Rapid #: -24244009

CROSS REF ID: **5927205690003996**

LENDER: **DCT (University of Dundee) :: Main Library**

BORROWER: **LF1 (University of Canberra) :: Main Library**

TYPE:	Book Chapter
BOOK TITLE:	Software Architecture
USER BOOK TITLE:	Software Architecture
CHAPTER TITLE:	A Better Way to Teach Software Architecture
BOOK AUTHOR:	Kazman, Rick ; Cai, Yuanfang ; Godfrey, Michael W.
EDITION:	
VOLUME:	
PUBLISHER:	Springer
YEAR:	2023
PAGES:	101 - 110
ISBN:	9783031368462
LCCN:	
OCLC #:	

Processed by RapidX: 3/19/2025 5:10:57 AM

A Better Way to Teach Software Architecture

Rick Kazman, Yuanfang Cai, Michael W. Godfrey, Cesare Pautasso, and Anna Liu

Abstract Software architecture education is a weak spot in many undergraduate programs in computer science and software engineering. While the concepts and practices used by software architects in industry are rich and varied, transferring this expertise into a university classroom has proved problematic. Bridging the gap between industry and academia requires ongoing, often heroic, effort. This is a "chicken and egg" problem: Because there is a lack of good teaching materials, architecture is seldom taught, and because it is seldom taught, there has been little incentive to create good materials. We would like to change that. Our goal is to establish guidelines for how software architecture practices should be taught—both technical and non-technical topics—and to suggest appropriate teaching methods to best prepare students to be software architects in practice.

R. Kazman (✉)
University of Hawaii, Honolulu, HI, USA
e-mail: kazman@hawaii.edu

Y. Cai
Drexel University, Philadelphia, PA, USA
e-mail: yfcai@drexel.edu

M. W. Godfrey
University of Waterloo, Waterloo, ON, Canada
e-mail: migod@uwaterloo.ca

C. Pautasso
University of Lugano, Lugano, Switzerland
e-mail: c.pautasso@ieee.org

A. Liu
Amazon Web Services, Seattle, WA, USA
e-mail: liuanna@amazon.com

P. Pelliccione et al. (eds.), *Software Architecture*,
https://doi.org/10.1007/978-3-031-36847-9_6

Introduction

Software architecture concerns the design of a software-intensive system at the highest level of abstraction. It is typically first addressed early on in development, as the decisions made at this level will pervade the design of the rest of the system and are thus the most difficult and costly to change later on. Of course, change is inevitable in most technical endeavors, and the architecture will also have to evolve as knowledge about the system and its operational context become clearer. However, understanding the costs and benefits of the architectural-level decisions can help to mitigate the overall risk of developing the software system.

The chosen architecture of a software system will exert strong forces on the system's quality attributes, such as modifiability, extensibility, availability, performance, scalability, sustainability, and security. It also determines the division of functionality in the system and consequently the structure of the development team. A large-scale system without a consciously developed and managed architecture can be disastrous, resulting in unhappy stakeholders—because their quality attribute goals are not being met—and an ever-increasing load of technical debt as the system evolves in an unprincipled and ad hoc way.

Software architectural expertise is vital to practicing software developers, but the topic is seldom taught in undergraduate computer science (CS) or software engineering (SE) programs. More often, it is found within MSc-level curricula [12], if it is found at all. The typical undergraduate CS/SE student is often a "good" programmer, but has little experience with developing or maintaining systems larger than a few hundred lines of code. It can be difficult to motivate students to care about the high-level design of the system—they don't know what they don't know—and it can be difficult to convey the importance of architecture and the risks of not paying attention to it, especially when dealing with large and long-lasting systems involving multiple developers and stakeholders. While some MSc-level students might have industry experience, this is not guaranteed. So anyone who is teaching software architecture to students at either level must grapple with their lack of real-world experience.

Furthermore, software architecture design is more of an engineering discipline than a mathematical one. Design is a "wicked" problem: there are rarely right/wrong answers, but rather, there are better/worse approaches, each with their own set of contextual tradeoffs. There are always many ways of designing an architecture and many ways of designing satisfactory solutions. However, there are many more ways of designing unsatisfactory solutions—it is an inherently complex problem, requiring experience and domain knowledge to address.

Complicating the problem further is that undergraduate CS/SE education is often taught in conceptual "silos", with classes and projects that are isolated from each other. For this reason, students have become accustomed to wrestling with problems of limited scope and scale; problems whose solutions have short timeframes; problems that can be solved in isolation, ignoring pre-existing solutions; and

problems with well-defined, stable requirements. None of these are typical features of real-world software systems.

A successful software architect needs to have a broad range of both technical and non-technical skills. However, there is often little attention paid to so-called "soft" skills in software engineering education. Our vision is that a course in software architecture must emphasize soft skills such as arbitration, brainstorming, facilitation, negotiation, listening and eliciting, delivering clear and convincing presentations, receiving and providing feedback [10], translating between technical and business worlds, leadership, decision-making, and risk analysis. Developing these skills will empower software architects to devise and communicate high-quality large-scale system designs.

To achieve all of these goals, we may need to expand on traditional thinking and go beyond the usual scope of classroom-based education, engaging in (or at least simulating the engagement in) large-scale real-world industrial or open-source projects. Only in these settings, students will experience how to work under a largely unknown context.

Teaching Resources

For many educators, finding a supporting textbook is a must before embarking on the design of a new course. Fortunately, there are already several good options for software architecture:

- *Software Architecture in Practice* [2]
- *Patterns of Software Architecture* [3]
- *Software Architecture: Foundations, Theory, and Practice* [15]
- *Just Enough Software Architecture: A Risk-Driven Approach* [8]
- *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives* [16]
- *Design It! From Programmer to Software Architect* [11]
- *Software Architecture: Visual Lecture Notes* [14]

Several of these textbooks provide supporting resources for instructors including PowerPoint slides, discussion questions, and classroom exercises; some also provide access to online pedagogical materials. And many of these resources have been written by practitioners, which lends an air of authority to their recommendations of best practices.

We feel that these example textbooks are of high quality and broad scope, addressing both the technical and non-technical topics mentioned above. While they do not solve our problem on their own, they can support the delivery of a well-designed course in software architecture.

But textbooks alone do not solve all of the challenges of software architecture education. To truly grasp a complex domain, the student needs a significant amount of experiential learning. They need to grapple with complex, non-obvious problems.

They need to grapple with uncertainty, with tradeoffs, with risk, and with the demands of a wide group of stakeholders. These kinds of experiences are difficult, but not impossible, to inject into the classroom experience.

Industry Talent Needs and Architecture Realities

There is already a significant lack of software engineering talent to fulfill industry's existing human resource needs. There is an even thinner pipeline of experienced and well-trained software architects to lead corporate digital transformation projects. The global market intelligence company IDC reports that the worldwide shortage of full-time software developers will increase from 1.4 million in 2021 to 4.0 million in 2025 [7]. This talent shortage problem of software developers is expected to grow worse, as industries continue to rely on experienced developers for delivering solutions that not only digitize business processes and operations but also enable innovative customer service and business models.

The talent shortage of skilled software architects is even more dire. The pipeline of experienced developers who can transition to software architect roles is thin to begin with, and the road to becoming an effective software architect requires significant large-scale system portfolio experience over many years. While software developer resources can often be successfully obtained as needed through an inter-organization or even international sourcing model, the role of software architect typically requires deep organizational and cultural insights to be effective and successful in the long term. That is, software architects cannot simply be "magicked up" on demand.

To further compound this talent pipeline issue, the plethora of digital transformation projects is rapidly growing across all industries. Organizations rely on the unique technical and interpersonal leadership of strong software architects to design, plan, navigate, communicate, and manage these complex projects, to guide the enterprise through their digital transformation journeys.

Industrial software projects are rarely greenfield projects. New CS/SE graduates expecting to be designing large-scale software systems from scratch may, unfortunately, have unrealistic expectations. This is not their fault; the problem lies mainly in how they have been educated, which focuses more on software creation than integration, maintenance, and evolution. An industrial software architect's day-to-day job can include some unglamorous tasks: ad hoc maintenance of brittle, ill-documented systems designed by employees who left the organizations long ago; software evolution planning, including strategies to retire legacy systems; and evaluation and acquisition of new capabilities to incorporate into the existing spaghetti-like enterprise architecture. This reality is at some distance from the often idealized and waterfall-like lifecycle descriptions of large-scale software system examples in some software engineering textbooks and curricula. This situation can lead to disappointment and sometimes even attrition of new computer science graduate talents landing in the deep end of enterprise architectural realities.

Duties, Skills, and Knowledge

An effective software architect requires a large skill set—both technical and non-technical—and a lot of experience. An architect needs to know a lot—about technologies, about organizations, about business, and about people. And an architect has many duties. Much of this can be taught, and that is the good news and the reason for hope in this chapter. Even the shyest, most introverted person can learn to become a better communicator, a better negotiator, and a better leader. It is beyond the scope of this chapter, and of the expertise of its authors, to speculate how much of an architect's duties, skills, and knowledge can be learned, but we take it as a given that everyone can get better in these three areas.

Let us look at them in more detail.

Knowledge

Knowledge is perhaps the first area that comes to mind when envisioning what software architecture education should require, and it is likely the easiest area to teach. This aspect of being a good software architect is more similar to the rest of software engineering education.

Some of the most important knowledge areas include:

- Technology knowledge: keeping abreast of state-of-the-art technologies, prototyping and experimentation, making appropriate tools, and framework selection decisions
- Evaluation methods knowledge: architecture analysis, performance testing, reliability evaluation, security and assurance methods, empirical methods, and evaluation criteria definition
- Technology-independent knowledge: deep understanding of patterns, design principles, architectural styles, large-scale distributed systems issues, and foundational design knowledge
- Fundamental knowledge: understanding what problems are inherent in the domain and which ones are transient and hence more amenable to non-architectural solutions
- Meta-knowledge: learning how to learn and be a lifelong learner. Awareness of one's own limits. Ability to collect domain-specific knowledge

A good software architecture education program should not just teach these knowledge areas but also put them to use in practical contexts [9]. The more abstract the knowledge, the more that its practical application is critical to understanding. For example, one study [4] showed that even after taking a semester-long course in software design and architecture, a majority of students still made fundamental design errors and were unaware that these violated the design principles that they had been taught. A project-based learning approach may help mitigate these issues [17].

Skills

Skills may be the most difficult area to teach, as they are often more challenging and less familiar to software educators. Consider a skill in another domain, say playing the piano. No one would expect that a student could learn to be an effective pianist from observing a lecture or two and doing an assignment. It takes consistent effort and practice to hone a skill. A software architecture course can at least be a starting point for this journey.

Here are some of the most important skills for an architect to acquire and master:

- Communication: architects must be able to communicate effectively, to sell their architectural vision both to management and to developers.
- Negotiation: architects often need to negotiate resources, requirements, priorities, and constraints. These can be the crux of key tradeoff decisions [5].
- Leadership: an architect must be able to rally people around the architectural vision. This is, perhaps, the most difficult skill to teach.
- Workload management: architects are often extremely busy, given the depth and breadth of their work roles.
- Flexibility/adaptability: architects must be able to react to changes. These could be changes in requirements, in strategic direction, in the external world (e.g., the technical ecosystem in which we all live and operate), in the workforce, in funding, in deadlines, and in many others as well. Any architect who is inflexible will fail sooner or later. This is, like leadership, a very challenging skill to teach and to acquire.

Duties

Architects have a large array of duties, which may vary according to the specific working context. In a small project, the architect may perform all of the duties listed below. In a large project, the architect may have direct responsibility for only a few key duties, but should have knowledge of and insight into the others.

- Creating, analyzing, and documenting an architecture: these are the most obvious duties of an architect. Who is an architect, if not the one doing highest-level design? But analysis duties may be done by outsiders (outside the team, outside the organization, outside the company), and documentation may not be done at all or may be done by everyone. An architect must be aware of and at least involved with all of these duties.
- Tool and technology selection: one of the most important decisions an architect can make in design is the choice of tools and technologies. This selection process is often complex, requiring the architect to be aware of many variables (cost, complexity, familiarity of the team, quality attributes, likely survivability of the technology, licensing, etc.).

- Leading or participating in design review and conformance exercises: if you are creating an architecture for a long-lived project, then it will almost certainly evolve. And if it evolves, there is a strong likelihood that it will diverge from its original intent and original documentation. This can often undermine the quality of the architecture. To guard against this, an architect should participate in code and design reviews, with an eye to spotting any decisions that may undermine or evolve the architecture.
- Requirements elicitation and management: while an architect may have little say in the functional requirements for a product, she should have a strong say in the elicitation, baselining, and negotiation of quality attribute requirements. These are among the strongest design drivers, so it is in the architect's self-interest and in the best interest of the project to manage these effectively.
- Managing and designing for quality assurance: when we think about testing, we typically focus on testing the system-level user-facing functionality of the system. But an architect is also responsible for the quality attributes of the system: its maintainability, security, performance, availability, and so forth. To ensure that these are met when the product is created and as it evolves, such non-functional requirements must be explicitly tested for. Thus, one of the architect's duties is to ensure that the architecture and its environment make such testing possible.

As we can see from the above, an architect's duties, skills, and knowledge form a comprehensive list. How can we hope to teach all of this to students? Perhaps we cannot. But we can prepare the ground so that our students understand the many dimensions of architecture and being an architect, and we can prepare them to be lifelong learners.

Challenges and Opportunities

An ample supply of well-trained software architects who are skilled at delivering robust software solutions for the long term is an established need from industry. The existence and maturity of a large number of open-source projects, such as Android, Chrome, and Hadoop, and the research advances that can be used to dissect the architecture of these systems provide a number of opportunities to better educate the much needed cohort of software architects for the future. Below are some opportunities that we see as being most critical and most promising.

1. Conveying real-world settings: Using longer-term projects opens up the opportunity to study the impact of changing requirements and experience maintenance and architecture refactoring [1]. The revision history and the code repository of these open-source projects provide ample examples of realistic settings, change requests, and bug fixing requests. Having students immerse themselves in these projects and asking them to recover the architecture so that they can claim a bug to fix or implement a new feature will put them into real industrial settings with direct impact.

2. Dealing with teamwork: Working in pairs; working in larger groups; and adding in new project members or swapping them along the way (e.g., [18]). Most of these prominent open-source projects are supported by companies like Google, Microsoft, Intel, Red Hat, Meta, Amazon Web Services, SAP, IBM, and many others. Having students work on these projects provides an opportunity for them to work in large groups and communicate with both open-source and industrial practitioners. Still, using open-source projects for undergraduate student education should be done limiting the cost imposed on the project maintainers and ensuring a win-win outcome benefiting both students and the projects to which they contribute to.
3. Design thinking: Architectural decision-making, creating and pruning alternatives, and exploring quality attribute tradeoffs. Numerous design decisions and tradeoffs are weaved into these open-source projects. Decisions and tactics used to ensure security, performance, and availability are indispensable for all these projects. We could leverage recent research tools to extract these decisions from source code and using these extracted source code as teaching examples.
4. New content organization: For each quality attribute and corresponding tactics, we could collect their design and implementations in these open-source projects and organize these materials based on quality attribute, viewpoint, and software development lifecycle. As a result, we could use these realistic examples to teach important architecture concepts.
5. Better use of notations: There are many modeling notations, such as architectural description languages or UML, that are seldom used in practice. It is useful to teach students to record and model their design decisions using architectural decision records. This can bring more precise thinking and analysis to the architecture design and analysis process as well as clarity, precision, and accuracy in the way it is represented. But instead of trying to model every aspect of a software architecture—which is very costly and almost never done in industry—for each project, we could extract a partial high-level model that is relevant to only one or a few key decisions or quality attributes. For example, we could extract and document a high-level component diagram related to security only. In this way, we can get the pedagogical benefits of models while avoiding the complexity and overhead of most models that prevent them from being widely adopted.
6. Grading/evaluation: It remains challenging to assess student learning on software architecture beyond superficial recall of definitions and toy design problems to be solved in an exam setting. It is very unlikely that real-world architects would need to go from zero knowledge about a domain to a complete specification of an architectural model in a few hours' time.

   To lower the teaching load, to remove subjectivity, and to provide regular feedback to students, we can and should automate design analysis [4]. Using recently published research tools, it becomes possible to automatically determine if students' homework assignments have followed proper design principles or whether they are well-modularized, so that their homework can be graded not only based on functionality but also based on design quality.

7. New teaching strategies: We can employ games (e.g., [6]), simulations, architectural katas [13], and live-through exercises, to try to give students the flavor of real-world architecting duties while keeping the overhead manageable for both students and educators.

Next Steps

This chapter serves as a call to action. If we are to improve the state of the art in teaching software architecture, then we need to get away from our reliance on "heroic" individual efforts, which is neither sustainable nor scalable. Our community of researchers and educators needs to come together, so that we can better educate the next generation of architects.

That means we need better example curricula, teaching materials, and tooling so that people can teach architecture as they would teach other courses, with more rigor and with more repeatable outcomes. We need to create and share resources such as curriculum guidelines, checklists, and—perhaps most importantly—large-scale teaching examples with curated datasets that connect business goals, requirements, architecture, documentation, and code. And these datasets should ideally include some history, so that we can reason about and teach how architectural artifacts and decisions evolve over time.

References

1. Bærbak Christensen, H.: Teaching microservice architecture using devops—an experience report. In: European Conference on Software Architecture, pp. 117–130. Springer, Berlin (2022)
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 4 edn. Addison-Wesley, Reading (2021)
3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture Volume 1: A System of Patterns. Wiley, New York (1996)
4. Cai, Y., Kazman, R., Jaspan, C., Aldrich, J.: Introducing tool-supported architecture review into software design education. In: Proceedings of the Conference on Software Engineering Education and Training (CSEE&T) (2013)
5. Capilla, R., Zimmermann, O., Carrillo, C., Astudillo, H.: Teaching students software architecture decision making. In: European Conference on Software Architecture, pp. 231–246. Springer, Berlin (2020)
6. Cervantes, H., Haziyev, S., Hrytsay, O., Kazman, R.: Smart decisions: an architectural design game. In: Proceedings of the International Conference on Software Engineering (ICSE) (2016)
7. Dayaratna, A.: Quantifying the worldwide shortage of full-time developers. In: IDC Market perspective (2021)
8. Fairbanks, G.: Just Enough Software Architecture: A Risk Driven Approach. Marshall and Brainerd, New York (2010)

9. Galster, M., Angelov, S.: What makes teaching software architecture difficult? In: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), pp. 356–359 (2016)
10. Hazzan, O.: The reflective practitioner perspective in software engineering education. J. Syst. Softw. **63**(3), 161–171 (2002)
11. Keeling, M.: Design It! From Programmer to Software Architect. Pragmatic Bookshelf, New York (2017)
12. Lago, P., Van Vliet, H.: Teaching a course on software architecture. In: Proceedings of the 18th Conference on Software Engineering Education and Training (CSEET'05), pp. 35–42. IEEE, New York (2005)
13. Neward, T.: Architectural katas (2012). https://www.architecturalkatas.com
14. Pautasso, C.: Software Architecture: Visual Lecture Notes. LeanPub, New York (2020)
15. Taylor, R.N., Nenad Medvidovic, E.D.: Software Architecture: Foundations, Theory, and Practice. Wiley, New York (2009)
16. Rozanski, N., Woods, E.: Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, Reading (2011)
17. Rupakheti, C.R., Chenoweth, S.V.: Teaching software architecture to undergraduate students: an experience report. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, pp. 445–454. IEEE, New York (2015)
18. Van Deursen, A., Aniche, M., Aué, J., Slag, R., De Jong, M., Nederlof, A., Bouwers, E.: A collaborative approach to teaching software architecture. In: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, pp. 591–596 (2017)