

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

Anonymous Author(s)
anonymized university

Abstract—Designing software systems is an essential technical skill of professional software engineering. However, recent graduates often lack important software design skills, such as generating alternative designs, communicating them effectively, and collaborating across teams. This skill gap motivates the need for a larger emphasis on the education of the activity and process of designing software systems. Teaching software design as an activity is challenging, as instructors have to find the right balance of teaching generalizable principles and design skills while still making the taught concepts actionable. They also have to balance teaching a sufficient number of different design techniques while also letting students gain enough practical experience with applying the taught design techniques in a realistic and sufficiently complex system. In this paper, we present our experience of designing and instructing a novel 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. Based on student feedback and our observations of the 17 students' learning success, the course has been successful overall. We describe actionable lessons learned and recommendations for how to effectively teach software design. To allow other instructors to adopt or improve our course design, we have made all teaching materials available.

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

I. INTRODUCTION

Designing software systems is an essential technical software engineering skill [2, 4, 71]. Software design plays a crucial role in the success of software products, as design decisions have a long-lasting impact on quality attributes, such as changeability, interoperability, reusability, robustness, scalability, and testability [68, 76, 79]. Furthermore, the ever-growing complexity of software systems increasingly requires software engineers to model software using various design abstractions to document, evaluate, and collaboratively discuss design options [18, 40, 57, 66].

However, recent graduates often lack important software design skills, such as generating alternative designs, communicating them effectively, and working across large teams [9, 30, 60]. Multi-national, multi-institutional experiments have shown that the majority of graduating students in computer science lack the skills of designing software systems [21, 46]. This gap between industry-needed software design competences and the skills of recent graduates has also been confirmed by surveys of software practitioners [2].

This skill gap motivates the need for a larger emphasis on software design education in universities [30, 31]. In many cases, software design is taught as just a small part of an overall software engineering course [2, 55, 69], giving students

little instruction on how to design and insufficient practice of these skills in projects that are large enough to expose students to practical software design challenges [9, 36, 56, 60]. In the cases in which software design is taught in a dedicated course, learning objectives mostly focus on design patterns, architectural styles, and quality attributes [56], which are important concepts to produce high-quality design artifacts. However, in contrast to design as an *artifact*, design as an *activity* [18] is rarely taught as a primary objective of the course [6, 56]. Therefore, students often lack the engineering mindset to systematically guide them while designing software [9, 30, 60]. Hence, more dedicated education on design activities is needed to foster an engineering mindset.

Teaching software design activities is challenging as instructors have to find the right balance between teaching generalizable processes and abstract design skills while still making the taught concepts actionable [36, 70]. Instructors also have to balance teaching a sufficient number of different design techniques while also allowing students gain enough practical experience with applying the taught design techniques to a realistic and sufficiently complex system [27, 36, 56, 70]. 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 [16, 17], such as compatibility of independently developed components, cross-team communication, component responsibility assignments, and workload distribution. Therefore, we argue that software design is most effectively taught with a large-scale multi-team project that provides students with opportunities to learn and practice skills needed in most professional software development projects.

In this paper, we present our experience of designing and instructing 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. In this course, students learn how to generate, communicate, and evaluate designs, how to generate multiple alternative designs, how to decompose a system into separately developed services, and how to integrate services and systems developed by different teams. In lectures, students learn design principles based on positive and negative real-world case studies using Constructivism Learning Theory [5]. In the course project, students teams collaboratively design, implement, test, and integrate a large-scale multi-service system and describe important design decisions in milestone reports.

Based on student feedback and our observations of the

17 students' learning success, the course has been successful overall. Students enjoyed the course and could successfully design, implement, and integrate a complex web application with components being developed and maintained by collaborating teams. As observed in the submitted assessments and exams, students' performance on design activities improved throughout the semester.

In this paper, we describe lessons learned and actionable recommendations for how to effectively teach software design. We discuss the teaching techniques we used and what we believe most strongly contributed to the positive outcomes. Furthermore, we identified some challenges that we observed and discuss potential improvements.

II. RELATED WORK

A. Software Design Courses

As software design is one of the major activities in software engineering, courses on this topic have been taught for multiple decades [56, 65].

Lecture-focused Courses: Many software design courses found in the literature focus on lecture-based learning without a major project component [56]. Some courses focus on teaching software design based on design patterns and stay closer to a source code [39, 74]. Other courses focus on high-level component interactions, architectural styles, and quality attributes [29, 48]. 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 [6] teaches software design primarily as a methodological activity [18].

Team-Project-based Courses: Some software design courses include a major team-project component [56, 65]. For example, in a course taught at Murdoch University students practice modular decomposition, architecture design, and learn to specify component interfaces in teams of six [3, 38]. UC Irvine includes two team projects in their software course during which students design and implement a system in teams of 14 students [6]. Courses taught at the University of Queensland [12] and Beihang University [78] 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 [73, 75]. In existing software design courses student teams generally work individually, rather than collaboratively developing a system across teams. In contrast, our course offers students the opportunity to experience a more challenges and realistic development context in which students have to integrate components built by other teams over multiple iterations.

B. Multi-Team Courses

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

Agile Processes: A course on scaling agile Scrum, which has been taught for multiple years at Hasso Plattner Institute, lets students build a web application with multiple interacting

teams [49, 50, 51]. The course teaches the Scrum process and modern software engineering 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 [49]. Students receive the role of either scrum master, product owner, or developers while customers are simulated by the teaching team [50]. Students learn by making decisions about their development process autonomously and reflecting on their decisions after each sprint [51]. Our course has been partially inspired by the teaching methods used in this course, while similar courses are taught at the College of William and Mary [16, 17], the University of Helsinki [47], and the University of Victoria [44]. However, in contrast to multi-team courses on agile processes, the learning objectives of our course focus on software design.

Global Software Development: Some courses teach even harder-to-practice skills of developing a product via collaborating globally-distributed teams [11, 15, 19, 35]. 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 includes intermediate programming skills and experience with developing and testing medium-size programs. The course builds on the programming skills that students have obtained through previously taken programming 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 [5] (Section IV).
- 2) A semester-long **multi-team project** in which all teams collectively build and integrate a system composed of different services (Section V).
- 3) Three **individual homework** assignments during which students practice skills taught in the lectures (Section VI).

A. Learning Objectives (LOs)

As there are few existing courses that 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 the most important ones to teach an engineering mindset [18] of software design.

Requirements analysis and specification are important skills for all software engineers [2, 4, 37, 60, 64], as prioritized requirements are the main drivers of software design [36, 56]. Therefore, a software design course should teach students how to elicit requirements:

LO 1 (Requirements)

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 skill of software design [18, 52]. Since considering multiple design alternatives is likely to lead to a better design [71], a software design course should teach students how to generate multiple solutions:

LO 2 (Generate)

Students should learn to: **Generate viable design solutions that appropriately satisfy the trade-offs between given requirements.**

Modeling is a central aspect of design [18, 23, 57] and essential for collaborative design [40, 66].

LO 3 (Communicate)

Students should learn to: **Apply appropriate abstractions & modeling techniques to communicate and document design solutions.**

Judging the quality of design options is essential for improving designs and assess requirements satisfaction [43].

LO 4 (Evaluate)

Students should learn to: **Evaluate design solutions based on their satisfaction of common design principles and trade-offs between different quality attributes.**

To build on existing knowledge and experiences, design principles provide guidance that helps students generate and evaluate design options [58, 61]. Hence, a software design course should provide students with a basic tool box of common design principles:

LO 5 (Design Principles)

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 scale.**

The process of how to design should be adjusted depending on the amount and types of risks in the domain, the organization, and other factors [22]. Therefore, a software design course should teach students a variety of agile and plan-driven design processes, their advantages, and disadvantages:

LO 6 (Process)

Students should learn to: **Explain how to adapt a software design process to fit different domains, such as robotics, web apps, mobile apps, and medical systems.**

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 [8, 9,

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

TABLE I: Lecture topics and learning objectives

60, 66]. Therefore, it is critical for a software design to teach students how to collaborate across teams:

LO 7 (Multi-Team)

Students should learn to: **Apply techniques of multi-team software design to design, develop, and integrate individually developed components into a complex system.**

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. The list of lectures and learning objectives that they fulfill is shown in Table I.

A. Teaching Software Design As An Activity

To teach students to design with an engineering mindset [18], we include lectures on the following design activities, which we summarize as the “*GCE-Paradigm*” (i.e., the *process* of iteratively *generating*, *communicating*, and *evaluating* design options based on *requirements*).

Requirements Analysis: As design has to consider the real-world problem for which a solution should be created, we teach students to identify important requirements and domain assumptions. 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

practice identifying important requirements and assumptions across different domains.

Communicating Designs via Abstractions: Since the communication of design is essential for collaboration and evaluation, we teach students how to model designs using appropriate abstractions. Interleaved [24] throughout Lectures 2, 3, 4, and 9, we teach students the abstractions that are most commonly used to communicate design solutions (context and component diagrams, sequence diagrams, data models, interface description, and CRC cards). As a use of spaced repetition [42], we use these abstraction throughout following lectures, homework, and project milestones.

Generating Design Alternatives: In Lecture 6, we provide an overview of techniques that help generating design options. First, we motivate the importance of thinking of different design alternatives, as this is likely to result in a better design [71]. Then, we teach brainstorming techniques (e.g., writing ideas on post-its, clustering, combination of 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 start 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 attributes dimensions. In Lecture 4, we introduce quality attributes based on the two different 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. As a use of spaced repetition [42], we ask students throughout many lectures to identify important quality attribute dimensions, specify quality attribute requirements, and evaluate design options.

Design Process: To convey the idea that the amount of design effort should depend on the criticality of the system being developed, we teach a risk-driven design approach [22]. We show how this approach fits into agile as well as more waterfall-like software development processes. We include activities in which students identify risks for different domains (e.g., online shops, games, medical software, spacecraft systems, startups, and social media systems). We also teach the human aspects of software design [67] by contrasting intuitive decision making with rational decision making [59], discussing bounded rationality [41], and emphasize that design is a collaborative hands-on activity.

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

The students generally responded positively about the lec-

tures. 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 (5 students), reuse (3 students), interoperability (2 students), testability (2 students), and changeability (2 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 anonymized university.”

No majority opinion emerges 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?” two students asked for more real-world examples in lectures, two students asked for more content on scalability, and one student each asked for more content on testability, security, robustness, and quality attributes broadly.

Lesson Learned 1 (Design as an Activity)	Lectures
Lectures on how to design large-scale software systems with the GCE-paradigm were well-received.	
<ul style="list-style-type: none"> • 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 [42] 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 software engineering education [28, 63, 72, 77] and have also been proposed for software design education in particular[13]. To illustrate the need for the design principles taught in the lectures (LO 5), we illustrated them based on the following real-world case studies of well-known software failures and success stories.

Global Distribution System (GDS) In the lecture on *design for interoperability*, we used GDS¹, the interface standard that is used by airlines and booking system 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 [10] case study as an example 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’ Simian Army: In the *design for testability* lecture, we used the Simian Army by Netflix as a positive

¹https://www.youtube.com/watch?v=1-m_Jse-cs

example for non-functional property testing of large-scale system [7].

Ariane 5 Rocket Launch Failure: In the *design with reuse* lecture, the well-known Ariane 5 failure (caused by an invalid assumption about the inertial reference unit in the software that was ported from Ariane 4) is used to illustrate the importance of identifying and checking assumptions made by reused components [45].

npm left-pad: In the *design with reuse* lecture, the suddenly unavailable, but highly dependent npm package `left-pad`² with trivial implementation is used to motivate the the design principle to strive for as few dependencies as possible.

Heartbleed: In the *design with reuse* lecture, the Heartbleed bug³ is used to motivated the importance of updating critical dependencies.

Twitter: In the *design for scalability* lecture, Twitter⁴ is used as a case study to demonstrate different approaches for scaling a system based on the amount of client demands.

Experience: Overall, we believe that the case studies were valuable for conveying the key course concepts and maintaining student engagement. In the middle of the semester, we collected student feedback on the course in an early-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 with quotes, such as “*Examples of design scenarios and code snippets make core ideas more concrete and easier to understand.*” and “*Use of real-world examples in lecture. 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 effective at teaching design principles (LO 5) and the software design process (LO 6) in this course.

- For complex case studies, such as GDS, and Netflix’ 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

Active student participation in lectures significantly improves learning outcomes [26, 33, 34]. As one form of active learning, we ask students to derive design principles themselves based on real-world case studies. Constructivism

learning theory [5] (i.e., letting students experience the step-by-step process of coming up with principles themselves) helps students gain a deeper understanding and feel a stronger connection with the principles they learn [62] and increases their meta-cognitive skills [54]. Therefore, we asked students to come up with design principles themselves based on real-world examples of successes and failures.

For example, in the *design for interoperability* lecture, we use a case study based on GDS, a system that is widely 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 towards design principles that apply to future projects, which they described as creating a shared data format or an interface between systems. Then, we show the students the example of the Mars Climate Orbiter failure [10] (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. Then, in written assignments and a course project, we encourage the students apply the newly learned design principles to a different system and describe how the design principle improves quality attributes of the system. For example, in the multi-team course project, the students describe the interfaces of the services that they need to implement using syntactic and semantic documentation. If they forget to mention the units of temperature values in their interface descriptions, we specifically provide feedback mentioning the Mars Climate Orbiter failure, to remind them of possible consequences of imprecise documentation.

To identify whether the students who did not speak up during lectures also understood the learning objectives, the end of each lecture, we included an exit ticket [25], i.e., a digital assignment in which students were asked to summarize the lecture’s main message in their own words.

Experience: In the mid-semester focus group run by an outside consultant, 46 % of students agreed that in-class discussions help them learn design principles more effectively,

Lesson Learned 3 (Constructivism)

Lectures

The use of constructivism for teaching design principles (LO 5) was overall successful in this course.

- Give students 5–10 min of silent thinking and small-group discussions before sharing 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.

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

³<https://heartbleed.com/>

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

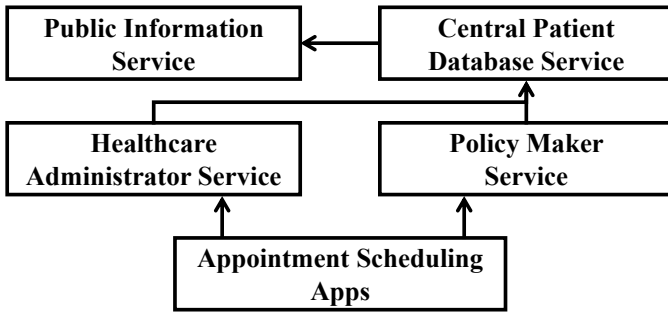


Fig. 1: Component Diagram of the project system.

with quotes such as “In class discussions help us think and reason over content” and “Reiteration of ideas; students have different perspectives”. Considering that students often subjectively under-value the effectiveness of active learning techniques [20], these results show initial evidence for the overall success of constructivism as an effective teaching for teaching design principles in this course. Based on the student quote “Don’t know what they expect as answers when they put us into discussion groups”, we identify clarity of questions as a potential challenge of the technique. When asking students to describe design principles, they might initially not know what type of answer is expected of them. Finally, all students agreed that exit tickets helped them learn, with quotes such as “Exit tickets are good since they help us summarize lectures”.

V. MULTI-TEAM PROJECT

While teamwork is one of the most important soft skills in professional software development [2], graduates in computer science often lack the skill of collaborating across teams [9, 60] or lack experience working on large projects [60]. To satisfy LO 7 (Multi-Team) and to give students an experience with designing and developing a system large enough that no single person fully understands the entire system, we decided to include a multi-team project in this course.

In the project, each team develops their own medical appointment scheduling app and one of the four services shown in Figure 1. The teams are eventually asked to integrate their scheduling app and service with other teams’ work.

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 [44], which we address with 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)

Experience: The four teams built a system with a total size of 19.5 KLOC. This amounts to 1.15 KLOC per students on average. Overall, the developed system was functionally correct and services integrated well with each other. The project work provided many insightful learning opportunities for students, which are discussed in the following sections.

A. Cross-Team Communicator

As identified in previous work on multi-project software engineering courses [16, 17, 44], communication between teams is a major challenge. Our approach to reducing communication overhead between teams includes three elements: (1) Dedicating a cross-team communicator role for each team, (2) Using class time for cross-team communication, (3) Providing a shared messaging channel for cross-team communication.

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 communicator, they did not fully understand the technical implications of these decisions on the team’s workload and required technical expertise. This lead 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 come up with a redesign of 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 teams.

- To reduce the risks of multi-team challenges (LO 7) let teams pick a cross-team communication who will serve as a facade of the team and interfaces 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 that did not contribute 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 another team and to use components developed by other teams, we let all teams describe the interface of their service before they start implementing it. Students are asked to create OpenAPI specifications that document the syntactic and semantics and review each others’ interfaces.

Experience: Students had surprisingly few service integration issues. Considering that each service was developed individually and most students were first experiencing a development project with that many people, we were generally surprised by the high interface compatibility between the

511 services. We see the reason for this mostly in the effective
512 use and discussion of interface descriptions.

513 Lesson Learned 5 (Interface Descriptions) Project

514 **Interface descriptions helped students in this course
515 independently develop compatible services (LO 7).**

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

522 C. Test Double Components

523 While all teams develop their own services, dependent-
524 on-services that their service is using are not immediately
525 available for testing. To address this challenge and to allow
526 students to simulate data sent from other components, we
527 teach students to implement *test double components* (i.e.,
528 components that mimic the interface of a required service to
529 control indirect inputs or verify indirect outputs [53]) based on
530 interface specifications in the *design for testability* lecture and
531 ask them to implement them for dependent-on components.

532 **Experience:** Test doubles helped students find some, but
533 not all, bugs before integration. Students also mentioned that
534 in the project, test double components helped “*isolating the
535 influence of external components*”. Many teams implemented
536 test doubles components via conditional logic within their
537 components, rather than as a separate component, which made
538 replacing them with real components slightly harder.

539 Lesson Learned 6 (Test Double Components) Project

540 **Test double components helped students in this course
541 independently develop and integrate services.**

- 542 • To ease replacing test double components with the real
543 components, recommend students to implement test dou-
544 ble components by mocking request / response messages
545 rather than simply mocking functions inside their own
546 component.
- 547 • To simplify implementation tasks, point students to li-
548 braries and frameworks that inject HTTP messages.

549 D. Milestone Reports

550 Asking students to submit multiple written reports on the
551 progress of their project lets students receive constructive
552 feedback and observe their own growth [32].

553 Many companies, such as Google, use Design Docs or other
554 architecture decision records [1] to describe their important
555 design decisions [14, 80], which are similar to the design
556 documents students create in this course. In mile stone reports,
557 students practice to generate (LO 2), communicate (LO 3), and
558 evaluate (LO 4) different design options, as we ask students to
559 discuss different design options, model them using appropriate
560 abstractions, and justify their decisions. The following sections
561 describe each milestone and our experience.

562 E. Milestone 1 (Domain Modeling & Initial System Design)

563 In the first milestone, students are given the description of
564 the context and requirements of the small system (medical
565 appointment scheduling app) that every team should imple-
566 ment. Based on this, students are asked to model a the
567 problem domain (LO 3), identify important quality attribute
568 requirements (LO 1), and model a first high-level design
569 solution of their system (LO 2 and LO 3). We gave students
570 two weeks for this milestone.

571 **Experience:** In an end-of-semester survey asking for feed-
572 back on every milestone, virtually all students said this mile-
573 stone was “*Good*” or “*Great*” and spent less time on the
574 milestone than we anticipated. Based on the submitted reports,
575 students made fewer design decisions (especially on the choice
576 of technologies and web frameworks) than we anticipated.
577 Therefore, we recommend to include more mandatory ques-
578 tions on particularly important decisions so that more design
579 decisions are made in this milestone.

580 F. Milestone 2 (First Prototype Development)

581 In the second milestone, students should refine (LO 2) and
582 implement the design they described in M1, implement tests
583 to evaluate the end-to-end functionality (LO 4), and reflect on
584 how the design changed and which other alternatives options
585 they considered (LO 2). We initially gave students two weeks
586 for this milestone.

587 **Experience:** Students took more time for this milestone
588 than we anticipated, requiring us to extend the milestone by
589 one week. In the end-of-semester survey many students said
590 the workload was too high (e.g., “*More time should be given
591 to this milestone because for some of the members in the
592 group are still in the learning stage of some frontend/backend
593 framework.*”). Moving more design work to milestone 1 can
594 help address this issue, as well as providing dedicated imple-
595 mentation support. We also recommend to plan three weeks
596 for this milestone, specifically telling students to start on the
597 first day, and regularly checking in with students to see how
598 much progress they made.

599 Lesson Learned 7 (Implementation Support) Project

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

- 603 • To reduce the amount of time students spend on coding
604 and allow them to focus more on design activities,
605 include coding templates that help students implement
606 their systems more efficiently.
- 607 • Link tutorials to common frameworks and libraries.
- 608 • Include a recitation in the beginning of the course that
609 introduces commonly used code generation techniques.
- 610 • Mention recommended pre-required knowledge in web
611 development, or the respective domain of a different
612 project system, in the course description.

G. Milestone 3 (Design for Changeability & Interoperability)

In the third milestone, students are first introduced to the four other services that they will design and implement to interoperate with other teams' services. The milestone provides a description of the functionality for each service as well as tips for cross-team collaboration via cross-team channels and a dedicated "interface person" (see Section V-A). Based on this description and service assignment per team, students are asked to design their service (LO 2), model it using interface descriptions (LO 3), and collaborate with other teams to ensure compatibility (LO 7). To further support service compatibility, students are asked to design test doubles (see Section V-C) for two of the most central services. Students are also asked to re-design their appointment scheduling app to support certain future changes (LO 2) and add test to evaluate the functionality (LO 4). In a design reflection students should report on design decisions they made during interface design, the changes they have made and describe a change impact analysis of two potential future changes.

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

H. Milestone 4 (Service Development & Integration)

The fourth milestone is split into two parts. In the first part teams are asked to implement their services, while collaborating with other teams to ensure compatibility (LO 7), and implement test doubles for adjacent services. Then they should deploy their services and provide other teams with the URL and port where their service can be accessed. 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 4). In a design reflection students should report on the design principles they used, 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 surprisingly smoothly. In the end-of-semester survey students mentioned this milestone "helped understand teamwork and how to collaboratively work with others".

I. Milestone 5 (Robustness Testing)

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. They should then report their findings to the team that developed the service. In the last task students were asked to describe at least two design options for at least two of the issues that other teams found and describe the improved designs. Due to time limitations and

due to this task strongly relying on the findings of other teams, this task was optional and only gave bonus points. However, all teams actually successfully completed this optional task.

Experience: Students thoroughly enjoyed breaking the services of other teams. In the end-of-semester survey, students particularly mentioned this milestone was "useful to understand what issues a system can potentially face and what could be potential loopholes". As students spend a bit less time on this than we expected, considering to expand the milestone by asking the students to identify issues for a larger variety (e.g., performance, correctness, availability, security) is one potential improvement.

Lesson Learned 8 (Milestone Reports)

Project

Milestone reports have been helpful at assessing 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 them quickly.
- Allow students to redo some milestone reports for an improved grade to incentivize students to take provided feedback seriously.

J. Final Presentations

To let students reflect on their overall experience in the project, we asked them to present their own lessons learned across the main learning objectives of the course. We encouraged teams to let every member present at least one insight.

Experience: All final presentations included insightful lessons learned, demonstrating that students used the opportunity to learn. As some presentations would have benefited from more concrete examples from which students learned their lessons, we recommend specifically asking students to demonstrate precise alignment of their lessons learned to concrete observations.

VI. HOMEWORK ASSIGNMENTS

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

A. HW2 - Domain and Design Modeling

The first homework is designed to let students practice domain analysis (LO 1) and modeling (LO 3). The homework is timed 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 assumption the made about the domain and design decision they made.

Experience: In an end-of-semester survey students overall liked the homework. Student quotes on the homework include: "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.", and "It was a useful learning experience, but I spend

a lot of time making the diagram.”, suggesting that some students spent more time on the homework than we anticipated, while still staying roughly in the range allocated for the course. Most submissions demonstrated accomplishment of the main learning objectives. The most common mistake was that 18 % of student submissions included domain entities in component diagrams, even though they belong in context diagrams.

B. HW2 - Design for Reuse

The second homework is designed to provide students with opportunities to practice the generation of multiple design alternatives (LO 2), communicate them using interface descriptions (LO 3), evaluate them for reusability (LO 4), and describe the design principles they support (LO 5).

We provide students with the source code of Python package PyPubSub (an implementation of the publish-subscribe connector). First, students are tasked to evaluate the package for reusability by identifying assumptions it is making about its reuse context, describe design principles most significantly contribute to the reusability of the package, and describe reuse scenarios (i.e., context of reuse, unit of reuse, maximum effort of adaptation required, and type of adaptation) in which reusing the package would be appropriate and not appropriate. Second, students are asked to pick one of the unsatisfied reuse scenarios and improve the package design to support that scenario. To communicate the design improvement, students should use interface descriptions and verbally describe how they would change the implementation. Finally students should describe how the redesign improves the reusability based on applied design principles or other arguments. The homework is intentionally designed to be open-ended to give students the opportunity to explore the reusability of the given module based on their own interests, domain experience.

Experience: In the end-of-semester survey students overall liked the homework. Student quotes on the homework include: “This was a good assignment to make us evaluate alternative designs.”, “Very good. Required much more thought about the reuse and how it works in practice.”, and “This hw is super useful. Letting me to make design decision is better than evaluating others’ design decisions”. Three students mentioned the homework was “a bit abstract” and “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 all students demonstrated sufficient accomplishment of the learning objectives. The most common mistakes were mostly related to the precise description of reuse scenarios we asked for (35 % of submissions), and partially lacking description of semantics in the interfaces (6 % of submissions). Due to the high number of students not following the four-part specification of reuse scenarios introduced in the lectures we hypothesize that many students tried to answer the homework without reviewing the corresponding lecture. Therefore, we recommend to explicitly remind students to check the slides for reuse scenarios.

C. HW3 - Design for Scalability

The third homework is designed to provide students with design generation (LO 2), communication (LO 3), and evaluation (LO 4) 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. Student quotes on the homework include: “It was a good balance between the time spend and learning outcome”, “Appropriate workload and provide a good learning experience”, and “It seems like it covers all the aspects of designing for scalability and has clear instructions”.

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, unclear justifications of design decisions, and unrealistic assumptions.

VII. DISCUSSION OF THE OVERALL COURSE EXPERIENCE

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. In this section we discuss to what degree we believe the course has accomplished this goal, based on our overall experience with the course, and describe potential improvements.

A. Design Abstractions

In both exams (mid-term and final exam) we asked students to describe at least two viable design options for each design problem, evaluate them, and discuss trade-offs.

In the mid-term exam students struggled most severely with interface descriptions. Only 58 % of submissions demonstrated sufficient accomplishment of the learning objective (6 % did not include an answer for the question, 12 % did not describe interfaces using an appropriate format, and 24 % lacked the descriptions of semantics). Submitted 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 received more practice with interface descriptions in the project and homework 3 (Section VI-C).

Therefore, we believe that adding an additional homework to practice interface descriptions in the first half of the semester would help students perform better in the mid-term.

Lesson Learned 9 (Design Abstractions)

Exams

In exams, students in this course struggled with precisely documenting interface descriptions (LO 3).

- Include multiple opportunities for students to practice interface descriptions in individual homework, recitations, and in-class exercises.

B. Generating Multiple Alternatives

In both exams (mid-term and final exam) we asked students to describe at least two viable design options for design problems, evaluate them, and discuss trade-offs between the two options: “Based on your identified risks, quality attribute requirements, and functional requirements specified above, generate design alternatives for this system. Describe two different design alternatives that satisfy the requirements. [...] Tip: You do not have to find the ‘best’ designs, but you should make a best effort to describe designs that reasonably address the requirements.”

In the submissions, especially in the mid-term, we noticed that 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 [71], we see this observation as a sign for needed improvements.

We see three potential explanations of the observation that likely all contributed: (1) Students did not receive enough practice with generating multiple viable design options; (2) The exam did not give them enough time to think of multiple good design options; (3) The task description did not clearly enough describe what a good design option is.

Lesson Learned 10 (Multiple Alternatives) Exams

In exams, students in this course struggled with describing multiple, viable design alternatives (LO 2).

- Include multiple individual homework and recitation exercises for students to practice generation of multiple design alternatives.
- Consider requesting a longer exam slot to give students more time to think of viable solutions.
- Consider including precise definitions of what considers a design option “good enough”.

C. Cross-Team Design Debate

One major challenge that students in this course encountered during the multi-team project was how to design the system in a way that the implementation effort of each service is roughly equal. Three of the teams came up with a design that would assign major responsibilities to the central database, who’s team was largely absent during these discussions. Unsurprisingly, 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 towards 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 on their preferred option and democratically reached a reasonable consensus.

While this discussion initially resulted from major frustrations and disagreements between teams, it provided one of the best learning opportunities for students to experience the complexity of real-world design considerations. 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 students prior to this incident. We believe that this discussion particularly helped students grow and integrate all major skills that are needed to effectively design large-scale software systems. Therefore, we recommend explicitly integrating more opportunities for students 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 data base team not attending, and students having had little time to generate design alternatives before this discussion, the debate following the heated Slack discussion was more productive.

Lesson Learned 11 (Design Debates) Project

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

- Include multiple opportunities for teams to debate cross-team design decisions during recitations or lectures.

VIII. CONCLUSIONS

In this paper we presented an innovative design of a course on designing large-scale software systems based on a real-world case studies and a multi-team project. We found that teaching software design based on the GCE-paradigm (i.e., the *process* of iteratively *generating*, *communicating*, and *evaluating* design options based on *requirements*) has shown promising initial results. In particular the use of real-world case studies and constructivism learning theory has been very successful, as students enjoyed this aspect of the lectures most. Our experience has shown that teaching software design as an activity to foster an engineering mindset has been promising, while more work is needed on improving the course to more effectively teach how to generate multiple alternatives and communicate them via appropriate abstractions.

We also found that the multi-team projects gave students many insightful learning opportunities that are rarely found in the context of university education. This experience motivates future work that empirically measures the effectiveness of a multi-team project on software design education, scales this approach to a larger number of students, and replicates this experience in other universities.

IX. DATA AVAILABILITY

To allow other instructors to adopt or improve our course design, we have made all teaching materials publicly available. As the slide templates would de-anatomize the authors’ affiliation, they are excluded from the review version. Non-aggregate data on student submissions is not shared to adhere to the highest privacy standards.

REFERENCES

- [1] B. Ahmeti, M. Linder, R. Groner, and R. Wohlrab. 2024. Architecture Decision Records in Practice: An Action Research Study. In *Software Architecture*, 333–349.
- [2] D. Akdur. 2022. Analysis of Software Engineering Skills Gap in the Industry. *ACM Trans. Comput. Educ.*, 23, 1, Article 16. DOI: 10.1145/3567837.
- [3] 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.
- [4] 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.
- [5] S. O. Bada and S. Olusegun. 2015. Constructivism Learning Theory: A Paradigm for Teaching and Learning. *Journal of Research & Method in Education*, 5, 6, 66–70. <https://iosrjournals.org/iosr-jrme/papers/Vol-5%20Issue-6/Version-1/I05616670.pdf>.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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>.
- [12] D. Carrington and S.-K. Kim. 2003. Teaching Software Design with Open Source Software. In *Frontiers in Education (FIE '03)*. Volume 3, SIC–9. DOI: 10.1109/FIE.2003.1265910.
- [13] 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)*, 23–28. DOI: 10.1109/ICSE-SEET58685.2023.00008.
- [14] M. Ciolkowski, O. Laitenberger, and S. Biffl. 2003. Software Reviews: The State of the Practice. *IEEE Software*, 20, 6, 46–51. DOI: 10.1109/MS.2003.1241366.
- [15] 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.
- [16] D. Coppit. 2006. Implementing Large Projects in Software Engineering Courses. *Computer Science Education*, 16, 1, 53–73. DOI: 10.1080/08993400600600443.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] G. Fairbanks. 2010. *Just Enough Software Architecture: A Risk-Driven Approach*. ISBN: 9780984618101.
- [23] G. Fairbanks. 2023. Software Architecture is a Set of Abstractions. *IEEE Software*, 40, 4, 110–113. DOI: 10.1109/MS.2023.3269675.
- [24] 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.
- [25] K. Fowler, M. Windschitl, and J. Richards. 2019. Exit Tickets. *The Science Teacher*, 86, 8, 18–26. DOI: 10.1080/00368555.2019.12293416.
- [26] 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.
- [27] 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.
- [28] 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.
- [29] 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.
- [30] 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.
- [31] 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.
- [32] 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.
- [33] 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.
- [34] 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.
- [35] 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.
- [36] C. Hu. 2013. The nature of software design and its teaching: an exposition. *ACM Inroads*, 4, 2, 62–72. DOI: 10.1145/2465085.2465103.
- [37] M. Jackson. 1995. The World and the Machine. In *International Conference on Software Engineering (ICSE '95)*, 283–292. DOI: 10.1145/225014.225041.
- [38] 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.
- [39] 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>.
- [40] 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.
- [41] D. Kahneman. 2003. Maps of Bounded Rationality: Psychology for Behavioral Economics. *American Economic Review*, 93, 5, 1449–1475. DOI: 10.1257/000282803322655392.

- [42] 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.
- [43] C. F. Kemmerer 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.
- [44] 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.
- [45] 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>.
- [46] 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.
- [47] 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.
- [48] 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.
- [49] C. Matthies. 2018. Scrum2Kanban: Integrating Kanban and Scrum in a University Software Engineering Capstone Course. In *International Workshop on Software Engineering Education for Millennials (SEEM ’18)*, 48–55. DOI: 10.1145/3194779.3194784.
- [50] 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.
- [51] 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.
- [52] 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.
- [53] G. Meszaros. 2007. *xUnit Test Patterns: Refactoring Test Code*. ISBN: 9780131495050. <http://xunitpatterns.com/Test%20Double.html>.
- [54] 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.
- [55] 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.
- [56] 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.
- [57] 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.
- [58] 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.
- [59] 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.
- [60] A. Radermacher and G. Wallia. 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.
- [61] G. Samarthyam, G. Suryanarayana, T. Sharma, and S. Gupta. 2013. MIDAS: A Design Quality Assessment Method for Industrial Software. In *International Conference on Software Engineering (ICSE-SEIP ’13)*, 911–920. DOI: 10.1109/ICSE.2013.6606640.
- [62] 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.
- [63] 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.
- [64] 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.
- [65] 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.
- [66] 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.
- [67] 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.
- [68] 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.
- [69] 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.
- [70] 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.
- [71] 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.
- [72] 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.
- [73] A. I. Wang. 2011. Extensive Evaluation of Using a Game Project in a Software Architecture Course. *ACM Trans. Comput. Educ.*, 11, 1, Article 5. DOI: 10.1145/1921607.1921612.
- [74] 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>.
- [75] B. Wu and A. I. Wang. [n. d.] A Guideline for Game Development-Based Learning: A Literature Review. *International Journal of Computer Games Technology*, 2012, 1, 103710. DOI: 10.1155/2012/103710.
- [76] 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.
- [77] 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.
- [78] 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.
- [79] 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.
- [80] 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.