that all white space related changes, e.g. line breaks, empty spaces and tabs are removed, so that the reviewer does not need to analyze them. Each programming language has its own linting strategies, utilizing existing tools like *eslint*[14] for Javascript or *checkstyle*[15] for Java. This hides pure white space and encoding changes and allows code reviewers to focus on the actual contributions by the students.

6. Combine commits: Reviewers provide code templates as a starting point for the programming exercise, in which the student has to make changes across multiple files. These changes can be small compared to the provided template and consequently hard to identify by the reviewer. In order to determine the student's contribution more effectively, it is helpful to see the exact difference between the template and the final submission instead of only looking at the final submission. VCSs provide easy comparison methods

[13] https://www.eclipse.org
[14] https://github.com/eslint/eslint
[15] https://github.com/checkstyle/checkstyle

```
1   - public int getSpeed(){
2   -       return this.speed;
3   - }
1   + public int getSpeed(){
2   +       return this.speed;
3   + }
```

Figure 5: There is no visual change in the two code blocks in this figure. However, non-visible *line breaks* cause the comparison tool to show these lines. This can make it time-consuming for the reviewer to identify relevant changes.

where the difference made by a single commit is visible. However, a submission can consist of multiple commits. The reviewer would have to compare each commit and memorize the changes themselves, which makes the standard comparison method impractical and error-prone.

The *combine commits* task combines the students commits into one single commit. The reviewer does not need to review multiple changes within the same code line and can omit changes that have been added in one commit and removed again in a later commit. This single commit also contains all Stager related changes (e.g. white space changes). As a result, it is easy for the reviewer to quickly identify the student's contribution and to decide if the solution is correct. In addition to the existing branches with the complete commit history, Stager adds the combined commit into a separate branch. Thus, information is only added and not removed from the repository and the reviewer could still see the whole commit history. Web-based VCSs like GitHub also offer a squash feature, however, the reviewer would have to trigger it manually for each repository.

Figure 6 illustrates this process with an example student *John Doe* and an *Instructor*. The *Instructor* provides a code template. *John Doe* works on the given exercise. Over a period of one day, *John* submits his work separated across multiple commits. As seen in the bottom right corner of Figure 6, one assignment was to *Add new car types to the game*. Since *John* submitted multiple code changes and removed the "TODO" lines within the code, the reviewer would have to actively scan all nine commits to identify *John*'s solution. Stager solves this time-consuming process by combining all student commits into one single commit that includes all changes by *John*. This single commit is selected in the top of Figure 6. The reviewer can see every file that has been modified by the student and quickly identify, whether *John* has completed the assignment correctly.

4 Experience Report

The following experience report describes the lecture-based course Introduction to Software Engineering (EIST[16]) in which we used Stager to improve the manual assessment of programming exercises. EIST is a second semester bachelor's course with a heterogeneous group of students including computer science, business informatics, and business students.

The course assumes that students have successfully completed an introductory course in computer science (e.g. CS1) and are familiar with object-oriented programming in Java. The course's learning goals are that students are able to apply relevant concepts and methods in all phases of software engineering projects including analysis, design, implementation, testing, and delivery. Further, students know the most important terms and concepts and can apply them in modeling and programming tasks. They are aware of the problems and issues that generally have to be considered in software engineering projects. Table 1 shows the schedule and the content of the course.

Week	Content
1	Introduction
2	Model-Based Software Engineering
3	Requirements Elicitation and Analysis
4	System Design I
5	System Design II
6	Object Design
7	Model Transformations and Refactorings
8	Pattern-Based Development
9	Lifecycle Modeling
10	Software Configuration Management
11	Testing
12	Project Management
13	Repetitorium

Table 1: The course *Introduction to Software Engineering* lasts 13 weeks.

1600 students were registered for the course in 2018. One lecturer and three exercise instructors were involved in the organization of the course. 45 teaching assistants were responsible for holding 74 exercise group sessions per week. Teaching assistants were mainly bachelor students in the fourth semester, who successfully completed the same course in the previous year.

The course design is based on interaction and assumes active participation from students. The interactive parts include in-class exercises, in-class quizzes, and exercise sessions. Students need to bring their laptops to the class and to exercise sessions. Students can earn bonus points for completing in-class and homework exercises successfully. They can use these bonus points to improve their final exam grade.

[16]The German title is "Einführung in die Softwaretechnik".

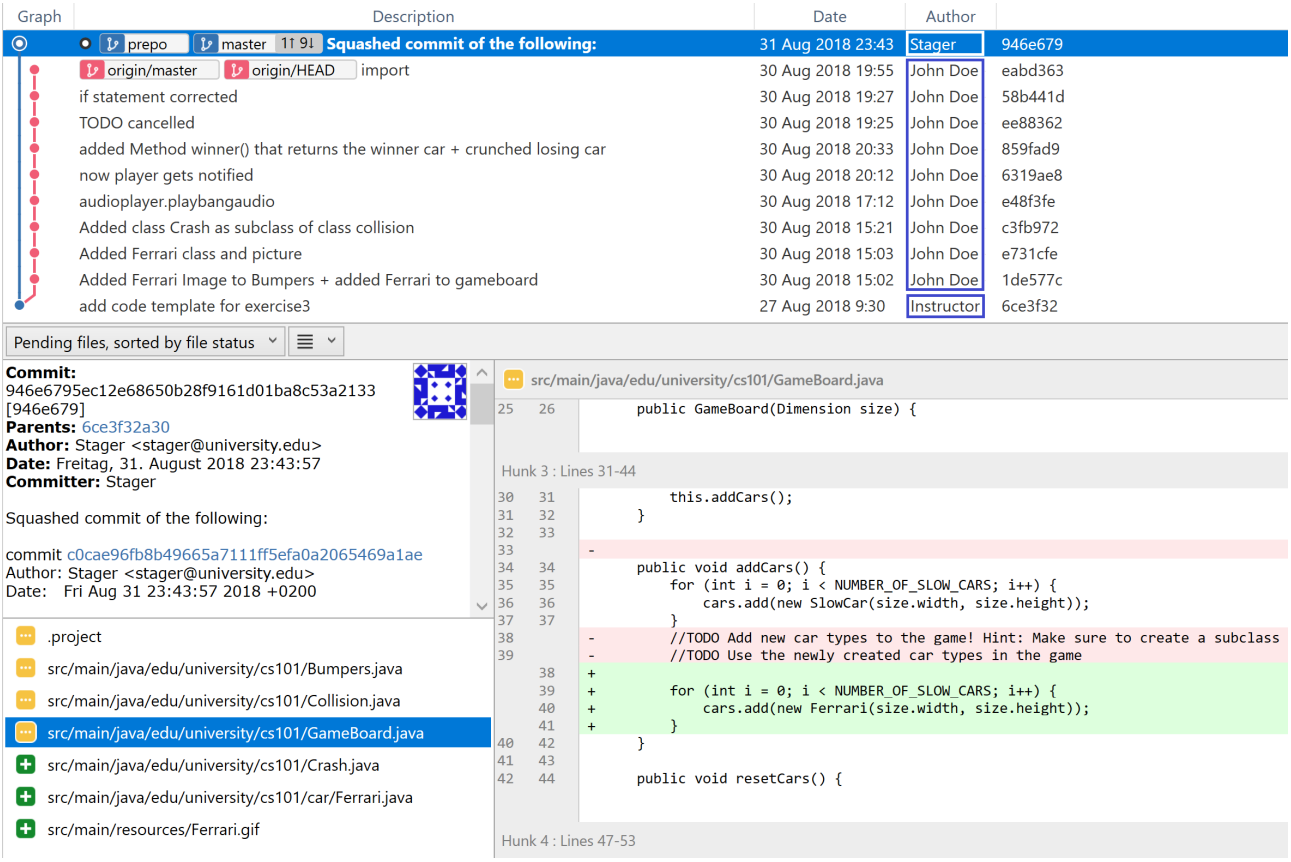


Figure 6: Student commits are combined into one discrete change set: the commit at the top highlighted in blue. This commit displays the difference between a provided code template by the instructor and the submitted solution by the student. All commits of the student John Doe are still available.

For instance, if they score more than 90 % of the total exercise points, their grade in the final exam is improved by 1.0. This possibility motivates the students to participate in the in-class exercises and in the homework exercises. In-class exercises consist of quizzes (similar to the quiz exercises described in [Krusche et al., 2017c]), modeling and programming exercises. Homework exercises include modeling, text and programming exercises.

4.1 Programming Exercises

Between 600 and 1200 students have actively participated in each programming exercise throughout the semester which is shown in Figure 7 and Figure 8. In each exercise, the students had to write new source code or adjust existing code based on a given problem statement. All students worked on the existing template code of an exercise in their individual git repository. The exercises were based on a 2D racing game called Bumpers. In the game, cars collide with each other and each collision has a winner. The course is designed so that each week's exercises focus on a different part of Bumpers in accordance with the lecture's content, e.g. in week 8, "Pattern-Based Development", exercises include the implementation of different design patterns to make the game extensible for new requirements.

To submit their solutions, the students commit their changes to a version control system. This automatically triggers test cases on a continuous integration server to verify the given solution. After the submission of their solution, students can automatically see the test results as individual feedback and improve their solution according to this feedback.

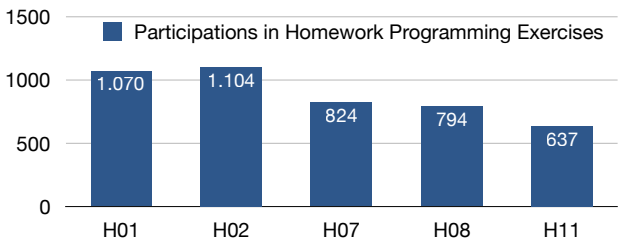


Figure 7: Number of students who submitted solutions to homework programming exercises

However, not all aspects of a problem statement can be automatically tested. Either it is difficult to test a certain aspect of a solution, for instance complex behavior tests, or the problem statement provides a

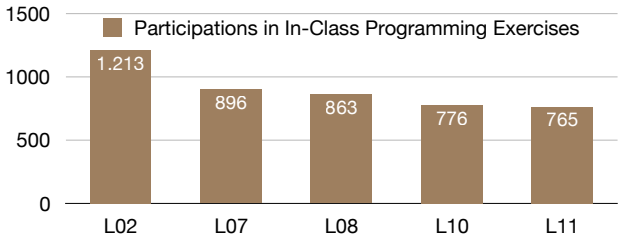


Figure 8: Number of students who submitted solutions to in-class programming exercises

high degree of freedom which makes it difficult to write test cases, e.g. open or visionary questions.

The following three homework programming exercises required manual assessment by the teaching assistants. The second and third exercises were graded semi-automated.

1. Collision Detection: The task was to implement a creative collision detection algorithm for cars in Bumpers. The students were given executable template code and had to extend it with a new class that included their solution. This exercise required manual correction to test whether the new collision algorithm performed as intended. Additionally, the most creative solutions were awarded and shown in class.

2. Serialization of Code: The students had to instantiate objects from two classes in Java. The main task was to serialize and deserialize these object using JSON. An automated assessment system was used to test the input and output of the serialization. However, the students wrote their own serialization code, so their solutions varied, e.g. in the naming of the objects or methods. This required the teaching assistants to assess the implementations manually.

3. Adapter Pattern: Based on a code template, the assignment was to extend the 2D car racing game Bumpers with legacy code using the adapter pattern. The legacy code for an existing analog speedometer panel was provided separately. An automated assessment system graded the students' solution. In addition, the teaching assistants had to verify if the speedometer panel was shown in the game user interface and displayed the velocity correctly.

4.2 Results

In order to determine how many manual steps during a homework assessment can be automated by Stager, we conducted a quantitative analysis for these three programming exercises. For the qualitative analysis we focused on:

1. Number of commits per student
2. Number of commits after the exercise deadline
3. Source code changes where only white spaces have been added or removed

Table 2 displays an overview of the number of participating students for each exercise together with submission metrics. The number of commits per student varies from 1.81 to 5.91 on average. Stager's *combine commits* task will combine student commits into one single commit so that reviewers can distinguish the difference between the provided code template and code submitted by the student immediately. There are respectively 34, 8, and 7 late submissions for the observed exercises. Stager will automatically filter commits that are contributed after the defined exercise deadline. There are between 118 and 183 students that submitted at least one commit where they only changed white spaces. While reviewing the student contributions, white space related changes are visually distracting to the reviewer (see Figure 5), since these changes are not relevant to the exercise.

In informal discussions, seven teaching assistants reported that Stager reduced their reviewing effort significantly. The workflow without Stager required the teaching assistants to first filter the repositories by student, then to check the commit dates and times, clone or download the code, and to fix potential white space problems in order to be able to assess the actual sumbission. Depending on the amount of exercise sessions, teaching assistants had to perform this manual workflow for up to 50 student submissions. Further, the repository names only include the student's identifiers, not names, so that mix-ups could occur when importing the solutions into an IDE.

4.3 Discussion

While using Stager, we identified four main advantages: (1) Combining commits is particularly helpful to review all changes of one student at a glance. This allows the reviewer to immediately identify whether the student has understood the problem statement and has implemented a proper solution. (2) Renaming the projects simplifies the assessment and comparison of multiple solutions. The reviewer can import multiple solutions at the same time with one click into an IDE. It increases the confidence of the reviewers, so that the assessment is associated with the correct student. (3) While most students follow the deadline of an exercise, some students have committed changes after the deadline. It would be possible to remove write permissions for all student git repositories at the given deadline, but this might be hard to realize. Enforcing the deadlines in Stager is easier and filters the cases where students try to circumvent the deadline. (4) Stager only depends on using git repositories for programming exercises and other instructors can use it without adaptions in their courses, e.g. in GitHub Classroom or other git environments[17]. As Stager is open-source, other instructors can adapt it to their own needs.

While Stager is easy to use as a standalone tool, reviewers need to configure it for each exercise as described in Section 3.1. It would further simplify the

[17] `https://classroom.github.com`

Metric	1. Collision Detection	2. Serialization of Code	3. Adapter Pattern
Total submission count	1104	657	794
Total commit count	1998	3880	2447
Average amount of commits per student	1.81	5.91	3.08
Total commits after exercise deadline	34	8	7
Total submission count with at least one white space related change	125	118	183

Table 2: Quantitative analysis of submission metrics for three programming exercises of the course

configuration if Stager would be integrated into the exercise management system, where the instructor sets up the programming exercise. Then Stager would automatically know the submission deadline, the latest commit of the instructor in the code template, and the remote repository URL. This would make the use of Stager easier and seamlessly.

4.4 Limitations

Our experience report only included three exercises that used Stager for code reviews. It would be interesting to analyze the concrete time-savings with a comparison and to use Stager throughout the whole course. While we have first indications, we did not evaluate whether the quality of the reviews improved through the use of Stager.

In addition, Stager's implementation currently has the following limitations: (1) Reviewers have to manually search for each student repository's key the first time they use Stager, before being able to use Stager for the remaining steps. The previously mentioned integration of Stager into an exercise management system would overcome this step. (2) For every exercise, the *config.json* file has to be changed accordingly with the deadline, URL-schema, and commit of the instructor. This could also be adapted to be automatically included when creating exercises by means of an exercise management system. (3) Reviewers have to install Stager on their computer and start it via a double-click or the command line interface. A web-based solution or a plugin into an IDE (e.g. Eclipse) in which the reviewers import the code would provide a more user-friendly experience.

5 Conclusion

Manual code reviews are important for the learning experience of students. While automatic tests can find typical problems and check whether code works as intended, they cannot find all problems, code smells, and implementation issues. Automatic assessment imposes certain solutions on the students and might limit their creativity. Stager supports code reviewers by automating steps in the manual assessment of programming exercises to reduce effort for the preparation and the conduction of code reviews. Stager downloads multiple students' submissions, renames folders and projects, filters out late submissions, and fixes typical white space problems. All commits of one student are combined into one discrete change-set that is easier to review. Code reviewers can better distinguish between the submissions of multiple students and identify students' contributions more quickly.

Our experience in a course with 1600 students and 45 teaching assistants shows that Stager reduced the reviewing effort and time for teaching assistants. The reviewers used the saved time to write better reviews and give more detailed feedback to the students. This improved the student's learning. A quantitative analysis in three programming exercises shows that Stager identifies several late submissions and fixes many white space issues.

Stager is free, open source, and available under the MIT license, so that other instructors can use it in their courses[18]. We will continue the development and aim to integrate the tool into the automated assessment system ArTEMiS [Krusche and Seitz, 2018]. Our future work also includes the integration of code quality metrics to support the actual code assessment. This could make it easier for reviewers to spot code quality issues in the students' solutions and be included, e.g. as a text file, into the feedback pipeline.

In addition, we would like to evaluate the quality of the code reviews when using Stager compared to pure manual reviews with respect to the completeness, helpfulness, and understandability of the review. Depending on the results of this evaluation, we could integrate strategies to semi-automatically propose common code review feedback. Automatic suggestions would further reduce the effort of reviewers but allow them to tailor these suggestions to the concrete situation.

References

[Ala-Mutka 2005] ALA-MUTKA, Kirsti M.: A Survey of Automated Assessment Approaches for Programming Assignments. In: *Computer Science Education* 15, pages 83–102, 2005.

[Cerioli and Cinelli 2008] CERIOLI, Maura ; CINELLI, Pierpaolo: GRASP: Grading and Rating ASsistant Professor. In: *Proceedings of the Informatics Education Europe III Conference*, 2008.

[18] https://github.com/arubacao/stager

[Chen 2004] CHEN, P. M.: An automated feedback system for computer organization projects. In: *IEEE Transactions on Education* 47, pages 232–240, 2004.

[Gerdes et al. 2017] GERDES, Alex ; HEEREN, Bastiaan ; JEURING, Johan ; BINSBERGEN, L. T. van: Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. In: *International Journal of Artificial Intelligence in Education* 27, pages 65–100, 2017.

[Heckman and King 2018] HECKMAN, Sarah ; KING, Jason: Developing Software Engineering Skills Using Real Tools for Automated Grading. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 794–799, 2018.

[Insa and Silva 2015] INSA, David ; SILVA, Josep: Semi-Automatic Assessment of Unrestrained Java Code: A Library, a DSL, and a Workbench to Assess Exams and Exercises. In: *Proceedings of the Conference on Innovation and Technology in Computer Science Education*, pages 39–44, 2015.

[Jackson 2000] JACKSON, David: A semi-automated approach to online assessment. In: *SIGCSE Bulletin* 32, pages 164–167, 2000.

[Jackson and Usher 1997] JACKSON, David ; USHER, Michelle: Grading Student Programs Using ASSYST. In: *Proceedings of the 28th Technical Symposium on Computer Science Education*, pages 335–339, 1997.

[Knobelsdorf and Romeike 2008] KNOBELSDORF, Maria ; ROMEIKE, Ralf: Creativity As a Pathway to Computer Science. In: *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, pages 286–290, 2008.

[Krusche et al. 2017a] KRUSCHE, Stephan ; BRUEGGE, Bernd ; CAMILLERI, Irina ; KRINKIN, Kirill ; SEITZ, Andreas ; WÖBKER, Cecil: Chaordic Learning: A Case Study. In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering Education and Training Track*, pages 87–96, IEEE, 2017.

[Krusche and Seitz 2018] KRUSCHE, Stephan ; SEITZ, Andreas: ArTEMiS: An Automatic Assessment Management System for Interactive Learning. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 284–289, 2018.

[Krusche et al. 2017b] KRUSCHE, Stephan ; SEITZ, Andreas ; BÖRSTLER, Jürgen ; BRUEGGE, Bernd: Interactive Learning: Increasing Student Participation through Shorter Exercise Cycles. In: *Proceedings of the 19th Australasian Computing Education Conference*, pages 17–26, 2017.

[Krusche et al. 2017c] KRUSCHE, Stephan ; VON FRANKENBERG, Nadine ; AFIFI, Sami: Experiences of a Software Engineering Course based on Interactive Learning. In: *Tagungsband des 15. Workshops "Software Engineering im Unterricht der Hochschulen"*, pages 32–40, 2017.

[Lawrance et al. 2013] LAWRANCE, Joseph ; JUNG, Seikyung ; WISEMAN, Charles: Git on the Cloud in the Classroom. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, pages 639–644, 2013.

[McCracken et al. 2001] MCCRACKEN, Michael ; ALMSTRUM, Vicki ; DIAZ, Danny ; GUZDIAL, Mark ; HAGAN, Dianne ; KOLIKANT, Yifat Ben-David ; LAXER, Cary ; THOMAS, Lynda ; UTTING, Ian ; WILUSZ, Tadeusz: A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. In: *Working Group Reports on Innovation and Technology in Computer Science Education*, pages 125–180, 2001.

[Pieterse 2013] PIETERSE, Vreda: Automated Assessment of Programming Assignments. In: *Proceedings of the 3rd Computer Science Education Research Conference*, pages 45–56, 2013.

[Poženel et al. 2015] POŽENEL, Marko ; FÜRST, Luka ; MAHNIČ, Viljan: Introduction of the automated assessment of homework assignments in a university-level programming course. In: *38th International Convention on Information and Communication Technology, Electronics and Microelectronics*, pages 761–766, IEEE, 2015.

[Robins et al. 2003] ROBINS, Anthony ; ROUNTREE, Janet ; ROUNTREE, Nathan: Learning and teaching programming: A review and discussion. In: *Computer Science Education* 13, pages 137–172, 2003.

[Staubitz et al. 2015] STAUBITZ, Thomas ; KLEMENT, Hauke ; RENZ, Jan ; TEUSNER, Ralf ; MEINEL, Christoph: Towards practical programming exercises and automated assessment in Massive Open Online Courses. In: *Teaching, Assessment, and Learning for Engineering*, pages 23–30, IEEE, 2015.

[Striewe and Goedicke 2013] STRIEWE, Michael ; GOEDICKE, Michael: Analyse von Programmieraufgaben durch Softwareproduktmetriken. In: *Tagungsband des 13. Workshops "Software Engineering im Unterricht der Hochschulen"*, pages 59–68, 2013.

8.8 An Interactive Learning Method to Engage Students in Modeling

This conference paper is the most significant contribution to the habilitation. Engaging students in modeling is challenging, especially in very large introductory courses. The paper describes an easy-to-use online modeling editor Apollon integrated into Artemis. Based on interactive learning, students learn modeling in guided tutorials in the lecture right after the theory is introduced and deepen their skills in group work and homework exercises. The instructors applied interactive learning in modeling in the introductory course Introduction to Software Engineering (EIST) with more than 1000 students. An empirical evaluation of the method demonstrated that the students learning outcome in modeling improved by up to 87 % compared to the previous year without interactive learning. As a result, students are motivated to use models in their future projects and understand how to approach problems with models. This publication includes the empirical evidence found in a largescale study that interactive learning can significantly improve students' learning outcomes.

Authors	**S. Krusche**, N. von Frankenberg, L. Reimer and B. Bruegge
Conference	42nd International Conference on Software Engineering
Publisher	ACM
Pages	11
Type	Conference: Full Research Paper
Review	Peer Reviewed (3 Reviewers)
Year	2020
Citation	[KvFRB20]
DOI	https://doi.org/10.1145/3377814.3381701

2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)

An Interactive Learning Method to Engage Students in Modeling

Stephan Krusche
krusche@in.tum.de
Technical University of Munich
Munich, Germany

Nadine von Frankenberg
nadine.frankenberg@in.tum.de
Technical University of Munich
Munich, Germany

Lara Marie Reimer
laramarie.reimer@tum.de
Technical University of Munich
Munich, Germany

Bernd Bruegge
bruegge@in.tum.de
Technical University of Munich
Munich, Germany

ABSTRACT

Modeling is an important skill in software engineering. However, it is often not tangible for students and not appreciated. Students prefer coding because they receive immediate feedback from the compiler. Engaging students in modeling is difficult, especially in large introductory courses.

We have developed an interactive learning method for modeling which is based on an easy to use online editor. Students learn modeling in guided tutorials in the lecture right after the theory is introduced and deepen their modeling skills in group work and homework exercises. This learning method was applied in a large introductory course with more than 1000 students.

An empirical evaluation of the method demonstrated that the students' learning outcome in modeling improved significantly by up to 87 %. Students are motivated to use models in their future projects and understand how to approach problems with models. The use of interactive models in programming exercises improves their understanding of the taught concepts.

CCS CONCEPTS

• **Social and professional topics → Software engineering education**; • **Applied computing → Interactive learning environments**; **Learning management systems**.

KEYWORDS

Software Engineering, Education, Learning Management System, Online Editor, Modeling, Learning Success, Interactive

ACM Reference Format:
Stephan Krusche, Nadine von Frankenberg, Lara Marie Reimer, and Bernd Bruegge. 2020. An Interactive Learning Method to Engage Students in Modeling. In *Software Engineering Education and Training (ICSE-SEET'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3377814.3381701

ICSE-SEET'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7124-7/20/05…$15.00
https://doi.org/10.1145/3377814.3381701

1 INTRODUCTION

Software engineering requires practical application of knowledge [6, 10, 34]. Modeling a system in the Unified Modeling Language (UML) is an important practical skill to facilitate communication between software engineers. Students typically get in touch with UML modeling in undergraduate courses in university. While examples and exercises play a central role in the early phases of cognitive skill acquisition [37], it is time consuming for instructors to create and assess modeling exercises that stimulate all cognitive skills, including creativity. Carefully developed and integrated examples improve the learning outcome [35, 36]. Providing individual feedback and allowing students to improve their knowledge through formative assessments are essential elements in learning [14, 15], so that students can improve their skills.

However, with the rising numbers of students in introductory courses, it is nearly impossible for instructors to teach the creative aspects of modeling and to provide individual feedback. Courses with hundreds of students create enormous efforts for instructors, especially in the correction of exercises and exams, and make it impossible to interact with each student on an individual level [26]. Modeling is a creative task where multiple solutions are possible and it is difficult to judge immediately whether a solution is correct or not [19, 24, 25]. It is not feasible to define all possible correct solutions for a modeling exercise. The choice of the modeling tool has many implications. The use of existing modeling tools such as Visual Paradigm, Gliffy, or draw.io overwhelms and demotivates students due to their complexity. Assessing models with these tools is not possible.

To overcome these problems, we designed an interactive learning method, where students start modeling in an easy to use online editor that focuses on learnability. In this online editor, students receive individual feedback directly next to the model elements in order to avoid media breaks. Students use modeling techniques such as UML to abstract ideas, reduce complexity, improve communication, and to solve concrete problems. The instructor of the course introduces different types of UML diagrams when they are first needed in the software lifecycle.

Through guided tutorials and in-class exercises, students learn the modeling notation of the respective UML diagram type in the lecture. They further practice modeling in tutor exercise sessions. Homework enables students to deepen their modeling skills in self-study. The presentation of their own modeling solution in the accompanying tutor exercise sessions improves communication skills,

which are essential in software engineering. Individual feedback on homework allows students to measure their learning progress and improve their skills further. In this paper, we investigate the following hypotheses:

- **H1 Learning success**: Interactive learning methods improve the learning success in modeling.
- **H2 Engagement**: Integrating modeling throughout the software lifecycle increases student engagement in modeling.
- **H3 Understanding**: Interactive models in programming exercises improve students' understanding of the concepts taught.

The remainder of the paper is structured as follows: Section 2 describes related work in relation to relevant learning concepts. Section 3 describes the interactive learning method and the tool support in detail. In Section 4, we present a large software engineering introduction course with more than 1000 students in which we used the interactive learning method. Section 5 outlines the evaluation and its results consisting of an online questionnaire, data analysis and a quasi experiment. Section 6 discusses the advantages and disadvantages of the proposed learning method. Section 7 concludes the paper and proposes future work.

2 RELATED WORK

Software engineering is an engaging, interactive, and collaborative activity [38]. In the educational sector, creating engaging and interactive curricula is an important topic. Content delivery and content exercise is often divided into lectures and exercises which can lead to knowledge gaps. This concept is described by Ebbinghaus's forgetting curve which follows the psychological proposition "... who learns quickly also forgets quickly" [12] and illustrates the knowledge retention rate over time [31]. The forgetting curve implies that after learning new information, within the first 24 hours, there is a retention loss of 40 % to 60 % – if this information is not practiced in short cycles.

Several pedagogical concepts of learning aim at closing this gap of content delivery and practice. Common concepts include: blended learning, a combination of E-learning with the traditional lecturing style that offers the course content through multiple delivery channels [4]; experiential learning, a methodology where students learn from experience [17]; active learning where students participate actively in all learning activities, rather than solely listening passively to a lecturer; and interactive learning, which is based on active and blended learning and further engages students in interactive activities using technology [8, 22].

Engaging students throughout the whole learning process has proved to improve the retention rate. Think-Pair-Share (TPS) is an approach where students work on a problem first individually, then in small groups, and eventually with the whole class [28]. In a study that involved a large introductory programming course, Kothiyal et. al found that a TPS approach yielded 83 % of student engagement [18]. Krusche et. al propose to introduce multiple short iterations between teaching and exercising concepts by combining lectures and exercises into interactive classes [21, 23], and report similar results as Kothiyal, an average of 80 % of student engagement throughout the semester.

Interactive learning concepts involve the direct hands-on application of knowledge, which is important for learning software engineering concepts [34]. The aforementioned concepts are particularly relevant for teaching modeling to students, considering modeling being an informal and creative activity in the software engineering process [7, 11, 19], rather than a concept that should be learned by heart. Some methodologies following this concept focus on collaborative learning environments for UML modeling [1, 9] which may help students to lower the entrance barrier to modeling. Others target the quality assessment and correctness of models [24, 27, 29, 30]. Few approaches propose tools for assessing models or offer scientific evidence on using such tools in practice.

3 LEARNING METHOD

Engaging students in modeling can be challenging. In this section, we describe an interactive learning method that integrates modeling into software lifecycle activities and guides students through their modeling experience.

The learning method is based on active learning [5] and combines lectures and exercises into small iterations to overcome the artificial separation of theory and practice often applied at universities. Instead, instructors introduce small chunks of theory and allow students to exercise them directly. Each theory-exercise cycle consists of five steps [22]:

Theory: The instructor explains a new modeling concept, e.g., a diagram type or modeling technique, and describes the theory behind while students listen and try to understand it.

Example: The instructor shows examples. Students relate the learned modeling concepts and techniques to a concrete situation.

In-class exercise: Students apply the concept in an exercise, e.g., a guided tutorial, and submit their solution.

Feedback: Students receive individual feedback on their own solutions. The instructor provides an example solution, e.g., within the guided tutorial, and explains it to the students to prevent misconceptions [16]. The instructor can also show exemplary student solutions and can discuss their strengths and weaknesses.

Reflection: The instructor facilitates a discussion about the theory and the exercise so that the students reflect on their first experience with the new modeling concept or modeling technique. This can be done, e.g., by discussing best practices, repeating advantages of a technique, or showing how the exercise instantiates the abstract concept.

3.1 Deepening the Learning Content

In addition to in-class exercises, the learning method includes group work exercises and homework. Small tutor exercise sessions with 15-20 students are supposed to deepen the understanding of the concepts explained in the lecture by means of suitable group or team exercises. Students experience the application of the taught concepts and methods with the help of manageable problems in the different phases of software engineering.

Homework exercises enable students to deepen their knowledge in self-study. The teaching concept motivates the students to participate in homework exercises by providing individual feedback and by granting exercise points that can be used to improve the final grade of the course. Individual feedback on homework allows students to measure learning progress and improve their skills.

Students present their solution to group work and homework exercises in the tutor exercise sessions. The presentation of the solution improves communication skills, which are essential in software engineering.

3.2 Tool Support with Artemis and Apollon

Artemis is an exercise system with individual feedback that supports interactive learning and is scalable to large courses [20]. It is open source[1] and used by multiple universities and courses. Artemis integrates an online modeling editor Apollon that is open source[2] and available as standalone and free web application[3]. Apollon supports seven UML diagrams: class diagrams, object diagrams, activity diagrams, use case diagrams, communication diagrams, component diagrams and deployment diagrams. It is lightweight and easy to use to lower the entrance barrier of digital modeling. It focuses on the learning experience of students. Figure 1 shows an example of a UML communication diagram. Students drag the model elements from the right into the diagram and double-click on an element to edit it in a small pop-up. They can drag and drop relationships (e.g. control flow) between model elements.

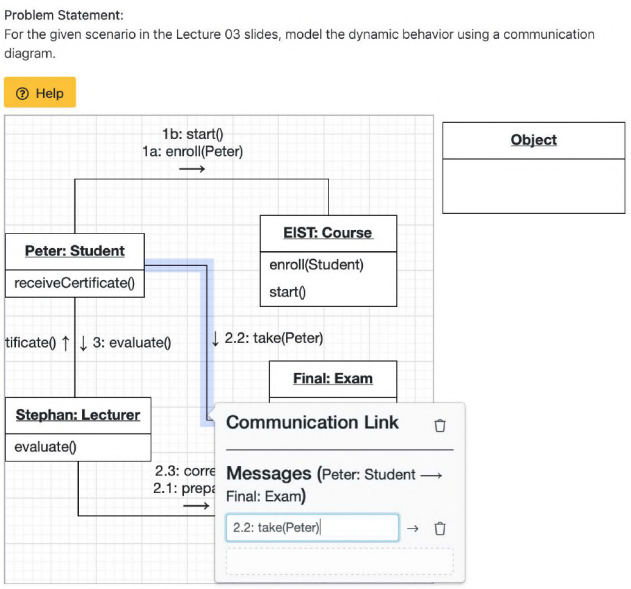


Figure 1: The online modeling editor Apollon is integrated into Artemis and supports the easy creation and assessment of digital models.

Instructors and teaching assistants assess models and provide feedback directly in Apollon. They double click on a model element and assess it in a popup with a score in points and with additional feedback comments to explain why a model element is correct or wrong. In addition, they can provide general feedback about the whole model or missing elements. Students can see this feedback directly in place next to the model elements and learn from it.

[1]https://github.com/ls1intum/Artemis
[2]https://github.com/ls1intum/Apollon
[3]https://apollon.ase.in.tum.de

Artemis includes an online code editor with interactive and dynamic exercise instructions on the right [20]. Figure 2 shows a screenshot. Interactive instructions change their color depending on the progress of students. Already completed tasks and correctly implemented model elements are marked in green, incomplete tasks and not yet implemented model elements are marked in red. This helps students to identify which parts of the exercise they have already solved correctly and improves the understanding of the source code on the model level. When they submit their current solution, the interactive instructions update dynamically.

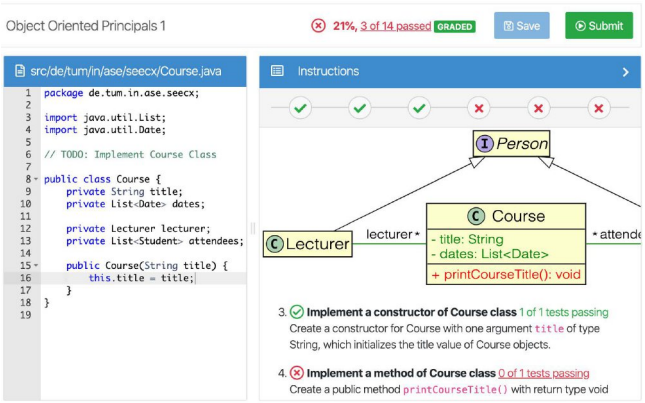


Figure 2: Online code editor with interactive instructions on the right side which include an interactive UML class diagram that changes the color from red to green after successfully completing the corresponding programming task.

4 COURSE

This section describes the undergraduate software engineering course SE1[4] at the Technical University of Munich. The instructors of SE1 introduce students to UML modeling using the interactive learning method described in Section 3.

The learning objectives of SE1 are to familiarize students with relevant concepts, workflows, and methods of software engineering and to apply them in all phases of software engineering projects. This includes analyzing and evaluating problems, e.g., modeling the problem, reusing classes and components, and testing the software. With respect to UML, students learn to communicate using models. They learn how and when to apply which model. They understand the relationship between modeling and programming and learn to abstract. Students learn to model and implement concrete problems in software engineering, for instance with the help of design patterns.

SE1 is a mandatory Bachelor's course offered in the second semester for a heterogeneous group of students from the fields of computer science, business informatics, and business, as well as students from other fields. A prerequisite of the course is that the students have basic programming experience, such as having successfully completed an introductory course in computer science (e.g., CS1). Course instructors use constructive alignment [2] to

[4]The course is called "Introduction to Software Engineering"

align teaching and assessment with the course objectives. For each lecture, a set of learning goals is defined based on the six cognitive skills in Bloom's taxonomy [3]. The course focuses in particular on higher cognitive skills so that students learn to apply the concepts in concrete situations. Students cannot pass the course by simply memorizing the course material.

4.1 Organization

One lecturer and two exercise instructors organize and teach the course with the help of around 45 student tutors. The tutors are bachelor and master students who successfully completed the course in previous years. The course takes place in the summer semester over 12 weeks. Table 1 shows the course content together with the UML models that are taught in the respective lecture. Around 1600 students register for the course. There is no lecture hall with enough seats for all students. The course uses a live stream and broadcasts the lecture to two additional overflow lecture halls. Most students either participate actively in the main lecture hall or watch the live stream at home. Therefore, the overflow lecture halls are closed after a few weeks. All students (within the main lecture hall, in overflow lecture halls and in the live stream) can ask questions using Slack[5]. Tutors answer these questions directly or pass on a question to the lecturer to repeat and answer it for all students.

Week	Content	UML Model
1	Introduction	Class
2	Model-based SE	Use Case, Class
3	Requirements Analysis	Object, Communication
4	System Design I	Component, Deployment
5	System Design II	Component, Deployment
6	Object Design I	Class
7	Object Design II	Class
8	Model Transformation and Refactoring	State Chart
9	Software Lifecycle Modeling	Activity
10	Software Configuration Management	Activity
11	Testing	Class
12	Project Management	Class, Activity

Table 1: Course Schedule: SE1 lasts 12 weeks. Each lecture includes specific UML models.

4.2 Design

Large courses present the challenge of keeping students motivated throughout the semester (without, e.g., enforcing mandatory attendance). Students are easily distracted by off-topic conversations with other students or social media, and stop paying attention to the lecture. To deal with such situations, the course includes interactive elements to activate the students and to keep students engaged throughout the course. The interactive components include in-class exercises, in-class quizzes, and group exercise sessions.

The study program of the university does not allow to include weekly assignments in the calculation of the final grade. Therefore, the course uses a bonus system that motivates students to participate in the course and its exercises: students can earn bonus points for completing in-class and homework exercises successfully. They need to present their homework twice in tutor exercise sessions to get the bonus applied.

If they pass the final exam, their exercise points are mapped to exam points that are then added to their final exam score to improve it. The German grading system consists of marks between 1.0 (similar to *A* in the US grading system) and 5.0 (similar to *F* in the US grading system), with 1.0 being the highest grade and 4.0 (similar to *D* in the US grading system) being the pass grade. For instance, if students score 30 % of the bonus points, they receive additional 3.0 points on top of their exam score, which improves their final grade by 0.3. If they score 100 % of the bonus points, they can receive a total bonus of 1.0, for instance, they can improve from a 2.3 to a 1.3. Students have reported that this increases their motivation to actively participate in the exercise system.

4.3 Exercises

The course includes quizzes, programming, modeling, and text exercises. In-class exercises include quizzes, to recapture previously learned content. They also include programming and modeling exercises as guided tutorials. Tutors help with student questions and problems during the in-class exercises. Group exercises mainly encompass modeling exercises, but also small programming exercises and text exercises that the students work out together during their group exercise sessions in small teams. Homework assignments include modeling, programming, and text exercises and enable students to deepen their knowledge in self-study.

4.3.1 Modeling Exercises. Students model a solution to concrete problems using UML. Modeling exercises stimulate higher cognitive skills and force students to analyze, evaluate and create. Apollon supports UML class, object, activity, use case, communication, component, and deployment diagrams. As shown in Table 1, each lecture includes different UML model types, aligned with the taught content. Figure 3 shows an example for a modeling exercise. Students drag and drop the model elements into the canvas, can add attributes, methods, and define associations between them. The advantage of Apollon is that students cannot use a UML element other than the ones specified for the specific model type.

4.3.2 Quiz Exercises. Students repeat already learned content during lectures and test their knowledge. They stimulate lower cognitive skills such as remembering and understanding the concepts. A quiz question can either be a multiple choice (MC) question, a drag and drop (DnD) question, or a short answer question. For questions related to modeling, the quizzes include MC and DnD questions where students drag elements to predefined spots on the canvas.

4.3.3 Programming Exercises. Students learn to make connections and see differences between models and their implementation in programming exercises. This stimulates their cognitive skills and students learn to apply the knowledge when implementing source code. A UML class diagram, e.g., represents the general structure of the source code and can be used as interactive problem statement in Artemis. Red model elements indicate that they are not implemented correctly, whereas green elements indicate correctly

[5]Slack is a cloud-based instant messaging platform: https://slack.com

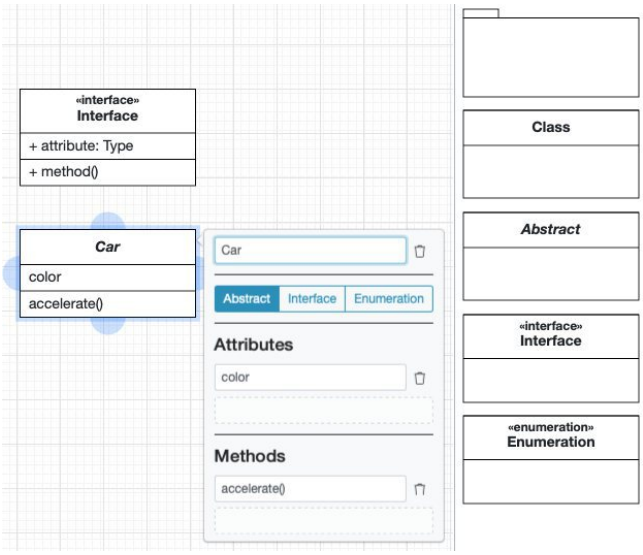


Figure 3: An example for a modeling exercise in Apollon.

implemented ones. This further helps students to understand what UML models should contain and what should be left out.

4.3.4 Text Exercises. Students need to answer questions about the learned concepts by writing open text responses. They, e.g., need to explain similarities and differences between design patterns in their own words and describe concrete situations how design patterns can be used. These exercises stimulate analysis and evaluation skills.

4.4 Communication

The course uses Slack as communication tool to facilitate discussions between students and teaching staff. Using instant messaging lowers the entrance barrier for students to ask questions, because it feels familiar to communicate (in reference to, e.g., social media chats) and students ask more questions if they notice that other students do the same. Students can communicate with each other and send direct messages to tutors and instructors in case they have a question or require help during the lectures and at any other time. Slack offers the ability to use channels with specific purposes:
#announcements — Instructors post course-wide announcements (students cannot post here), e.g., reminders that a lecture is cancelled on a public holiday
#organization — Questions about the organization of the course
#lecture — Questions regarding the lecture slides
#exercise — Questions regarding the exercises

The instructors further encourage students to answer questions themselves. This increases a 'sense of belonging' (which is hard to achieve in such a large setting), when students communicate with each other, but can also deepen their understanding of a topic, e.g., in discussions that pursue questions. Students receive fast replies, which increases the interactivity. Tutors help to moderate discussions. They ensure a positive atmosphere, reprimand and prevent bullying. They answer questions and point students to previously asked questions, if it has already been asked before.

4.5 Tutor Exercise Sessions

45 tutors hold 80 weekly occurring tutor exercise sessions, each with around 20 students. The main focus for tutors is to activate the students in these sessions, to moderate discussions and to explain the learned concepts again in case the students ask questions. Tutors have the following responsibilities: attend a weekly tutor meeting with the instructors[6], assess exercises and hold one or two tutor exercise sessions per week. Tutors also help with moderating the Slack channels, answer questions on Artemis, help students during in-class exercises, or review slides and exercise content.

In the tutor exercise sessions, students apply the knowledge acquired in the lecture. Each tutor exercise session is structured as follows:

(1) **Review of previous lecture** [5 - 10 min]: students discuss the learning goals, outline, and summary.
(2) **Homework presentation** [30 - 45 min]: students present their solution to homework exercises. Tutors asks questions about the solution, point out typical mistakes and provide additional feedback.
(3) **Group work** [30 - 45 min]: Students work on predefined group exercises in groups (3-6 students).
(4) **Discussion of next homework** [5 - 10 min]: the new homework exercises are briefly discussed.

The tutor exercise sessions review a specific topic that was covered in the lecture before and prepare the students for the next homework assignment. They help to deepen the understanding of the taught concepts. Group exercises show the application of the learned methods with the help of concrete problems in the different phases of software engineering. Homework assignments deepen the knowledge in self study. Students receive individual feedback on their homework submission, which allows them to measure their learning progress and improve their skills. The presentation of their own solution improves the communication skills of the students, an essential skill in software engineering.

For instance, in lecture *Object Design II*, the course covered the Strategy Design Pattern [13] by means of an example and the general structure. In the corresponding tutor exercise session, there was one group work, where students discussed the pattern's problem, solution, benefits, consequences, etc. In the subsequent group work, the students modeled a real-world example of the strategy pattern as a UML class diagram. This exercise was designed to teach students how to approach a concrete problem, how to analyze it, and how to model this problem. In one homework assignment, the students were given a similar problem: to model different encryption strategies as a UML class diagram, using the strategy pattern. In another assignment in a programming exercise, the students had to implement sorting algorithms using the strategy pattern.

4.6 Grading

While programming and quiz exercises are automatically evaluated, modeling and text submissions are graded manually. Each tutor grades about 25 submissions per exercise per week. Artemis offers a double-blind grading system, which opts for less bias while grading. Every week on Monday at noon, the homework is published. The

[6]Instructors discuss issues and present the next group work and homework.

students then have one week to create and upload their solutions. In the following week, students present their homework in the tutor exercise sessions. Tutors use example solutions and detailed grading criteria to assess the students' submissions and provide individual feedback. In the grading criteria, instructors point out that multiple solutions to modeling exercises can be correct.

When students are given sample solutions, they often do not think about their own solutions, but tend to take the sample solution as the single truth. To encourage self-reflection and revision, the example solutions are not distributed to the students. The feedback students receive about their solutions is crucial for them to understand how they can improve. As shown in Figure 4, the feedback for modeling exercises is comprised of the (1) points they receive for one concrete element, (2) feedback for this element, and (3) general feedback regarding the whole model.

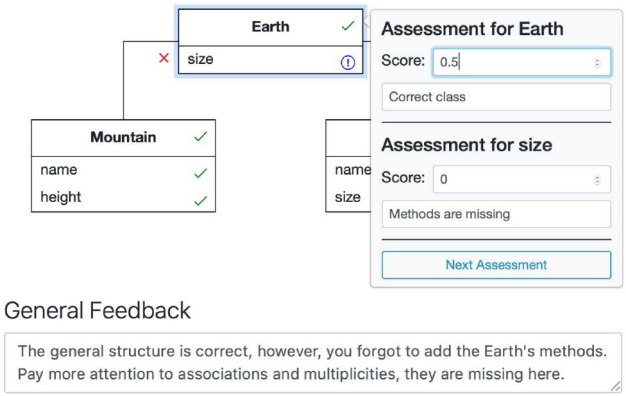


Figure 4: An example for a modeling exercise with individual feedback.

5 EVALUATION

This section presents the evaluation of the interactive learning method in the SE1 course in 2019 based on the hypotheses in Section 1. We describe the research method, present the results and discuss the findings and limitations.

5.1 Research Method

In order to test our hypotheses, we applied the following three research methods:

(1) **Online Questionnaire**: we created an online questionnaire and asked all participants of the SE1 course in 2019 to participate.
(2) **Data Analysis**: we analyzed the participation and exercise performance of the students in the modeling exercises in the SE1 course in 2019.
(3) **Quasi Experiment**: we compared the results of modeling tasks in the SE1 exams from 2018 and 2019. In 2018, the SE1 course did not include the interactive learning method for modeling exercises and therefore serves as the control group.

5.1.1 Online Questionnaire. All participants of the course SE1 in 2019 were asked to complete a questionnaire about their experience with the modeling exercises. The questionnaire consisted of four main parts, each containing several questions: (1) demographic information, in particular field of study, current semester, and the student's achieved prerequisites for SE1; (2) previous modeling and programming experience (before taking SE1); (3) participation in SE1; (4) modeling in SE1, including motivation, the interactive learning method, and the tools being used.

5.1.2 Data Analysis. We analyzed the results of all modeling exercises in SE1 in 2019. This includes a total of 17 modeling exercises; 9 of which were conducted as in-class exercises, the other 8 as homework. Each of the modeling exercises was assigned a difficulty level[7] as well as a score. To analyze each modeling exercise, we have created a dataset that contains all participating students as well as their individual scores per exercise. Based on the scores of the participating students, we calculated the average success rate in % of the total score.

5.1.3 Quasi Experiment. We carried out a quasi experiment with post-testing of two student groups, i.e., students who took SE1 in 2018 and students who took SE1 in 2019, by comparing their scores in the modeling tasks of the final exam. Both course instances in 2018 and 2019 had the same learning goals, the same course schedule with the same content and the same exercise structure except for modeling exercises: In 2018, SE1 did not use the interactive learning method and instead relied on practicing modeling only in homework. In 2019, SE1 used the interactive learning method and introduced in-class modeling exercises. In terms of the quasi experiment, the interactive learning method is the intervention. Apart from that, there were no substantial differences in other variables.

The control group is comprised of the 2018 students, the experimental group of the 2019 students. We did not execute a pre-test. Our assumption was that students from both groups had similar knowledge regarding modeling before taking part in the SE1 course, mainly because the majority of both student groups was comprised of second-semester bachelor students, with both groups following the same curriculum. In both years, the course was attended by over 1000 students, so that a normal distribution of the results can be assumed. Both exams included five similar modeling tasks:

(1) **Functional model**: Create a UML use case diagram based on a given problem statement (easy)
(2) **Structural model**: Create an analysis object model using a UML class diagram based on a given problem statement (medium)
(3) **Dynamic model**: Create an UML activity diagram (2018) / UML communication diagram (2019) based on a given problem statement (medium)
(4) **Architecture**: Create a UML communication diagram of an architectural style (2018, medium) / model the architecture based on a given problem statement using a UML component diagram (2019, hard)
(5) **Model refactoring**: Analyze an existing model, propose a model refactoring and explain the reasoning (easy)

In the post test, we compared these five modeling tasks in two-sample one-tailed t-tests to evaluate, whether the 2019 students performed significantly better than the 2018 students. For all model tasks, the null hypothesis H_0 is that the 2019 students performed

[7] Possible difficulty levels: **E** = easy, **M** = medium, or **H** = hard

less or equal as compared to the 2018 group with a significance level of $\alpha = 0.01$. H_1 hypothesizes that the 2019 results are better than the results from 2018. 1128 students completed the exam in 2018, 1225 completed the exam in 2019. To make the results comparable (two tasks differed by one point) we calculated the mean and standard deviation as relative values.

5.2 Results

This subsection shows the results of the three research methods.

5.2.1 Online Questionnaire. In total, 954 students participated in the online questionnaire (response rate: 68 %). 90 % of the students are enrolled in a Bachelor's program, while 10 % of the participants are enrolled in a Master's program. 69 % of the students take SE1 in their second semester, followed by 18 % forth-semester students.

The first question (Q1) refers to the experience the student's had with UML modeling before taking the SE1 course. Figure 5 depicts the answer distribution of Q1. 50 % of the participants stated that they have "somewhat experience", which means that they have modeled using UML once in a previous course, followed by 29 % of students students that stated that they have "little experience", which means that they had heard about UML models before. 17 % had no experience at all, only 4 % stated that they model regularly.

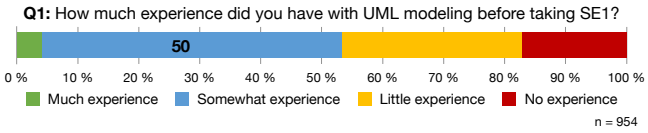


Figure 5: Limited modeling experience before SE1.

Q2 refers to the frequency of participation in a tutor exercise session. 52 % of the students stated that they always attend the tutor exercise sessions, while 27 % stated that they visit them very often. 12 % participate sometimes, 7 % rarely attend and 2 % never participate. Figure 6 depicts the distribution of answers.

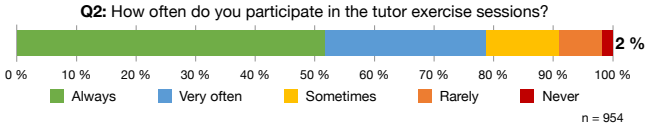


Figure 6: Regular participation in tutor exercise sessions.

In Q3, students were asked how often they submit their solutions to the modeling exercises. The majority, 74 %, stated that they submit always, 10 % submit very often. 9 % submit sometimes, while 5 % submit rarely, and 2 % have never submitted any modeling exercise. The results are shown in Figure 7.

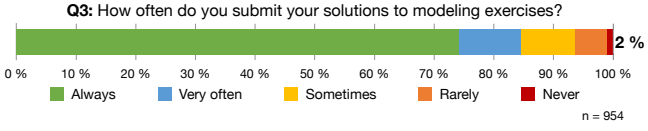


Figure 7: Regular submissions of modeling exercises.

Q4 stated that the interactive models used in Artemis programming exercises have helped the students to solve the exercises. The rating was done by using a 5-point Likert scale as shown in Figure 8. 54 % strongly agree with the statement, 36 % agree. 6 % have a neutral opinion on the statement, whereas 3 % disagree and 1 % of the participants strongly disagree.

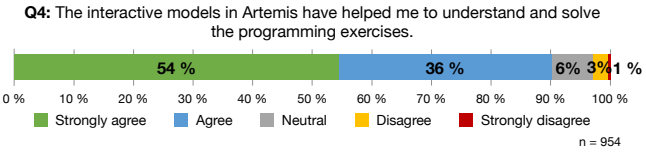


Figure 8: Interactive models in programming exercises.

Q5 asked whether students would use models in their future projects using a 5-point Likert scale (shown in Figure 9). 24 % of the students stated that they strongly agree, 51 % stated that they agree with this statement. 20 % have a neutral opinion. 4 % of the students disagree and 1 % strongly disagree.

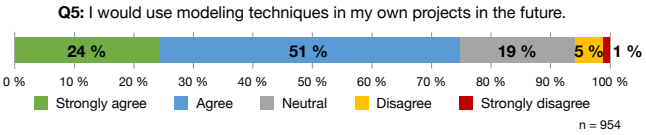


Figure 9: Use of modeling in future projects.

Q6 analyzed to which extent the students considered the different exercise types as helpful (shown in Figure 10). Homework exercises were rated most helpful for deepening and understanding modeling after the theory was introduced (37 % strongly agree and 47 % agree), followed by in-class exercises (28 % strongly agree, 49 % agree). Group works were considered least helpful, with 13 % of students strongly agreeing to the statement, 36 % agreeing, 31 % neutral opinions, 16 % disagreeing and 4 % strongly disagreeing.

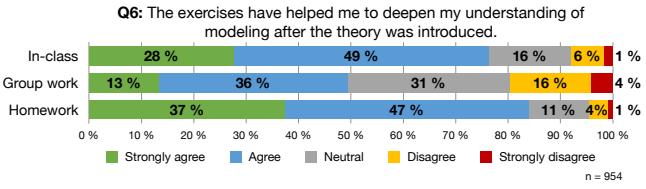


Figure 10: Helpfulness of exercise types to deepen the understanding of modeling.

Q7 asked the participants to state their opinion on different statements regarding the modeling exercises and concepts using a 5-point Likert scale. Figure 11 shows the results. The exercises and concepts especially helped the students to understand why to use models (31 % strongly agree, 53 % agree) and improve their modeling skills (32 % strongly agree and 53 % agree). 72 % state that modeling helped them to understand how to approach problems in software engineering.

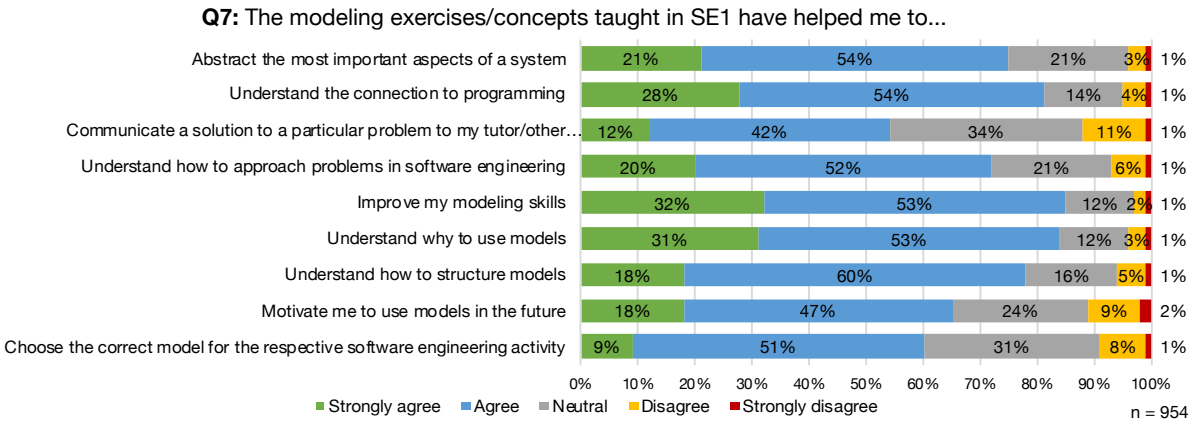


Figure 11: Helpfulness of modeling exercises in various aspects.

5.2.2 Data Analysis. In the second part of the evaluation, we analyzed data from the Artemis system regarding the modeling exercises conducted during the SE1 2019 course. SE1 included 17 modeling exercises, 9 of them were guided tutorials done in the lecture, 8 of them were executed by the students on their own as homework. The results are summarized in Table 2.

#	Exercise	Pts	Difficulty	Participations	Avg. Score
1	L1 Modeling Tutorial	4	E	1147	97 %
2	H1 Use Case Model	4	E	1116	55 %
3	H2 Analysis Object Model	6	E	1090	75 %
4	L2 Communication Diagram	6	E	1136	88 %
5	H3 Communication Diagram	7	M	992	54 %
6	L3 Model View Controller	4	H	1070	99 %
7	L4 Component Diagram	3	M	1049	98 %
8	H4 Choose a Design Pattern	6	M	899	83 %
9	H5 Strategy Pattern	5	M	886	91 %
10	L5 Model Refactoring	2	E	950	90 %
11	L6 Java to UML	2	E	898	67 %
12	H6 Tables to a Model	4	M	851	74 %
13	L7 Scrum as Activity Diagram	5	M	860	89 %
14	H7 Activity Diagram	8	M	884	82 %
15	H8 Build & Release Management Workflow	10	M	851	95 %
16	L8 Functional Model	4	E	819	62 %
17	L9 Analysis Object Model	6	M	810	74 %

Table 2: Students actively participate throughout the course. They score more points in in-class exercises (L) than in homework exercises (H) on average.

The table depicts the different exercises, including a unique identifier, e.g. L1, where "L" represents that the exercise was conducted during a lecture. Exercises conducted as Homework are marked with a leading "H". All exercises are assigned a difficulty, there were 7 easy exercises, 9 medium ones and 1 hard one in the area of software architectures. Each exercise in the table has a certain amount of points (between 2 and 10) that the students can achieve, as well as the number of participations and the average score achieved in %. On average, the guided exercises performed in-class received a higher average score (85 %) than the homework exercises, that the students had to solve on their own (76 %).

5.2.3 Quasi Experiment. We compared the exam results SE1 in 2018 against the results of SE1 in 2019. Both exams covered the same exercise types about functional, structural, dynamic, and architecture models as well as refactoring of an existing model. We calculated the average score the students reached in the exercises and compared them against each other. The results are shown in Figure 12.

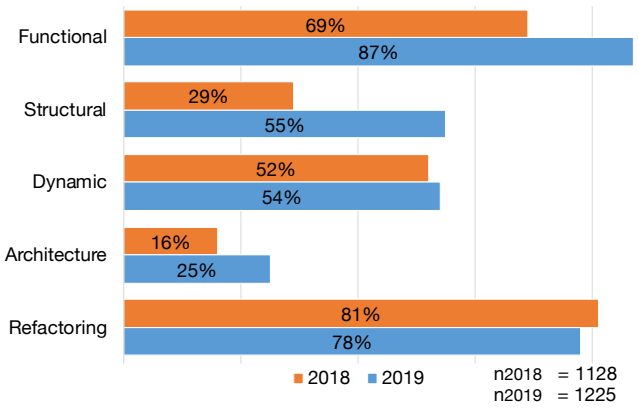


Figure 12: Students scored significantly better in 3 modeling tasks in the 2019 exam than in the 2018 exam.

Except for the exercise about refactoring, where students received 4 % fewer points on average in 2019 (78 %) than in 2018 (81 %), the students performed better in the SE1 2019 exam than in the 2018 exam in the analyzed modeling tasks. For the functional model, the average score is at 87 %, which is 26 % higher than in the 2018 exam, where students received 69 % of available points on average. In the structural model, students received 55 % of the points on average, which is 87 % more than in 2018, where they

received 29 % on average. The results for the dynamic model are 4 % higher (54 %) than in 2018 (52 %). The average score of the architecture exercise is 55 % higher in 2019 with 25 % compared to 16 % in 2018.

To evaluate the significance of the results, we performed a two-sample one-tailed t-test with a significance level of $\alpha = 0.01$. The results of the t-tests per exercise are depicted in Table 3. If the p-value for the exercise was below alpha, we rejected H_0 and considered the results from 2019 as significantly better than 2018. For the functional and structural models, we calculated a p-value of $2{,}2e^{-16}$ and for the architecture model a p-value of $6{,}4e^{-15}$, which means, that students were significantly better in the 2019 exam for the same model type in 2018. For the dynamic model, we calculated a p-value of 0,09, for the refactoring, the p-value was 0,99. This means, that the 2019 results were not significantly better than in 2018.

Model	$\overline{x}_{2018}$	$\overline{x}_{2019}$	σ_{2018}	σ_{2019}	p
Functional	0,689	0,867	0,334	0,237	$2{,}2e^{-16}$
Structural	0,293	0,549	0,265	0,310	$2{,}2e^{-16}$
Dynamic	0,519	0,538	0,327	0,365	0,09
Architecture	0,161	0,249	0,273	0,278	$6{,}4e^{-15}$
Refactoring	0,812	0,776	0,281	0,208	0,99
$n_{2018} = 1128$, $n_{2019} = 1225$					$\alpha = 0,01$

Table 3: T-tests of the SE1 2018 and SE1 2019 underline that the 2019 students were significantly better for functional, structural and architecture models.

5.3 Findings

SE1 serves as introduction to modeling for most students in the computer science curriculum. The modeling experience of the students before the course is low.

The results show that the interactive learning method motivates the students to attend the tutor exercise sessions and submit their modeling exercises. The performance in the modeling exercises indicates that, depending on the individual exercises, the amount of good and successful submissions is high, with an average score of 85 % for guided in-class exercises and 76 % in homework exercises. The main reason for the differences in the success rates between in-class and homework exercises lies in the fact that in-class exercises are usually performed together with the instructor. The instructor explains and performs the exercise together with the students using a projector. The students are able to see the correct steps of creating the model by following the instructions.

The in-class exercises serve as a tutorial and aim at giving the students a first hands-on experience in the specific UML model type. Later on, the students have to apply their new knowledge on their own in the homework. There are no instructions other than the exercise description, so that they have to find a solution without the help of the instructor. As this is usually more difficult and more error-prone than following tutorial steps, the average score is lower. When comparing the examinations of SE1 in the 2019 with the control group in 2018, the results in the modeling exercises were significantly better for the 2019 group, which used the interactive learning method.

> **Finding 1:** Interactive learning improves the learning success in modeling.

SE1 covers different UML models, that are being created throughout the whole software lifecycle. The teaching schedule and modeling exercises are aligned to cover different types of modeling in the corresponding lecture and following tutor exercise sessions and homework exercises. The students get an overview about when to use which UML model. The results show that most participating students submit their modeling exercises every week or at least on a regular basis.

The number of participations between the first (1147 participations) and the last modeling exercises (810 participations) stayed on a high level compared to traditional courses, where usually only about 25 % of the students still actively participate in the last lectures and exercises. The results of the questionnaire show, that students are highly motivated because of the interactive learning method and the use of modeling throughout the whole software engineering lifecycle. Higher motivation causes students to engage more in the lecture.

> **Finding 2a:** Integrating modeling throughout the whole software lifecycle increases student engagement in modeling.

> **Finding 2b:** Interactive learning increases student engagement in modeling.

Most of the participating students in the survey reported that the interactive learning method has helped them to approach different problems in software engineering. They understand the connection between modeling and programming. Especially the usage of the interactive UML class diagrams in the programming exercises helped them to better understand the programming exercises and find the correct solution.

> **Finding 3:** Interactive models in programming exercises improve the students' understanding of the taught concepts.

Most of the students also state they would use modeling techniques in their own projects in the future and that it helped them to better understand how to approach problems in software engineering. This is also emphasized by the success of the students in the 2019 exam compared to their results in the 2018 exam, where the interactive learning method was not used.

5.4 Threats to Validity

Internal validity: The evaluation does not measure all variables that could lead to better exam results. Existing knowledge, motivation, the exact wording of an exam question and other external factors might influence how students perform a modeling task in an exam. The internal validity of the results in the quasi experiment might be limited [32]. However, the online questionnaire and the data analysis support the findings.

External validity: The SE1 course is one specific example of courses, where modeling is taught. It is a mandatory course that is offered to many students, who may differ in their field of studies as well as in their previous experiences. We assume that the interactive learning method can be successfully applied in other software engineering courses and is generalizable. However, other study programs and regulations might make it difficult to adopt the approach.

Construct validity: The validity of the questionnaire might be affected by the wording of the questions or due to the fact that students like the approach of getting feedback, which does not necessarily improve their learning outcome. To limit the influence, we carefully designed the questions, used Likert scales as answer options and multiple researchers reviewed the wording. The measures in the quasi experiment and the data analysis support the findings.

6 DISCUSSION

While individual feedback for modeling exercises improves the students understanding and retention rate in terms of modeling and problem solving, as shown in Section 5.3, assessing each model submission increases the workload. Due to the amount of exercises to be assessed, it is hard to provide meaningful and understandable feedback comments, which can lead to partly incomprehensible and insufficient feedback. We have observed that tutors take more time at the beginning and take less time towards the end. Some tutors lack the required expertise or motivation which can result in inconsistent assessments in terms of fairness and correctness.

With either missing or insufficient feedback comments, students may get confused and would require a more extensive feedback to understand their faults. We have added a "request more feedback" functionality to Artemis, so that students can ask specific questions and another tutor answers them. A lack of feedback may also motivate students to participate more in discussions during the group work exercises, where they can ask their tutor for clarification. Artemis helps to assess student submissions with less bias than traditional methods, because student and tutor names are hidden. A random allocation of assessments also improves the fairness of assessments.

One of our main goals is to teach creativity in modeling which can further enhance problem solving abilities as well as understanding of object-orientated principles. For students new to modeling, this can be difficult. First, students are used to solve programming exercises where the compiler tells them exactly where they made a mistake, in real-time. This allows students to find and fix mistakes easily. Second, students are often used to precise problem statements and assume only one correct solution. They may have trouble in understanding that multiple solutions can be correct, which is the case in modeling. Third, students new to modeling may also lack the confidence in creating, presenting and discussing their own solutions with tutors or peer students. In our learning method, we approach these challenges by instructing tutors to actively encourage discussions and asking students to present their solutions to homework exercises at least twice during the tutor exercise sessions in order to qualify for the bonus system.

There are certain benefits for modeling on paper first, rather than using online editors, regarding syntax, semantic, and aesthetics [33]. Students may become used to using an online editor that already provides the syntax. In the paper-based exam, they would have to remember the syntax themselves and still draw their solutions on paper. Online editors, on the other hand, offer the function to change models easily, e.g., by adding a new element and replacing existing elements. On paper, this can quickly become unreadable and cumbersome.

Extrinsic motivation might also be a factor, because learning may be affected by assessments and bonus points for their work, rather than solely focusing on feedback and learning outcome. In our experience, especially students in their first year can have trouble in differing between learning and receiving feedback. The bonus might motivate those students more to participate in the exercises than the effect of practicing and improving their skills.

7 CONCLUSION

In this paper, we presented an interactive learning method that teaches modeling in an interactive setting, where students learn the problem solving aspects of modeling, can iterate through different types of models, and discuss their results in tutor exercise sessions. We present how Artemis and Apollon support students, instructors, and tutors in learning and teaching modeling throughout the whole software engineering lifecycle. We applied the interactive learning method in a large introductory software engineering course with more than 1000 students. Our empirical evaluation consisting of an online questionnaire, data analysis, and a quasi experiment shows that the interactive learning method improves the learning success of the students significantly and increases their motivation in modeling.

In the future, we want to integrate team projects with realistic problem statements and modeling tasks in our courses that are supported by the interactive learning method. Modeling with real-time synchronization in Apollon would allow students to collaborate on the model creation even in distributed settings. We also want to incorporate peer reviews for modeling exercises in Artemis so that students also have to grade models and provide feedback to other students because evaluating a model further improves the own modeling skills.

REFERENCES

[1] Mohammed Basheri, Liz Burd, and Nilufar Baghaei. 2012. Collaborative software design using multi-touch tables. In *4th International Congress on Engineering Education*. IEEE, 1–5.
[2] John Biggs. 2003. Aligning teaching and assessing to course objectives. *Teaching and learning in higher education: New trends and innovations* 2, 13–17.
[3] Benjamin Bloom, Max Engelhart, Edward Furst, Walker Hill, and David Krathwohl. 1956. Taxonomy of Educational Objectives: The Classification of Educational Goals.
[4] Curtis Bonk and Charles Graham. 2012. *The handbook of blended learning: Global perspectives, local designs.* John Wiley & Sons.
[5] Charles Bonwell and James Eison. 1991. *Active Learning: Creating Excitement in the Classroom.* ASHE-ERIC Higher Education Reports.
[6] Bernd Bruegge, Stephan Krusche, and Lukas Alperowitz. 2015. Software Engineering Project Courses with Industrial Clients. *ACM Transactions on Computing Education* 15, 4, 17:1–17:31.
[7] Bernd Bruegge, Stephan Krusche, and Martin Wagner. 2012. Teaching Tornado: from communication models to releases. In *Proceedings of the MODELS Educators' Symposium.* ACM, 5–12.
[8] Doug Buehl. 2017. *Classroom strategies for interactive learning.* Stenhouse Publishers.

[9] Weiqin Chen, Roger Heggernes Pedersen, and Øystein Pettersen. 2006. CoLeMo: A collaborative learning environment for UML modelling. *Interactive Learning Environments* 14, 3, 233–249.
[10] Thomas Connolly, Mark Stansfield, and Thomas Hainey. 2007. An application of games-based learning within software engineering. *British Journal of Educational Technology* 38, 3, 416–428.
[11] Dora Dzvonyar, Stephan Krusche, and Lukas Alperowitz. 2014. Real Projects with Informal Models. In *Proceedings of the MODELS Educators Symposium*. 39–45.
[12] Hermann Ebbinghaus. 2013. Memory: A contribution to experimental psychology. *Annals of neurosciences* 20, 4, 155.
[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*. Springer, 406–431.
[14] Richard Higgins, Peter Hartley, and Alan Skelton. 2002. The conscientious consumer: Reconsidering the role of assessment feedback in student learning. *Studies in higher education* 27, 1, 53–64.
[15] Alastair Irons. 2007. *Enhancing learning through formative assessment and feedback*. Routledge.
[16] Paul Kirschner, John Sweller, and Richard Clark. 2006. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist* 41, 2, 75–86.
[17] Alice Kolb and David Kolb. 2005. Learning styles and learning spaces: Enhancing experiential learning in higher education. *Academy of management learning & education* 4, 2, 193–212.
[18] Aditi Kothiyal, Rwitajit Majumdar, Sahana Murthy, and Sridhar Iyer. 2013. Effect of think-pair-share in a large CS1 class: 83% sustained engagement. In *Proceedings of the 9th annual international conference on International computing education research*. ACM, 137–144.
[19] Stephan Krusche, Bernd Brügge, Irina Camilleri, Kirill Krinkin, Andreas Seitz, and Cecil Wöbker. 2017. Chaordic Learning: A Case Study. In *39th International Conference on Software Engineering: Software Engineering Education and Training*. IEEE, 87–96.
[20] Stephan Krusche and Andreas Seitz. 2018. ArTEMiS: An Automatic Assessment Management System for Interactive Learning. In *Proceedings of the 49th Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 284–289.
[21] Stephan Krusche and Andreas Seitz. 2019. Increasing the Interactivity in Software Engineering MOOCs - A Case Study. In *52nd Hawaii International Conference on System Sciences*. 1–10.
[22] Stephan Krusche, Andreas Seitz, Jürgen Börstler, and Bernd Bruegge. 2017. Interactive learning: Increasing student participation through shorter exercise cycles. In *Proceedings of the 19th Australasian Computing Education Conference*. ACM, 17–26.
[23] Stephan Krusche, Nadine von Frankenberg, and Sami Afifi. 2017. Experiences of a Software Engineering Course based on Interactive Learning. In *Tagungsband des 15. Workshops Software Engineering im Unterricht der Hochschulen*. 32–40.
[24] Christian Lange and Michel Chaudron. 2004. An empirical assessment of completeness in UML designs. In *Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering*. 111–121.
[25] Christian Lange, Bart DuBois, Michel Chaudron, and Serge Demeyer. 2006. An experimental investigation of UML modeling conventions. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 27–41.
[26] Harold Leavitt and Bernard Bass. 1964. Organizational psychology. *Annual Review of Psychology* 15, 1, 371–398.
[27] WenQian Liu, Steve Easterbrook, and John Mylopoulos. 2002. Rule-based detection of inconsistency in UML models. In *Workshop on Consistency Problems in UML-Based Software Development*, Vol. 5.
[28] Frank Lyman. 1987. Think-pair-share: An expanding teaching technique. *Maa-Cie Cooperative News* 1, 1, 1–2.
[29] Jacqueline McQuillan and James Power. 2006. On the application of software metrics to UML models. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 217–226.
[30] Kashif Mehmood and Samira Si-Said Cherfi. 2009. Evaluating the Functionality of Conceptual Models. In *ER Workshops*.
[31] Jaap Murre and Joeri Dros. 2015. Replication and analysis of Ebbinghaus' forgetting curve. *PloS one* 10, 7.
[32] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. 2012. *Case Study Research in Software Engineering: Guidelines and Examples* (1st ed.). Wiley Publishing.
[33] Doris Schmedding and Anna Vasileva. 2017. Reviews-ein Instrument zur Qualitätsverbesserung von UML-Diagrammen.. In *SEUH*. 8–19.
[34] David Shaffer. 2004. Pedagogical praxis: The professions as models for postindustrial education. *Teachers College Record* 106, 7, 1401–1421.
[35] John Sweller and Graham A. Cooper. 1985. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction* 2, 1, 59–89.
[36] J. Gregory Trafton and Brian J. Reiser. 1993. *Studying Examples and Solving Problems: Contributions to Skill Acquisition*. Technical Report. Naval HCI Research Lab, Washington, DC, USA.
[37] Kurt VanLehn. 1996. Cognitive Skill Acquisition. *Annual Review of Psychology* 47, 513–539.
[38] Jim Whitehead. 2007. Collaboration in Software Engineering: A Roadmap. *FOSE* 7, 214–225.

8.9 Towards the Automation of Grading Textual Student Submissions to Open-ended Questions

The conference paper outlines an approach for the semi-automatic assessment of open-ended textual exercises using supervised machine learning. Based on manual assessments, the approach learns which aspects of solutions are correct and wrong. This concept supports multiple correct solutions instead of only one sample solution. It enables a wider variety of possible exercise types with automatic assessments, including creative aspects. The approach uses natural language processing: topic modeling to split student answers into text segments and language embeddings to transform those segments. It then applies clustering to group the text segments by similarity. Finally, it applies the same feedback to all text segments within the same cluster.

Jan Philip Bernius and Anna Kovaleva proposed the machine learning based approach for open-ended textual exercises. The author of this habilitation supervised Jan Philip Bernius throughout the development of the semi-automatic assessment approach and writing the paper.

Authors	J. Bernius, A. Kovaleva, **S. Krusche** and B. Bruegge
Conference	4th European Conference on Software Engineering Education
Publisher	ACM
Pages	10
Type	Conference: Full Research Paper
Review	Peer Reviewed (4 Reviewers)
Year	2020
Citation	[BKKB21]
URL	https://doi.org/10.1145/3396802.3396805

Towards the Automation of Grading Textual Student Submissions to Open-ended Questions

Jan Philip Bernius
Department of Informatics
Technical University of Munich
Munich, Germany
janphilip.bernius@tum.de

Anna Kovaleva
Department of Informatics
Technical University of Munich
Munich, Germany
anna.kovaleva@tum.de

Stephan Krusche
Department of Informatics
Technical University of Munich
Munich, Germany
krusche@in.tum.de

Bernd Bruegge
Department of Informatics
Technical University of Munich
Munich, Germany
bruegge@in.tum.de

ABSTRACT

Growing student numbers at universities worldwide pose new challenges for instructors. Providing feedback to textual exercises is a challenge in large courses while being important for student's learning success. Exercise submissions and their grading are a primary and individual communication channel between instructors and students. The pure amount of submissions makes it impossible for a single instructor to provide regular feedback to large student bodies. Employing tutors in the process introduces new challenges. Feedback should be consistent and fair for all students. Additionally, interactive teaching models strive for real-time feedback and multiple submissions.

We propose a support system for grading textual exercises using an automatic segment-based assessment concept. The system aims at providing suggestions to instructors by reusing previous comments as well as scores. The goal is to reduce the workload for instructors, while at the same time creating timely and consistent feedback to the students. We present the design and a prototypical implementation of an algorithm using topic modeling for segmenting the submissions into smaller blocks. Thereby, the system derives smaller units for assessment and allowing the creation of reusable and structured feedback.

We have evaluated the algorithm qualitatively by comparing automatically produced segments with manually produced segments created by humans. The results show that the system can produce topically coherent segments. The segmentation algorithm based on topic modeling is superior to approaches purely based on syntax and punctuation.

ECSEE '20, June 18–19, 2020, Seeon/Bavaria, Germany

ACM ISBN 978-1-4503-7752-2/20/06…$15.00
https://doi.org/10.1145/3396802.3396805

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**; • **Computing methodologies** → *Natural language processing*.

KEYWORDS

Software Engineering Education, Automatic Assessment, Textual Exercise, Assessment Support Systems

ACM Reference Format:
Jan Philip Bernius, Anna Kovaleva, Stephan Krusche, and Bernd Bruegge. 2020. Towards the Automation of Grading Textual Student Submissions to Open-ended Questions. In *European Conference on Software Engineering Education (ECSEE '20), June 18–19, 2020, Seeon/Bavaria, Germany*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3396802.3396805

1 INTRODUCTION

In the past, there has been a growing number of students enrolled at universities worldwide[1]. Large courses have thousands of students participating, especially when using virtual classrooms. Figure 1 shows a typical mixed classroom setup for 1.700 software engineering students used in the summer semester 2019 at Technical University of Munich (TUM).

In introductory computer science and software engineering courses, classroom sizes with up to 1.700 students are no longer an exception, with growth by factor five in the last ten years. The free Stanford Massive Open Online Course (MOOC) "Intro to Artificial Intelligence,"[2] started in 2011, quickly reaching 160,000 students [42]. Large lectures pose a problem for instructors when grading textual exercises. This is partially solved in MOOCs by peer reviews [19]. The main problem is the asynchronous assessment, which usually requires a week, or even longer. A major disadvantage of MOOCs is the delay between giving the exercise and grading. To reduce this delay, we teach interactive lectures where we include exercises live during the lectures, grade them immediately, and provide quick feedback to students [24]. This increases student

[1]United Nations, "UN Global Assessment on Higher Education Reveals Broad Socio-Economic, Gender Disparities," https://news.un.org/en/story/2017/04/555642-un-global-assessment-higher-education-reveals-broad-socio-economic-gender, 2017.
[2]Peter Norvig and Sebastian Thrun, "Intro to Artificial Intelligence," https://www.udacity.com/course/intro-to-artificial-intelligence--cs271, 2011.

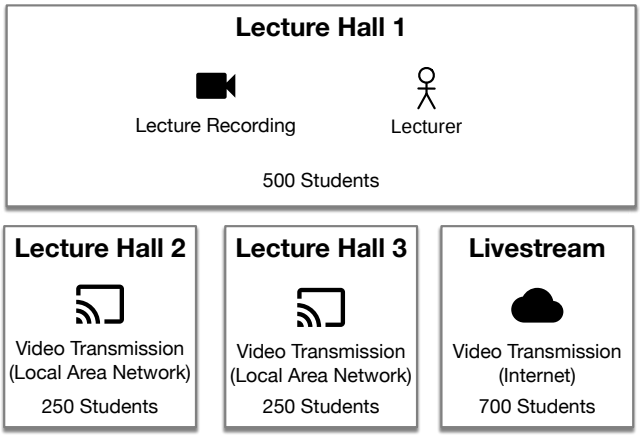


Figure 1: Mixed on-campus and virtual classroom setup employed in the summer semester 2019 at TUM for the "Introduction to Software Engineering" course.

comprehension and deepens understanding [19, 24], "significantly by up to 87 %" in the domain of modeling [25].

Technology to foster interaction and discussion within large lectures does exist [19, 29], as well as a scalable exercise system for programming and modeling exercises with automatic assessments [22, 23]. Textual exercises are commonly used in the examination, but no automatic assessment solution is available to instructors.

Conducting open answer questions requires time-consuming activities from instructors, including designing exercises and manual assessment, due to the high variability in student answers. To reduce efforts, instructors tend to reuse exercises from previous years. Grading is a repeatable process, instructors look for common mistakes or predefined solution patterns. The students learning success benefits from detailed and personalized feedback [37]. To enable large scale courses, the need to reuse feedback comments arises. Individual feedback can still rely on the domain expertise of the teacher. A single instructor cannot provide regular individual feedback due to large student bodies with more than 1.000 students. Ofter, tutors are employed to distribute the workload. Multiple graders require means to create consistent feedback for learners. This holds especially if the assessments are relevant for the final grade, e.g. as part of a grade bonus system.

This paper focuses on the segmentation of submissions into topically coherent parts, to enable reuse of feedback. Section 2 describes foundations on assessment systems and Section 3 summarizes related work on text segmentation. We present an algorithm in Section 4 that learns the topics of the submissions and then splits up the answers accordingly. The Evaluation in Section 5 analyzes the quality of the algorithm's performance, in a study with 10 participants. Section 6 summarizes the paper and outlines future work.

2 ASSESSMENT SYSTEMS

Assessment systems are a common tool used in universities. Software systems available to instructors vary from simple submission of work, over grade review, towards automated systems. We first explore interactive learning, a teaching methodology that can be supported by assessment systems. Second, we inspect Artemis as an example of an assessment system geared towards automatic assessment. Last, we look at an approach to apply automatic assessments on textual exercises.

2.1 Interactive Learning

A traditional university approach based on real-time communication demands students to be present in the lecture hall to participate. With growing numbers of enrollments in universities, the interaction in classes is getting more difficult as more staff is needed, and new ways for communication in large audiences are required [19]. One of the first approaches to incorporate technology into the classroom was the introduction of clickers for answering questions [29]. Mayer et al. describe a method for forcing interaction with the help of "response systems". The proposed system allows students to "click" an answer to a multiple-choice question. The instructor can evaluate the answers and a discussion on the topic can follow. Bonwell and Eison analyze the impact of in-class discussions and questions during lectures and exercises [6]. They found out that through constantly applying knowledge, students gain a deeper understanding of the content. The interactive learning approach combines theory, typically presented in lectures, with practical exercises [23]. Reflections based on feedback help to comprehend knowledge. In an iterative process, frequent feedback enables students to resubmit and learn from their mistakes [24].

2.2 Artemis

Artemis[3] is an automatic assessment management system developed at TUM [22]. It was built specifically to enable interactive lectures, following the idea of interactive learning. The aim of this system was primarily to allow students that are enrolled in software engineering classes to participate in interactive programming exercises. The system provides quick automatic feedback, thereby helping the students to acquire knowledge better and, as a result, achieve better grades in the final exam [24, 25]. During the past years, the system constantly evolved and is now also used at other educational institutions and in MOOCs. Programming exercises can be submitted and assessed with the help of unit tests. Additionally, modeling exercises are supported by a UML editor and a semi-automatized assessment component. The system provides full support of multiple-choice quizzes, including creating, conducting, and correcting them. For a deeper understanding of a lecture's theoretical basis, open-ended questions are more suitable than multiple-choice questions [13]. Artemis allows us to conduct textual exercises and submit answers, but instructors need to grade student answers manually. This is a time-consuming process that can lead to longer feedback loops, which decrease the students' motivation. With a growing number of students, the number of assessments increases, too. This results in bigger workloads for instructors and usually requires hiring more people. In this case, the consistency of the assessment may decrease. While there is usually only one sample solution, an unbound variety of students' answers exists. In mathematical problems or multiple-choice questions, the

[3] "Artemis: Interactive Learning with Individual Feedback," https://github.com/ls1intum/Artemis, 2020.

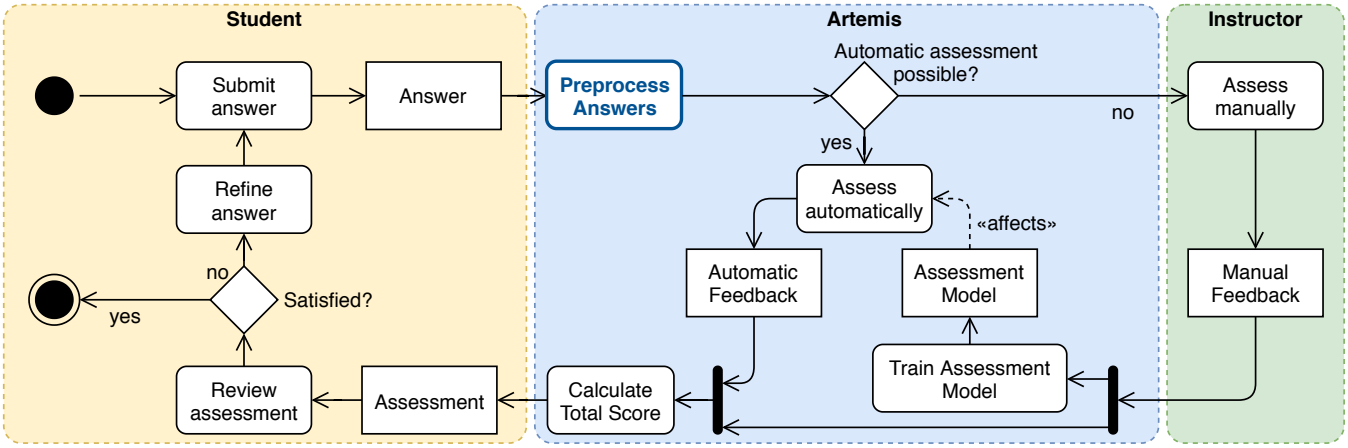


Figure 2: Workflow of the automatic assessment system for textual exercises. The "Preprocessing Answers" activity (Figure 4) includes the algorithm presented in this paper. UML activity diagram based on Bernius and Bruegge [2].

correct solution is mostly unique, whereas, for open questions, multiple interpretations are possible.

2.3 Automatic Assessment of Textual Exercises

Bernius and Bruegge describe a feedback concept built to produce reusable and consistent feedback targeted for automatic assessments of textual exercises [2]. Feedback is provided to topically coherent text blocks, resulting in uniform and consistent feedback across all assessments from multiple instructors. The concept aims at reducing work for instructors and increasing consistency, reducing complaints from a peer-to-peer comparison between students. In this approach, text blocks are manually highlightable by the instructor, but this is not applicable to automated computations. Splitting student answers based on delimiter characters[4] is not a reliable solution, because of missing punctuation, abbreviations, the use of bullet point answers, or long sentences. Also, a single feedback item is sometimes more suitable for a whole paragraph or a single clause or bullet point, which is not covered by the syntactical separator approach and requires manual adjustments.

Based on this concept, we developed a system to reuse instructor feedback across students by analyzing the similarity of text blocks [2]. The system simplifies the grading process by providing grading suggestions to instructors. Feedback suggestions are based on similarity between answers, allowing the training of an assessment model used to automatically assess answers as depicted in Figure 2. Training and using this system relies on topically coherent text blocks so that feedback is well scoped and can be shared between many submissions.

3 TEXT SEGMENTATION

Text segmentation is considered to be one of the tasks of Natural Language Processing (NLP). The term is used differently in literature and is not clearly defined. For example, document processing to extract typed or handwritten text by distinguishing it from graphics and blank spaces is referred to as text segmentation [17]. In other cases, text segmentation is the process of extracting text from video in order to index the recordings in a database [26]. Pak and Teh conducted an analysis of literature on text segmentation published between 2007 and 2017 [34] and categorize different approaches found in literature as depicted in Figure 3. The authors additionally categorize the papers according to used documents, language, and the goal of applying text segmentation. They identify the following application domains for text segmentation: "emotion extraction, sentiment mining, opinion mining, topic identification, language detection and information retrieval" [34].

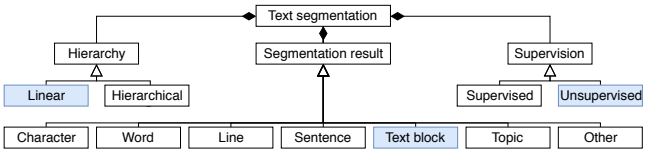


Figure 3: Taxonomy for text segmentation adapted from Pak and Teh [34]. Text segmentation types relevant for this paper are highlighted in blue color.

Information retrieval has many different applications, for example, reducing large documents to relevant fragments based on desired subtopics. The different desired results of text segmentation, the segment, is another interesting aspect Pak and Teh point out. According to their analysis, a word is considered a segment most often in literature, slightly less frequent are characters, topics, sentences, lines. In other cases, phrases, paragraphs, or tags can be used. We define the term "text block" in this paper as either clause, bullet point, sentence, or paragraph.

Text segmentation can be additionally divided into linear, text split into non-overlapping linear segments, and hierarchical, where segments also have hierarchical relationships [9, 44]. The latter is sometimes used for discourse retrieval. Along with most literature on text segmentation, we only focus on linear text segmentation.

There also exists a differentiation based on the supervision of the algorithm. Unsupervised approaches do not require any external

[4]Delimiter characters such as . : ; ? !

information to be trained, whereas supervised algorithms learn from big datasets, such as Wikipedia, for example [21].

3.1 Topic Modeling

Latent Dirichlet Allocation (LDA) was introduced by Blei et al. in 2003 [5]. LDA is used by many authors [7, 9, 32, 33] and proven to be suitable when training data is from the same domain as test data [32].

TopicTiling is an extension of Hearst's TextTiling algorithm that uses LDA to assign topic IDs to text blocks [15, 41]. Each block is represented by a T-dimensional vector, where T is the number of topics in the dataset. A coherence score is then calculated between neighboring blocks inside of a "window" with cosine similarity. Depth scores of the smallest coherence scores are then calculated depending on the highest coherence score to the left and the right. The highest depth scores indicate sub-topic boundaries.

Chen et al. use LDA and a K-nearest neighbor algorithm to classify short texts which gives evidence that LDA can also be applied to text consisting of only several words [7].

Tu et al. use LDA and word-embeddings to segment educational texts for online learning with a domain-independent algorithm [43]. They train their model on a small dataset and state that LDA can be used with a comparatively small number of topics. They also compare different similarity measures, such as cosine similarity, depth score, spectrum. They additionally analyze the impact of different values of input parameters of LDA. A similar analysis is done by Riedl and Biemann [40].

3.2 Keyword Extraction

Ramos uses Term Frequency Inverse Document Frequency (TF-IDF) to determine whether of a word is significant to a user's query when searching documents [38]. Intuitively, a word's frequency is linked to its importance. TF-IDF proposes that not only the absolute frequency is relevant, but also the number of occurrences in different documents. If a word occurs often across many documents, it is most probably not significant. In the previous section, the concept of stop words, which deals with the same problem, is described. The application of TF-IDF is rather straightforward: every document is run through and the two relevant frequencies are computed. The significance of a word is proportional to the frequency inside of the document but decreases if the word is found across different texts.

Another way to extract keywords is by using a thesaurus [30]. This can be especially helpful when there is only one document, thus, the TF-IDF approach is not suitable. A thesaurus also provides external knowledge which, on one hand, allows extracting keywords without any training but, on the other hand, requires additional maintenance and fails if there is no match available.

An ontology, a relational representation between concepts, can also be used to extract topics from text [11]. Embley et al. take unstructured documents and application ontology as input. Then they use a "keyword recognizer" to spot keywords with the help of regular expressions, afterward, restructuring the extracted information with the help of the ontology. They use this approach, for example, to extract information from car advertisements. This can be a suitable solution if the domain is known, keeping in mind that creating an ontology requires time. However, it is not applicable if the algorithm is to be applied to many different domains, and the main concepts are not known in advance.

Matsuo and Ishizuka propose another method for keyword extraction that bases on the χ^2-measure [28]. They first count co-occurrences of words and word sequences. "If a term appears frequently with a particular subset of terms, the term is likely to have an important meaning" [28]. Then a co-occurrence matrix is calculated. To improve the χ^2-computation "variety of sentence length and robustness of the χ^2-value" are considered. To improve the quality of the χ^2-measure two types of clustering are applied. Similarity-based clustering gathers words with similar roles in a sentence, pairwise clustering picks words from the same domain. The words with the largest χ^2-value are given as the result.

Most of the previous approaches only focus on the frequencies but cannot detect synonyms, even different forms of a verb can decrease the quality of the algorithms. Hulth adjusts the previous approaches by introducing syntactical information, such as part-of-speech (PoS) tagging, and data preprocessing, for example, stemming, stop words removal [16]. They introduce a pattern approach: based on the training set, there is evidence that most keywords have nouns and follow a particular pattern, for example, "adjective noun" uncountable or in the singular [16]. To calculate the relevance of a phrase four features are used: frequency within a single document, frequency in the whole set of documents, the position where the term appears first in a document, and the PoS-tag. The machine learning model is then based on a set of inductive rules that are derived with the help of "recursive partitioning (or divide-and-conquer), which has as the goal to maximize the separation between the classes for each rule" [16].

3.3 Dataset

Most authors use labeled and segmented, often artificially generated datasets, such as Choi's labeled dataset for evaluating their algorithms [8]. Often news articles or news broadcast transcripts are used as there are clear topic boundaries that can be then compared [1, 7, 20, 35, 45]. The evaluation algorithm is often based on the approach by Beeferman et al. [1, 12, 39].

In this paper, as part of a text grading application for a university environment, we focus on data collected from the lecture "Patterns in Software Engineering" (PSE) at TUM in 2018/19. The dataset consists of two exercises with 121 and 124 student submissions. The exercises were conducted in-class and were announced as a mock exam.

4 SEGMENTING STUDENT ANSWERS

Based on the literature, several existing approaches were applied to the proposed problem. For testing the approaches we used a set of students' answers from our dataset. The exercise on the difference between patterns and anti-patterns received answers with an average length of 3.6 sentences. Tested approaches were, TopicTiling [41] and Bayes-seg developed by Eisenstein and Barzilay [10]. The first is based on topic modeling with LDA. The latter uses Bayesian probabilities and entropy to segment the texts. However, these algorithms could deliver no or only poor results on our dataset, most probably because of the short length and specific vocabulary distribution, which they are not fitted for.

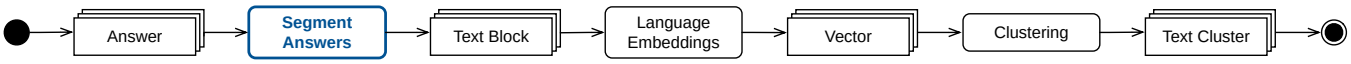


Figure 4: A detailed view into the "Preprocess Answers" activity (Figure 2) performed by the assessment system before the grading, depicted using a UML activity diagram.

We abstracted the topic modeling approach and preserve the idea that every answer is a collection of topics, and many topics are distributed among different answers [5]. However, instead of calculating a topic model, we claim that a topic can be reduced to a keyword. This way, the scarcity of the words in the answers can be compensated for. Another strategy adapted from other works is the "vocabulary introduction" [15]. As soon as new keywords are introduced, a new segment begins. The presented approach differs from thesaurus or ontology in a way that we do not know what the keywords are going to be, and they are calculated for every problem separately.

In the assessment system, the algorithm is one step in a preprocessing phase, depicted in Figure 4. Answers are segmented into text blocks before language embeddings and clusters are computed.

The algorithm can be separated into three phases: Text Preprocessing, Keyword Extraction, and Segmentation. Figure 5 depicts the algorithm's flow of events, which is described in detail in the following sections.

4.1 Text Preprocessing

Most algorithms for NLP are applied to preprocessed text-data. In the assessment context, data is of rather low quality and cannot be preprocessed manually. The available data contains lots of typing mistakes, poor formatting, missing punctuation, and misspelled words. Student submissions must not be modified, formatting being the only exception. Applying existing algorithms to our data showed that bullet points, wrong punctuation, such as using newlines instead of points, can quickly reduce the quality of the outcome. Hence, we try to cover the most common irregularities and transform them into a format suitable for further calculations.

4.1.1 Stop Words. Removing stop words from text is a very common way to clean textual data for NLP [15, 16, 41]. We use the set of stop words provided as part of the Natural Language Toolkit for Python (NLTK) [4]. The English collection consists of 179 words, like "I", "the", "what", "did", that do not contain much lexical content and can, therefore, be removed from the corpus. Although this implementation only supports students' submissions written in English, the German set of 232 words is also included because occasionally students hand in answers in the German language. This cannot provide full support of submissions in German but can reduce their negative effect on further processing.

4.1.2 Lemmatization. Lemmatization is the process of reducing a word to its meaningful root. Keeping in mind, that we want to extract keywords from a text and that the stop words are already removed, we now have a set of words where the most significant terms need to be found. Naturally, we use different forms of a word: either the plural or the singular, different tenses for verbs, degrees of comparison for adjectives, etc. Without preprocessing, the system would consider the words "view" and "views" as two different ones. With the help of WordNet, which is provided as part of the NLTK, the algorithm reduces the second word to "view" [4, 31].

The result of the text preprocessing is thereby a set of lemmatized lower-case words without any punctuation or stop words.

4.2 Keyword Extraction

The chosen approach for segmenting the students' answers into text blocks is partially based on keyword extraction. We generalize the idea of topic modeling that claims that every document is a distribution over topics, and every topic is distributed over words. We claim that every student's submission is a collection of topics, and statements, that are common among different answers. However, we do not calculate a topic model. As already described, existing approaches based on topic modeling are not suitable for our kind of data because of rather short answers (3.6 sentences long on average) and very different vocabulary used among different submissions. That is why we reduce a topic to a keyword, thereby, compensating for the data scarcity.

For keyword calculation, we adopt an approach based on word frequency[5]. We tested the frequently used TF-IDF approach [38], which proved to be inefficient in our case. The reason for it is the specific character of the data. The TF-IDF method assumes that words, frequent among different documents, are not significant for keyword extraction, as they are too common. In the considered context, the important words, definitions, for example, are present in most of the answers. Another examined approach was an extension of the word frequency measure [16, 39]. Instead of searching for significant words, they consider n-grams. This method did not suit the data either. We tested the algorithm with bi- and tri-grams, the resulting segmentation was worse than with single words. The resulting keywords are the 10 most frequently used words in the texts. The number was chosen empirically based on our data. Dynamically determining the optimal number of keywords could be researched in the future to improve the algorithm.

4.3 Segmentation

The segmentation of the texts is split up into two steps. First, the answers are split up into initial text blocks. Then, adjacent text blocks are considered and merged if there are no new keywords introduced. The result of this is a set of segments for each answer that can be used by the rest of the system.

4.3.1 Sentence Tokenization. For identifying sentences we use a pre-trained model of "punkt tokenizer" from the NLTK [4, 18]. However, it cannot handle bulleted lists, that is why we need to additionally split the text on new lines. We also want to work with clauses if a sentence is long. We decided not to use any algorithm for that

[5] Sowmya Vivek, "Automated Keyword Extraction from Articles using NLP", https://medium.com/analytics-vidhya/automated-keyword-extraction-from-articles-using-nlp-bfd864f41b34, 2018.

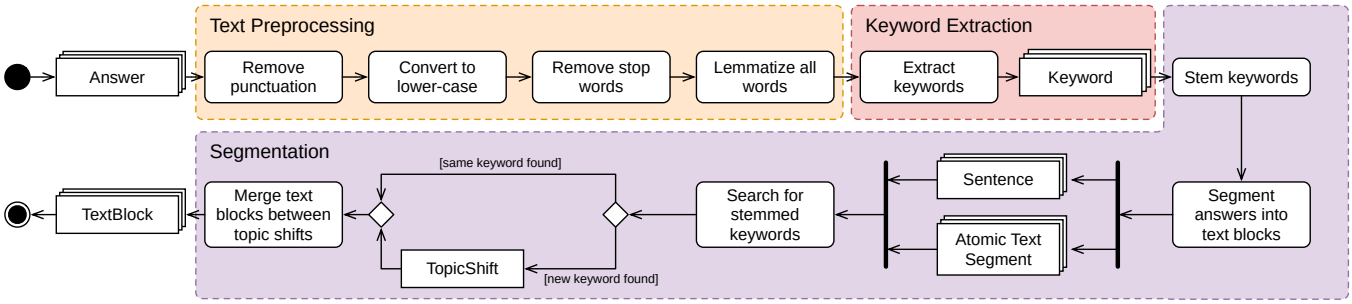


Figure 5: The segmentation algorithms flow of events depicted using a UML activity diagram based on Bernius et al. [3].

but search for conjunctions. We use subordinating conjunctions and assume that they indicate a new clause. This approach is not complete and cannot be considered proper clause identification, however, for this use case, we assume, it is enough. To minimize false positives when identifying clauses, we only consider sentences that are longer than 20 words.

4.3.2 Finding Segments. Before searching for keywords in the text blocks, we use a stemmer from the NLTK [4], called PorterStemmer [36]. Similarly to lemmatization, stemming is applied to avoid different forms of a word in a text. The latter, however, reduces a term to a part, that in some cases may not be a correct word. An example of this is "similarit", as the result of stemming the word "similarity". Hence, it can be very helpful when searching for words, as you can then find both "similarities" and "similarity".

For the definition of segments, we use the lexical cohesion approach and the vocabulary introduction method [14, 15]. The algorithm iterates over all text blocks defined in a submission. We use the original texts at this stage, not the preprocessed versions. In every segment, stemmed keywords are sought. If two adjacent segments have the same keywords or the second text block has none, they are merged into one block and the algorithm proceeds. As soon as new keywords are introduced, the algorithm puts a segment boundary before the current text block. This way the whole process can be defined as a "divide & conquer" approach, because we first divide the answer into initial text blocks, as small as possible, and then merge them according to the defined boundaries.

5 EVALUATION

In order to evaluate the segmentation quality of the algorithm, we conducted a qualitative study with 10 participants. We compare the segmentation of the new algorithm with the existing approach and the segmentation generated by the participants. We present anecdotal evidence on the performance of the new algorithm.

5.1 Design

The evaluation is designed as a 15-minute interview. Participants first get an introduction to semi-automatic text assessment, the assessment concept [2] and segmentation. However, for reasons of internal validity, no details of the segmentation algorithm or further processing are given. The questionnaire consists of two parts: segmentation tasks and questions about the subjective impressions of the approach.

The first part requires five segmentation tasks. Participants are given five student submissions from our dataset and asked to find and mark topic shifts. The same task is performed by two systems, one based on the syntactical separator approach and a second one based on our topic modeling algorithm.

Each participant performs the task of finding and marking topic shifts, as the system would do. These results are then quantitatively analyzed and compared to the segmentation results of the existing solution and the proposed algorithm. The performance measure consists of the two criteria recall and precision [1]:

$$recall = \frac{\text{number of estimated topic shifts that are actual topic shifts}}{\text{number of true topic shifts}}$$

$$precision = \frac{\text{number of estimated topic shifts that are actual topic shifts}}{\text{number of estimated topic shifts}}$$

The submissions are taken from the PSE dataset and are of various format that is common among students' answers. There are bulleted and numbered lists, as well as text mixed with bulleted lists, also two submissions that consist of multiple sentences and paragraphs are included. The submissions are taken with original grammar and punctuation.

The third part addresses the impressions of the surveyed. They are asked to state their personal opinion on the approach and give their judgment whether this solution can improve the instructors' and students' experience with textual exercises. The possible answers are on a five-point scale based on Likert [27].

The study was conducted with ten students from the Department of Informatics at TUM, who previously passed software engineering courses from our chair four of which have previous experience working as a tutor. These students have reasonable domain knowledge to determine segments. Also, they are potential tutors for future editions of the courses.

5.2 Objectives

We define the following hypotheses for the evaluation:

- H1 The designed segmentation algorithm performs better than the syntactical separator approach measured using the performance criterion recall and precision.
- H2 Students understand the approach and find it intuitive.

H3 Students consider the approach an improvement of their understanding of feedback and the comprehension of a lecture's content.
H4 The segmentation algorithm produces the same segmentation as humans.

5.3 Results

Based on the computed segmentations depicted in Figure 6, we conducted a performance evaluation using recall and precision. A topic shift position was considered if more than 50% of the students marked the position. Results in Table 1 show an increased recall and precision values for the topic modeling based algorithm.

Table 1: Performance analysis of the new topic modeling based algorithm and the previous approach based on syntactical separators measured according to precision and recall [1].

Submission	Topic Modeling		Syntactical Separators	
	Recall, %	Precision, %	Recall, %	Precision, %
S1	100	100	100	50
S2	75	60	100	67
S3	75	100	50	50
S4	100	100	100	100
S5	67	100	30	100
Average	83.4	92	76	73.4

We analyze the number of detected topic shifts in Figure 7. We compare the number of topic shifts found by the proposed algorithm and the current solution to the number of topic shifts marked by the participants. We also depict statistics for the most frequent topic shifts, meaning positions that were present in six or more answer sheets.

In the questionnaire, nine out of ten students agreed that the presented approach of segmenting answers is intuitive (see Figure 8), supporting our hypothesis H2. Students also claimed that finding topic shifts' positions was not very easy which can probably be linked to the unambiguity of the task. The results also depend on the style of the assessment of a participant. Therefore, we compared the average number and the number of the most frequent segments, where one can see that these two numbers sometimes vary. Especially, for submission S2, where the proposed system failed to improve the result of the current system, the difference between the two numbers is big. This can also be justified with the fact, that some participants tended to mark more positions than other students for most of the submissions. The data shows that the topic modeling-based algorithm resembles human perception better than the syntactical separation approach.

Since most of the students stated to value the assessment of textual exercises as helpful, there were downsides like general or short feedback, as well as long correction periods. Participants agree that the assessment process can be accelerated by applying our approach. All of our participants considered structured feedback to be an improvement for the students' comprehension, eight participants agree strongly. The responses support the third hypothesis (H3).

S1

Differences: [S]
Antipatterns: [S]
-Have one problem [8] and two solutions [8] (one problematic [8] and one refactored) [1-4, 7-10, S, T]
-Antipatterns are a sign of bad architecture [8] and bad coding [1-10, S, T]
Pattern: [S]
-Have one problem and one solution [1-5, 7, 9, 10, S, T]
-Patterns are a sign of elaborated architecutre and coding

S2

The main difference between patterns and antipatterns is, [8] that [6, 7] patterns show you a good way to do something [8, 7] and antipatterns show a bad way to do something. [1, 2, 4-10, S] Nevertheless [7] patterns may become antipatterns in the course of changing understanding of how good software engineering looks like. [1, 2, 5-10, S, T] One example for that is functional decomposition, [5] which used to be a pattern and "good practice". [1, 2, 5, 8, S, T] Over the time it turned out that it is not a goog way to solve problems, so it became a antipattern. [1-10, S, T]

A pattern itsself is a proposed solution to a problem that occurs often and in different situations. [1-3, 5-10, S, T]
In contrast to that a antipattern shows commonly made mistakes when dealing with a certain problem. [2, 7-9, S, T] Nevertheless a refactored solution is aswell proposed.

S3

1.Patterns can evolve into Antipatterns when change occurs [1-8, 10, S, T]
2. [S] Pattern has one solution, [2, 5-8, 10, T] whereas anti pattern can have subtypes of solution [1, 3, 4, 6, 8, 10, S, T]
3. [S] Antipattern has negative consequences [8] and symptom, [2, 6-8, 10] where as patterns looks only into benefits [8] and consequences

S4

Patterns: A way to Model code in differents ways [1-10, S, T]
Antipattern: A way of how Not to Model code

S5

Antipatterns are used when there are common mistakes in software management [5] and development to find these, [1-10, T] while patterns by themselves are used to build software systems [8] in the context of frequent change [8] by reducing complexity and isolating the change. [1-10, S, T]
Another difference is that the antipatterns have problematic solution [5, 8] and then refactored solution, [2, 5, 6, 8-10] while patterns only have a solution.

Figure 6: Submissions S1-S5 from our PSE data set. The submissions were segmented by two algorithms, as well as ten participants. The detected segment borders are marked inline with the text in square brackets: Topic Modeling Algorithm [T], Syntactical Separator Approach [S], and Participants [1-10].

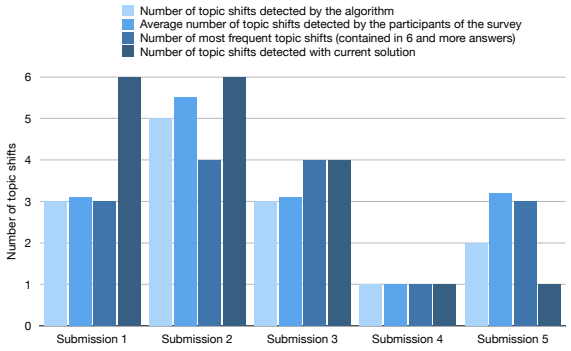


Figure 7: Comparison of the number of detected topic shifts by the current and proposed systems as well as the participants.

The topic modeling algorithm found 14 topic shifts in our sample of five submissions. The participants derived 15 topic shifts. As visible in Figure 6, 13 topic shifts (92%) are equally detected by the newly proposed algorithm and a majority of the participants. Only two topic shifts are not detected by the algorithm (false negative), and one topic shifts detected by the algorithm in S2 has no majority with the participants (false positive). This analysis does support the fourth hypothesis (H4).

5.4 Discussion

We could test the performance of the proposed system and compare it to the current solution based on the topic shifts' positions marked by students. Two interesting details could be discovered.

First, submission S2 was the only case where the proposed solution performed slightly worse than the existing approach. We can explain this with the character of this submission. The student submitted a rather long answer. It consists of seven sentences whereas the average number in our dataset is 3.6. In general, submissions with a lot of sentences where the same information is repeated multiple times can become a challenge. The student also gives an example of an anti-pattern. Answers with examples can become a problem for the proposed solution since there is an unbound set of examples that can be provided and thus it is difficult to judge if the keyword approach suits this case. A solution to this could be dynamically determining the number of keywords.

Second, when reviewing the students' segmentation, there were several answer sheets with significantly more topic shifts than found in other responses. This is usually because the participant saw an "and" in a sentence and decided that there are two different objects or verbs, hence, two different statements. One such case was the following part of an answer: "Antipatterns are used when there are common mistakes in software management and development to find these". Some participants put a boundary between the words "management" and "and". However, this kind of segmentation can lead to problems for further processing and assigning feedback to the text blocks. Though this part of the sentence does have two objects and they could, for example, be correct and incorrect or the other way around, the two resulting text blocks are both incomplete. The first text block misses the "find these" part, the second one — the subject of the sentence. This proves that it is possible to get text blocks that do not make any sense without context. A possible solution could be augmenting the parts of the sentence with the subject or the object from the other part. This, however, demands a deeper analysis of the sentence structure.

During the evaluation, we could make some interesting observations. There are two different types of text blocks that could be treated in another way. First, phrases that express the student's personal opinion about the question or the lecture, like "I do not understand this" or "oh, that's easy", do not need to be assessed. A possibility could be to discover them and exclude them from the corpus to improve the quality of the data for further processing. Incomplete sentences and clauses can also be treated differently. Compound sentences with several clauses often contain multiple different statements. Currently, we do not want to split them up. A sentence like "I like apples and bananas" does have two objects, but a text block "and bananas" does not make any sense without context, the subject and the verb in this case. So a possible solution could be augmenting incomplete text blocks with the corresponding missing context. This could be addressed by implementing PoS tagging.

5.5 Threats to Validity

One of the problems of the evaluation is the small size of the population. The validity could be improved by either increasing the population to include more tutors with different experience levels or by choosing a more experienced population of instructors. In addition, selected submissions for the segmentation task are a threat to external validity since they are from a single lecture. Third, submissions are chosen according to the formatting of the answer, as we allowed different answering formats such as bullet points or full sentences. The study therefore only provides anecdotal evidence on the performance of the assessment algorithm.

6 SUMMARY

In this paper, we have formalized a new algorithm based on topic modeling and text segmentation to segment student answers into topically coherent text blocks. A prototypical implementation has been integrated as part of the open-source Athene project[6] into the automatic assessment management system Artemis. A performance evaluation with ten students has shown that the new algorithm performs better than an algorithm using syntactical separators such as delimiters.

6.1 Conclusion

The presented algorithm is a small building block towards a semi-automated assessment support system for textual exercises, as well as the vision of fully automated assessments of textual exercises. Producing coherent text blocks from student submissions improves the experience for instructors, tutors, and students:

For instructors, a structured form of feedback makes it easier to compare against grading criteria. The increasing degree of automation reduces the workload necessary to conduct textual exercises.

[6] "Athene: A library to support (semi-)automated assessment of textual exercises," https://github.com/ls1intum/Athene, 2020.

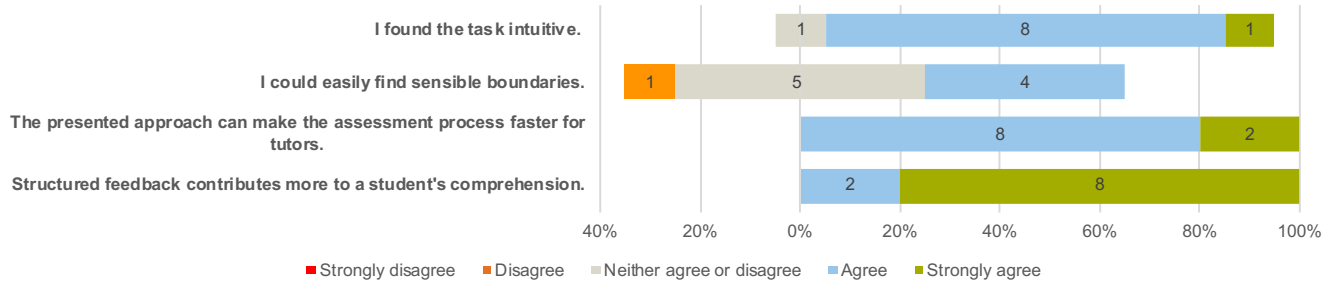


Figure 8: Participants response on their subjective impression of the approach ranked on a five-point scale based on Likert [27]. ($n = 10$)

For tutors, the algorithm allows to automate the first step of the grading process and removes some of the overhead related to the segment-based assessment concept. Generated feedback suggestions improve the value of each feedback element, as it can be easily reused for multiple students, even by other tutors. Suggestions reduce the workload, as a partial assessment is already pre-filled. A semi-automated system should encourage tutors to create extensive and high-quality explanations.

For students, feedback will be more concise. A direct link between a segment of their submission and feedback helps students to understand the feedback and their mistakes. They profit from improvements for tutors, which we envision to lead to quicker and more extensive feedback.

6.2 Future Work

The result of the algorithm's application can be improved in two areas: keywords and text blocks using statistical models, topic models, or decision trees. Additionally, a thesaurus could be used to recognize synonyms.

The effect of the algorithm on the assessment system can be evaluated in two aspects: The usability for tutors when grading text blocks and the impact of the segmentation on the quality of feedback suggestions.

REFERENCES

[1] Doug Beeferman, Adam L. Berger, and John D. Lafferty. 1997. Text Segmentation Using Exponential Models. *CoRR* (1997). http://arxiv.org/abs/cmp-lg/9706016

[2] Jan Philip Bernius and Bernd Bruegge. 2019. Toward the Automatic Assessment of Text Exercises. In *2nd Workshop on Innovative Software Engineering Education (ISEE)*. Stuttgart, Germany, 19–22.

[3] Jan Philip Bernius, Anna Kovaleva, and Bernd Bruegge. 2020. Segmenting Student Answers to Textual Exercises Based on Topic Modeling. In *17th Workshop Software Engineering im Unterricht der Hochschulen (SEUH)*. Innsbruck, Austria, 72–73.

[4] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python* (1st ed.). O'Reilly Media, Inc.

[5] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *The Journal of Machine Learning Research* 3 (2003), 993–1022.

[6] Charles C. Bonwell and James A. Eison. 1991. *Active Learning: Creating Excitement in the Classroom*. ERIC Clearinghouse on Higher Education.

[7] Qiuxing Chen, Lixiu Yao, and Jie Yang. 2016. Short text classification based on LDA topic model. In *2016 International Conference on Audio, Language and Image Processing (ICALIP)*. IEEE, 749–753. https://doi.org/10.1109/icalip.2016.7846525

[8] Freddy Y. Y. Choi. 2000. Advances in Domain Independent Linear Text Segmentation. In *Proceedings of the 1st North American Chapter of the Association for Computational Linguistics Conference* (Seattle, Washington) *(NAACL 2000)*. Association for Computational Linguistics, USA, 26–33.

[9] Jacob Eisenstein. 2009. Hierarchical Text Segmentation from Multi-Scale Lexical Cohesion. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, Boulder, Colorado, 353–361. https://www.aclweb.org/anthology/N09-1040

[10] Jacob Eisenstein and Regina Barzilay. 2008. Bayesian Unsupervised Topic Segmentation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing* (Honolulu, Hawaii) *(EMNLP '08)*. Association for Computational Linguistics, USA, 334–343.

[11] David W. Embley, Douglas M. Campbell, Randy D. Smith, and Stephen W. Liddle. 1998. Ontology-based Extraction and Structuring of Information from Data-rich Unstructured Documents. In *Proceedings of the seventh international conference on Information and knowledge management - CIKM '98*. ACM Press, 52–59. https://doi.org/10.1145/288627.288641

[12] Pavlina Fragkou, Vassilios Petridis, and Athanasios Kehagias. 2004. A Dynamic Programming Algorithm for Linear Text Segmentation. *Journal of Intelligent Information Systems* 23, 2 (2004), 179–197. https://doi.org/10.1023/b:jiis.0000039534.65423.00

[13] Arthur C. Graesser, Peter Wiemer-Hastings, Katja Wiemer-Hastings, Derek Harter, Tutoring Research Group Tutoring Research Group, and Natalie Person. 2000. Using Latent Semantic Analysis to Evaluate the Contributions of Students in AutoTutor. *Interactive Learning Environments* 8, 2 (2000), 129–147. https://doi.org/10.1076/1049-4820(200008)8:2;1-b;ft129

[14] Michael A. K. Halliday and Ruqaiya Hasan. 1976. *Cohesion in English*. Longman, London.

[15] Marti A. Hearst. 1997. TextTiling: Segmenting Text into Multi-Paragraph Subtopic Passages. *Computational Linguistics* 23, 1 (1997), 33–64.

[16] Anette Hulth. 2003. Improved Automatic Keyword Extraction Given More Linguistic Knowledge. In *Proceedings of the 2003 conference on Empirical methods in natural language processing -*. Association for Computational Linguistics, 216–223. https://doi.org/10.3115/1119355.1119383

[17] Anil K. Jain and Sushil Bhattacharjee. 1992. Text segmentation using gabor filters for automatic document processing. *Machine Vision and Applications* 5, 3 (1992), 169–184. https://doi.org/10.1007/bf02626996

[18] Tibor Kiss and Jan Strunk. 2006. Unsupervised Multilingual Sentence Boundary Detection. *Computational Linguistics* 32, 4 (2006), 485–525. https://doi.org/10.1162/coli.2006.32.4.485

[19] Jan Knobloch and Enrico Gigantiello. 2017. AMATI: Another Massive Audience Teaching Instrument. In *15th Workshop Software Engineering im Unterricht der Hochschulen (SEUH)*. Hannover, Germany, 63–68.

[20] Takafumi Koshinaka, Ken ichi Iso, and Akitoshi Okumura. 2005. An HMM-based text segmentation method using variational Bayes approach and its application to LVCSR for broadcast news. In *Proceedings. (ICASSP '05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, Vol. 1. IEEE, 485–488. https://doi.org/10.1109/icassp.2005.1415156

[21] Omri Koshorek, Adir Cohen, Noam Mor, Michael Rotman, and Jonathan Berant. 2018. Text Segmentation as a Supervised Learning Task. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, Vol. 2. Association for Computational Linguistics, 469–473. https://doi.org/10.18653/v1/n18-2075

[22] Stephan Krusche and Andreas Seitz. 2018. ArTEMiS: An Automatic Assessment Management System for Interactive Learning. In *49th ACM Technical Symposium on Computer Science Education*. ACM, 284–289. https://doi.org/10.1145/3159450.3159602

[23] Stephan Krusche and Andreas Seitz. 2019. Increasing the Interactivity in Software Engineering MOOCs - A Case Study. In *31th Conference on Software Engineering Education and Training (CSEE&T)*.

[24] Stephan Krusche, Andreas Seitz, Jürgen Börstler, and Bernd Bruegge. 2017. Interactive Learning: Increasing Student Participation Through Shorter Exercise Cycles. In *19th Australasian Computing Education Conference*. ACM, 17–26. https://doi.org/10.1145/3013499.3013513

[25] Stephan Krusche, Nadine von Frankenberg, Lara Marie Reimer, and Bernd Bruegge. 2020. An Interactive Learning Method to Engage Students in Modeling. In *Proceedings of the 42nd International Conference on Software Engineering - Software Engineering Education and Training (ICSE-SEET'20)*. Seoul, South Korea.

[26] Rainer Lienhart and Wolfgang Effelsberg. 2000. Automatic text segmentation and text recognition for video indexing. *Multimedia Systems* 8, 1 (2000), 69–81. https://doi.org/10.1007/s005300050006

[27] Rensis Likert. 1932. A Technique for the Measurement of Attitudes. *Archives of Psychology* 22, 140 (1932), 1–55.

[28] Yutaka Matsuo and Mitsuru Ishizuka. 2003. Keyword Extraction from a Single Document using Word Co-occurrence Statistical Information. *International Journal on Artificial Intelligence Tools* 13, 01 (2003), 157–169. https://doi.org/10.1142/s0218213004001466

[29] Richard E. Mayer, Andrew Stull, Krista DeLeeuw, Kevin Almeroth, Bruce Bimber, Dorothy Chun, Monica Bulger, Julie Campbell, Allan Knight, and Hangjin Zhang. 2009. Clickers in college classrooms: Fostering learning with questioning methods in large lecture classes. *Contemporary Educational Psychology* 34, 1 (2009), 51–57. https://doi.org/10.1016/j.cedpsych.2008.04.002

[30] Olena Medelyan and Ian H. Witten. 2006. Thesaurus Based Automatic Keyphrase Indexing. In *Proceedings of the 6th ACM/IEEE-CS Joint Conference on Digital Libraries* (Chapel Hill, NC, USA) *(JCDL '06)*. Association for Computing Machinery, New York, NY, USA, 296–297. https://doi.org/10.1145/1141753.1141819

[31] George A. Miller. 1995. WordNet: A Lexical Database for English. *Commun. ACM* 38, 11 (1995), 39–41. https://doi.org/10.1145/219717.219748

[32] Hemant Misra, François Yvon, Olivier Cappé, and Joemon Jose. 2011. Text segmentation: A topic modeling perspective. *Information Processing & Management* 47, 4 (2011), 528–544. https://doi.org/10.1016/j.ipm.2010.11.008

[33] Hemant Misra, François Yvon, Joemon M. Jose, and Olivier Cappe. 2009. Text Segmentation via Topic Modeling: An Analytical Study. In *Proceeding of the 18th ACM conference on Information and knowledge management - CIKM '09* (Hong Kong, China). ACM Press, 1553–1556. https://doi.org/10.1145/1645953.1646170

[34] Irina Pak and Phoey Lee Teh. 2017. Text Segmentation Techniques: A Critical Review. In *Innovative Computing, Optimization and Its Applications: Modelling and Simulations*. Springer International Publishing, 167–181. https://doi.org/10.1007/978-3-319-66984-7_10

[35] Jay M. Ponte and W. Bruce Croft. 1997. Text segmentation by topic. In *Research and Advanced Technology for Digital Libraries*. Springer Berlin Heidelberg, 113–125.

[36] Martin F. Porter. 1980. An algorithm for suffix stripping. *Program: electronic library and information systems* 14, 3 (1980), 130–137.

[37] Ann Poulos and Mary Jane Mahony. 2008. Effectiveness of feedback: the students' perspective. *Assessment & Evaluation in Higher Education* 33, 2 (2008), 143–154. https://doi.org/10.1080/02602930601127869

[38] Juan Enrique Ramos. 2003. Using TF-IDF to Determine Word Relevance in Document Queries. In *1st instructional Conference on Machine Learning*.

[39] Jeffrey C. Reynar. 1999. Statistical Models for Topic Segmentation. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics -*. Association for Computational Linguistics, 357–364. https://doi.org/10.3115/1034678.1034735

[40] Martin Riedl and Chris Biemann. 2012. Sweeping through the Topic Space: Bad Luck? Roll Again!. In *Proceedings of the Joint Workshop on Unsupervised and Semi-Supervised Learning in NLP* (Avignon, France) *(ROBUS-UNSUP '12)*. Association for Computational Linguistics, USA, 19–27.

[41] Martin Riedl and Chris Biemann. 2012. TopicTiling: A Text Segmentation Algorithm Based on LDA. In *Proceedings of ACL 2012 Student Research Workshop* (Jeju Island, Korea) *(ACL '12)*. Association for Computational Linguistics, USA, 37–42.

[42] C. Osvaldo Rodriguez. 2012. MOOCs and the AI-Stanford like Courses: Two Successful and Distinct Course Formats for Massive Open Online Courses. *European Journal of Open, Distance and E-Learning* (2012).

[43] Yuwei Tu, Ying Xiong, Weiyu Chen, and Christopher Brinton. 2018. A Domain-Independent Text Segmentation Method for Educational Course Content. *IEEE International Conference on Data Mining Workshops* (2018). https://doi.org/10.1109/icdmw.2018.00053

[44] Yaakov Yaari. 1997. Segmentation of Expository Texts by Hierarchical Agglomerative Clustering. *CoRR* (1997). arXiv:9709015 [cmp-lg]

[45] Jon P. Yamron, Ira Carp, Larry Gillick, Steve Lowe, and Paul van Mulbregt. 1998. A hidden Markov model approach to text segmentation and event tracking. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, Vol. 1. IEEE, IEEE, 333–336. https://doi.org/10.1109/icassp.1998.674435

8.10 A Machine Learning Approach for Suggesting Feedback in Textual Exercises in Large Courses

The conference paper is based on [BKKB21] and describes more details of the machine learning approach for suggesting computer-aided feedback and the open-source application Athene. It includes an empirical evaluation of Athene in 17 textual exercises across two courses with 2.300 students and 53 tutors. The empirical evaluation results show that Athene can suggest up to 70 % of the feedback with an average accuracy of 85 %. Ratings provide the first indications that the perceived quality improves when compared to purely manual assessments. The evaluation shows that the efficiency depends on the type of textual exercise and the variability of the possible solutions. A higher variance of correct solutions leads to less coverage because of fewer similarities in the student answers.

Jan Philip Bernius implemented the approach in the open-source application Athene and integrated it into Artemis. The author of this habilitation supervised Jan Philip Bernius throughout the development of the semi-automatic assessment approach and writing the paper.

Authors	J. Bernius, **S. Krusche** and B. Bruegge
Conference	8th Conference on Learning at Scale
Publisher	ACM
Pages	10
Type	Conference: Full Research Paper
Review	Peer Reviewed (3 Reviewers)
Year	2021
Citation	[BKB21]
URL	https://doi.org/10.1145/3430895.3460135

A Machine Learning Approach for Suggesting Feedback in Textual Exercises in Large Courses

Jan Philip Bernius, Stephan Krusche, and Bernd Bruegge
Department of Informatics
Technical University of Munich
Munich, Germany
janphilip.bernius@tum.de, krusche@in.tum.de, bernd.bruegge@tum.de

ABSTRACT

Open-ended textual exercises facilitate the comprehension of problem-solving skills. Students can learn from their mistakes when teachers provide individual feedback. However, courses with hundreds of students cause a heavy workload for teachers: providing individual feedback is mostly a manual, repetitive, and time-consuming activity.

This paper presents **CoFee**, a machine learning approach designed to suggest computer-aided feedback in open-ended textual exercises. The approach uses topic modeling to split student answers into text segments and language embeddings to transform these segments. It then applies clustering to group the text segments by similarity so that the same feedback can be applied to all segments within the same cluster.

We implemented this approach in a reference implementation called **Athene** and integrated it into Artemis. We used Athene to review 17 textual exercises in two large courses at the Technical University of Munich with 2,300 registered students and 53 teachers. On average, Athene suggested feedback for 26% of the submissions. Accordingly, 85% of these suggestions were accepted by the teachers, 5% were extended with a comment and then accepted, and 10% were changed.

Author Keywords

Software Engineering, Education, Interactive Learning, Automatic Assessment, Grading, Assessment Support System, Learning, Feedback

CCS Concepts

•**Social and professional topics** → **Software engineering education;** •**Computing methodologies** → *Natural language processing;*

L@S '21, June 22–25, 2021, Virtual Event, Germany

ISBN 978-1-4503-8215-1/21/06…15.00

DOI: https://doi.org/10.1145/3430895.3460135

INTRODUCTION

The rise in student numbers in universities has led to an increase in course management efforts, and made it challenging to provide high-quality individual feedback to students [19]. Recent approaches, such as online platforms and live streaming, allow teachers[1] to cope and interact with a large amount of students on an individual level, regardless of the respective course size.

In particular, large university courses with hundreds of students rely on teaching assistants to provide feedback on exercises, e.g., multiple-choice quizzes and textual exercises. Multiple-choice quizzes are easy to assess, and tools are broadly available in learning management systems (LMSs) and for paper-based assessment. However, mastery of these quizzes does not require problem-solving skills because they typically target only lower cognitive skills, in particular, *knowledge* recall and *comprehension*. Most quiz types include predefined options and do not reflect work practices in industry. It is difficult to create quizzes that stimulate higher cognitive skills, such as problem-solving, which are important in computer science [1, 32].

Open-ended textual exercises allow instructors to teach problem-solving skills and allow students to improve their knowledge. These exercises do not have a single correct solution, but rather allow answers within a particular solution space which can be characterized by words and phrases. The searchlight theory of scientific knowledge [27] states that students increase their knowledge through observations, especially observations that prove their assumptions wrong. Students profit from having an individual feedback relationship with their teachers [10]. Individual feedback and formative assessments are essential elements in learning [11, 12]. Feedback on open-ended exercises allows students to try out problem-solving and to experience failure. Students need guidance in the form of feedback in their learning activities to prevent misconceptions [13].

[1]For this paper, we define teachers as both instructors and teaching assistants (see Figure 1). Instructors are employees of the university such as professors, lecturers, and doctoral candidates. Teaching assistants are experienced students who have passed the same course previously with a good grade and who are motivated to help in the teaching process. Some universities also use the term "tutor" to refer to a teaching assistant.

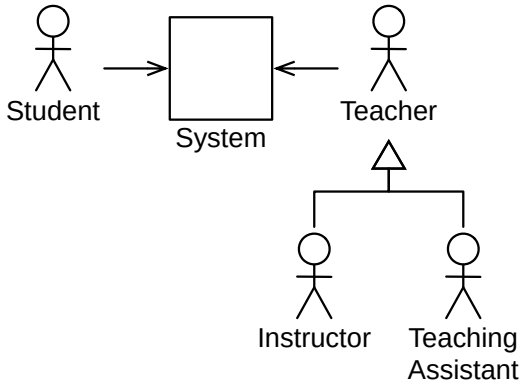


Figure 1. Use case diagram of the Athene and Artemis system. Students and teachers interact with the system. Teachers are instructors, employees of the university, or teaching assistants, who are previous students hired to assist in teaching.

However, textual exercises lead to greater variability because students need to formulate individual answers to problems. This results in high manual effort when reviewing students' answers. Assuring consistent feedback is difficult with a large number of teaching assistants. In this paper, we present an approach for computer-aided feedback for textual exercises that addresses these challenges. We implemented the approach in an open-source reference implementation, used it in multiple exercises, and evaluated its effects on the learning experience of students. In particular, we investigated the following research questions (RQ):

- RQ1 **Coverage:** How much feedback can be automatically suggested?
- RQ2 **Accuracy:** How accurate is the suggested feedback?
- RQ3 **Quality:** How do students perceive the quality of the automatically suggested feedback?

The paper is organized as follows: Section 2 describes the background of this work which consists of machine learning concepts, in particular language models. In Section 3, we show similar approaches and relate them to the approach presented in this paper. Section 4 presents the approach computer-aided feedback for textual exercises (CoFee) based on supervised learning to deal with the greater variability in the student answers. Language embeddings and clustering are used to provide individual feedback based on similarity. Section 5 shows the open-source reference implementation **Athene**[2] which is integrated into the open-source LMS Artemis[3]. Section 6 describes the courses in which the approach was used, shows the study design of the empirical evaluation with respect to Bloom's revised taxonomy [2], presents results and limitations, and discusses the findings. Section 7 concludes the paper with its main contributions and future work.

BACKGROUND: LANGUAGE MODELS

Assessing text submissions automatically requires comparing segments of those submissions and identifying similar pieces of text. Therefore, we need a measurable abstraction of a texts meaning as an intermediate representation. This paper relies on existing approaches and techniques from the domain of natural language processing (NLP), most notably language models and word embeddings, to convert a piece of text into a comparable format. Student answers can contain unknown words, incorrect use of grammar and punctuation, and false statements.

Word embedding is a feature learning technique in NLP, where words or phrases from the vocabulary are mapped to vectors of real numbers (each word is associated with a point in a vector space) [21]. The feature vector represents different aspects of the word and consequently, words that have the same meaning are assigned similar vector representations. Additionally, word embeddings are capable of capturing word analogies by examining various dimensions of the differences between word vectors [24]. For example, the analogy "king is to queen as man is to woman" should be encoded in the vector space by the vector equation $king - queen = man - woman$.

The distributed representation is learned based on the usage of the words. This allows words that are used in similar contexts to have similar representations, naturally capturing their meaning. ELMo [26] is a word embedding constructed as a task-specific combination of the intermediate layer representations in a bidirectional language model (biLM). It models complex characteristics of words-use in the language dictated by the syntax and semantics. It also captures how these uses vary across linguistic contexts, which is important for addressing polysemy in natural languages.

In a deep language model (LM), the higher-level long short term memory (LSTM) states are shown to capture context-dependent aspects of word meaning while lower-level states model aspects of the syntax. By constructing a representation out of all the layers of the LM, ELMo is able to capture both characteristics of the language. ELMo representations have three main characteristics that allow them to achieve state-of-the-art results in most common NLP downstream tasks. First, ELMo representations are contextual: the representation for each word depends on the entire context in which it is used. They are also deep: the word representations combine all layers of a deep, pre-trained language model neural network. Finally, ELMo representations are purely character based, allowing the network to use morphological clues to form robust representations for out-of-vocabulary tokens, unseen in training.

RELATED WORK

Automated essay scoring (AES) computes scores on written solutions based on previous submissions. AES systems require a perfect solution to be available up front [23, 31]. They primarily consider the distance to a perfect solution to determine the grade. Feedback is not the focus. Manual clustering and shared grading are concepts used in research [25] and commercial tools (i.e., Gradescope). Managing clusters is hard at scale, communicating the exact differences between clusters between many graders.

[2]Athene: https://github.com/ls1intum/Athene
[3]Artemis: https://github.com/ls1intum/Artemis

Atenea is a computer-assisted assessment system for scoring short answers in computer science [25] and is integrated into a web-based application. Atenea uses a database of questions with a correct sample solution for each, either written by a teacher or taken from a highly graded student's answer. When a student accesses Atenea, a random question from this pool is asked and compared to the given sample solution by utilizing a hybrid for syntax as well as semantic similarity. The system works by combining latent semantic analysis (LSA) and a modified bilingual evaluation understudy (BLEU) algorithm, with the hypothesis that syntax and semantics complement each other naturally. The combination of both NLP tools always performs better (with a higher hit rate) than their individual parts, with the authors believing that combinations of syntactical and semantical analysis can lead to even greater results for automatic text assessment.

Atenea compares student answers to a set of predefined answers. The grade is determined by its similarity to these predefined answers. This approach is limited to exercises with a narrow answer space where possible answers are known beforehand. A high variability in answers requires a large set of predefined answers, which limits the applicability of the system. The focus of the Atenea system is grading, whereas Athene is primarily focused on individual feedback. Athene does not require a predefined solution but collects knowledge on correct and incorrect solutions during the manual assessment. The evaluation of the Atenea authors focuses on a comparison of NLP techniques in the grading context and is based on a dataset. We evaluate Athene by using it in multiple courses and measuring its performance.

Powergrading is an automatic assessment approach [3]. Instead of solely focusing on providing a numerical score or a right or wrong grade, Powergrading tries to justify a certain given grade by providing feedback in the form of a comment as to why an answer is right or wrong, similar to how a teacher would do it in a classroom setting. Basu et al. propose a system, that clusters similar answers to a question so that teachers can "divide and conquer" the correction process by assigning a whole cluster with the same score and comment, therefore reducing the correction time significantly. Clustering answers to a question should happen based on a distance function, which is composed of different features and tries to learn a similarity metric between two students' answers automatically. Some of the implemented and used features that are weighted in developing this distance function used for clustering are, e.g., the difference in length between two answers, the term frequency-inverse document frequency (TF-IDF)[4] similarity of words, or the LSA vectorial score based on the entirety of Wikipedia as a training text corpus. The authors have tested their implementation with test data from the United States Citizenship Exam in 2012 with 697 examinees and concluded that around 97% of all submissions can be grouped into similar clusters so that teachers would only have to provide feedback for a single cluster and would still be able to reach and correct multiple submissions at once, therefore reducing assessment time significantly [3].

[4]TF-IDF: An information extraction statistic which indicates how significant a word is to a document [28].

Powergrading is focused on short-answer grading, where a typical answer does not exceed two sentences. Athene is not limited to a certain answer length and uses segmentation to work with multiple sentences or paragraphs. Similar to Powergrading, Athene groups segments into clusters. Both systems assume hierarchical cluster structures. Powergrading allows teachers to grade clusters rather than submissions, whereas Athene will use the cluster structure to suggest feedback for following assessments.

Gradescope is a system geared toward the assessment of handwritten homework and exam exercises [30] by scanning paper-based work. Teachers review the submissions online. Gradescope allows the teacher to dynamically create grading rubrics at the assessment time. For the assessment, teachers can group similar submissions manually for shared grading or relay on suggested groups.[5]

Athene follows a similar idea by sharing feedback with groups of answers; however, Athene groups individual segments, whereas Gradescope groups entire submissions. Gradescope allows the grader to grade multiple submissions as one, similar to Powergrading, whereas Athene shares individual feedback elements across multiple submissions. Athene requires teachers to inspect every submission and supports by suggesting feedback items. Neither system requires a training dataset of previously assessed answers. For exercises with a limited answer spectrum, Gradescope does allow the grader to assess several submissions efficiently as it reduces the number of solutions to grade. However, for exercises with high variability in answers (e.g., when asking for examples), this approach is more limited as more groups with less elements need to be reviewed.

APPROACH: COMPUTER-AIDED FEEDBACK (COFEE)

CoFee uses supervised machine learning to learn correct answers and related feedback. Figure 2 shows the main workflow how CoFee can automatically propose computer-aided feedback to students' answers. CoFee learns which answers to an exercise are correct and which are incorrect. For further submissions, the learning platform can automatically generate suggestions for similar answers or even evaluate the answers fully automatically. In doing so, the learning platform uses the knowledge of previous assessments by lecturers. The more students participate in an exercise, the more knowledge is generated and the better feedback the learning platform can suggest.

Figure 3 shows the details of the activity "preprocess answers" shown in Figure 2 and represents the basis for the three objectives mentioned above. The system analyzes incoming text (responses) using NLP, divides them into text segments, and uses them to create text clusters with similar text segments from different responses. This is done using a combination of segmentations and linguistic embeddings, in particular deeply contextualized word representations (ELMo). This allows for an understanding students' responses and for the generation of individualized feedback. In this way, a learning platform can automatically reuse manual feedback for contributions

[5]https://gradescope.com

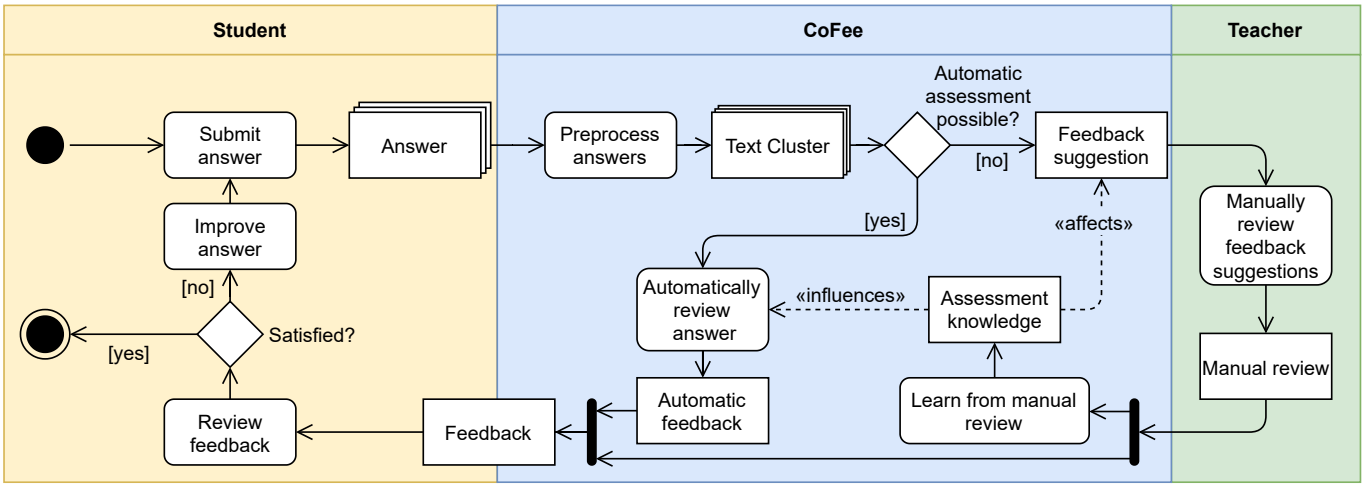


Figure 2. Workflow of automatic assessment of submissions to textual exercises based on the manual feedback of teachers. CoFee analyzes manual assessments and generates knowledge for the suggestion of computer-aided (automatic) feedback (UML activity diagram).

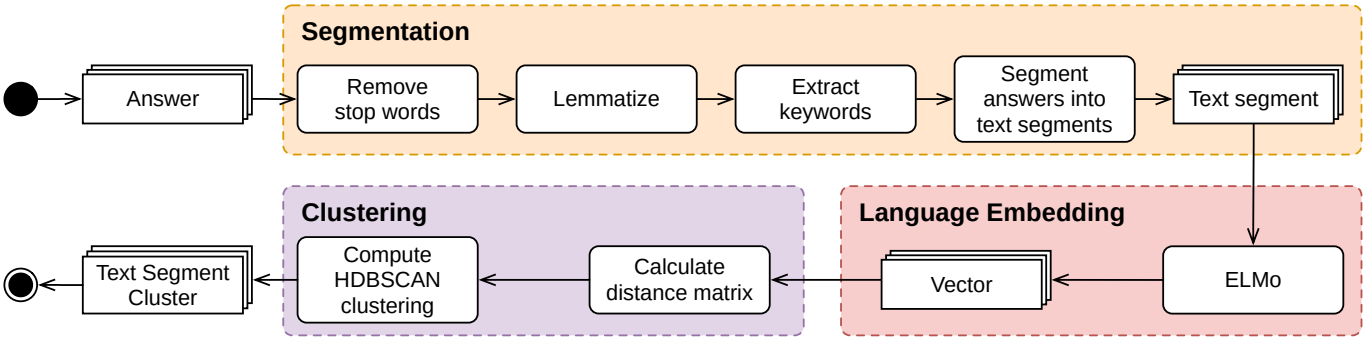


Figure 3. Detailed overview of the machine learning activities as part of the "preprocess answers" activity in Figure 2. These are used to extract text segments and build text clusters for scoring and similarity analysis (UML activity diagram).

from different students. This can reduce the workload for teachers and increase the consistency and quality of feedback to improve students' understanding.

The goal is to increase the quality and quantity of the feedback provided to students while decreasing the overall assessment time. CoFee integrates into existing learning platforms that need to provide an interface for students to submit their textual answers. We utilize a segment-based feedback concept [5], requiring assessors to provide feedback and score in relation to a segment of student's answer, resulting in relatable and reusable feedback elements.

CoFee trains its assessment model with every feedback element and thereby becomes more accurate with every new feedback element. After the assessment process, the system can detect conflicting assessments in both comments and scores. Therefore, CoFee computes the similarity among feedback comments. We claim that the distance between two text segments should be proportional to the distance between the feedback comments. If this relation is violated, CoFee prompts the teacher to review the pair of submissions and allows them to update the assessment as needed. The learning platform may only release the feedback to students after the teachers have the chance to resolve inconsistencies.

Compared to existing work, our system segments and clusters student solutions automatically. By training the system during the assessment process, we do away with the need for a reference dataset before the assessment. Furthermore, by training with highly and lowly scored solutions, we maintain a dataset to provide helpful feedback comments to support the learning process. Dynamically collecting the dataset during assessment keeps the system independent of any domain and allows for use of the system with new exercises to incorporate the latest knowledge into teaching.

REFERENCE IMPLEMENTATION (ATHENE)

We implemented CoFee in a reference implementation called Athene [4] that is integrated into the learning platform Artemis [15]. After the exercise deadline, Artemis sends the students' answers to Athene for processing. Athene will preprocess the answers before the assessment begins and will identify segments suitable for the same feedback. Figure 3 depicts the preprocessing activities. This represents the basis for the three objectives mentioned above. The system analyzes incoming student answers using NLP, divides them into text segments, and uses them to create text clusters with similar text segments from different responses. Figure 4 depicts the top-level design of the system which consists of three steps: segmentation, language embedding and clustering.

First, Athene analyzes the answers to identify segments [6, 7]. Therefore, Athene identifies common topics described in the answers from all students. A topic is represented by a keyword. To identify the important topics for an exercise, Athene counts the occurrences of lemmatized words across all students and selects the 10 most common words [7]. Within each student answer, Athene will break down all submissions into clauses. Adjacent clauses that share the same topic, represented by the use of a keyword and absence of a new keyword, are merged to form a segment. If a new keyword appears an a following clause, we identify a topic shift and start a new segment. This results in a set of topically coherent segments.

Second, Athene uses an ELMo model to convert each segment to vector form. ELMo vectors have 1,024 dimensions representing the information extracted from the segment. The vector representation allows for a comparison of segments and for the identification of similarities. Athene uses a pre-trained ELMo model [26] based on a dataset consisting of 5.5B tokens from Wikipedia and news articles.[6]

Third, Athene employs the Hierarchical Density-Based Spatial Clustering (HDBSCAN) clustering algorithm [22] to identify classes of similar text segments. Within a cluster, Athene shares manually created feedback as suggestions. The hierarchical clustering algorithm allows for a determination of the required number of clusters dynamically. Further, the hierarchical structure is used to dynamically narrow or widen the search radius depending on the availability of feedback. Narrow clusters provide more accurate feedback on the one side; however, they also limit the possible coverage. Larger clusters increase the possibility to find existing feedback to compose a suggestion; however, they also increase the risk of false feedback.

During the manual assessment, Athene sorts submissions so that it priorities submissions with the highest effect on automated grading. Submissions with several segments in clusters without feedback are prioritized, maximizing the possible coverage for automatic feedback suggestions. For each segment, Athene searches their respective clusters for existing feedback and suggests the closest feedback. Furthermore, credit points associated with feedback are used to prioritize based on the clusters' credit average. Athene's automatic feedback suggestions are displayed to teachers within Artemis as part of the review interface [5], as depicted in Figure 5. Teachers can add additional feedback to unreviewed parts of the student solution. They can either approve of the feedback suggestions or update them as they see fit.

EVALUATION

After several teachers used Athene in initial experiments in smaller courses with around 500 students, they found anecdotal evidence that the system improves the quantity and quality of feedback. The next step was to evaluate the approach in multiple exercises in the course *Introduction to Software Engineering (SE1)* with 1,800 students and 49 teaching assistants and in a second course *Networks for Monetary Transactions*. In this section, we describe the two courses and the study design of the evaluation. We show the results of the usage of Athene and discuss the findings, implications, and limitations.

[6] AllenNLP – ELMo: https://allennlp.org/elmo

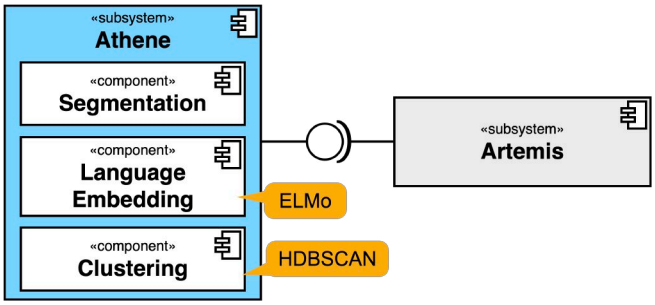


Figure 4. Top-level design of Athene, which is decomposed into three subsystems for segmentation, language embedding, and clustering and offers an API to be used in existing LMS (UML component diagram).

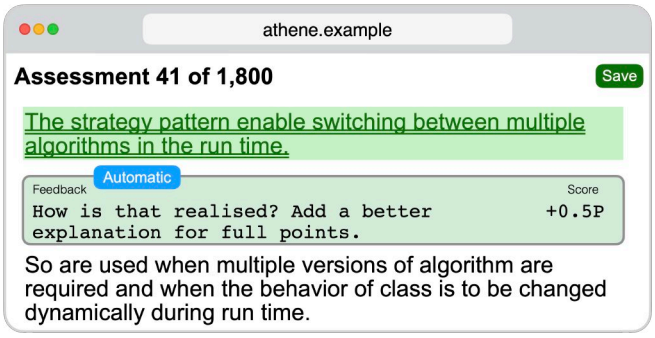


Figure 5. Example of the teacher interface: Athene presents a feedback suggestion for the first text segment with a feedback comment and a score.

Courses

The course SE1 is an introductory software engineering course, with around 1,800 registered students who are mainly computer science bachelor's students in their second semester. Students with computer science as a minor can also enroll in the course. The course covers software engineering concepts, such as requirements analysis, system and object design, testing, lifecycles, configuration management, and project management and covers UML modeling [19]. To participate in the course, students need to have fundamental programming experience (e.g., CS1). The instructors use constructive alignment [8] to align the teaching concepts and exercises with the course objectives. For each lecture, they define learning goals based on six cognitive processes in Bloom's revised taxonomy [2]. The course focuses on higher cognitive processes: students apply the concepts in concrete exercises.

Following an interactive learning approach, SE1 teaches software engineering concepts with multiple, small iterations of theory, example, exercise, solution and reflection [16]. It utilizes exercises to foster student participation [17] and to motivate the students to attend the lectures [18]. The course involves different kinds of exercises:

1. Lecture exercises as part of the (virtual) lectures
2. Group exercises solved in small ad hoc groups
3. **Homework exercises** to be solved throughout the week individually

4. Team exercises to be solved in a team in five 2-week periods
5. **Exam exercises** to assess the students' knowledge after the course has finished in multiple variants

Students were asked to submit their solutions to all exercises but group exercises to Artemis to receive an assessment with feedback and points. The students could gain bonus points for the final exam when participating in the exercises. To train software engineering and problem-solving skills, the instructors utilize programming, modeling, **textual**, and quiz exercises in the course. Automatic assessment suggestions based on Athene have been enabled for 11 textual homework exercises and six textual exam exercises.

The course *Networks for Monetary Transactions* has the learning goals to understand and assess the fundamentals, architecture, and security of domestic and international payment networks and their legal frameworks. Around 500 students participated. The teachers used Artemis to conduct an online exam during the COVID-19 pandemic. The exam consisted of 11 quiz exercises and three text exercises. Automatic assessment suggestions based on Athene were enabled for one **textual** exam exercise: *IT-Attacks*.

Bloom created the taxonomy of educational objectives, defining six categories: Knowledge, Comprehension, Application, Analysis, Synthesis, and Evaluation [9]. The *revised taxonomy* shifts the focus from static educational objectives toward a classification of cognitive processes students encounter when solving exercises [2]. The exercises conducted as part of the evaluation can be classified to train the cognitive processes *Remember*, *Understand*, but *Apply*, and *Analyze* (see Figure 6).

Table 1 lists the textual exercises and includes the cognitive process (as a category) that receives the most training in terms of the revised taxonomy. Some exercises such as *H09E02* and *H10E01* facilitate understanding by asking student to explain concepts. *H10E01*, e.g., states: "Name and explain similarities and differences between the Unified Process and Scrum in your own words". Other exercises such as *Exam 3* focus on the application of knowledge. Students need to apply their requirements elicitation skills in order to create use case descriptions based on a given problem statement.

Study Design

Figure 7 shows the study design of the evaluation that was instantiated for each exercise in which Athene was used for grading. The teacher defines the exercise in Artemis with a problem statement, grading criteria, example solutions, and a due date. The students can insert their solution in plain text on Artemis. After the due date, Artemis sends all student answers to Athene to preprocess the answers as described in the Approach section. The teachers can start reviewing the student answers as soon as Athene completes the preparation and stores the text clusters. For every student answer, the teachers create a review consisting of multiple feedback items. During the review phase, the teachers used a chat room to discuss the grading criteria as needed.

Every review can either be computer-aided, if at least one feedback item is suggested by the system, or manual. Furthermore,

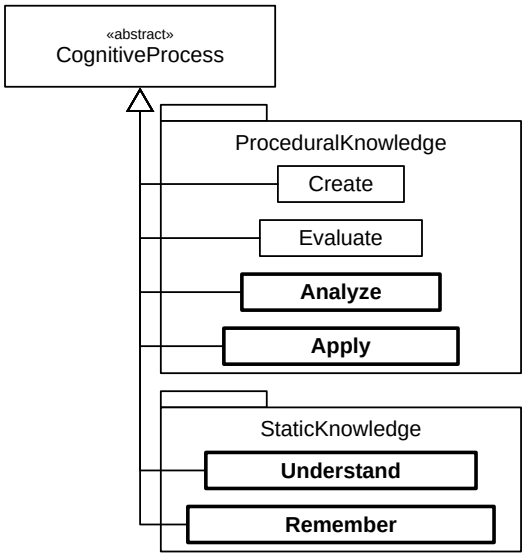


Figure 6. Exercises in the evaluation assess different cognitive processes. This taxonomy, based on the revised Bloom's taxonomy [2], depicts the hierarchy of skills. Exercises test static knowledge by testing the *remember* and *understand* skills but also *apply* and *analyze*, e.g., by identifying design issues from a system.

Athene stores intermediate versions of all feedback items to evaluate how teachers work with feedback suggestions.

After the teachers completed the review, we retrieved the classification of the reviews from the Artemis database using SQL queries. Two researchers verified the correctness of the queries. We collected the statistics on the feedback items from Athene. We inserted the measurements in a spreadsheet for further analysis and graphing. Two researchers reviewed the results for consistency and plausibility and took several samples to check individual feedback entries.

Results

The presentation of the results is based on the research questions stated at the beginning of the paper on the coverage, accuracy and quality of the approach.

Review Coverage

First, we classify the reviews into two categories: manual and computer-aided. A review is considered computer-aided, if at least one feedback item was suggested by Athene. Figure 8 depicts the classification of the reviews. On average, 26% of all reviews were computer-aided. The system performed best in exercise `Exam 1`, with 70% computer-aided reviews. Exercises `Exam 4` and `Exam 6` have the least coverage, with 2% and 8% computer-aided reviews, respectively. However, Athene was disabled for exercise `Exam 4` after a few assessments.

Finding 1: Coverage: Athene can cover up to 70% of reviews with feedback suggestions without previous training data or a predefined solution.

Exercise	Title	Category
H04E01	Coupling and Cohesion	Understand
H04E02	Analysis Models & System Design	Analyze
H04E03	Design Goal Trade-offs	Apply
H05E02	Centralized vs Decentralized Designs	Understand
H06E03	Specification & Implementation Inheritance	Apply
H06E04	Inheritance vs. Delegation	Understand
H07E03	MVC & Observer Pattern	Understand
H09E01	Advantages and Disadvantages of Scrum	Understand
H09E02	Unified Process and Scrum	Understand
H10E01	Problems using Git	Understand
H10E02	Merge Conflicts & Best Practices	Understand
Exam 1	Requirements	Apply
Exam 2	Visionary Scenarios	Apply
Exam 3	Use Cases	Apply
Exam 4	Access Control	Apply
Exam 5	Design Goal Trade-offs	Apply
Exam 6	Centralized vs. Decentralized Control	Apply
Exam 7	IT-Attacks	Remember

Table 1. Homework and exam textual exercises and their categorization following Bloom's revised taxonomy [2] used in the evaluation.

Feedback Accuracy

Second, we classified feedback items based on the intermediate versions collected during the review process. Feedback items can be classified as follows:

1. A feedback suggestion that remains unchanged is classified as *automatic*.
2. For changed suggestions, Athene computes the Levenshtein distance [20] between feedback comments. Athene classifies a changed feedback as a *typo* fix for a Levenshtein distance > 0.9.
3. Athene uses the longest common substring length and the Jaro–Winkler distance [33] to recognize feedback suggestions with a manual *extended* comment.
4. Feedback not classified in these metrics is considered as *changed*.

We analyzed the teachers' assessment work for two homework and seven exam exercises. The results depicted in Figure 9 show that on average, 85% of computer-aided feedback comment suggestions remained unchanged in their final assessment or only had minor modifications, such as corrections to typing mistakes. Furthermore, 5% of suggested comments were extended with additional feedback at the end of the suggestion to provide more details for the student. The remaining 10% of comments were changed. In these cases, the comment was either rewritten from scratch or was heavily revised.

Finding 2: Accuracy: On average, 85% of the feedback suggestions are accurate and can be published to students without modification.

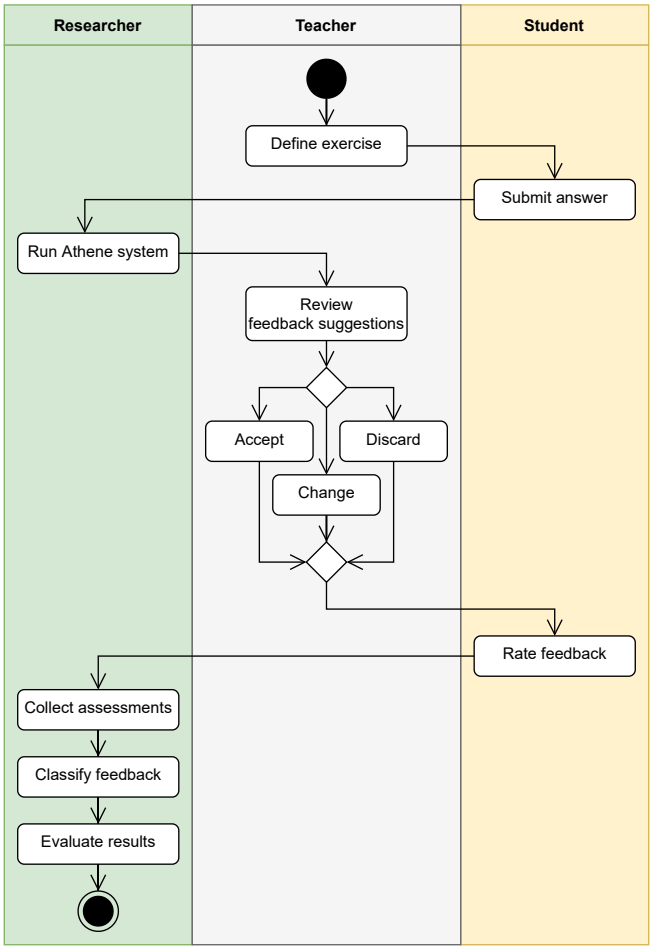


Figure 7. Research approach depicted with the involved actors and flow of events (UML activity diagram).

Perceived Quality

Third, we asked students to rate their received feedback on a 5-star scale. The students rated a total of 396 reviews out of 10,240 total reviews done by the teachers. Artemis presents the rating input underneath the feedback and asks, *"How useful is the feedback for you?"* Figure 10 depicts the distribution by star rating. In the study, 85% of the ratings were either 1-star or 5-star ratings. Students with computer-aided feedback were more likely to give a 5-star rating (72%) when compared to students who received manual feedback (62%). On the same page, computer-aided feedback received 1-star ratings less often (14%) than manual feedback (25%). Students giving a 5-star rating on average (92% and 89%, respectively) had better scores than students giving 1-star ratings (69% and 66%, respectively).

Finding 3: Quality: The computer-aided feedback in Athene has at least the same quality as manual feedback.

Limitations

This section discusses threats to the trustworthiness of the presented results, and whether the results are biased based

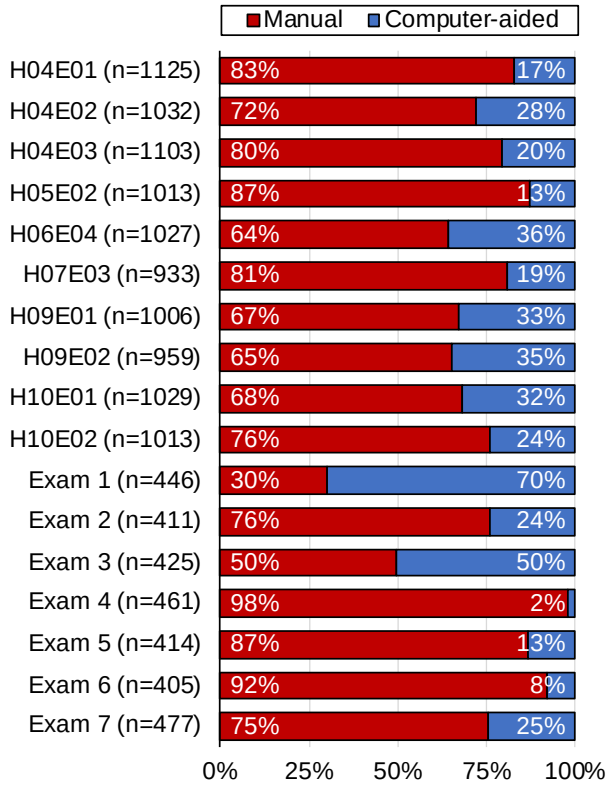


Figure 8. Exercises with their assessment ratios. Computer-aided reviews received automated grading suggestions which were reviewed by a teacher. On average, 26% reviews were computer-aided.

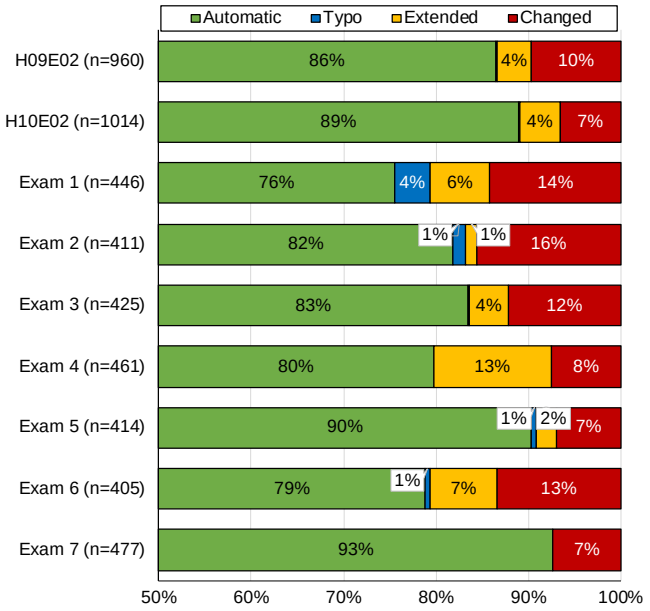


Figure 9. On average, 85% of computer-aided feedback comments remained unchanged (green) or only included minor typo fixes (blue). Furthermore, 5% were extended (yellow), and 10% were changed (red).

on the researchers' subjective point of view. We distinguish between three aspects of validity: internal validity, external validity, and construct validity [29].

Internal Validity: The accuracy of the feedback suggestions is measured by the acceptance of the teacher. A second review from a control teacher would allow for a more accurate measurement of accuracy. The teacher might be biased to confirm a feedback suggestion as it requires less effort than providing a new comment. We noticed that most teachers took the review of the automatic feedback suggestions seriously, but we cannot guarantee that some of the 49 involved teaching assistants failed to fully review the automatic feedback suggestions.

Two of the authors of this paper have been involved in teaching the course SE1 and might have influenced the empirical evaluation. However, we tried to clearly separate the research and instructor perspective. Two additional instructors have been involved in the course SE1 who are not authors of this paper, and the third author of the paper reviewed the results carefully without being involved in the course. In addition, we observed similar results in the second course, which was taught by an independent instructor who was not involved in the research.

External Validity: Most analyzed exercises have been in the domain of software engineering and computer science in the same university. While we believe that the approach is generalizable for other domains, we have not shown this in this study.

Construct Validity: The validity of the ratings might be affected by the wording of the question and by the score that the students received. Students with a higher score are typically more satisfied and less likely to complain about the quality of the feedback. Therefore, a good rating does not necessarily mean that the feedback had a good quality. Another limitation could be the fact that students like the approach of getting feedback. The ratings measure the perceived quality which is subjective. We can only infer the quality based on the ratings. Therefore, we consider Finding 3 on the quality of the ratings as anecdotal evidence.

Discussion

The review coverage of Athene is higher for exercises that do not ask for examples, but rather require students to work based on a given example. In the exercises `Exam 1` and `Exam 3`, students were asked to extract requirements or use cases from a given problem statement. In those exercises, the coverage was above the average with 70% and 50%, respectively. These questions still require students to apply problem-solving skills, but limit the variability of the answers. This leads to more similar answers and more reusable feedback.

Exercises asking for examples, such as the SE1 homework exercises, have lower review coverage of between 17% and 36%. This may be due to the increased variability of answers with different examples. As Athene tries to find similar text segments, it is more difficult to find a group with shared segments as students can describe all possible examples. Therefore, it is less likely to find reusable feedback among students.

Athene reuses reviews from teachers. The quality of the feedback suggestions depends on the quality of the manual feed-

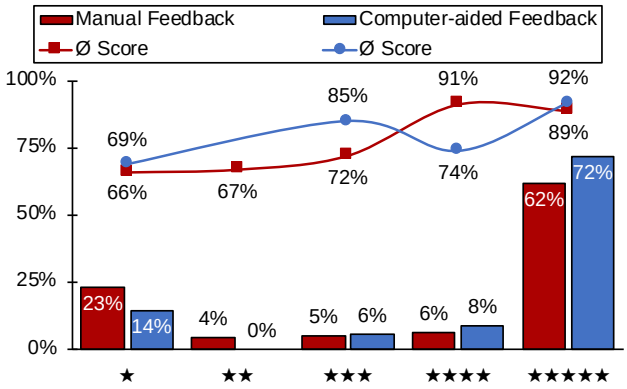


Figure 10. All ratings for SE1 homework (HXX) exercises by star rating. In this figure, ratings are grouped by the assessment type *Manual* ($n = 325$) or *Computer-aided* ($n = 71$). The average score in percent is depicted per rating and assessment type. In the study, 396 out of 10,240 reviews were rated by students.

back provided during the teacher reviews. If teachers provide incorrect manual feedback, Athene will not be able to provide correct feedback suggestions. In the example of SE1, the teachers who review the submission consist primarily of teaching assistants, who have limited experience in grading or providing feedback.

Nevertheless, the approach can improve the review process as it allows instructors to handle larger amounts of reviews or to inspect examples. Other systems presented in the Related Work section suggest comparing answers only with a sample solution provided by an instructor [25], thus reducing the variability in the solution space, which might limit the creativity of the students. However, creativity is an important aspect in software engineering education [14].

CONCLUSION

This paper presents three main contributions:

1. The machine-learning based **approach** CoFee was presented to suggest feedback for textual exercises. The approach is based on segmentation and similarity-based clustering. It reuses feedback on segments within the same cluster and learns which aspects of student answers are correct during the assessment.

2. Athene, a **reference implementation** of CoFee, using the ELMo language model and the HDBSCAN clustering algorithm was presented. Athene is integrated into Artemis and published as open-source software under the MIT license.[7]

3. An **empirical evaluation** of Athene in two courses with 2,300 students and 53 teachers in 17 textual exercises was conducted. The results of the quantitative evaluation in these exercises show that Athene can suggest up to 70% of the feedback with an average accuracy of 85%. Ratings provide first indications that the quality improves when compared to purely manual assessments.

[7] Athene: https://github.com/ls1intum/Athene

The evaluation also shows that these numbers depend on the type of the textual exercise and on the variability of the possible solutions. A higher variance in correct solutions leads to less coverage because of fewer similarities in the student answers.

Athene does not require training data before the reviewing process to learn correct answers and feedback suggestions. Instead, it collects knowledge during the assessment. This incremental process allows instructors to change or introduce new exercises as needed, preventing students from submitting solutions from previous years. However, when reusing past exercises, Athene could profit from additional knowledge captured in these reviews. Future work needs to evaluate whether training data from the same exercise in previous years can improve the coverage or accuracy of feedback suggestions.

REFERENCES

[1] Carlos Alario-Hoyos, Carlos Kloos, Iria Estévez-Ayres, Carmen Fernández-Panadero, Jorge Blasco, Sergio Pastrana, and J Villena-Román. 2016. Interactive activities: the key to learning programming with MOOCs. *European Stakeholder Summit on Experiences and Best Practices in and Around MOOCs* 319 (2016).

[2] Lorin W. Anderson, David R. Krathwohl, Peter W. Airasian, Kathleen A. Cruikshank, Richard E. Mayer, Paul R. Pintrich, James Raths, and Merlin C. Wittrock. 2001. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longmans Green.

[3] Sumit Basu, Chuck Jacobs, and Lucy Vanderwende. 2013. Powergrading: a clustering approach to amplify human effort for short answer grading. *Transactions of the Association for Computational Linguistics* 1 (2013), 391–402.

[4] Jan Philip Bernius. 2021. Toward Computer-Aided Assessment of Textual Exercises in Very Large Courses. In *52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21)*. 1386.

[5] Jan Philip Bernius and Bernd Bruegge. 2019. Toward the Automatic Assessment of Text Exercises. In *2nd Workshop on Innovative Software Engineering Education*. Stuttgart, Germany, 19–22.

[6] Jan Philip Bernius, Anna Kovaleva, and Bernd Bruegge. 2020a. Segmenting Student Answers to Textual Exercises Based on Topic Modeling. In *17th Workshop Software Engineering im Unterricht der Hochschulen (SEUH)*. Stuttgart, Germany, 72–73.

[7] Jan Philip Bernius, Anna Kovaleva, Stephan Krusche, and Bernd Bruegge. 2020b. Towards the Automation of Grading Textual Student Submissions to Open-ended Questions. In *European Conference on Software Engineering Education*. ACM, 61–70.

[8] John Biggs. 2003. Aligning teaching and assessing to course objectives. *Teaching and learning in higher education: New trends and innovations* 2 (2003), 13–17.

[9] Benjamin S. Bloom, Max D. Engelhart, Edward J. Furst, Walker H. Hill, and David R. Krathwohl. 1956. *Taxonomy of educational objectives. The classification of educational goals. Handbook 1: Cognitive domain*. Longmans Green.

[10] Richard P. Feynman. 1994. *Six Easy Pieces*.

[11] Richard Higgins, Peter Hartley, and Alan Skelton. 2002. The conscientious consumer: Reconsidering the role of assessment feedback in student learning. *Studies in higher education* 27, 1 (2002), 53–64.

[12] Alastair Irons. 2007. *Enhancing learning through formative assessment and feedback*. Routledge.

[13] Paul Kirschner, John Sweller, and Richard Clark. 2006. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist* 41, 2 (2006), 75–86.

[14] Stephan Krusche, Bernd Bruegge, Irina Camilleri, Kirill Krinkin, Andreas Seitz, and Cecil Wöbker. 2017a. Chaordic Learning: A Case Study. In *39th International Conference on Software Engineering: Software Engineering Education and Training*. IEEE, 87–96.

[15] Stephan Krusche and Andreas Seitz. 2018. ArTEMiS: An Automatic Assessment Management System for Interactive Learning. In *49th ACM Technical Symposium on Computer Science Education (SIGCSE)*. 284–289.

[16] Stephan Krusche and Andreas Seitz. 2019. Increasing the Interactivity in Software Engineering MOOCs - A Case Study. In *52nd Hawaii International Conference on System Sciences*. 1–10.

[17] Stephan Krusche, Andreas Seitz, Jürgen Börstler, and Bernd Bruegge. 2017b. Interactive learning: Increasing student participation through shorter exercise cycles. In *19th Australasian Computing Education Conference*. ACM, 17–26.

[18] Stephan Krusche, Nadine von Frankenberg, and Sami Afifi. 2017c. Experiences of a Software Engineering Course based on Interactive Learning. In *Tagungsband des 15. Workshops Software Engineering im Unterricht der Hochschulen (SEUH)*. CEUR, 32–40.

[19] Stephan Krusche, Nadine von Frankenberg, Lara Marie Reimer, and Bernd Bruegge. 2020. An interactive learning method to engage students in modeling. In *International Conference on Software Engineering: Software Engineering Education and Training*. 12–22.

[20] Vladimir I. Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics-Doklady* 10, 8 (1966), 707–710.

[21] Yang Li and Tao Yang. 2018. *Word Embedding for Understanding Natural Language: A Survey*. Springer International Publishing, Cham, 83–104.

[22] Leland McInnes and John Healy. 2017. Accelerated Hierarchical Density Based Clustering. In *International Conference on Data Mining Workshops*. 33–42.

[23] Tom Mitchell, Terry Russell, Peter Broomhead, and Nicola Aldridge. 2002. Towards robust computerised marking of free-text responses. (2002).

[24] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global Vectors for Word Representation. In *Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Doha, Qatar, 1532–1543.

[25] Diana Perez, Alfio Gliozzo, Carlo Strapparava, Enrique Alfonseca, Pilar Rodríguez, and Bernardo Magnini. 2005. Automatic Assessment of Students' Free-Text Answers Underpinned by the Combination of a BLEU-Inspired Algorithm and Latent Semantic Analysis. 358–363.

[26] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep Contextualized Word Representations. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2227–2237.

[27] Karl Raimund Popper. 1972. *Objective Knowledge*.

[28] Juan Ramos. 2003. Using TF-IDF to Determine Word Relevance in Document Queries. In *1st instructional conference on machine learning*, Vol. 242. Piscataway, NJ, 133–142.

[29] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. 2012. *Case Study Research in Software Engineering*. John Wiley & Sons, Inc.

[30] Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. 2017. Gradescope: A Fast, Flexible, and Fair System for Scalable Assessment of Handwritten Work. In *4th Conference on Learning @ Scale*. ACM, 81–88.

[31] Jana Sukkarieh, Stephen G Pulman, and Nicholas Raikes. 2003. Automarking: using computational linguistics to score short, free-text responses. (2003).

[32] Reed Williams and Thomas Haladyna. 1982. Logical Operations for Generating Intended Questions (LOGIQ): A typology for higher level test items. *A technology for test-item writing* (1982), 161–186.

[33] William E. Winkler. 1990. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. In *Section on Survey Research*. 354–359.

Appendix A

Licenses

Multiple publications published through IEEE, ACM, ScholarSpace, and CEUR are the basis for this habilitation. The author of the habilitation requested permission to reuse all publication material from the co-authors and the publishers via email. All co-authors and publishers granted permission for reuse. In addition, students of supervised bachelor and master theses permitted the reuse of content (e.g., text, figures, and tables) in this habilitation.

A.1 IEEE

Permission grants have been retrieved from the IEEE Xplore Digital Library, as shown in Figure A.1.

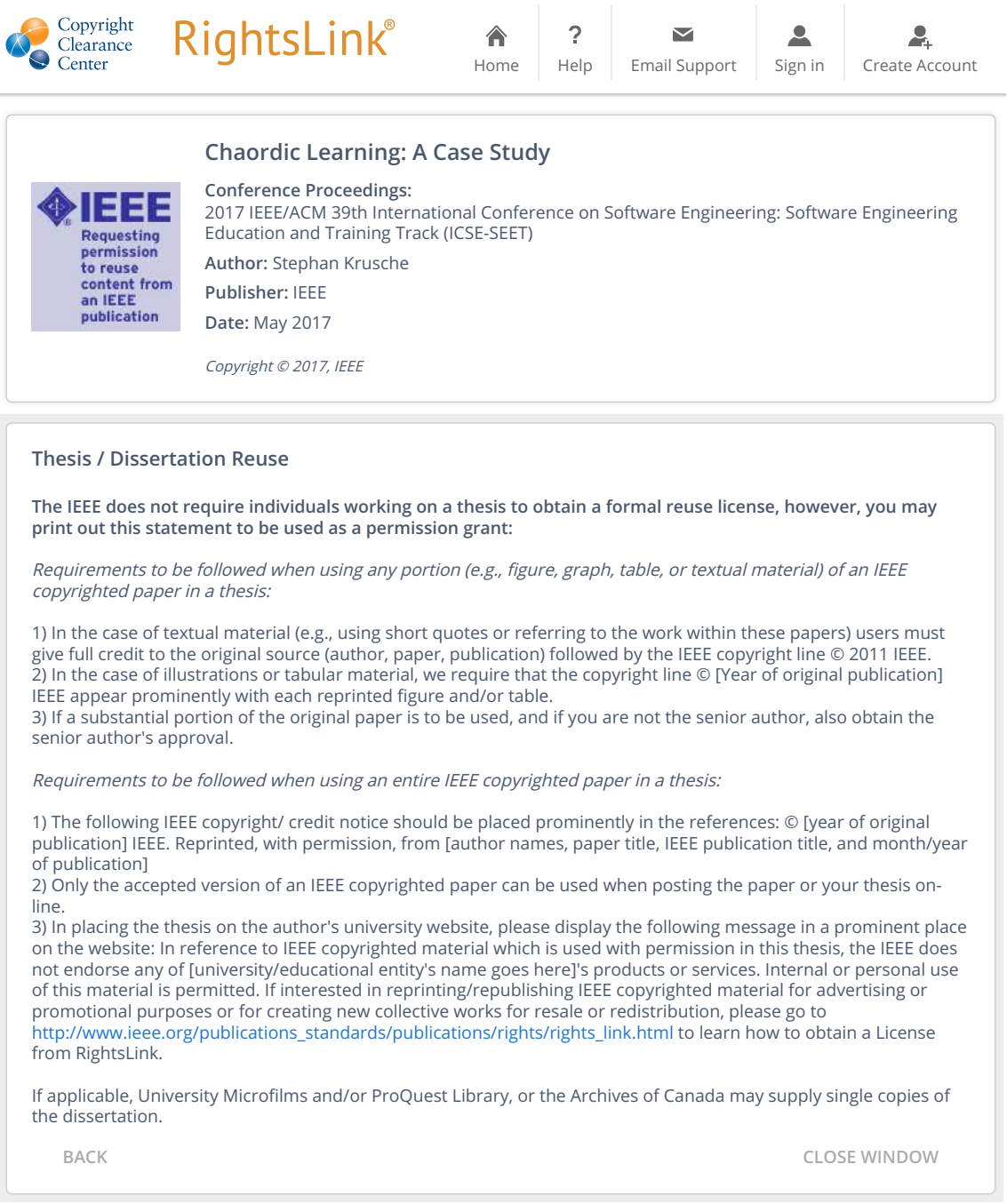


Figure A.1: IEEE permission grant for the publication [KBC+17].

A.2 ACM

Content of the ACM-published conference papers [KSBB17], [KS18], [KvFRB20] and [BKKB21]) and journal article [KDXB18] is used in this habilitation based on ACM Author Rights, which are available on https://authors.acm.org/author-resources/author-rights. The relevant paragraphs are the following:

REUSE

Authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is not the editor, requires permission and usually a republication fee.
- **Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included.** Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).
- Commercially produced course-packs that are sold to students require permission and possibly a fee.

A.3 ScholarSpace

The paper [KS19] has been submitted to the Hawaii International Conference on System Sciences (HICSS), which hosted a special track on Software Engineering Education and Training (as part of the CSEE&T community[1]). HICSS published the proceedings on ScholarSpace, an open-access, digital, and institutional repository for the University of Hawaii at Manoa community[2].

HICSS papers have been submitted with Creative Commons licenses (CC-BY-NC-ND[3]). Authors own the copyright of their papers. They can disseminate the paper freely after the conference has taken place.

[1] https://conferences.computer.org/cseet
[2] https://scholarspace.manoa.hawaii.edu.
[3] https://creativecommons.org/licenses/by-nc-nd/2.0/

A.4 CEUR

The workshop papers [KvFA17] and [LKvFB19] have been published on CEUR Workshop Proceedings (CEUR-WS.org), a free open-access publication service at Sun SITE Central Europe operated under the umbrella of RWTH Aachen University. CEUR-WS.org is a recognized ISSN publication series More information can be obtained on http://ceur-ws.org/HOWTOSUBMIT.html.

The relevant paragraphs are the following:

- The copyright and any similar right for the proceedings and **all included material remain with the papers' authors/owners** (for the individual papers).
- User rights are expressed by the license dedication of a volume and its contents. New volumes after summer 2019 adopt the Creative Commons Attribution (CC-BY 4.0) license. Older volumes permit the use for private and academic purposes. See the appropriate license dedication in the volume and/or the digital artefact/paper in a volume.
- Re-publication of material published in CEUR-WS.org volumes requires the permission by the copyright holders, i.e. to the paper's authors.
- CEUR-WS.org provides published proceedings and papers as 'open access'. CEUR-WS.org employs no access control.

List of Figures

List of Tables

Bibliography

[AB99] David Arnow and Oleg Barshay. Webtoteach: an interactive focused programming exercise system. In *29th Annual Frontiers in Education Conference*, volume 1, pages 12A9–39. IEEE, 1999.

[ACF+13] Rondall E Allen, Jeffrey Copeland, Andrea S Franks, Reza Karimi, Marianne McCollum, David J Riese, and Anne YF Lin. Team-based learning in us colleges and schools of pharmacy. *American journal of pharmaceutical education*, 77(6), 2013.

[AFR08] M. Amelung, P. Forbrig, and D. Rösner. Towards Generic and Flexible Web Services for E-Assessment. *SIGCSE Bulletin*, pages 219–224, 2008.

[AHEAK+17] Carlos Alario-Hoyos, Iria Estévez-Ayres, Delgado Kloos, Raquel M Crespo-García, Julio Villena-Román, and Jorge Ruiz-Magaña. Integration of external tools to foster learner interaction in moocs: The example of codeboard. In *Proceedings of the International Conference MOOC-Maker*, pages 1–10, 2017.

[Ald77] Clayton Alderfer. Group and intergroup relations. *Improving life at work*, 227:296, 1977.

[ALSCiMP08] Benjamin Auffarth, Maite López-Sánchez, Jordi Campos i Miralles, and Anna Puig. System for automated assistance in correction of programming exercises (sac). In *International Congress University Teaching and Innovation*, pages 104–113, 2008.

[AM05] K. Ala-Mutka. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, pages 83–102, 2005.

[Bal16] Ziad Balaa. Developing an exercise management system for e-learning. In *Sixth International Conference on Digital Information Processing and Communications*, pages 93–96. IEEE, 2016.

[BE91] Charles Bonwell and James Eison. *Active Learning: Creating Excitement in the Classroom*. ASHE-ERIC Higher Education Reports, 1991.

[Bea10] Colin Beard. *The experiential learning toolkit: Blending practice with concepts*. Kogan Page Publishers, 2010.

[BEF+56] Benjamin Bloom, Max Engelhart, Edward Furst, Walker Hill, and David Krathwohl. Taxonomy of educational objectives: The classification of educational goals. 1956.

[BF98] David Boud and Grahame Feletti. *The challenge of problem-based learning*. Psychology Press, 1998.

[BFPS17] Oliver Bott, Peter Fricke, Uta Priss, and Michael Striewe. *Automatisierte Bewertung in der Programmierausbildung*. Waxmann, 2017.

[BGNM04] Michael Blumenstein, Steve Green, Ann Nguyen, and Vallipuram Muthukkumarasamy. Game: A generic automated marking environment for programming assessment. In *International Conference on Information Technology: Coding and Computing*, volume 1, pages 212–216. IEEE, 2004.

[BHS17] Joachim Breitner, Martin Hecker, and Gregor Snelting. Der Grader Praktomat. In *Automatisierte Bewertung in der Programmierausbildung*, number 10, pages 159–172. Waxmann, 2017.

[Big03] John Biggs. Aligning teaching and assessing to course objectives. *Teaching and learning in higher education: New trends and innovations*, 2:13–17, 2003.

[BK05] Christian Bauer and Gavin King. *Hibernate in action*, volume 1. Manning Greenwich, 2005.

[BKA15] Bernd Bruegge, Stephan Krusche, and Lukas Alperowitz. Software engineering project courses with industrial clients. *ACM Transactions on Computing Education*, 15(4):17:1–17:31, 2015.

[BKB21] Jan Philip Bernius, Stephan Krusche, and Bernd Bruegge. A Machine Learning Approach for Suggesting Feedback in Textual Exercises in Large Courses. In *Proceedings of the 8th Conference on Learning at Scale*. ACM, 2021.

[BKKB21] Jan Philip Bernius, Anna Kovaleva, Stephan Krusche, and Bernd Bruegge. Towards the Automation of Grading Textual Student Submissions to Open-ended Questions. In *Proceedings of the 4th European Conference on Software Engineering Education*, pages 61–70. ACM, 2021.

[BM04] Carolyn Birmingham and Mary McCord. Group process research: Implications for using learning groups. *Team-based learning: A transformative use of small groups in college teaching*, pages 73–93, 2004.

[CBB18] Ricardo Conejo, Beatriz Barros, and Manuel Bertoa. Automated assessment of complex programming tasks using siette. *IEEE Transactions on Learning Technologies*, 12(4):470–484, 2018.

[CBH91] Allan Collins, John Seely Brown, and Ann Holum. Cognitive apprenticeship: Making thinking visible. *American educator*, 1991.

[CGM+04] Ricardo Conejo, Eduardo Guzmán, Eva Millán, Mónica Trella, José Luis Pérez-De-La-Cruz, and Antonia Ríos. Siette: A web-based tool for adaptive testing. *International Journal of Artificial Intelligence in Education*, 14(1):29–61, 2004.

[CHCG14] Jennifer Campbell, Diane Horton, Michelle Craig, and Paul Gries. Evaluating an inverted cs1. In *Proceedings of the 45th technical symposium on Computer science education*, pages 307–312. ACM, 2014.

[CKLO03] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. On automated grading of programming assignments in an academic institution. *Computers & Education*, 41(2):121–131, 2003.

[Col92] Allan Collins. Toward a design science of education. In *New directions in educational technology*, pages 15–22. Springer, 1992.

[Cra46] Harold Cramer. Mathematical methods of statistics. *Princeton*, 1946.

[Dal99] Charlie Daly. Roboprof and an introductory computer programming course. *SIGCSE Bulletin*, 31(3):155–158, 1999.

[Dew38] John Dewey. *Experience & Education*. IBM, 1938.

[DGR+15] Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. Automatic grading of programming exercises in a mooc using the inginious platform. *European Stakeholder Summit on experiences and best practices in and around MOOCs*, pages 86–91, 2015.

[DJF09] Pierre Dillenbourg, Sanna Järvelä, and Frank Fischer. The evolution of research on computer-supported collaborative learning. In *Technology-enhanced learning*, pages 3–19. Springer, 2009.

[DLO05] C. Douce, D. Livingstone, and J. Orwell. Automatic Test-Based Assessment of Programming: A Review. *Journal on Education Resources in Computing*, 2005.

[dSMB11] Draylson Micael de Souza, José Carlos Maldonado, and Ellen Francine Barbosa. An environment for the submission and evaluation of programming assignments based on testing activities. *Conference on Software Engineering Education and Training*, 2011.

[DSP10] Steffi Domagk, Ruth Schwartz, and Jan Plass. Interactivity in multimedia learning: An integrated model. *Computers in Human Behavior*, 26(5), 2010.

[Edw03] S. Edwards. Improving student performance by evaluating how well students test their own programs. *Journal on Education Resources in Computing*, 2003.

[EFF+21] Christina Ehrlinger, Thomas Fritsch, Michael Fruth, Franz Lehner, and Stefanie Scherzinger. Toolunterstützung für den Übungsbetrieb in der datenbanklehre: Erfahrungen mit der software praktomat. *Datenbank-Spektrum*, pages 1–8, 2021.

[EKN+11] Emma Enström, Gunnar Kreitz, Fredrik Niemelä, Pehr Söderman, and Viggo Kann. Five years with kattis—using an automated assessment system in teaching. In *Frontiers in Education Conference*. IEEE, 2011.

[FD84] Susan Brown Feichtner and Elaine Actis Davis. Why some groups fail: A survey of students' experiences with learning groups. *Organizational Behavior Teaching Review*, 9(4):58–73, 1984.

[Fel11] Patrick Felicia. *Handbook of research on improving learning and motivation through educational games: Multidisciplinary approaches: Multidisciplinary approaches*. iGi Global, 2011.

[FH+97] Stephen Fisher, Terri Hunter, et al. Team or group? managers' perceptions of the differences. *Journal of Managerial Psychology*, 1997.

[FTHS99] Eric Foxley, A Tsintsifas, CA Higgins, and Pavlos Symeonidis. Ceilidh, a system for the automatic evaluation of students programming work. In *Proceedings of CBLISS*, 1999.

[Gar16] Robert Garmann. Graja-autobewerter für java-programme. Technical report, Hochschule Hannover, 2016. Retrieved May 16, 2021 from https://serwiss.bib.hs-hannover.de/frontdoor/deliver/index/docId/941/file/BerichtGraja20160331.pdf.

[GB87] Thomas Good and Jere Brophy. *Looking in classrooms*. Harper & Row, 1987.

[GBSO17] Florian Grummel, Ayla Brettle, David Schuster, and Rainer Oechsle. Automatic evaluation of python applications. In *Proceedings of the 3rd Workshop Automatische Bewertung von Programmieraufgaben*, 2017.

[GD95] Scott Grabinger and Joanna Dunlap. Rich environments for active learning: A definition. *Research in learning Technology*, 3(2), 1995.

[Gee20] Michael Geers. *Micro Frontends in Action*. Manning Publications, 2020.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson, 1994.

[GHV+19] Daniel Galan, Ruben Heradio, Hector Vargas, Ismael Abad, and Jose Cerrada. Automated assessment of computer programming practices: The 8-years uned experience. *IEEE Access*, 7:130113–130119, 2019.

[GK04] Randy Garrison and Heather Kanuka. Blended learning: Uncovering its transformative potential in higher education. *The internet and higher education*, 2004.

[GMT+92] Anne Goodsell, Michelle Maher, Vincent Tinto, Barbara Leigh Smith, and Jean MacGregor. Collaborative learning: A sourcebook for higher education. 1992.

[GMTW13] F. Grünewald, C. Meinel, M. Totschnig, and C. Willems. Designing MOOCs for the Support of Multiple Learning Styles. In *European Conference on Technology Enhanced Learning*, pages 371–382. Springer, 2013.

[GR10] Peter Goodyear and Symeon Retalis. *Technology-enhanced learning: Design Patterns and Pattern Languages*. Sense Publishers, 2010.

[GTR+10] Eladio Gutiérrez, María A Trenas, Julián Ramos, Francisco Corbera, and Sergio Romero. A new moodle module supporting automatic verification of vhdl-based assignments. *Computers & Education*, 54(2):562–577, 2010.

[Gus17] Helmar Gust. Der Grader VEA. In *Automatisierte Bewertung in der Programmierausbildung*, number 14, pages 225–239. Waxmann, 2017.

[Hec15] Sarah Heckman. An empirical study of in-class laboratories on student learning of linear data structures. In *Proceedings of the 11th annual International Conference on International Computing Education Research*, pages 217–225. ACM, 2015.

[HGST05] Colin Higgins, Geoffrey Gray, Pavlos Symeonidis, and Athanasios Tsintsifas. Automated assessment and experiences of teaching programming. *Journal on Educational Resources in Computing*, 5(3), 2005.

[HM93] Juha Hyvönen and Lauri Malmi. Trakla-a system for teaching algorithms using email and a graphical editor. In *HYPERMEDIA, Vaasa*, pages 141–147. 1993.

[Hoa02] Christopher Hoadley. Creating context: Design-based research in creating and understanding cscl. 2002.

[HR20] Jimmy Ming Hong and Preman Rajalingam. Geographic trends in team-based learning (tbl) research and implementation in medical schools. *Health Professions Education*, 6(1):47–60, 2020.

[HT07] John Hattie and Helen Timperley. The power of feedback. *Review of educational research*, 77(1):81–112, 2007.

[HVVP21] Telle Hailikari, Viivi Virtanen, Marjo Vesalainen, and Liisa Postareff. Student perspectives on how different elements of constructive alignment support active learning. *Active Learning in Higher Education*, 2021.

[IAKS10] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling Conference on Computing Education Research*, pages 86–93. ACM, 2010.

[IDDBI15] Lukas Iffländer, Alexander Dallmann, Philip Daniel-Beck, and Marianus Ifland. Pabs - a programming assignment feedback system. In *Proceedings of the 2nd Workshop Automatische Bewertung von Programmieraufgaben*, 2015.

[J+91] David Johnson et al. *Cooperative Learning: Increasing College Faculty Instructional Productivity. ASHE-ERIC Higher Education Report*. ERIC, 1991.

[JBK06] Grace Johnson, Gordon Bruner, and Anand Kumar. Interactivity and its facets revisited: Theory and empirical test. *Journal of Advertising*, 35(4), 2006.

[JF11] Wang Jing and Rui Fan. The research of hibernate cache technique and application of EhCache component. In *3rd International Conference on Communication Software and Networks*, pages 160–162. IEEE, 2011.

[JG67] Edward Jones and Harold Gerard. Foundations of social psychology. 1967.

[JGB05] Mike Joy, Nathan Griffiths, and Russell Boyatt. The boss online submission and assessment system. *Journal on Educational Resources in Computing*, 5(3), 2005.

[JJS91] David Johnson, Roger Johnson, and Karl Smith. *Active learning: Cooperation in the college classroom*. Interaction Book Company, 1991.

[JJWH06] Stavenga Jong, A Jan, Ronny Wierstra, and José Hermanussen. An exploration of the relationship between academic and experiential learning approaches in vocational education. *British Journal of Educational Psychology*, 76(1):155–169, 2006.

[JL99] Mike Joy and Michael Luck. Plagiarism in programming assignments. *IEEE Transactions on education*, 42(2):129–133, 1999.

[Joh13] Mat Johns. *Getting Started with Hazelcast*. Packt Publishing, 2013.

[JU97] David Jackson and Michelle Usher. Grading student programs using assyst. In *Proceedings of the twenty-eighth technical symposium on Computer science education (SIGCSE)*, pages 335–339, 1997.

[KA14] Stephan Krusche and Lukas Alperowitz. Introduction of Continuous Delivery in Multi-Customer Project Courses. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 335–343. IEEE, 2014.

[KABW14] Stephan Krusche, Lukas Alperowitz, Bernd Bruegge, and Martin Wagner. Rugby: An Agile Process Model Based on Continuous Delivery. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pages 42–50. ACM, 2014.

[KBB16] Stephan Krusche, Mjellma Berisha, and Bernd Bruegge. Teaching Code Review Management using Branch Based Workflows. In *Companion Proceedings of the 38th International Conference on Software Engineering*. IEEE, 2016.

[KBC+17] Stephan Krusche, Bernd Bruegge, Irina Camilleri, Kirill Krinkin, Andreas Seitz, and Cecil Wöbker. Chaordic Learning: A Case Study. In *39th International Conference on Software Engineering: Software Engineering Education and Training*, pages 87–96. IEEE, 2017.

[KDXB18] Stephan Krusche, Dora Dzvonyar, Han Xu, and Bernd Bruegge. Software Theater—Teaching Demo-Oriented Prototyping. *Transactions on Computing Education (TOCE)*, 18(2):1–30, 2018.

[KFT+14] Barry Kurtz, James Fenwick, Rahman Tashakkori, Ahmad Esmail, and Stephen Tate. Active learning during lecture using tablets. In *Proceedings of the 45th technical symposium on computer science education*, pages 121–126. ACM, 2014.

[KHK+15] Ville Karavirta, Riku Haavisto, Erkki Kaila, Mikko-Jussi Laakso, Teemu Rajala, and Tapio Salakoski. Interactive learning content for introductory computer science course using the ville exercise framework. In *International Conference on Learning and Teaching in Computing and Engineering*, pages 9–16. IEEE, 2015.

[KKS+20] Angelika Kaplan, Jan Keim, Yves Schneider, Maximilian Walter, Dominik Werle, Anne Koziolek, and Ralf H Reussner. Teaching programming at scale. In *Tagungsband des 17. Workshops Software Engineering im Unterricht der Hochschulen (SEUH)*, pages 2–6. CEUR, 2020.

[KMMI13] Aditi Kothiyal, Rwitajit Majumdar, Sahana Murthy, and Sridhar Iyer. Effect of think-pair-share in a large cs1 class: 83% sustained engagement. In *Proceedings of the 9th annual international conference on International computing education research*, pages 137–144. ACM, 2013.

[KMS03] Ari Korhonen, Lauri Malmi, and Panu Silvasti. Trakla2: a framework for automatically assessed visual algorithm simulation exercises. In *Proceedings of the 3rd Koli calling international conference on computing education research*, pages 48–56. University of Joensuu and University of Helsinki, 2003.

[Kol84] David Kolb. *Experiential learning: Experience as the source of learning and development*, volume 1. Prentice Hall, 1984.

[KP14] Adrian Kirkwood and Linda Price. Technology-enhanced learning and teaching in higher education: what is 'enhanced'and how do we know? a critical literature review. *Learning, media and technology*, 39(1):6–36, 2014.

[Kra02] David Krathwohl. A revision of bloom's taxonomy: An overview. *Theory into practice*, 41(4):212–218, 2002.

[KRTB16] Stephan Krusche, Barbara Reichart, Paul Tolstoi, and Bernd Bruegge. Experiences from an experiential learning course on games development. In *Proceedings of the 47th Technical Symposium on Computer Science Education (SIGCSE)*, pages 582–587, 2016.

[KS18] Stephan Krusche and Andreas Seitz. ArTEMiS: An Automatic Assessment Management System for Interactive Learning. In *Proceedings of the 49th Technical Symposium on Computer Science Education (SIGCSE)*, pages 284–289. ACM, 2018.

[KS19] Stephan Krusche and Andreas Seitz. Increasing the Interactivity in Software Engineering MOOCs - A Case Study. In *52nd Hawaii International Conference on System Sciences*, pages 1–10, 2019.

[KSBB17] Stephan Krusche, Andreas Seitz, Jürgen Börstler, and Bernd Bruegge. Interactive Learning: Increasing Student Participation Through Shorter Exercise Cycles. In *Proceedings of the 19th Australasian Computing Education Conference*, pages 17–26. ACM, 2017.

[KSC06] Paul Kirschner, John Sweller, and Richard Clark. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist*, 41(2):75–86, 2006.

[KSZ02] Jens Krinke, Maximilian Störzer, and Andreas Zeller. Web-basierte programmierpraktika mit praktomat. *Softwaretechnik-Trends*, 22(3):51–53, 2002.

[KvFA17] Stephan Krusche, Nadine von Frankenberg, and Sami Afifi. Experiences of a Software Engineering Course based on Interactive Learning. In *Tagungsband des 15. Workshops Software Engineering im Unterricht der Hochschulen (SEUH)*, pages 32–40. CEUR, 2017.

[KvFRB20] Stephan Krusche, Nadine von Frankenberg, Lara Marie Reimer, and Bernd Bruegge. An Interactive Learning Method to Engage Students in Modeling. In *Proceedings of the 42nd International Conference on Software Engineering: Software Engineering Education and Training*, pages 12–22. ACM, 2020.

[LJ95] Michael Luck and Mike Joy. Automatic submission in an evolutionary approach to computer science teaching. *Computers & Education*, 25(3):105–111, 1995.

[LKLB16] Yang Li, Stephan Krusche, Christian Lescher, and Bernd Bruegge. Teaching global software engineering by simulating a global project in the classroom. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 187–192. ACM, 2016.

[LKvFB19] Christopher Laß, Stephan Krusche, Nadine von Frankenberg, and Bernd Bruegge. Stager: Simplifying the Manual Assessment of Programming Exercises. In *Tagungsband des 16. Workshops Software Engineering im Unterricht der Hochschulen (SEUH)*, pages 34–43. CEUR, 2019.

[LL12] Marjan Laal and Mozhgan Laal. Collaborative learning: what is it? *Procedia-Social and Behavioral Sciences*, 31:491–495, 2012.

[LRJG09] Paul Lowry, Nicholas Romano, Jeffrey Jenkins, and Randy Guthrie. The CMC interactivity model: How interactivity enhances communication quality and process satisfaction in lean-media groups. *Journal of Management Information Systems*, 26(1):155–196, 2009.

[LS03] José Paulo Leal and Fernando Silva. Mooshak: A web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, 2003.

[Luc20] Naemi Luckner. *Enabling Peer Review in Large University Courses*. PhD thesis, TU Wien, 2020.

[May05] Richard Mayer. *The Cambridge handbook of multimedia learning*. Cambridge university press, 2005.

[MB97] Ference Marton and Shirley Booth. *Learning and awareness*. Psychology, 1997.

[MBP19] Matej Madeja, Miroslav Biňas, and Lukáš Prokein. Continuous analysis of assignment evaluation results from automated testing platform in iterative-style programming courses. In *17th International Conference on Emerging eLearning Technologies and Applications*, pages 486–492. IEEE, 2019.

[Mic06] Joel Michael. Where's the evidence that active learning works? *Advances in physiology education*, 30(4):159–167, 2006.

[MPAC20] Evan Maicus, Matthew Peveler, Andrew Aikens, and Barbara Cutler. Autograding interactive computer graphics applications. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 1145–1151, 2020.

[MS08] Larry K Michaelsen and Michael Sweet. The essential elements of team-based learning. *New directions for teaching and learning*, 2008(116):7–27, 2008.

[MS13] Oliver Müller and Sven Strickroth. Gate-ein system zur verbesserung der programmierausbildung und zur unterstützung von tutoren. In *Proceedings of the 1st Workshop Automatische Bewertung von Programmieraufgaben*, 2013.

[MSS+18] Till Massing, Nils Schwinning, Michael Striewe, Christoph Hanck, and Michael Goedicke. E-assessment using variable-content exercises in mathematical statistics. *Journal of Statistics Education*, 26(3):174–189, 2018.

[Mul15] Ingrid Mulder. A pedagogical framework and a transdisciplinary design approach to innovate hci education. *Interaction Design and Architecture(s) Journal*, (27):115–128, 2015.

[MWCDF82] Larry K Michaelsen, Warren Watson, John P Cragin, and L Dee Fink. Team learning: A potential solution to the problems of large classes. *Exchange: The organizational behavior teaching journal*, 7(1):13–22, 1982.

[New15] Sam Newman. *Building microservices: designing fine-grained systems*. O'Reilly, 2015.

[OKP17] Norbert Oster, Marius Kamp, and Michael Philippsen. Audoscore: Automatic grading of java or scala homework. In *Proceedings of the 3rd Workshop Automatische Bewertung von Programmieraufgaben*, 2017.

[PL14] Peter Purgathofer and Naemi Luckner. Aurora-exploring social online learning tools through design. In *Proceedings of The Seventh International Conference on Advances in Computer-Human Interactions*, 2014.

[PLVV13] Martin Pärtel, Matti Luukkainen, Arto Vihavainen, and Thomas Vikberg. Test my code. *International Journal of Technology Enhanced Learning*, 5(3-4):271–283, 2013.

[PMP+02] Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, 8(11):1016, 2002.

[PRC+17] Jordi Petit, Salvador Roura, Josep Carmona, Jordi Cortadella, Jordi Duch, Omer Gimnez, Anaga Mani, Jan Mas, Enric Rodrguez-Carbonell, Enric Rubio, et al. Jutge.org: Characteristics and experiences. *Transactions on Learning Technologies*, 11(3):321–333, 2017.

[Pri04] Michael Prince. Does active learning work? a review of the research. *Journal of Engineering Education*, 93(4):223–231, 2004.

[PTB+17] Matthew Peveler, Jeramey Tyler, Samuel Breese, Barbara Cutler, and Ana Milanova. Submitty: An open source, highly-configurable platform for grading of programming assignments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 641–641, 2017.

[QL12] R. Queirós and J. Leal. Programming Exercises Evaluation Systems – An Interoperability Survey. In *Conference on Computer Supported Education*, pages 83–90, 2012.

[Raf88] Sheizf Rafaeli. From new media to communication. *Sage annual review of communication research: Advancing communication science*, 16:110–134, 1988.

[Ram83] Arkalgud Ramaprasad. On the definition of feedback. *Behavioral Science*, 28(1):4–13, 1983.

[RHRR12] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley Publishing, 1st edition, 2012.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language. Reference Manual*. Addison Wesley, 1999.

[RSZ15] Rohaida Romli, Shahida Sulaiman, and Kamal Zuhairi Zamli. Improving automated programming assessments: User experience evaluation using fast-generator. *Procedia Computer Science*, 72:186–193, 2015.

[SBOG17] David Schuster, Ayla Brettle, Rainer Oechsle, and Florian Grummel. Automatic evaluation of javafx applications. In *Proceedings of the 3rd Workshop Automatische Bewertung von Programmieraufgaben*, 2017.

[SC85] John Sweller and Graham Cooper. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2(1):59–89, 1985.

[Sch95] Ken Schwaber. Scrum development process. In *Proceedings of the OOPSLA Workshop on Business Object Design and Information*, 1995.

[SH14] Tina Stavredes and Tiffany Herder. *A guide to online course design: Strategies for student success*. John Wiley & Sons, 2014.

[Sha04] David Shaffer. Pedagogical praxis: The professions as models for postindustrial education. *Teachers College Record*, 106(7):1401–1421, 2004.

[SHH10] Charles Severance, Ted Hanss, and Josepth Hardin. IMS learning tools interoperability: Enabling a mash-up approach to teaching and learning tools. *Technology, Instruction, Cognition and Learning*, 7(3-4):245–262, 2010.

[Shi12] Ruey Shieh. The impact of technology-enabled active learning (teal) implementation on student learning and teachers' teaching in a high school context. *Computers & Education*, 59(2):206–214, 2012.

[SHLD16] Sumukh Sridhara, Brian Hou, Jeffrey Lu, and John DeNero. Fuzz testing projects in massive courses. In *Proceedings of the Third ACM Conference on Learning at Scale*, pages 361–367, 2016.

[SHP+06] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, . Hollingsworth, and N. Padua-Perez. Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. *SIGCSE Bulletin*, pages 13–17, 2006.

[SKGA17] Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. Gradescope: a fast, flexible, and fair system for scalable assessment of handwritten work. In *Proceedings of the fourth Conference on Learning at Scale*, pages 81–88, 2017.

[SKT+16] T. Staubitz, H. Klement, R. Teusner, J. Renz, and C. Meinel. Codeocean - a versatile platform for practical programming exercises in online environments. In *Global Engineering Education Conference*, pages 314–323, 2016.

[SL19] Arkendu Sen and Calvin K. C. Leong. *Technology-Enhanced Learning*, pages 1–8. Springer International Publishing, 2019.

[Son05] Nishikant Sonwalkar. Adaptive learning technologies: From one-size-fits-all to individualization. *EDUCAUSE Center for Applied Research, Research Bulletin*, 2005.

[SOP11] Sven Strickroth, Hannes Olivier, and Niels Pinkwart. Das gate-system: Qualitätssteigerung durch selbsttests für studenten bei der onlineabgabe von Übungsaufgaben? In *9. E-Learning Fachtagung Informatik DELFI*, 2011.

[SSLP12] Joachim Schramm, Sven Strickroth, Nguyen-Thinh Le, and Niels Pinkwart. Teaching uml skills to novice programmers using a sample solution based intelligent tutoring system. In *FLAIRS Conference*, 2012.

[Sta03] Robert Stake. *Standards-based and responsive evaluation*. Sage, 2003.

[Ste13] Sune Steffensen. Human interactivity: problem-solving, solution-probing and verbal patterns in the wild. In *Cognition beyond the brain*, pages 195–221. 2013.

[SWA03] Saul Schleimer, Daniel Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the International Conference on Management of Data*, pages 76–85, 2003.

[SYK+13] Manabu Suno, Toshiko Yoshida, Toshihiro Koyama, Yoshito Zamami, Tomoko Miyoshi, Takaaki Mizushima, and Mitsune Tanimoto. Effectiveness of team-based learning (tbl) as a new teaching approach for pharmaceutical care education. *Yakugaku zasshi: Journal of the Pharmaceutical Society of Japan*, 133(10):1127–1134, 2013.

[TGPS08] Guy Tremblay, F Guérin, A Pons, and Aziz Salah. Oto, a generic and extensible tool for marking programming assignments. *Software: Practice and Experience*, 38(3):307–333, 2008.

[Thi15] Dominique Thiébaut. Automatic evaluation of computer programs using moodle's virtual programming lab (vpl) plug-in. *Journal of Computing Sciences in Colleges*, 30(6):145–151, 2015.

[TLL13] Sho-Huan Tung, Tsung-Te Lin, and Yen-Hung Lin. An exercise management system for teaching programming. *Journal of Software*, 8(7):1718–1725, 2013.

[TR93] Gregory Trafton and Brian Reiser. Studying examples and solving problems: Contributions to skill acquisition. Technical report, Naval HCI Research Lab, 1993.

[VA02] Esa Vihtonen and Eugene Ageenko. Viope-computer supported environment for learning programming languages. In *International Symposium on Technologies of Information and Communication in Education for Engineering and Industry*, pages 371–372. Citeseer, 2002.

[Van96] Kurt VanLehn. Cognitive skill acquisition. *Annual Review of Psychology*, 47:513–539, 1996.

[Ver98] Jan Vermunt. The regulation of constructive learning processes. *British journal of educational psychology*, 68(2):149–171, 1998.

[VRV+12] Elena Verdú, Luisa M Regueras, María J Verdú, José P Leal, Juan P de Castro, and Ricardo Queirós. A distributed system for learning programming on-line. *Computers & Education*, 58(1):1–10, 2012.

[VSTK18] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Deploying microservice based applications with kubernetes: experiments and lessons learned. In *11th international conference on cloud computing*, pages 970–973. IEEE, 2018.

[VVLP13] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. Scaffolding students' learning using test my code. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 117–122, 2013.

[Wal17] Johannes Waldmann. Generating and grading exercises on algorithms and data structures automatically. In *Proceedings of the 3rd Workshop Automatische Bewertung von Programmieraufgaben*, 2017.

[WBL06] Gita Williams, Richard Bialac, and Yi Liu. Using online self-assessment in introductory programming classes. *Journal of Computing Sciences in Colleges*, 22(2):115–122, 2006.

[Wes10] Wim Westera. Technology-enhanced learning: review and prospects. *Serdica Journal of Computing*, 4(2):159–182, 2010.

[WH82] Reed Williams and Thomas Haladyna. Logical operations for generating intended questions (logiq): A typology for higher level test items. *A technology for test-item writing*, pages 161–186, 1982.

[Whi07] Jim Whitehead. Collaboration in software engineering: A roadmap. *FOSE*, 7(2007):214–225, 2007.

[Win13] Daniel Wind. *Instant Effective Caching with Ehcache*. Packt Publishing, 2013.

[WKM93] Warren Watson, Kamalesh Kumar, and Larry Michaelsen. Cultural diversity's impact on interaction process and performance: Comparing homogeneous and diverse task groups. *Academy of management journal*, 36(3):590–602, 1993.

[WSM+11] Tiantian Wang, Xiaohong Su, Peijun Ma, Yuying Wang, and Kuanquan Wang. Ability-training-oriented automated assessment in introductory programming course. *Computers & Education*, 56(1):220–226, 2011.

[Yac00] Michael Yacci. Interactivity demystified: A structural definition for distance education and intelligent CBT. *Educational Technology*, 40(4):5–16, 2000.