Teaching How to Design Large-Scale Software in a Multi-Team Project Course

Tobias Dürschmid
Carnegie Mellon University, Pittsburgh, PA, USA

Eunsuk Kang
Carnegie Mellon University, Pittsburgh, PA, USA

***Abstract*—Designing software systems is an essential technical skill for professional software engineers. However, recent graduates often lack important software design skills, such as generating, effectively communicating, and evaluating design options and collaborating across teams to build large systems.**

In this paper, we present our experience designing and delivering a new course that teaches how to design large-scale software systems via case-study-driven lectures and a project in which multiple teams collaboratively build a complex software system. We propose the GCE-paradigm (i.e., the process of iteratively *generating*, *communicating*, and *evaluating* design options) as a guiding framework to systematically teach software design.

Overall, the course has been well-received by the 17 students. They particularly valued the use of real-world case studies and in-class discussions. The multi-team project gave students insightful learning opportunities on cross-team communication that are rarely found in university education. Using interface descriptions and test double components, students could successfully integrate separately developed components. While most students' performance improved throughout the semester, some students continued to struggle with generating multiple viable alternatives and clearly communicating them via appropriate abstractions.

Based on our lessons learned, we discuss recommendations to improve the course. To allow other instructors to adopt or improve our course, we will publish all teaching materials.

***Index Terms*—Software Design, Software Engineering, Teaching, Education, Team Project, Case Studies, Constructivism**

I. Introduction

Designing software systems is an essential technical software engineering (SE) skill [3, 5, 72] that includes generation, communication, and evaluation of design options and working across teams to build complex systems [19, 42, 60, 68].

However, recent graduates often lack important software design skills, such as generating and comparing alternative designs, communicating them effectively, and collaborating across teams [10, 32, 63]. Multi-national, multi-institutional experiments have shown that the majority of graduating students in computer science lack the skills to design software systems [22, 50]. This gap between industry-needed competencies [3, 5, 72] and the design skills of recent graduates has also been confirmed by surveys of software practitioners [3].

This skill gap motivates a larger emphasis on software design education in universities [32, 33]. In many cases, software design is taught as just a small part of an overall SE course [3, 58, 70]. However, general SE courses give students little instruction and insufficient practice of software design skills in projects that are large enough to expose students to practical design challenges [10, 38, 59, 63]. In the cases in which software design is taught in a dedicated course, learning objectives focus on design patterns and architectural styles [59], which are important concepts for producing high-quality design artifacts. However, in contrast to design as an *artifact*, design as an *activity* [19] is rarely taught as a primary course objective [7, 59]. Therefore, students often lack the skills and mindset to systematically design complex software [10, 32, 63].

Teaching software design activities is challenging. Instructors have to find the right balance between teaching theoretical knowledge while also allowing students to gain enough practical experience with applying the taught design techniques to a realistic and sufficiently complex system [28, 38, 48, 59, 71]. In small software projects, students do not experience the challenges and learning opportunities that arise when no single person can fully understand the entire system [17, 18], such as compatibility of independently developed components [30], cross-team communication, component responsibility assignments, and workload distribution. Therefore, we believe software design is most effectively taught with a large-scale multi-team project that closely simulates the complexity and challenges of professional software development projects.

In this paper, we present our experience designing and delivering a new course that teaches undergraduate and graduate students how to design large-scale software systems via case-study-driven lectures and a semester-long multi-team project. We propose the "*GCE-paradigm*" (i.e., the process of iteratively ***g**enerating*, ***c**ommunicating*, and ***e**valuating* design options) as a guiding framework to systematically teach software design activities. In lectures, students learn design principles based on positive and negative real-world case studies using constructivism learning theory [6] and active learning [12]. Further, we teach multi-team software design using interface descriptions and test double components. In the course project, student teams collaboratively design, implement, test, and integrate a large-scale multi-service web application and describe important design decisions in milestone reports.

Based on our lessons learned, we discuss recommendations to improve the course design. Overall, the course was well-received by students. 17 students across 4 teams successfully designed and implemented a complex system. Most students' performance in design activities improved throughout the semester. However, some students continued to struggle with generating multiple viable alternatives and clearly communicating them via appropriate abstractions. This suggests that students need more formative assessments and more concrete guidelines for these design activities.

II. Related Work

A. Software Design Courses

Due to the importance of software design skills, courses on software design have been taught for decades [59, 67].

Lecture-focused Courses: Many software design courses in the literature focus on lecture-based learning without a major project component [59]. Some courses focus on teaching software design based on design patterns and remain closer to a source code [41, 75]. Other courses focus on high-level component interactions, architectural styles, and quality attributes [31, 52]. While these courses teach important skills that are relevant to producing good design artifacts, to the best of our knowledge, only one course at UC Irvine [7] teaches software design primarily as a systematic activity [19].

Team-Project-based Courses: Some software design courses include a major team-project component [59, 67]. For example, in a course taught at Murdoch University, students practice modular decomposition and learn to specify component interfaces in teams of six [4, 40]. UC Irvine includes two team projects in their software course during which students design and implement a system in teams of 14 students [7]. Courses taught at the University of Queensland [14] and Beihang University [79] provide students with open-source systems that students should read, model, and extend. A common domain for team projects in software design courses is game projects [76]. In existing software design courses student teams generally work individually, rather than collaboratively developing a system across teams. In contrast, our course allows students to experience cross-team communication challenges and a more realistic development context in which students have to integrate components built by other teams.

B. Multi-Team Courses

The teaching concept of using multiple interacting teams in SE education has been proposed and implemented in courses not focused on software design before.

Agile Processes: A course on scaling Scrum, which has been taught for multiple years at Hasso Plattner Institute, lets students build a web application with multiple interacting teams [53, 54]. The course teaches the Scrum process and modern SE practices (e.g., test-driven development, behavior-driven development, continuous integration, and version control) in a realistic environment with self-organizing teams in a semester-long project [54]. Students receive the role of either Scrum Master, Product Owner, or developer while customers are simulated by the teaching team [53]. Students learn by making decisions about their development process autonomously and reflecting on their decisions after each sprint [54]. The multi-team project of our course has been partially inspired by this course. Similar courses are taught at the College of William and Mary [17, 18], the University of Helsinki [51], and the University of Victoria [47]. However, in contrast to multi-team courses on Agile processes, the learning objectives of our course focus on software design activities. This creates additional challenges, as the time available for teaching development processes and interactions is more limited in a software design course.

Global Software Development: Some courses teach even harder-to-practice skills of developing a product via collaborating, globally distributed teams [13, 16, 20, 37]. However, similar to the courses on Agile processes, they do not specifically focus their learning objectives on software design.

III. Course Design Overview

The course presented in this paper is a full-semester elective aimed at graduate and undergraduate students in computer science and majors related to computer science (e.g., information systems). Prerequisite knowledge of the course included intermediate programming skills and experience with developing and testing medium-sized programs. The course builds on the programming skills that students have obtained through previously taken courses, internships, or other industry experience and teaches them the highly demanded skills of designing large-scale software systems by making trade-offs between different quality attributes, considering different design alternatives, and communicating design using appropriate models. The course consists of three major instructional methods:

1) Active-learning-style **lectures** using real-world case studies to teach design principles based on constructivism learning theory [6] (Section IV).
2) A semester-long **multi-team project** in which all teams collectively design, implement, and integrate a system composed of different services and describe their design decisions in five milestone reports (Section V).
3) Three **individual homework** assignments during which students practice skills taught in the lectures (Section VI).

A. Learning Objectives (LOs)

As few existing courses teach software design primarily as an activity, deciding what to teach in this course is one of the contributions of this paper. We decided that the following learning objectives are most important to teach an engineering mindset [19] of software design.

Requirements analysis and specification are important skills for all software engineers [3, 5, 39, 63, 66], as prioritized requirements are the main drivers of software design [38, 59]. Therefore, a software design course should teach students how to elicit, specify, and prioritize requirements.

LO R (Requirements)	Bloom's Level [1]: **Analyzing**
Students should learn to: **Identify, describe, and prioritize relevant requirements for a given design problem.**	

Starting from requirements, design space exploration via constructive thinking and creative problem solving is the next required software design skill [19, 55]. Since considering multiple design alternatives is likely to lead to a better design [72], a software design course should teach students how to generate multiple viable solutions.

LO G (Generate)	Bloom's Level [1]: **Creating**
Students should learn to: **Generate multiple viable design solutions that appropriately satisfy the trade-offs between given requirements.**	

Modeling is a central aspect of design [19, 24, 46, 60] and essential for collaborative design [42, 68]. Hence, we should teach students to effectively communicate design ideas.

LO C (Communicate)	Bloom's Level [1]: **Creating**
Students should learn to: **Communicate the essential aspects of design solutions by choosing and visualizing appropriate abstractions and models.**	

Judging the quality of design options is essential to improve designs and assess requirements satisfaction [45]. Therefore, a software design course should teach design evaluation.

LO E (Evaluate)	Bloom's Level [1]: **Evaluating**
Students should learn to: **Evaluate design solutions based on their satisfaction of common design principles and trade-offs between different quality attributes.**	

Design decisions have a long-lasting impact on quality attributes, such as changeability, interoperability, reusability, robustness, scalability, and testability [48, 69, 77, 80]. To build on existing knowledge and experiences, teaching design principles can guide students to generate and evaluate design options for various quality attributes [46, 61].

LO DP (Design Principles)	Bloom's Level [1]: **Applying**
Students should learn to: **Describe, recognize, and apply principles for: Design for reuse, design with reuse, design for change, design for robustness, design for testability, design for interoperability, and design for scalability.**	

The software design process should be adjusted depending on the context, the overall amount of risk, and the types of risks in the domain [23]. Therefore, a software design course should teach students how to adjust the design process to fit into Agile, plan-driven, and risk-driven development processes across different domains.

LO P (Process)	Bloom's Level [1]: **Applying**
Students should learn to: **Determine and explain how to adapt a software design process to fit different development contexts and domains.**	

Finally, to build complex, large-scale software systems, skills of cross-team design and development are essential, as most modern software is built by more than one team [9, 10, 63, 68]. Thus, it is critical for a software design course to teach students how to collaborate across teams.

LO MT (Multi-Team)	Bloom's Level [1]: **Creating**
Students should learn to: **Collaborate with other teams to design, develop, and integrate individually developed components into a complex system.**	

Date	Topic	LOs
L 1	Introduction and Motivation	
L 2	Problem vs. Solution Space	LO R, LO C
L 3	Design Abstractions	LO C
L 4	Quality Attributes and Trade-offs	LO R, LO E, LO C
L 5	Design Space Exploration	LO G
L 6	Generating Design Alternatives	LO G
L 7	Design for Change	LO DP, LO E, LO G
L 8	Design for Change	LO DP, LO E, LO G
L 9	Design for Interoperability	LO DP, LO C, LO E
L 10	Design for Testability	LO DP, LO E, LO G
L 11	Design with Reuse	LO DP, LO G, LO E
L 12	Reviewing Designs	LO E, LO C
	Midterm	
L 13	Cross-team Interface Design	LO MT
L 14	Design for Reuse	LO DP, LO E, LO G
L 15	Design for Scalability	LO DP, LO E, LO G
L 16	Design for Scalability	LO DP, LO E, LO G
L 17	Design for Robustness	LO DP, LO E, LO G
L 18	Design for Robustness	LO DP, LO E, LO G
L 19	Design Processes	LO P
L 20	Design for Security	LO DP, LO E, LO G
L 21	Design for Usability	LO DP, LO E, LO G
L 22	Ethical and Responsible Design	LO DP, LO E
L 23	Designing AI-based Systems	LO DP, LO E
L 24	Course Review	
	Project Presentations	LO C
	Final Exam	

TABLE I: Lecture topics and addressed learning objectives.

IV. Lecture Design

This section describes how the lectures in this course teach design primarily as an *activity* based on real-world case studies and constructivism learning theory. The list of lectures and learning objectives that they address is shown in Table I.

A. *Teaching Design as an Activity via the* GCE-Paradigm

We propose the "*GCE-paradigm*" as a guiding framework for systematically teaching software design activities. The GCE-paradigm describes software design as the process of iteratively ***g****enerating*, ***c****ommunicating*, and ***e****valuating* design options based on requirements. We introduce the GCE-paradigm via lectures and in-class activities on the individual design activities. Then, we teach how to combine these activities in an iterative design process while providing specific instruction on designing for individual quality attributes in "design for X" lectures. To help students connect new knowledge to the respective design activity, each slide highlights the associated activity in the cycle of the GCE-paradigm.

Requirements Analysis: To understand the problem and context of design tasks, we teach students to identify important requirements and domain assumptions (LO R). In Lecture 2, we illustrate the importance of domain assumptions based on the case study of the Lufthansa 2904 runway crash (caused by the assumption that the plane is on the ground if and only if the wheels are spinning, which was violated by a wet runway). We then ask students to identify important requirements and assumptions across different domains.

Communicating Designs via Abstractions: To support design collaboration and evaluation, we teach how to communicate designs using appropriate abstractions (LO C). Interleaved [25] throughout Lectures 2, 3, 4, and 9, we introduce context diagrams, component diagrams, sequence diagrams, data models, interface descriptions, and Class-Responsibility-Collaboration (CRC) cards. As a use of spaced repetition [44], we use these abstractions in following the lectures, recitations, homeworks, and project milestones.

Generating Design Alternatives: In Lecture 6, we survey techniques that help generate design options (LO G). First, we motivate the importance of thinking of different design alternatives, as this is likely to result in a better design [72]. Then, we teach brainstorming techniques (e.g., writing ideas on post-its, clustering, combining ideas, avoiding anchoring), which students practice during an in-class exercise. Based on the thereby introduced pattern of model-view-controller, we teach that design generation often starts with building on existing designs described in patterns.

Evaluating Design via Quality Attribute Trade-offs: As design often has to compromise between multiple conflicting objectives, we teach students how to identify and evaluate important quality attribute dimensions (LO E). In Lecture 4, we introduce quality attributes based on the connectors, publish-subscribe and call return, which can be used to implement the same functionality with different quality attributes. Thereby, we illustrate that design decisions can impact extensibility, robustness, and understandability. We then teach how to specify quality attribute requirements via measurable scenarios and show examples of trade-offs and synergies between quality attributes. In Lecture 12, we teach how to review designs via adversarial thinking and how to argue for design options. Via spaced repetition [44], we ask students throughout many lectures to identify important quality attribute dimensions, specify measurable scenarios, and evaluate design options.

Design Process: To convey the principle that the amount of design effort should depend on the criticality of the system being developed (LO P), we teach a risk-driven design approach [23] and show how this approach fits into Agile as well as more waterfall-like software development processes. Then, we conduct in-class activities to identify relevant risks for different domains (e.g., online shops, games, medical software, spacecraft systems, startups, and social media systems). Further, we teach the human aspects of software design [68, 73] by contrasting intuitive decision-making with rational decision-making [62], discussing bounded rationality [43], and emphasizing that design is a collaborative hands-on activity [73].

Experience: At the end of the semester, we conducted an anonymous survey to request feedback on the course, including the lectures. 13 out of 17 students responded.

The students responded positively to the lectures. To the question "Which topics/lectures were valuable and should be kept for future versions of the course?" four students responded with "*all*" and two students responded with all "design for X" lectures. Lectures that students enjoyed in particular were the lectures on scalability (five students), reuse (three students), interoperability (two students), testability (two students), and changeability (two students). One student wrote: "*I think all the theoretical portion of the lectures were very well structured and should be all kept. Like this course is one of the best logically flowing courses I have taken at CMU.*"

No majority opinion emerged on which topics should be covered more/less. In response to the question "*To improve the course, which topics should we cover additionally, cover more, or cover less?*" students asked for more real-world examples in lectures (two students); more content on scalability (two students); and more content on testability, security, robustness, and quality attributes broadly (one student each).

Lesson Learned 1 (Design as an Activity) — Lectures

Lectures on how to design large-scale software systems via the GCE-paradigm were well-received.

- Include a mix of lectures on individual design activities (requirements specification, design generation, design communication via abstractions, design evaluation, and design process adjustment) and on "design for X"
- To provide students with multiple practice opportunities, apply spaced repetition [44] by including the major activities in each "design for X" lecture while explicitly marking the corresponding slides with the activity name.

B. Real-World Case Studies

Case studies have been shown to be an effective teaching method in general SE education [29, 65, 74, 78] and have also been proposed for software design education in particular [15]. To convey the need for the design principles taught in the lectures (LO DP), we instructed them based on the following real-world case studies of well-known software failures and success stories, some of which we assigned as required readings before the corresponding lecture.

Global Distribution System (GDS) In the lecture on *design for interoperability*, we used GDS[1] (the interface standard that is used by airlines and booking systems to transfer data between independently developed systems) as a case study for a multi-decade success of hundreds of interoperating systems (but with limited changeability).

Mars Climate Orbiter After discussing techniques to achieve syntactic interoperability, we used the Mars Climate Orbiter [11] case study to illustrate the importance of semantic interoperability (a mix of imperial units and metric units caused the system to crash for a multi-million dollar loss).

Netflix's Simian Army: In the *design for testability* lecture, we used the Simian Army by Netflix as a positive example for quality attribute testing of large-scale systems [8].

Ariane 5 Rocket Launch Failure: In the *design with reuse* lecture, the well-known Ariane 5 failure (caused by an invalid assumption about the maximum velocity in the inertial reference system that was ported from Ariane 4)

[1] https://www.youtube.com/watch?v=1-m_Jjse-cs

is used to illustrate the importance of identifying and checking assumptions made by reused components [49].

npm `left-pad`: In the *design with reuse* lecture, the suddenly unavailable, but widely reused npm package `left-pad`[2] with trivial implementation was used to motivate the design principle to strive for few dependencies.

Heartbleed: In the *design with reuse* lecture, the Heartbleed bug[3] (a security vulnerability in OpenSSL) motivated the importance of updating critical dependencies.

Twitter: In the *design for scalability* lecture, Twitter[4] (now X) was used as a case study to teach approaches for scaling a system based on estimated demand.

Experience: Overall, we believe the case studies were valuable for conveying the key course concepts and maintaining student engagement. We collected student feedback on the course in a mid-semester course feedback focus group session. To ensure students can speak freely and to anonymize all responses, the feedback was collected by an outside consultant who was not part of the course teaching team. In that session, all students unanimously agreed that the real-world case studies helped them learn, because "*examples of design scenarios and code snippets make core ideas more concrete and easier to understand*" and "*use of real-world examples in lecture[s] ties concepts to reality, helps retain info (e.g. the npm library)*". As instructors, we also noticed an increased level of student attention and participation specifically when discussing the case studies during lectures.

Lesson Learned 2 (Real-World Case Studies) Lectures

The use of real-world case studies of positive and negative examples for design principles has been well-received for teaching design principles (LO DP) and the software design process (LO P) in this course.

- For complex case studies, such as GDS and Netflix's Simian Army, assign required reading with a reading quiz before the lecture, so that all students are familiar with the important details of the case study.

C. Teaching Software Design Principles using Constructivism

In contrast to directly presenting design principles to students up-front, in this course, we let students themselves actively construct design principles by generalizing from real-world case studies of positive and negative examples (LO DP). Delivering lectures centered around student participation uses *active learning* [12], which has been shown to significantly improve learning outcomes in computer science and other fields [27, 35, 36]. Letting students construct design principles from examples is rooted in *constructivism learning theory*, which posits that teachers cannot simply transmit knowledge to students, but students need to actively construct knowledge in their own minds [6]. According to constructivism learning theory, students learn best by discovering information, checking new information against old information, and revising rules when they do not longer apply [6]. Based on the best available evidence in educational literature, constructivism improves retention [64], students' academic success [64], and metacognitive skills [57].

As software design principles are abstract concepts for which it is important to internalize why they exist and what their limitations are, we believe a constructivist teaching approach is most effective. By letting students follow the step-by-step process of formulating design principles based on positive and negative examples, we believe students gain a deeper understanding of how the design principles impact system design, why they often improve design, and in which cases they would not improve design.

For example, in the *design for interoperability* lecture, we use a case study based on GDS, a system that is used by nearly all airlines and booking systems to exchange data. First, we ask the students to discuss in small groups what specifically makes this example so successful and share their thoughts in the class. Second, we ask them to generalize their insights toward design principles that apply to future projects, which they described as creating a shared data format or an interface between systems. This is a part of the final design principle, but still missing an important element. Hence, we show the students the example of the Mars Climate Orbiter failure [11] (which resulted from the inconsistent use of metric and imperial units) to demonstrate that just having syntactic interoperability alone is not sufficient, but that semantics have to be defined precisely as well. Students appropriately inferred the design principle of documenting the meaning and units of interfaces. Finally, we let students describe the shortcomings of GDS. They correctly identified limited changeability of the interface that is implemented across hundreds of systems. In doing so, students identified and addressed the concrete challenges, generalized them, and constructed the design principles that the lecture was intended to teach. We follow the same approach to teaching design principles throughout the course.

To identify whether non-participating students also understood the design principles, we end each lecture with an *exit ticket* [26] (a digital assignment in which students are asked to summarize the lecture's main message in their own words and apply it to a small, different example).

Experience: In the mid-semester focus group, 46 % of students agreed that in-class discussions helped them learn, since "*in class discussions help us think and reason over content*" and facilitate "*reiteration of ideas; students have different perspectives*". Considering that students often subjectively under-value the objective effectiveness of active learning techniques [21], these results suggest that constructivism likely supported the students' learning of design principles. Based on the quote "*[we] don't know what they expect as answers when they put us into discussion groups*", we identify the clarity of questions as a potential challenge of the technique, as students might not have always known what type of answer was expected of them. Finally, all students agreed that exit tickets

[2] https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program/

[3] https://heartbleed.com/

[4] https://blog.x.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how

helped them learn, because "*exit tickets help us reconsider what we learned in the class right after class*".

Lesson Learned 3 (Constructivism) — Lectures

The use of constructivism for teaching design principles (LO DP) was overall well-received in this course.

- Give students 2 – 5 min of silent thinking and small-group discussions before discussing with the whole class.
- Soon after describing design principles, give students another problem to practice applying the principles in recitations or homework.
- To give students an idea of what type of answer is expected, give them examples of answers to a similar question that they are already familiar with.
- At the end of each lecture, include an exit ticket with one summary task and one small task for applying the learned techniques to a different example.

V. Multi-Team Project

While teamwork is one of the most important soft skills in professional software development [3], graduates in computer science often lack the skill to collaborate across teams [10, 63] or work on large projects [63]. To let students practice collaborative software design in a realistic context, in which no single developer fully understands all components, we decided to include a large-scale multi-team project in this course (LO MT). In the project, each team developed its own medical appointment scheduling app and one of four collaborating services. The *medical scheduling app* should allow users to book appointment slots, see their results, and receive quarantine requests. The *healthcare administrator service* should let healthcare professionals enter patients' test results and other medical data. The *policymaker service* should allow government officials to modify the policy that determines whether and for how long a patient should undergo quarantine. The *central database service* provides storage and retrieval of patient information across multiple scheduling apps. The *public information service* should allow users to view aggregated statistics). The teams were eventually asked to integrate their scheduling app and service with other teams' services.

The decision to let students collaboratively design and develop a large-scale system comes with unique challenges that should be addressed by course design to ensure students focus their time and effort on the main learning objectives and can gain a mostly positive experience with the design techniques. These major challenges include:

- Challenges of cross-team communication [47], which we address by letting teams pick a dedicated member to be responsible for cross-team communication (Section V-A)
- Potentially incompatible interfaces of individually developed services, which we address using interface descriptions (Section V-B)
- Challenges of testing services while dependent-on services have not been implemented, which we address using test double components (Section V-C)

To better support students with the project, we offered weekly project office hours (15 min slots per team) during which students could present their progress, ask clarification questions, and receive targeted feedback from instructors.

Experience: Students particularly valued the weekly project office hours, with quotes such as "*I really gained a lot from your feedback and discussion with you during the office hours. It enhanced my learning and thinking about previous or undergoing milestones.*". The four teams built a system with a total size of 19.5 KLOC. This amounts to 1.15 KLOC per student on average. Overall, the developed system was functionally correct, and services integrated well with each other. The course project provided many insightful learning opportunities, which are discussed in the following sections.

A. Cross-Team Communicator

As identified in previous work on multi-project SE courses [17, 18, 47], communication between teams is a major challenge. To reduce communication overhead between teams (LO MT) we decided to use class time for cross-team communication, provided a shared Slack channel for cross-team communication, and dedicated a *cross-team communicator* role for each team. Cross-team communicators should serve as interfaces of the team and represent the wishes and needs of their team. When multiple teams need to make decisions together, instead of all students meeting, discussions can be limited to only cross-team communicators.

Experience: We believe some teams did not pick the ideal person to serve as the cross-team communicator. During the initial design of the high-level architecture, cross-team communicators met to assign component responsibilities. As some teams picked students who were less involved in the team's technical design discussions as cross-team communicators, they did not fully understand the technical implications of these decisions on the team's workload and required technical expertise. This led to unpleasant surprises when the students learned that their cross-team communicator agreed to them working on tasks that they did not feel equipped to work on in the given time frame, requiring a new meeting to redesign the system's overall architecture.

Lesson Learned 4 (Cross-Team Communicator) Project

The effectiveness of cross-team communicators depends on how well they can evaluate design trade-offs and how well they know the skill set of their team.

- To reduce the risks of multi-team challenges (LO MT), let teams pick a cross-team communication that will serve as a facade of the team and interface with other teams.
- Clearly describe the responsibilities and desired traits of a cross-team communicator.
- Ensure that cross-team communicator is not a role that teams assign to the member who has not contributed enough yet, but a role that should be given to a student who is prepared to represent the team's needs and wishes in important technical design decisions.

B. Service Interface Description

To give students the experience of building a component that is used by other teams and using components developed by other teams (LO MT), we let teams describe OpenAPI specifications describing syntax and semantics of their interfaces (LO C) and review each others' interfaces (LO E).

Experience: Students had only a few integration issues. Considering that each service was developed individually and most students experienced a large-scale development project with multiple teams for the first time, we were surprised by the high interface compatibility between the services. We believe interface descriptions contributed to this success.

Lesson Learned 5 (Interface Descriptions) — Project

Interface descriptions likely helped students independently develop compatible services (LO MT).

- As part of the project milestone in which teams design their individual services (Milestone 3), include a task for students to precisely specify interface descriptions.
- To increase the probability of major compatibility issues being caught before implementation, ask student teams to give each other feedback on their interface descriptions.

C. Test Double Components

While all teams develop their own services, dependent-on-services are not immediately available for testing. To address this challenge and to allow students to simulate data sent from other components (LO MT), we taught students to implement *test double components* (components that mimic the interface of a required service to control indirect inputs or verify indirect outputs [56]) based on interface specifications in the *design for testability* lecture. During the project, we asked students to implement test doubles for dependent-on components.

Experience: Test doubles helped students find some, but not all, bugs before integration. Students also mentioned that in the project, test double components helped "*isolating the influence of external components*". Many teams implemented test doubles via conditional logic within their components, rather than as a separate HTTP-communicating component, which impeded replacing them with real components later.

Lesson Learned 6 (Test Double Components) — Project

Test double components helped students independently develop and integrate services (LO MT).

- To ease replacing test double components with the real components, recommend students to implement test double components by mocking HTTP messages rather than simply mocking functions inside their own component.
- To simplify implementation tasks, point students to libraries and frameworks that inject HTTP messages.

D. Milestone Reports

Many companies, such as Google, use Design Docs or other architecture decision records [2] to describe their important design decisions [81]. Students practiced writing similar documents in milestone reports for which we asked them to generate (LO G), communicate (LO C), and evaluate (LO E) multiple design options for project tasks. The following sections describe each milestone and our experience.

Milestone 1 (Domain Modeling & Initial System Design)

In the first milestone, students were given the description of a small design problem (designing a medical appointment scheduling app). Based on the given requirements and context, students were asked to model a problem domain (LO C), identify important quality attribute requirements (LO R), and describe a first high-level design solution (LO G and LO C).

Experience: In an end-of-semester survey asking for feedback on every milestone, virtually all students said this milestone was "*Good*" or "*Great*" and spent less time on the milestone than we anticipated. Based on the submitted reports, students made fewer design decisions (especially on the choice of technologies and web frameworks) than we anticipated. Therefore, we recommend including more mandatory questions on particularly important decisions so that more design decisions are made in this milestone.

Milestone 2 (First Prototype Development)

In the second milestone, students should refine (LO G), model (LO C), and implement the design they described in Milestone 1, implement tests to evaluate the end-to-end functionality (LO E), and reflect on how the design changed and which other alternatives options they considered (LO G).

Experience: Students took more time for this milestone than we anticipated, requiring us to extend the milestone by one week. In the end-of-semester survey, many students said "*More time should be given to this milestone because ... some of the members in the group are still in the learning stage of some frontend/backend framework.*". Furthermore, due to the higher workload of picking and learning a framework, students' time efforts shifted more towards implementation than design, leaving less time to consider alternatives and evaluate the impact of implementation decisions on the system design [48]. Providing more implementation support, specifically on frameworks that might be useful for the project, might help address this issue.

Lesson Learned 7 (Implementation Support) — Project

The relative portion of project time spent on coding rather than design was higher than desired, resulting in students investing less time into the main LOs.

- To reduce the time students spend on coding and allow them to focus more on design activities, include coding templates that help students implement their systems more efficiently.
- Link tutorials to common frameworks and libraries.
- Include a recitation at the beginning of the course that introduces commonly used code generation techniques.

Milestone 3 (Design for Changeability & Interoperability)

In the third milestone, students were first introduced to the four services that they were going to design and implement to interoperate with each other. The milestone provides a description of the functionality of each service as well as tips for cross-team collaboration via cross-team channels and a dedicated cross-team communicator. Based on this description and service assignment per team, students are asked to design their service (LO G), model it using interface descriptions (LO C), and collaborate with other teams to ensure compatibility (LO MT). To further support service compatibility, students are asked to design test doubles for two of the most central services. Students are also asked to re-design their appointment scheduling app to support certain future changes (LO G) and add tests to evaluate the functionality (LO E). In a design reflection students should report on design decisions they made during interface design, the changes they made and describe a change impact analysis of two potential changes.

Experience: Students had major discussions and disagreements, which increased the workload of the milestone while providing insightful learning opportunities. We recommend providing multiple opportunities for students to have cross-team discussions in recitations or setting some lecture time aside for this, as some students mentioned they had "*not enough time to discuss design decisions with other students*".

Milestone 4 (Service Development & Integration)

In the first part of the fourth milestone, we asked teams to implement their services, while collaborating with other teams to ensure compatibility (LO MT), and implement test doubles for adjacent services. Then they should deploy their services and provide other teams with the URL and port of their service instance. In the second part, students should integrate their services by replacing the test double components with the real deployed services of other teams. Then they should perform rigorous integration testing to evaluate the functionality of the overall system (LO E). In a design reflection students should report on the design principles they used (LO DP), how they reused existing libraries, how cross-team collaboration affected their design decisions, and how starting from a fixed interface impacted their implementation.

Experience: The integration of services went largely smoothly. The most common integration issues were related to different capitalization and the use of dashes in data formats that resulted from interface changes that were not explicitly communicated but were easy to fix. In the end-of-semester survey, students mentioned this milestone "*helped understand teamwork and how to collaboratively work with others*".

Milestone 5 (Robustness Testing)

In the last milestone each team is assigned the service of another team for which they should conduct intense robustness testing by trying to break the service (LO E). They should report their findings to the team that developed the service. In an optional task, students were asked to describe at least two design options for at least two of the issues found by other teams and describe the improved designs (LO G and LO C). Due to time limitations and due to this task strongly relying on the findings of other teams this task only gave bonus points. However, all teams completed this optional task.

Experience: Students thoroughly enjoyed breaking the services of other teams and said it was "*useful to understand what issues a system can potentially face and what could be potential loopholes*". As students spend less time on this than we expected, expanding the milestone by asking the students to identify a large variety of issues (e.g., performance, correctness, availability, security) is one potential improvement.

E. Assessment of Milestone Report Submissions

Asking students to submit multiple written reports on the progress of their project lets students receive constructive feedback and observe their own growth [34]. The main shortcomings of submissions were related to LO G and LO C.

The discussion of design alternatives was often quite superficial. In some cases, students just described their final design without discussing potential alternatives. In other cases, students described alternative designs that clearly would not satisfy the requirements and thereby missed the opportunities to meaningfully discuss design trade-offs.

The models of design solutions often did not communicate the essential aspects of the corresponding design. In many cases, the textual arguments of students were largely disconnected from the presented diagrams, suggesting that students did not sufficiently consider what aspects of their design should be communicated at which level of detail. In other cases, models were too ambiguous or unclear.

We allowed students to redo some milestones to improve their design discussions. We saw significant growth in redone milestones, later milestones, and during final presentations, suggesting that feedback helped students improve.

Lesson Learned 8 (Milestone Reports) — Project

Milestone reports have helped assess students' progress and their satisfaction of learning objectives and have been great opportunities to provide targeted feedback to teams in this course.

- To allow students to apply feedback in the next milestone, try to grade submissions quickly.
- Allow students to redo some milestone reports for an improved grade to incentivize students to take provided feedback seriously.

VI. Homework Assignments

This section describes our design and experience of complementing the project with individual homework assignments.

A. HW1 - Domain and Design Modeling

The first homework is designed to let students practice domain analysis (LO R) and modeling (LO C). The homework is scheduled so that students receive feedback on this homework before working on the first project milestone.

In the homework, students were presented with a case study of a home security system and asked to model the system using a context model, component diagram, data model, and sequence diagram. Students should also describe assumptions made about the domain and design decisions they made.

Experience: In an end-of-semester survey students overall liked the homework while mentioning a higher-than-expected workload (e.g., "*This was useful and a must learn skill for design documentation. Although it took me around 6 – 7 hours as opposed to 2 – 3 hours.*"). Most submissions demonstrated accomplishment of the learning objectives. The most common mistake was that 18 % of submissions included domain entities in component diagrams rather than context diagrams.

B. HW2 - Design for Reuse

The second homework practiced generating multiple design alternatives (LO G), communicating them using interface descriptions (LO C), evaluating them for reusability (LO E), and describing the design principles they support (LO DP).

Students were tasked to evaluate an open-source package for reusability by identifying its assumptions and reuse context, describing design principles that contribute to its reusability, and describing reuse scenarios in which reusing it would be appropriate and inappropriate. Then, students were asked to improve the package design for an unsatisfied reuse scenario and communicate the new design with interface descriptions and a description of required implementation changes. Finally, students should describe how the redesign improves the reusability based on applied design principles or other arguments. The homework was designed to be open-ended to allow students to freely explore the reusability of the given module based on their interests and domain expertise.

Experience: In the end-of-semester survey, students overall liked the homework (e.g., "*Very good. Required much more thought about the reuse and how it works in practice.*"). Three students mentioned that "*the instruction was very open-ended*", suggesting that some students prefer more concrete instructions rather than an open-ended format.

In the graded submissions, most students demonstrated sufficient accomplishment of the learning objectives. The most common mistakes were related to the precise description of reuse scenarios (35 % of submissions), and partially lacking description of semantics in the interfaces (6 % of submissions).

C. HW3 - Design for Scalability

The third homework was designed to provide students with design generation (LO G), communication (LO C), and evaluation (LO E) skills related to scalability. Based on the case study of the project, students should specify scalability requirements, make design decisions (e.g, what data to store, what storage model to use, what type of scaling to use, how to distribute the data, which data to cache), model them using component diagrams, and evaluate the designs.

Experience: In the end-of-semester survey all students liked the homework (e.g., "*It was a good balance between the time spend and learning outcome*").

In the graded submissions, almost all students demonstrated sufficient accomplishment of the learning objectives. Common mistakes were mostly minor, such as the use of generic rather than domain-specific component names, insufficient justifications of design decisions, and unrealistic demand estimations.

VII. Open Challenges of Teaching Design

The main goal of this course was to teach students how to design large-scale software systems by fostering an engineering mindset and teaching design as an activity. Overall, students struggled most with learning objectives LO G, LO C, and LO MT. As all three LOs are at the highest cognitive level of Bloom's revised taxonomy [1] (Creating), they are particularly challenging to teach effectively. In this section, we discuss the concrete challenges we observed and suggest ideas to overcome them in future courses.

A. Generating Multiple Viable Alternatives

As mentioned in Section V-E, in milestone reports, students struggled with generating multiple viable alternative design options (LO G). We observed similar trends in both exams (mid-term and final exam), in which we asked students to describe at least two viable design options for a design problem, evaluate them, and discuss trade-offs between the two options. In both exams, especially in the mid-term, many students presented one viable option and one straw-man option that was a deliberate degradation of their other option.

As generating multiple viable design options is an important software design skill [72], we see this as an important challenge when teaching design. While students' ability to discuss alternatives noticeably improved throughout the course, we believe providing more dedicated instruction on design generation is still an open challenge. Potential improvements could teach more design generation and brainstorming techniques throughout the course paired with exercises of generating as many viable ideas as possible to give students more practice and spaced repetition. Furthermore, as students asked for "*more concrete tactics*" to design systems, a curated list of more specific design recipes, cautiously annotated with limitations of their applicability, could help students learn the generation of more design options.

Lesson Learned 9 (Multiple Viable Alternatives) LO G

Many students in this course struggled with describing multiple, viable design alternatives.

- Include multiple individual homeworks, recitations, and in-class exercises for students to practice generating multiple design alternatives.
- Teach more concrete guidelines on how to generate multiple viable design alternatives.

B. Design Communication via Appropriate Abstractions

As mentioned in Section V-E, in milestone reports, students struggled with identifying appropriate abstractions to communicate the essential aspects of their design (LO C). We observed similar trends in both exams, in which we asked students to communicate designs using component diagrams, interface diagrams, and sequence diagrams.

In the mid-term exam, students struggled most severely with interface descriptions and component diagrams. Only $58\,\%$ of submissions demonstrated sufficient accomplishment of the learning objective ($6\,\%$ did not include an answer to the question, $12\,\%$ did not describe interfaces using an appropriate format, and $24\,\%$ lacked descriptions of semantics). Interface descriptions improved in the final exam with $82\,\%$ of submissions demonstrating sufficient accomplishment of the learning objective. The improvement is most likely due to students having had more practice with interface descriptions in the project and Homework 3. Therefore, we believe adding additional homework to practice interface descriptions in the first half of the semester would help students. The additional homework workload might be offset via Lesson Learned 7. Common mistakes for component diagrams included unclear responsibility assignments, missing arrows, and missing connection labels. Mid-term submissions included more severe cases of diagrams being too ambiguous to appropriately convey design choices, suggesting some growth. Furthermore, in both exams, some diagrams were inconsistent (i.e., design choices communicated in different models contradicted each other).

Based on these observations, we identified that teaching the identification of appropriate abstraction to model the most essential aspects of design is still an open challenge. Potential improvements could use interleaving [25] of different model types to train students to identify which aspects of a design are best represented using which type of model. Many exercises in modeling different design aspects throughout the course could give students more practice and spaced repetition.

Lesson Learned 10 (Communicate Abstractions) LO C

Many students in this course struggled with communicating design options via appropriate abstractions.

- Include multiple opportunities for students to practice interface descriptions and component diagrams in individual homeworks, recitations, and in-class exercises.
- Include guidelines and exercises on selecting abstractions that communicate the essential aspects of a given design.

C. Cross-Team Design Debate

One major challenge during the multi-team project was how to design the system in a way that the implementation effort of each service is roughly equal (LO MT). Three teams devised a design that would assign major responsibilities to the central database, whose team was largely absent during these discussions. Understandably, the database team was opposed to taking on a higher workload. Faced with this conflict in a situation in which the three other teams invested considerable effort into a design that was not going to get approved by the other team, a heated discussion took place on Slack. To lead students toward a more constructive resolution, we recommended an in-person meeting. With instructors only passively observing, the teams self-organized a collaborative discussion of potential design options and evaluated them across self-identified dimensions (code modifications needed, interface complexity, extensibility, and workload balance). Based on their evaluations, teams then voted for their preferred option and democratically reached a reasonable consensus.

While this discussion initially resulted from frustrations and disagreements between teams, it provided one of the best learning opportunities to experience the complexity of real-world design considerations [68, 73]. During this meeting, students demonstrated excellent application of advanced software design skills, such as trade-off evaluation, design communication, iterative refinement, and a deep understanding of the non-technical implications of their decisions, skills that we did not observe in the students before. We believe this discussion particularly helped students grow and integrate all major design skills more than they would have otherwise.

Therefore, we recommend explicitly integrating more opportunities for student teams to collectively debate cross-team decisions. While we allocated one lecture at the beginning of Milestone 3 for this activity, due to most students of the database team not attending, and students having had little time to generate design alternatives before this discussion, it was less productive than the debate following the heated Slack discussion. A challenge in integrating cross-team debates is to identify the right balance between leaving enough opportunities for constructive disagreements between teams to encourage debates while moderating the discussions enough to ensure that students still have a positive experience.

Lesson Learned 11 (Design Debates) LO MT

Students gained the most substantial practice with multi-team software design activities during an unplanned cross-team design debate.

- Include multiple opportunities for teams to debate cross-team design decisions during recitations or lectures.
- Embrace (constructive) disagreements between teams as an opportunity to practice group decision-making.
- While avoiding too much interference with student autonomy, ensure that disagreements are resolved peacefully.

VIII. Conclusions & Future Work

In this paper, we presented the design of a novel course on designing large-scale software systems via the GCE-paradigm using real-world case studies, constructivism, and a multi-team project. Our experience motivates future work that empirically measures the effectiveness of these approaches on software design education, scales the course to a larger number of students, and replicates this experience at other universities.

IX. Data Availability

To allow other instructors to adopt or improve our course design, we have made all teaching materials publicly available here: https://cmu-swdesign.github.io/. Non-aggregate data on student submissions is not shared to adhere to the highest privacy standards.

REFERENCES

[1] ACM Committee for Computing Education in Community Colleges (CCECC). 2023. *Bloom's for Computing: Enhancing Bloom's Revised Taxonomy with Verbs for Computing Disciplines*. ISBN: 9798400707636. DOI: 10.1145/3587276.

[2] B. Ahmeti, M. Linder, R. Groner, and R. Wohlrab. 2024. Architecture Decision Records in Practice: An Action Research Study. In *Software Architecture*, 333–349. DOI: 10.1007/978-3-031-70797-1_22.

[3] D. Akdur. 2022. Analysis of Software Engineering Skills Gap in the Industry. *ACM Trans. Comput. Educ.*, 23, 1, Article 16. DOI: 10.1145/3567837.

[4] J. Armarego. 2002. Advanced Software Design: a Case in Problem-based Learning. In *Conference on Software Engineering Education and Training* (CSEE&T '02), 44–54. DOI: 10.1109/CSEE.2002.995197.

[5] N. Assyne, H. Ghanbari, and M. Pulkkinen. 2022. The state of research on software engineering competencies: A systematic mapping study. *Journal of Systems and Software*, 185, 111183. DOI: 10.1016/j.jss.2021.111183.

[6] S. O. Bada and S. Olusegun. 2015. Constructivism Learning Theory: A Paradigm for Teaching and Learning. *Journal of Research & Method in Education* (IOSR-JRME), 5, 6, 66–70. https://iosrjournals.org/iosr-jrme/papers/Vol-5%20Issue-6/Version-1/I05616670.pdf.

[7] A. Baker and A. van der Hoek. 2009. An Experience Report on the Design and Delivery of Two New Software Design Courses. In *Technical Symposium on Computer Science Education* (SIGCSE '09), 519–523. DOI: 10.1145/1508865.1509045.

[8] A. Basiri, L. Hochstein, N. Jones, and H. Tucker. 2019. Automating Chaos Experiments in Production. In *International Conference on Software Engineering: Software Engineering in Practice* (ICSE-SEIP '19), 31–40. DOI: 10.1109/ICSE-SEIP.2019.00012.

[9] A. Begel, N. Nagappan, C. Poile, and L. Layman. 2009. Coordination in Large-Scale Software Teams. In *ICSE Workshop on Cooperative and Human Aspects on Software Engineering* (CHASE '09), 1–7. DOI: 10.1109/CHASE.2009.5071401.

[10] A. Begel and B. Simon. 2008. Novice Software Developers, All Over Again. In *International Workshop on Computing Education Research* (ICER '08), 3–14. DOI: 10.1145/1404520.1404522.

[11] M. I. Board. 1999. Mars Climate Orbiter Mishap Investigation Board Phase I Report November 10, 1999. (1999). https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf.

[12] C. C. Bonwell and J. A. Eison. 1991. *Active Learning: Creating Excitement in theClassroom. 1991 ASHE-ERIC Higher EducationReports*. ISBN: 1878380087. https://eric.ed.gov/?id=ED336049.

[13] T. Carleton and L. Leifer. 2009. Stanford's ME310 Course as an Evolution of Engineering Design. In *CIRP Design Conference – Competitive Design*. http://hdl.handle.net/1826/3648.

[14] D. Carrington and S.-K. Kim. 2003. Teaching Software Design with Open Source Software. In *Frontiers in Education* (FIE '03). Volume 3, S1C–9. DOI: 10.1109/FIE.2003.1265910.

[15] C. Y. Chong, E. Kang, and M. Shaw. 2023. Open Design Case Study - A Crowdsourcing Effort to Curate Software Design Case Studies. In *International Conference on Software Engineering: Software Engineering Education and Training* (ICSE-SEET '23), 23–28. DOI: 10.1109/ICSE-SEET58685.2023.00008.

[16] T. Clear, S. Beecham, J. Barr, M. Daniels, R. McDermott, M. Oudshoorn, A. Savickaite, and J. Noll. 2015. Challenges and Recommendations for the Design and Conduct of Global Software Engineering Courses: A Systematic Review. In *ITiCSE on Working Group Reports* (ITICSE-WGR '15), 1–39. DOI: 10.1145/2858796.2858797.

[17] D. Coppit. 2006. Implementing Large Projects in Software Engineering Courses. *Computer Science Education*, 16, 1, 53–73. DOI: 10.1080/08993400600600443.

[18] D. Coppit and J. M. Haddox-Schatz. 2005. Large Team Projects in Software Engineering Courses. In *Technical Symposium on Computer Science Education* (SIGCSE '05), 137–141. DOI: 10.1145/1047344.1047400.

[19] N. Cross. 1982. Designerly ways of knowing. *Design Studies*, 3, 4, 221–227. Special Issue Design Education. DOI: 10.1016/0142-694X(82)90040-0.

[20] D. Damian, A. Hadwin, and B. Al-Ani. 2006. An Experiment on Teaching Coordination in a Globally Distributed Software Engineering Class. In *International Conference on Software Engineering* (ICSE '06), 685–690. DOI: 10.1145/1134285.1134391.

[21] L. Deslauriers, L. S. McCarty, K. Miller, K. Callaghan, and G. Kestin. 2019. Measuring actual learning versus feeling of learning in response to being actively engaged in the classroom. *Proceedings of the National Academy of Sciences*, 116, 39, 19251–19257. DOI: 10.1073/pnas.1821936116.

[22] A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, and C. Zander. 2006. Can Graduating Students Design Software Systems? In *Technical Symposium on Computer Science Education* (SIGCSE '06), 403–407. DOI: 10.1145/1121341.1121468.

[23] G. Fairbanks. 2010. *Just Enough Software Architecture: A Risk-Driven Approach*. ISBN: 9780984618101.

[24] G. Fairbanks. 2023. Software Architecture is a Set of Abstractions. *IEEE Software*, 40, 4, 110–113. DOI: 10.1109/MS.2023.3269675.

[25] J. Firth, I. Rivers, and J. Boyle. 2021. A systematic review of interleaving as a concept learning strategy. *Review of Education*, 9, 2, 642–684. DOI: 10.1002/rev3.3266.

[26] K. Fowler, M. Windschitl, and J. Richards. 2019. Exit Tickets. *The Science Teacher*, 86, 8, 18–26. DOI: 10.1080/00368555.2019.12293416.

[27] S. Freeman, S. L. Eddy, M. McDonough, M. K. Smith, N. Okoroafor, H. Jordt, and M. P. Wenderoth. 2014. Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences*, 111, 23, 8410–8415. DOI: 10.1073/pnas.1319030111.

[28] M. Galster and S. Angelov. 2016. What makes teaching software architecture difficult? In *International Conference on Software Engineering Companion* (ICSE '16), 356–359. DOI: 10.1145/2889160.2889187.

[29] K. Garg and V. Varma. 2007. A Study of the Effectiveness of Case Study Approach in Software Engineering Education. In *Conference on Software Engineering Education and Training* (CSEE&T '07), 309–316. DOI: 10.1109/CSEET.2007.8.

[30] D. Garlan, R. Allen, and J. Ockerbloom. 1995. Architectural Mismatch: Why Reuse is so Hard. *IEEE Software*, 12, 6, 17–26. DOI: 10.1109/52.469757.

[31] D. Garlan, M. Shaw, C. Okasaki, C. M. Scott, and R. F. Swonger. 1992. Experience with a Course on Architectures for Software Systems. In *Software Engineering Education*, 23–43. DOI: 10.1007/3-540-55963-9_38.

[32] V. Garousi, G. Giray, E. Tüzün, C. Catal, and M. Felderer. 2019. Aligning software engineering education with industrial needs: A meta-analysis. *Journal of Systems and Software*, 156, 65–83. DOI: 10.1016/j.jss.2019.06.044.

[33] C. Ghezzi and D. Mandrioli. 2006. The Challenges of Software Engineering Education. In *Software Engineering Education in the Modern Age*, 115–127. DOI: 10.1007/11949374_8.

[34] R. S. Hansen. 2006. Benefits and Problems With Student Teams: Suggestions for Improving Team Projects. *Journal of Education for Business*, 82, 1, 11–19. DOI: 10.3200/JOEB.82.1.11-19.

[35] Q. Hao, B. Barnes, E. Wright, and E. Kim. 2018. Effects of Active Learning Environments and Instructional Methods in Computer Science Education. In *Technical Symposium on Computer Science Education* (SIGCSE '18), 934–939. DOI: 10.1145/3159450.3159451.

[36] S. Hartikainen, H. Rintala, L. Pylväs, and P. Nokelainen. 2019. The Concept of Active Learning and the Measurement of Learning Outcomes: A Review of Research in Engineering Higher Education. *Education Sciences*, 9, 4. DOI: 10.3390/educsci9040276.

[37] R. Hjelsvold and D. Mishra. 2019. Exploring and Expanding GSE Education with Open Source Software Development. *ACM Trans. Comput. Educ.*, 19, 2, Article 12. DOI: 10.1145/3230012.

[38] C. Hu. 2013. The nature of software design and its teaching: an exposition. *ACM Inroads*, 4, 2, 62–72. DOI: 10.1145/2465085.2465103.

[39] M. Jackson. 1995. The World and the Machine. In *International Conference on Software Engineering* (ICSE '95), 283–292. DOI: 10.1145/225014.225041.

[40] S. Jarzabek. 2013. Teaching Advanced Software Design in Team-Based Project Course. In *International Conference on Software Engineering Education and Training* (CSEE&T '13), 31–40. DOI: 10.1109/CSEET.2013.6595234.

[41] C. W. Johnson and I. Barnes. 2005. Redesigning the Intermediate Course in Software Design. In *Australasian Conference on Computing Education - Volume 42* (ACE '05), 249–258. https://crpit.scem.westernsydney.edu.au/confpapers/CRPITV42Johnson.pdf.

[42] R. Jolak, A. Wortmann, M. Chaudron, and B. Rumpe. 2018. Does Distance Still Matter? Revisiting Collaborative Distributed Software

Design. *IEEE Software*, 35, 6, 40–47. DOI: 10.1109/MS.2018.290100920.

[43] D. Kahneman. 2003. Maps of Bounded Rationality: Psychology for Behavioral Economics. *American Economic Review*, 93, 5, 1449–1475. DOI: 10.1257/000282803322655392.

[44] S. H. K. Kang. 2016. Spaced Repetition Promotes Efficient and Effective Learning: Policy Implications for Instruction. *Policy Insights from the Behavioral and Brain Sciences*, 3, 1, 12–19. DOI: 10.1177/2372732215624708.

[45] C. F. Kemerer and M. C. Paulk. 2009. The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. *IEEE Transactions on Software Engineering* (TSE), 35, 4, 534–550. DOI: 10.1109/TSE.2009.27.

[46] A. N. Kumar, R. K. Raj, S. G. Aly, M. D. Anderson, B. A. Becker, R. L. Blumenthal, E. Eaton, S. L. Epstein, M. Goldweber, P. Jalote, D. Lea, M. Oudshoorn, M. Pias, S. Reiser, C. Servin, R. Simha, T. Winters, and Q. Xiang. 2024. *Computer Science Curricula 2023*. ISBN: 9798400710339. DOI: 10.1145/3664191.

[47] Z. S. Li, N. N. Arony, K. Devathasan, and D. Damian. 2023. "Software is the Easy Part of Software Engineering" — Lessons and Experiences from A Large-Scale, Multi-Team Capstone Course. In *International Conference on Software Engineering: Software Engineering Education and Training* (ICSE-SEET '23), 223–234. DOI: 10.1109/ICSE-SEET58685.2023.00027.

[48] J. T. Liang, M. Arab, M. Ko, A. J. Ko, and T. D. LaToza. 2023. A Qualitative Study on the Implementation Design Decisions of Developers. In *International Conference on Software Engineering* (ICSE '23), 435–447. DOI: 10.1109/ICSE48619.2023.00047.

[49] J.-L. Lions, L. Luebeck, J.-L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran. 1996. Ariane 5 flight 501 failure report by the inquiry board. (1996). https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

[50] C. Loftus, L. Thomas, and C. Zander. 2011. Can Graduating Students Design: Revisited. In *Technical Symposium on Computer Science Education* (SIGCSE '11), 105–110. DOI: 10.1145/1953163.1953199.

[51] M. Luukkainen, A. Vihavainen, and T. Vikberg. 2012. Three Years of Design-based Research to Reform a Software Engineering Curriculum. In *Annual Conference on Information Technology Education* (SIGITE '12), 209–214. DOI: 10.1145/2380552.2380613.

[52] T. Mannisto, J. Savolainen, and V. Myllarniemi. 2008. Teaching Software Architecture Design. In *Working Conference on Software Architecture* (WICSA '08), 117–124. DOI: 10.1109/WICSA.2008.34.

[53] C. Matthies, J. Huegle, T. Dürschmid, and R. Teusner. 2019. Attitudes, Beliefs, and Development Data Concerning Agile Software Development Practices. In *International Conference on Software Engineering: Software Engineering Education and Training Track* (ICSE-SEET '19), 158–169. DOI: 10.1109/ICSE-SEET.2019.00025.

[54] C. Matthies, T. Kowark, and M. Uflacker. 2016. Teaching Agile the Agile Way — Employing Self-Organizing Teams in a University Software Engineering Course. In *ASEE International Forum*. DOI: 10.18260/1-2--27259.

[55] M. E. McMurtrey, J. P. Downey, S. M. Zeltmann, and W. H. Friedman. 2008. Critical Skill Sets of Entry-Level IT Professionals: An Empirical Examination of Perceptions from Field Personnel. *Journal of Information Technology Education: Research*, 7, 1, 101–120. DOI: 10.28945/181.

[56] G. Meszaros. 2007. *xUnit Test Patterns: Refactoring Test Code*. ISBN: 9780131495050. http://xunitpatterns.com/Test%20Double.html.

[57] R. Negretti. 2012. Metacognition in Student Academic Writing: A Longitudinal Study of Metacognitive Awareness and Its Relation to Task Perception, Self-Regulation, and Evaluation of Performance. *Written Communication*, 29, 2, 142–179. DOI: 10.1177/0741088312438529.

[58] S. Ouhbi and N. Pombo. 2020. Software Engineering Education: Challenges and Perspectives. In *Global Engineering Education Conference* (EDUCON '20), 202–209. DOI: 10.1109/EDUCON45650.2020.9125353.

[59] W. L. Pantoja Yépez, J. A. Hurtado Alegría, A. Bandi, and A. W. Kiwelekar. 2023. Training software architects suiting software industry needs: A literature review. *Education and Information Technologies*. DOI: 10.1007/s10639-023-12149-x.

[60] M. Petre. 2009. Insights from Expert Software Design Practice. In *Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering* (ESEC/FSE '09), 233–242. DOI: 10.1145/1595696.1595731.

[61] R. Plösch, J. Bräuer, C. Körner, and M. Saft. 2016. MUSE: A Framework for Measuring Object-Oriented Design Quality. *Journal of Object Technology*, 15, 4, 2:1–29. DOI: 10.5381/jot.2016.15.4.a2.

[62] C. Pretorius, M. Razavian, K. Eling, and F. Langerak. 2024. When rationality meets intuition: A research agenda for software design decision-making. *Journal of Software: Evolution and Process*, 36, 9, e2664. DOI: 10.1002/smr.2664.

[63] A. Radermacher and G. Walia. 2013. Gaps Between Industry Expectations and the Abilities of Graduates. In *Technical Symposium on Computer Science Education* (SIGCSE '13), 525–530. DOI: 10.1145/2445196.2445351.

[64] Ç. Semerci and V. Batdi. 2015. A Meta-Analysis of Constructivist Learning Approach on Learners' Academic Achievements, Retention and Attitudes. *Journal of Education and Training Studies*, 3, 2, 171–180. DOI: 10.11114/jets.v3i2.644.

[65] M. Shaw. 2000. Software Engineering Education: A Roadmap. In *Conference on The Future of Software Engineering* (ICSE '00), 371–380. DOI: 10.1145/336512.336592.

[66] M. Shaw, J. Herbsleb, and I. Ozkaya. 2005. Deciding What to Design: Closing a Gap in Software Engineering Education. In *International Conference on Software Engineering* (ICSE '05), 607–608. DOI: 10.1145/1062455.1062563.

[67] M. Shaw and J. E. Tomayko. 1991. Models for undergraduate project courses in software engineering. In *Software Engineering Education*, 33–71. DOI: 10.1007/BFb0024284.

[68] A. Tang, M. Razavian, B. Paech, and T.-M. Hesse. 2017. Human Aspects in Software Architecture Decision Making: A Literature Review. In *International Conference on Software Architecture* (ICSA '17), 107–116. DOI: 10.1109/ICSA.2017.15.

[69] A. Tang, M. H. Tran, J. Han, and H. van Vliet. 2008. Design Reasoning Improves Software Design Quality. In *Quality of Software Architectures. Models and Architectures*, 28–42. DOI: 10.1007/978-3-540-87879-7_2.

[70] S. Tenhunen, T. Männistö, M. Luukkainen, and P. Ihantola. 2023. A systematic literature review of capstone courses in software engineering. *Information and Software Technology*, 159, 107191. DOI: 10.1016/j.infsof.2023.107191.

[71] C. Thevathayan and M. Hamilton. 2017. Imparting Software Engineering Design Skills. In *Australasian Computing Education Conference* (ACE '17), 95–102. DOI: 10.1145/3013499.3013511.

[72] D. Tofan, M. Galster, and P. Avgeriou. 2013. Difficulty of Architectural Decisions — A Survey with Professional Architects. In *Software Architecture*, 192–199. DOI: 10.1007/978-3-642-39031-9_17.

[73] H. van Vliet and A. Tang. 2016. Decision making in software architecture. *Journal of Systems and Software*, 117, 638–644. DOI: 10.1016/j.jss.2016.01.017.

[74] V. Varma and K. Garg. 2005. Case Studies: The Potential Teaching Instruments for Software Engineering Education. In *International Conference on Quality Software* (QSIC '05), 279–284. DOI: 10.1109/QSIC.2005.18.

[75] I. Warren. 2005. Teaching Patterns and Software Design. In *Australasian Conference on Computing Education - Volume 42* (ACE '05), 39–49. https://crpit.scem.westernsydney.edu.au/confpapers/CRPITV42Warren.pdf.

[76] B. Wu and A. I. Wang. 2012. A Guideline for Game Development-Based Learning: A Literature Review. *International Journal of Computer Games Technology*, 2012, 1, 103710. DOI: 10.1155/2012/103710.

[77] S. S. Yau and J. J.-P. Tsai. 1986. A Survey of Software Design Techniques. *Transactions on Software Engineering* (TSE), SE-12, 6, 713–721. DOI: 10.1109/TSE.1986.6312969.

[78] J. Zhang and J. Li. 2010. Teaching Software Engineering Using Case Study. In *International Conference on Biomedical Engineering and Computer Science* (ICBECS '10), 1–4. DOI: 10.1109/ICBECS.2010.5462378.

[79] L. Zhang, Y. Li, and N. Ge. 2020. Exploration on Theoretical and Practical Projects of Software Architecture Course. In *International Conference on Computer Science & Education* (ICCSE), 391–395. DOI: 10.1109/ICCSE49874.2020.9201748.

[80] X. Zhang and H. Pham. 2000. An analysis of factors affecting software reliability. *Journal of Systems and Software*, 50, 1, 43–56. DOI: 10.1016/S0164-1212(99)00075-8.

[81] C. Ziftci and B. Greenberg. 2023. Improving Design Reviews at Google. In *International Conference on Automated Software Engineering* (ASE '23), 1849–1854. DOI: 10.1109/ASE56229.2023.00066.