Teaching Statement

ROBERT DICKERSON

I take a pragmatic approach to teaching informed by years of industry experience. My goal is to help students build strong theoretical foundations motivated by practical applications. Drawing on my background as a software engineer, I use real-world examples to foster active discussions and student engagement. When lecturing, I work to impart both an understanding of a topic or technique and how it fits into a larger goal or agenda. I believe effective computer scientists require strong communication skills, an ability to work collaboratively, and an understanding of professional ethics, and I will work to integrate the development of these skills into every course I teach.

TEACHING APPROACH

Lectures should build toward concrete goals. I believe students engage with and retain lecture content more readily when actively following along with a line of thought building toward a goal. Whenever possible, I organize lectures as a narrative that culminates in accomplishing a specific goal or answering a specific question. For example, when giving a lecture on property-based testing in *CS 456 Programming Languages*, I began by stating that maximizing test coverage was the central goal of the lecture. We began with basic unit testing and then attempted to cover increasing amounts of the input space, with each approach building on the shortcomings of the previous one, until property-based testing naturally emerged. At each step of this process, I wanted students to understand how the current approach brought us closer to our goal, to actively recognize the shortcomings of that approach, and to anticipate how we might address those shortcomings in our next attempt. Asking students to help advance the lecture's narrative (for example, by asking students for suggestions on how to cover more state space) not only provides a useful check that the lecture material is being understood, but makes for a more lively and memorable overall discussion. Ensuring the thread of a lecture is always building toward specific goals naturally leads to this narrative flow.

Lesson plans should incorporate practical knowledge when possible. I believe a strong computer science education should not only give students a solid theoretical foundation in important topics across the field, but also equip students to apply that knowledge after graduation. When mentoring recent computer science graduates during my career as a software engineer, I found that new hires who were able to "hit the ground running" with knowledge of modern version control systems, continuous deployment practices, and popular development frameworks enjoyed both a noticeable confidence boost and a positive starting reputation among their peers. In addition to my personal experience, I maintain contact with friends and colleagues in the software industry to understand its shifting trends and demands; this helps me structure lessons in ways that will serve the majority of computer science undergraduates who plan to become professional software engineers. For example, a survey of popular testing methodologies was not the primary learning objective of the *CS 456 Programming Languages* lecture on property-based testing mentioned above. However, building toward property-based testing through these testing methodologies was not only pedagogically useful, but leveraged an opportunity to expose students to techniques they were likely to encounter in their careers. (In fact, I received a heartfelt thank-you after this lecture from a student who had been feeling unsure of the software testing landscape he might encounter in an upcoming internship.)

Students should be equipped to effectively and ethically apply their knowledge. Success in any computer science career path is about more than technical acumen; "softer" skills like the ability to clearly communicate complex ideas and the ability to work well within (or even lead) a team play a

significant role. The work outside of "hard" technical skills I put into my own liberal arts education has served me well, and I will endeavor to develop these attributes in my students. As a TA, I have worked closely with critical writing assignments and long-term group projects designed to instruct students on contextualizing and effectively applying their knowledge, and I plan on including these elements in my own teaching. In addition to understanding how to work well within a team, I believe it is important for computer science students to understand how the systems they build fit into the context of the larger world around them. With so many aspects of society tied in some way to a software system, we place significant trust in the people creating these systems, and computer science educators have an obligation to produce graduates who understand the scope of the role their work will play in the world. Purdue's software engineering course covers the ACM and IEEE codes of ethics, which I believe is a great way to introduce students to these issues; I plan on including these topics when teaching similar courses.

TEACHING EXPERIENCE

Before starting my graduate studies, I served as a Java instructor at EnCircle, a nonprofit organization in Missouri dedicated to teaching career skills to young adults on the autism spectrum. Designing and teaching a Java course alongside Encircle's experts on neurodivergent education was an eye-opening experience, providing me an opportunity to closely examine my assumptions on how students experience my instruction. This theme later resurfaced for me while attending a series of workshops at Purdue on creating inclusive learning environments, in part by examining how instruction can be perceived through the lens of background cultural assumptions. I strive to continually examine the assumptions I make about my students, and I find crafting lessons which retain impact across a wide range of student perspectives to be an enjoyable challenge.

At Purdue, I have served as a teaching assistant for a diverse range of courses:

Large undergraduate course. I have twice served as head teaching assistant for *CS 307 Software Engineering*, an undergraduate course with an enrollment of around 200 students. This course is centered around semester-long projects which are chosen, designed, and implemented by a group of 4 – 6 students with the help of a TA assigned to coordinate that group. My duties as head TA included overseeing a team of around 10 graduate and undergraduate TAs, managing group assignments and grading rotations, serving as a project coordinator for several teams, scheduling project pitch meetings and final presentations, and generally aiding course logistics. In addition, I gave several guest lectures on software testing and project management. I enjoyed the opportunity this course afforded me to directly apply my career software engineering experience while working with small groups of students. As an example, when one of the groups I coordinated told me they wanted to use REST in their project but did not know much about the approach, not only was I able to give an impromptu lesson to some motivated students, I was also able to discuss how the approach shows up in a practical setting. I found sharing this industry experience with aspiring software engineers especially rewarding.

Mid-sized undergraduate elective. I have also been a teaching assistant for *CS 456 Programming Languages*, an elective course for upper-division undergraduates with an enrollment of around 30 students. In addition to grading and holding office hours, I gave guest lectures on property-based testing and embedded DSLs. The course culminated in a tournament loosely based on an ICFP contest[1], with an open-ended requirement to develop programming language tooling to help craft a tournament entry. With this project, students were constrained in the shape of output their project could produce, but had few constraints on how they built the tooling to get there. I found this an

[1]https://en.wikipedia.org/wiki/ICFP_Programming_Contest

interesting contrast to my experience in *CS 307 Software Engineering*, where students have few constraints on what their project actually does, but are graded on following a rigorous process to build it. I see the advantage of latter approach as making evaluation more objective and well-defined, but at the cost of causing students to focus mostly on following the process and hitting predefined checkpoints. (This is clearly an appropriate tradeoff for a course on software engineering, which is focused on process.) I see the advantage of the former approach as allowing more creative freedom in how students attack the problem, although it complicates evaluation for the course staff, as grading depends on more subjective questions about how well a team applied principles from the course in their approach. Ultimately, I believe different approaches to constraining student projects are appropriate in different contexts, and these experiences serve as valuable data points.

Mid-sized graduate elective. Finally, I have served as a teaching assistant for *CS 560 Reasoning About Programs*, an elective course for graduate students with an enrollment of around 30 students. This course was undergoing a significant revision when I TAed, and as part of that process I designed new homework assignments from the ground up targeting the updated material and course objectives. Assignments included both pen-and-paper and coding components, with evaluation of coding portions automated via grading scripts. I also created and delivered a guest lecture on congruence closures. This lecture was a memorable challenge, as it covered a theoretically dense topic that culminated in a technically sophisticated, but somewhat dry, algorithm with several subcomponents; I did not want to end up reciting this algorithm to a room of checked-out students. The approach I took was to frame the lecture in terms of a motivating challenge: write a decision procedure using only a handful of available axioms. This challenge was, in turn, motivated by a larger goal of automatically deciding satisfiability modulo theories established in previous lectures. I then introduced the algorithm in pieces that cleared successive hurdles to having our decision procedure in hand. By tying the lecture's objective to a larger course goal, I received some good questions at the top of the lecture on the problem setup. By introducing the approach in pieces, I was able to prompt students to drive the lecture forward by identifying gaps in our solution (e.g., I could ask if a set of congruence classes in a running example was correct to have a student point out that certain classes had not been properly merged). I found the net result to be a lively and engaging lecture.

MENTORING EXPERIENCE

I find the personal connection of mentorship particularly rewarding. While working in industry, I mentored several new computer science graduates who were just entering the software engineering profession. When in an engineering manager role, I provided project and career guidance to my reports. At Purdue, I worked directly with an undergraduate computer science student on one of my research projects; this project resulted in a publication[2]. While initially unsure of how the mentoring approach I had developed in industry would carry over to an academic setting, my experience has been that the core ideas behind building a rapport, fostering confidence, and providing actionable guidance are the same. Since this student was working on our verification tool, I made sure to get them to their first code commit quickly, a confidence-building technique I carried over directly from my industry experience.

I have also served as a mentor to incoming computer science graduate students through Purdue's Computer Science Graduate Student Association mentorship program. This program paired me with three incoming graduate students, and was focused on orienting these students to life as a

[2]Dickerson, R., Ye, Q., Zhang, M. K., & Delaware, B. (2022, November). RHLE: modular deductive verification of relational $\forall\exists$ properties. In Asian Symposium on Programming Languages and Systems (pp. 67-87). Cham: Springer Nature Switzerland.

graduate student and helping them identify resources available to them as they began their studies at Purdue.

TEACHING INTERESTS

As my research is generally in programming languages and formal methods, I am well-positioned to teach courses on *Systems Programming*, *Compilers*, and *Discrete Math*. My industry background aligns well with courses on *Software Engineering*, preferably in a course centered around semester-long team projects. If the opportunity exists, I am interested in teaching a *Programming Languages* course aimed at junior and senior level students. This course would explore formal foundations of programming language syntax and semantics, cover basic type theory, and expose students to functional programming paradigms. In the interest of producing graduates who can "hit the ground running" in industry, I am also interested in teaching or developing a small (1-hour) course on a topic such as *Modern Software Engineering Practices*, a survey of best practices for operational concerns like automated deployment, containerization, and service monitoring, or *Programming with Linux*, an introduction to the Linux command line, file system, and basic scripting techniques useful for working programmers.