Research Statement

Ayaan M. Kazerouni

My research is at the intersection of software engineering and computing education. I identify the effective and ineffective software development practices of students, using quantitative and qualitative empirical software engineering methods like IDE-log analysis, software repository mining, and semi-structured interviews. In education contexts, assessment of software currently tends to focus on qualities like correctness, code coverage from test suites, and code style. Little attention or tooling has been developed to assess the software development process. The primary vision of my research is to build software evaluation techniques that focus not only on the final product, but also on the process undertaken to produce it. I publish my work at principal computing education venues, including the Technical Symposium of the Special Interest Group on Computer Science Education (**SIGCSE**), the International Computing Education Research (**ICER**) conference, and the conference on Innovation and Technology in Computer Science Education (**ITiCSE**). I have been honored with a SIGCSE Best Paper Award and 1st place in the SIGCSE Student Research Competition.

Current and Past Work

My research is driven by the belief that a stronger pedagogical focus on the software development process will better equip CS graduates for roles in industry. Graduating CS students have faced well-documented difficulties upon entering the workforce, with reports of a gap between what they learn and what is expected of them in industry. Perhaps an early manifestation of this is the challenges faced by intermediate CS students while trying to complete large and complex programming projects. For example, the Data Structures & Algorithms course at Virginia Tech requires developing projects over a three to four week life-cycle, and it is common to see 25–30% of students drop the course. Even among students who continue in the course, there is high variance in their success rates. Effective feedback about the programming process could improve students' self-awareness about their programming habits, promoting metacognition and self-regulation, which have been linked to numerous benefits such as increased productivity and improved project outcomes [1].

The primary focus of my research has been to model the programming process undertaken by students, with the goal of providing them with feedback about their process during development. I use data from numerous sources to formulate this feedback, including click-stream data and periodic code snapshots from the students' integrated development environment (IDE), and information about correctness and code quality from our automated assessment tool, Web-CAT. I have also conducted numerous in-depth interviews with students to gain an understanding of the more complex challenges they face while working toward project completion. Using these data, I have worked on assessing students' development in terms of time management, test writing, test quality, and other "self-checking" behaviors like running the program locally or submitting to an oracle of instructor-written test cases. This research is relevant to software engineering researchers and practitioners; the student developers I study are typically only two or three semesters removed from professionals entering the industry.

Time management on programming projects. Difficulties faced by students while working on programming projects have been linked to poor time management practices. Using data from sources described above, I modelled procrastination as a measure of how early and often students worked on programming projects [2]. I found that when students worked earlier and more often, they tended to produce more correct programs, with no difference in the total amount of time spent on the project [3]. This hints at the intuitive notion that procrastination negatively impacts the quality of work. Additionally, this measure has been validated through qualitative interviews. Students indicated that it matched with their own perceptions of how they approached specific programming assignments [2]. I am currently working on methods for early identification of students who are procrastinating, to facilitate just-in-time interventions that that could nudge them into action.

Incremental software testing. Numerous challenges hinder the pedagogy of software testing. These include a lack of focus on process, a lack of consensus in the software engineering community on what that process should look like, and widespread use of weak test adequacy criteria to drive assessments. We cannot give students effective feedback without validated assessments of their software testing process and the thoroughness of their software tests. I have

worked to address these challenges by developing and evaluating assessments of students' test writing practices, and building on prior research in mutation analysis to facilitate rapid incremental feedback on the thoroughness of their software tests.

Students do not practice software testing, despite the fact that it has been linked to improved project outcomes. Many CS courses require students to submit software tests along with their solutions, but data from IDE activity and student interviews indicate that tests tend to be (grudgingly) written toward the end of the project life-cycle. These tests do not benefit the students, and writing them at this late stage serves only to fulfill what is perceived as an arbitrary requirement. My preliminary investigations using IDE click-stream data indicated that on projects where students wrote tests earlier in the project life-cycle, they ended up with better outcomes than when those same students delayed testing until later in other projects [3]. Investigating further, I developed measures of incremental testing practices [4]. I found that students did not write tests continuously as they worked on solutions, even though eventual project outcomes improved with more continuous engagement with testing. These measures of incremental testing are naturally amenable to intermediate feedback. They represent an important step toward a continuous feedback loop that nudges students towards continuous engagement with software testing.

Software test quality. In addition to software testing *process*, there is a need for more effective methods of evaluating the *quality* of student-written tests. Code coverage measures — although frequently used in education and industry — have well-documented deficiencies when it comes to measuring test adequacy. There is often a disconnect between the quality of a student's test suite (as measured by code coverage) and its "actual" quality (e.g., its defect detection capability). This results in feedback that is not pedagogically valuable to students. I am currently working on methods to incorporate *mutation analysis*, a stronger test adequacy criterion, into our automated assessment system. Specifically, I am working on reducing its considerable run-time cost and evaluating its pedagogical value to students.

Collaborations. I have worked on *CodeWorkout*, a system for novices to practice programming exercises, and used it to study the impact of voluntary programming practice on exam performance [5]. I have made significant efforts to aid in its adoption both within and outside Virginia Tech by ensuring its interoperability with the Canvas learning management system [6], the OpenDSA e-textbook system [7], and the MasteryGrids [8] and Realizeit [9] learning and analytics platforms. I also applied my empirical software engineering experience to collaborate on a paper investigating the generalizability of regular expressions, published at **ASE 2019** [10].

Research Agenda

Short-term: Formative assessment of the programming process. Students have indicated in interviews that they know the benefits of incremental development and testing, but don't put them into practice. Perhaps as a consequence of this, students face reduced project success in the classroom, and struggles upon entering the industry. In the short term, I plan to address this by pursuing the following research:

Understanding why students find it difficult to self-regulate their programming behaviors. What makes good programming process difficult to learn? Once good process is "learned", what makes it difficult to consistently put into practice? I have already learned *how* students work toward project completion. Next, we must understand *why* students exhibit the behaviors that they do. What are the social, cognitive, and situational factors that affect their programming behaviors? How can we design tooling and instruction to ameliorate their negative impacts and encourage the positive ones?

Designing programming process feedback. Assessing students' programming process only addresses part of the problem. In order to change students' programming behaviors, assessments must be translated into feedback. For example, I developed measures of working early and often and incremental software testing. How can these assessments be used to guide students toward better programming practices? My lab will develop and evaluate feedback delivery mechanisms grounded in educational design principles. I will collaborate with researchers in educational psychology and user-interface design to aid in this effort.

Explicitly emphasizing reflection during and about the software development process. Metacognition, self-regulation, and reflection have been linked to improved program comprehension, problem-solving ability, and programming

productivity in both expert and novice programmers [11, 12, 1]. How can we design instructional strategies to encourage reflection about one's software development process? How might this impact students' project outcomes as they work on large and complex software projects?

Studying professional software developers. I plan to collaborate with software engineering researchers and industry professionals to study the development habits of professional developers. Using methods and interventions such as those described above, can we include feedback about the development process in the feedback loop (e.g., in-IDE static analyses, code coverage, continuous integration, etc.) that governs much software development today? What effects would this have on software quality?

Long-term: Teaching good programming process for end-user software development. The development of software — historically the sole purview of trained software professionals — is increasingly being carried out in a professional capacity by people with varying intents and motivations [13]. There are far more *end-user programmers* today than professional software developers [14]. For example, data analysts often maintain computational notebooks or spreadsheets to help make sense of data. What does effective programming process look like for people in these roles? How is this similar to or different from "traditional" software engineering best practices? How can we design instruction for students preparing to enter these roles, keeping in mind their *motivations* and *intents* for practicing computing?

Research Philosophy

Diversity in research. I respect the importance of diversity in research. My work as a researcher in computing education and software engineering is inherently human-centered. As a matter of ethics and pragmatics, it is imperative that all people are equitably represented in this research. Our findings are inherently weaker when they leave out particular demographics — such as women and other underrepresented minorities — and it is ethically unsound to put them into practice when they could potentially affect how people learn or practice their profession. I put this commitment to practice when I conduct ethnographic studies, by inviting diverse groups of students to participate in interviews.

Impact. A good indicator of success in scholarship is its *impact*, particularly in human-centered research like education and software engineering. I will continually work to put my findings to practice, in the form of new tools, engineering processes, teaching practices, or pedagogical material. For example, I am actively working to integrate formative feedback about software development based on my research in our Data Structures & Algorithms course at Virginia Tech.

References

[1] Dastyni Loksa, Amy J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. Programming, problem solving, and self-awareness: Effects of explicit guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 1449–1461, New York, NY, USA, 2016. ACM.

[2] Ayaan M. Kazerouni, Stephen H. Edwards, T. Simin Hall, and Clifford A. Shaffer. DevEventTracker: Tracking development events to assess incremental development and procrastination. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, pages 104–109, New York, NY, USA, 2017. ACM.

[3] Ayaan M. Kazerouni, Stephen H. Edwards, and Clifford A. Shaffer. Quantifying incremental development practices and their relationship to procrastination. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, pages 191–199, New York, NY, USA, 2017. ACM.

[4] Ayaan M. Kazerouni, Clifford A. Shaffer, Stephen H. Edwards, and Francisco Servant. Assessing incremental testing practices and their impact on project outcomes. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 407–413, New York, NY, USA, 2019. ACM.

[5] Stephen H. Edwards, Krishnan P. Murali, and Ayaan M. Kazerouni. The relationship between voluntary practice of short programming exercises and exam performance. In *Proceedings of the ACM Conference on Global Computing Education*, CompEd '19, pages 113–119, New York, NY, USA, 2019. ACM.

[6] Canvas the learning management platform. Accessed: 2019-11-05.

[7] Clifford A. Shaffer, Ville Karavirta, Ari Korhonen, and Thomas L. Naps. Opendsa: Beginning a community active-ebook project. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, pages 112–117, New York, NY, USA, 2011. ACM.

[8] Tomasz D Loboda, Julio Guerra, Roya Hosseini, and Peter Brusilovsky. Mastery grids: An open source social educational progress visualization. In *European conference on technology enhanced learning*, pages 235–248. Springer, 2014.

[9] Com Howlin and Danny Lynch. A framework for the delivery of personalized adaptive content. In *2014 International Conference on Web and Open Access to Learning (ICWOAL)*, pages 1–5. IEEE, 2014.

[10] James C. Davis, Daniel Moyer, Ayaan M. Kazerouni, and Dongyoon Lee. Testing regex generalizability and its implications. In *Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering, ASE 19, San Diego, CA*, 2019.

[11] Stephen H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. *SIGCSE Bull.*, 36(1):26–30, March 2004.

[12] Anneli Eteläpelto. Metacognition and the expertise of computer program comprehension. *Scandinavian Journal of Educational Research*, 37(3):243–254, 1993.

[13] Bonnie A Nardi. *A small matter of programming: perspectives on end user computing*. MIT press, 1993.

[14] Margaret M. Burnett and Brad A. Myers. Future of end-user software engineering: beyond the silos. In *Proceedings of the on Future of Software Engineering - FOSE 2014*, pages 201–211, Hyderabad, India, 2014. ACM Press.