Teaching Software Design in a Multi-Team Project Course

Anonymous Author(s)

Abstract

We taught a course.

Furthermore, the course teaches students how to design systems that are larger than a single developer can understand via the collaborative course project that gives students first experiences working on components that should integrate with components developed by other teams, which is one of the most important skills in industry, as almost all software that is professionally developed is developed in multiple teams.

CCS Concepts

• **Social and professional topics** → **Software engineering education**; *Model curricula*; • **Software and its engineering** → **Designing software**; *Collaboration in software development*; System modeling languages.

Keywords

Software Design, Software Engineering, Teaching, Education, ...

ACM Reference Format:

Anonymous Author(s). 2025. Teaching Software Design in a Multi-Team Project Course. In *Proceedings of IEEE Conference on Software Engineering Education and Training (CSEE&T).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Software design is an essential technical skill of professional software engineers [1, 3, 58]. 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 [55, 62, 64].

However, recent graduates often lack important software design skills, such as generating alternative designs, communicating them effectively, and working across large teams [7, 25, 48]. Multi-national, multi-institutional experiments have shown that the majority of graduating students in computer science lacks the skills of designing software systems [19, 37]. This gap between industry-needed software design competences and the skills of recent graduates has also been confirmed by surveys of software practitioners [1].

This skill gap motivates the need for a larger emphasis on the education of software design in universities [25, 26]. In many cases, software design is taught just as a small part of an overall software engineering course [1, 44, 56], 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 [7, 31, 45, 48]. Therefore, more dedicated software design education is needed.

Teaching software design is challenging, as instructors have to find the right balance between teaching a sufficient number of different design techniques and letting students gain enough practical experience with applying the taught design techniques in a realistic and sufficiently complex system [22, 31, 45, 57].

We believe that the most successful approach to learn software design is via first-hand experience in sufficiently complex software projects in conjunction with studying design principles. In small software projects students do not experience the issues that arise when no one person can fully understand the entire system[15, 16]. So, small software projects do not offer enough exposure to practice design techniques that are required for working on large software systems that are developed and iteratively maintained by multiple teams, such as interface descriptions, component responsibility assignments, and cross-team communication. Therefore, we argue shorter that teaching software design is most effectively done with a large-scale multi-team project, as almost all software that is professionally developed is developed in multiple teams.

In this paper, we present the comprehensive design and experience of a newly developed course that teaches 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 [4]. In the software project, students design, implement, and test a large-scale multi-service software system and describe their important design decisions in milestone reports that are inspired by Design Docs used by Google and other companies [13, 59, 65]. This course design also had the benefit of teaching teamwork, another under-taught software engineering skill [7, 48] that is in high demand in industry [1].

Our experience teaching this course has been mostly positive, while some unexpected observations motivate improvements of future iterations. We discuss actionable lessons learned from our course and recommendations for how to effectively teach software design. The teaching material will be made publicly available when the paper is published.

2 Related Work

[53]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CSEE&T, April 27– May 3, 2025, Ottawa, Ontario, Canada

© 2025 ACM.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.1 Multi-Team Courses

The teaching concept of using multiple interacting teams in software engineering education has been proposed and implemented before [12, 14, 15, 16, 18, 36, 39, 40].

Agile Processes: A course on scaling agile Scrum, which has been taught for multiple years at Hasso Plattner Institute in Germany, lets students build a web application with multiple interacting teams [12, 39, 40]. 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 [39]. Students receive the role of either scrum master, product owner, or developers while customers are simulated by the teaching team [40]. Students learn by making decisions about their development process autonomously and reflecting on their decisions after each sprint [12]. Our course has been partially inspired by the teaching methods used in this course. However, in contrast to multi-team courses on agile processes, the learning objectives of course focus on software design.

[15, 16]

Global Software Development: global software development courses [14, 18]

Stanford's ME310 [9]

2.2 Software Design Courses

As software design is one of the major activities in software engineering, courses on this topic have been taught over multiple decades [2, 5, 10, 24, 32, 33, 38, 45, 61].

[33]

[2]

[61]

[5]

[10, 38]

[32]

Software Architecture Course Survey [45] [24]

3 Course Design

The course is a full-semester elective aimed at graduate and undergraduate students at anonymized university in computer science and majors related to computer science. Prerequired 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 students the highly demanded skills 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 uses active-learning-style lectures and a full semester team project in which all teams collectively build and integrate a system composed of different services.

3.1 Learning Objectives (LOs)

The course has the following learning objectives:

LO 1 (Requirements)

Students should be able to: **Identify, describe, and prioritize relevant requirements for a given design problem.**

Rationale: Requirements analysis and specification are important skills for all software engineers [1, 3, 48, 52], as prioritized requirements are the main drivers of software design [31, 45].

LO 2 (Generate)

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

Rationale: Starting from requirements, design space exploration via constructive thinking and creative problem solving is the next required skill of software design [17, 41]. Since considering multiple design alternatives leads to better design [58], an important goal of a software design course is to teach the generation of multiple solutions.

LO 3 (Communicate)

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

Rationale: Modeling is a central aspect of design [17, 46] and required for collaborative design [34, 54].

LO 4 (Evaluate)

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

Rationale: Judging the quality of design options is essential to improve designs and assess requirements satisfaction [35].

LO 5 (Design Principles)

Students should be able 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.**

Rationale: Design principles provide guidance that help students generate and evaluate design options that generally lead to better designs [47, 49].

LO 6 (Process)

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

Rationale: Depending on the amount and types of risks in the domain, the software design process has to be adjusted [20].

LO 7 (Multi-Team)

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

Rationale: To build complex, large-scale software systems, skills of cross-team design and development are essential [6, 7, 48, 54].

to satisfy LO 3

3.2 Course Philosophy

Hands-on Experience in a Collaborative Project: Interesting design challenges usually arise in complex systems and can only be fully understood via first-hand experience. Therefore, teams in this course will work together on a single project to collaboratively build a complex system that involves interesting design decisions and discussions of interfaces between different teams. We will provide support structures to help you navigate potential cross-team issues.

Growth Mindset & Learning from Failures: When working on a complex software project, design issues and challenges will inevitably arise. In this course, we embrace a growth mindset and encourage students to see challenges in the project as an opportunity to learn based on the motto: "If you fail and you know why, you've succeeded. If you succeed and you don't know why, you've failed". We will focus on continuous, semester-long improvements, rather than achieving perfection in the first attempt.

Active Student Participation: Educational research shows that active student participation in lectures significantly improves learning outcomes [8, 21, 27, 29, 30]. So rather than just presenting design principles in a traditional lecture style, our lectures are based on real-world case studies from which we will derive design principles together.

Considering multiple design alternatives leads to better design [58]

Constructivism Learning Theory [4] increases meta-cognitive skills [43] and has a large, positive effect on academic achievements and retention[50].

3.3 Overview of Instructional Methods

(continued)

4 Multi-Team Project

While teamwork is one of the most important soft skills in professional software development [1], graduates in computer science often lack the skill of working in large teams [7, 48] or lack experience in working on large projects [48]. Therefore, a course with a multi-team project has to be designed to address challenges of cross-team communication [36].

To satisfy LO 7 (Multi-Team) and to give students the potentially their first experience with designing and developing a system large enough that nobody fully understands the system, we chose to

Why have a multi-team project?: In small team projects consisting of just four to six developers, students do not experience the issues that arise when no single person can fully understand the entire system [15, 16]. Furthermore, the comparatively low size and complexity of a project that is developed by a single team within a single semester would give students less practice with design techniques that are required for ensuring the correct composition of independently developed software components.

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.

4.1 Team Organization

4.2 Milestone Reports

Many companies, such as Google, use Design Docs to describe their important design decisions [13, 59, 65], which are very similar to the design documents students create in this course. The skill to write clear and convincing Design Docs is a strong predictor of the success of developers. in these companies, as they are often used as supporting documents in promotions.

4.3 Reducing Communication Overhead

As identified in previous work on multi-project software engineering courses [15, 16, 36], 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:

Based on our interpretation, some teams did not pick the ideal person to serve as the cross-team communicator. This interpretation is based on the observation that the cross-team communicators met for an important.

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
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	
	Final Exam	

Table 1: Course Schedule

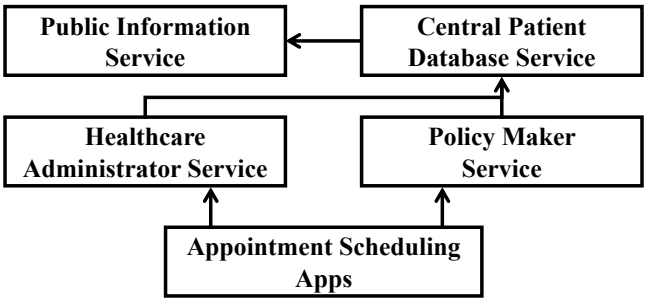

Figure 1: Component Diagram of the project system. Each team develops their own appointment scheduling app and one of the four services. Later all services have to integrate with all appointment scheduling apps and other services.

Lesson Learned 1 (Cross-Team Communicator) LO 7

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. Make the responsibilities and desired traits of a cross-team communicator more clear.

Recommendations:

- Tell students that the person who should represent their team should know what is going on in the team and should represent the team's needs and wishes.
-

4.4 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 create OpenAPI specifications that document the syntactic and semantics.

4.5 Test Doubles

For dependent-on-services that their service is using, students code *test doubles* (i.e., component that mimic the interface of a required service to control indirect inputs or verify indirect outputs [42]) based on the provided interface specification. Students

OpenAPI
Milestones[28]
Social bonus points
Team Contracts

5 Lecture Design

5.1 Real-World Case Studies

Case studies have been shown to be an effective teaching method in general software engineering education [23, 51, 60, 63] and have also been proposed for software design education in particular[11].

To illustrate the need for the design principles taught in the lectures (LO 5), we illustrated them based on real-world case studies of either prominent software failures or major success stories.

For example, in the lecture on *design for interoperability*, we used the **Global Distribution System (GDS)**, 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. After discussing techniques to achieve syntactic interoperability, we used the **Mars Climate Orbiter** case study as an example to illustrate the importance of semantic interoperability (a mix of imperial units and metric units caused the system to cash for a multi million dollar loss). Other real-world case studies used in the lectures are:

Netflix' Simian Army: In the *design for testability* lecture, we used the Simian Army by Netflix as a positive example for non-functional property testing of large-scale system.

Ariane 5 Rocket Launch Failure: In the *design with reuse* lecture, the violated assumption of 16 bit Horizontal velocity of the reused inertial reference unit from Ariane 4 in Ariane 5 is used to illustrate the importance of checking the assumptions of reused packages.

npm `left-pad`: In the *design with reuse* lecture, the suddenly unavailable, but highly dependent package 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 bug is used to motivated the importance of updating critical dependencies.

Twitter: In the *design for scalability* lecture, Twitter is used as a case study to demonstrate

EK: describe case study

Add student quotes

Outcomes: Based on the experience with our course, we can confirm that the effectiveness of case studies holds true for teaching software design.

In a mid-semester course feedback focus group session run by an outside consultant, all students agreed that the real-world case studies helped them learn. As instructors, we also noticed more student attention and more student participation when specifically discussing the case studies in lectures.

Lesson Learned 2 (Real-World Case Studies) LO 5

The use of real-world case studies of positive and negative examples for design principles has been effective at teaching design principles (LO 5).

Recommendations:

- 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

5.2 Teaching Software Design Principles using Constructivist

Lesson Learned 3 (Constructivism) LO 5

The use of constructivism for teaching design principles (LO 5) was successful.
Recommendations:

- Give students five to ten minutes of silent thinking and small group discussion before starting the whole class discussion.
- Soon after describing design principles, give students another problem to practice applying the principles in recitations or homework.

6 Homework Assignments

7 Experience

7.1 Observed Challenges of Multi-Team Projects

Letting multiple teams work together on a single system is not without any challenges.

8 Lessons Learned

[*Architecture Design and Coding Responsibilities are More Connected Than Students Realize*]

[*Code Templates for faster Coding*]

[*Students Struggle with Finding Multiple Design Alternatives*]

[*Students Ask for Concrete Steps Design Recipes*]

[*Interface Descriptions Helped Component Design*]

[*Students Enjoyed Breaking Services*]

[*Students Still Struggle with Abstractions*]

[*More Individual Assessments in Labs instead of Recitations*]

9 Conclusions

References

[1] Deniz Akdur. 2022. Analysis of software engineering skills gap in the industry. *ACM Trans. Comput. Educ.*, 23, 1, Article 16. DOI: 10.1145/3567837.

[2] Jocelyn 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.

[3] Nana Assyne, Hadi Ghanbari, and Mirja 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.

[4] Steve Olusegun Bada and Steve 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.

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

[6] Andrew Begel, Nachiappan Nagappan, Christopher Poile, and Lucas 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.

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

[8] Peter C Brown, Henry L Roediger III, and Mark A. McDaniel. 2014. *Make it stick: The Science of Successful Learning*. ISBN: 9780674729018.

[9] Tamara Carleton and Larry Leifer. 2009. Stanford's ME310 Course as an Evolution of Engineering Design. In *CIRP Design Conference – Competitive Design*. http://hdl.handle.net/1826/3648.

[10] 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.

[11] Chun Yong Chong, Eunsuk Kang, and Mary Shaw. 2023. Open design case study - a crowdsourcing effort to curate software design case studies. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, 23–28. DOI: 10.1109/ICSE-SEET58685.2023.00008.

[12] Thomas Kowark Christoph Matthies and Matthias 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.

[13] Marcus Ciolkowski, Oliver Laitenberger, and Stefan Biffl. 2003. Software Reviews: The State of the Practice. *IEEE Software*, 20, 6, 46–51. DOI: 10.1109/MS.2003.1241366.

[14] Tony Clear, Sarah Beecham, John Barr, Mats Daniels, Roger McDermott, Michael Oudshoorn, Airina Savickaite, and John 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.

[15] David Coppit. 2006. Implementing large projects in software engineering courses. *Computer Science Education*, 16, 1, 53–73. DOI: 10.1080/08993400600600443.

[16] David Coppit and Jennifer 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.

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

[18] Daniela Damian, Allyson Hadwin, and Ban 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.

[19] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Carol Zander. 2006. Can Graduating Students Design Software Systems? In *Technical Symposium on Computer Science Education* (SIGCSE '06), 403–407. DOI: 10.1145/1121341.1121468.

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

[21] Scott Freeman, Sarah L. Eddy, Miles McDonough, Michelle K. Smith, Nnadozie Okoroafor, Hannah Jordt, and Mary Pat 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.

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

[23] Kirti Garg and Vasudeva 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.

[24] David Garlan, Mary Shaw, Chris Okasaki, Curtis M. Scott, and Roy 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.

[25] Vahid Garousi, Görkem Giray, Eray Tüzün, Cagatay Catal, and Michael 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.

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

[27] Tyler Greer, Qiang Hao, Mengguo Jing, and Bradley Barnes. 2019. On the Effects of Active Learning Environments in Computing Education. In *Technical Symposium on Computer Science Education* (SIGCSE '19), 267–272. DOI: 10.1145/3287324.3287345.

[28] Randall 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.

[29] Qiang Hao, Bradley Barnes, Ewan Wright, and Eunjung 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.

[30] Susanna Hartikainen, Heta Rintala, Laura Pylväs, and Petri 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.

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

[32] Stan 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.

[33] C. W. Johnson and Ian 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.

[34] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. 2018. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software*, 35, 6, 40–47. DOI: 10.1109/MS.2018.290100920.

[35] Chris F. Kemerer and Mark 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*, 35, 4, 534–550. DOI: 10.1109/TSE.2009.27.

[36] Ze Shi Li, Nowshin Nawar Arony, Kezia Devathasan, and Daniela 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.

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

[38] Tomi Mannisto, Juha Savolainen, and Varvana Myllarniemi. 2008. Teaching Software Architecture Design. In *Working Conference on Software Architecture* (WICSA '08), 117–124. DOI: 10.1109/WICSA.2008.34.

[39] Christoph Matthies. 2018. Scrum2Kanban: Integrating Kanban and Scrum in a UniversitySoftware Engineering Capstone Course. In *International Workshop on Software Engineering Education for Millennials* (SEEM '18), 48–55. DOI: 10.1145/3194779.3194784.

[40] Christoph Matthies, Johannes Huegle, Tobias Dürschmid, and Ralf Teussner. 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.

[41] Mark E. McMurtrey, James P. Downey, Steven M. Zeltmann, and William 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.

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

[43] Raffaella 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.

[44] Sofia Ouhbi and Nuno Pombo. 2020. Software engineering education: challenges and perspectives. In *Global Engineering Education Conference* (EDUCON '20), 202–209. DOI: 10.1109/EDUCON45650.2020.9125353.

[45] Wilson Libardo Pantoja Yépez, Julio Ariel Hurtado Alegría, Ajay Bandi, and Arvind W. Kiwelekar. 2023. Training software architects suiting software industry needs: A literature review. *Education and Information Technologies*. DOI: 10.1007/s10639-023-12149-x.

[46] Marian Petre. 2009. Insights from Expert Software Design Practice. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (ESEC/FSE '09), 233–242. DOI: 10.1145/1595696.1595731.

[47] Reinhold Plösch, Johannes Bräuer, Christian Körner, and Matthias 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.

[48] Alex Radermacher and Gursimran 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.

[49] Ganesh Samarthyam, Girish Suryanarayana, Tushar Sharma, and Shrinath 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.

[50] Çetin Semerci and Veli 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.

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

[52] Mary Shaw, Jim Herbsleb, and Ipek Ozkaya. 2005. Deciding What to Design: Closing a Gap in Software Engineering Education. In *Proceedings of the 27th International Conference on Software Engineering* (ICSE '05), 607–608. DOI: 10.1145/1062455.1062563.

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

[54] Antony Tang, Maryam Razavian, Barbara Paech, and Tom-Michael 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.

[55] Antony Tang, Minh H. Tran, Jun Han, and Hans 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.

[56] Saara Tenhunen, Tomi Männistö, Matti Luukkainen, and Petri 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.

[57] Charles Thevathayan and Margaret Hamilton. 2017. Imparting software engineering design skills. In *Australasian Computing Education Conference* (ACE '17), 95–102. DOI: 10.1145/3013499.3013511.

[58] Dan Tofan, Matthias Galster, and Paris 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.

[59] Malte Ubl. 2020. Design Docs at Google. Retrieved 05/17/2024 from https://www.industrialempathy.com/posts/design-docs-at-google/.

[60] Vasudeva Varma and Kirti 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.

[61] Ian 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.

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

[63] Jianmin Zhang and Jian 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.

[64] Xuemei Zhang and Hoang 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.

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