Austin Henley — Research Statement 2018

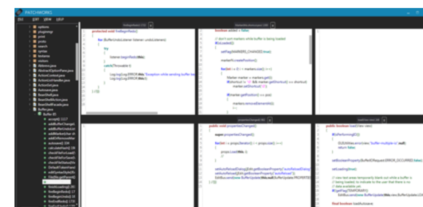
azh\@utk.edu

All software failures are fundamentally the fault of humans—either the design was ill formed or the implementation contained a bug. These failures were estimated to have cost companies worldwide over $1.1 trillion dollars in 2016 and impacted over 4.4 billion users[1]. The high cost of such failures ultimately results in developers having to design, implement, and test fixes, which all take considerable time and effort, and may result in more failures. This is a concern for both code quality and developer productivity. These complex challenges in software maintenance indicate tremendous opportunity to better support developers by enabling them to produce better code with less bugs in less time with less mental burden.

I *design* and *build* more usable software development tools by applying **human–computer interaction** methodologies to **software engineering**. In particular, I aim to design software development tools that provide enhanced cognitive support to developers and improve productivity on development tasks. My approach to tool design is a rigorous human-oriented one, involving quantitative and qualitative empirical studies of developers and user evaluations. In addition to producing well-validated tool designs, my empirical work has led to a better understanding of developer behaviors and barriers in software development. Impact on practice is a priority of my work and I have engaged in research internships and collaborations with a variety of industry research colleagues from **IBM Research, Microsoft Research**, and **National Instruments**. My research has been published in top venues including CHI and FSE, winning an **ACM SIGSOFT Distinguished Paper Award** [6], an **IEEE Best Paper Award** [4], and an **IEEE Honorable Mention Award** [8]. Toward these goals, I have applied my human-centric approach to three key tasks that developers perform during software maintenance: navigating, changing, and reviewing code.

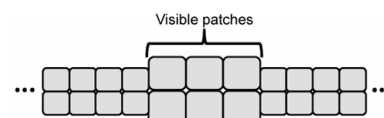
1. Tools for Navigating Code

One key problem in software maintenance is navigating source code. Research shows that existing tools for navigating are insufficient. For example, recent studies have found that programmers spend 30–50% of their time seeking information in the code. Current affordances for navigating code are little more than standard text editor features. Given that code is so structured and inter-connected, why do programmers find it so tedious to juxtapose relevant code fragments? Another burden is that programmers often navigate back and forth between a small set of code, and programmers must resort to manually cycling between document tabs and scrolling up and down.



The Patchworks code editor for Java featuring a fixed grid of patches containing code fragments.

The Patchworks code editor was designed to specifically support these navigation behaviors. By providing a fixed grid with a code fragment in each "patch" of the grid, programmers can easily juxtapose. Additionally, the grid slides to the left and right, revealing more patches that can also contain code. This design allows programmers to quickly re-visit code.



A conceptual illustration of Patchworks' grid of patches, with off-screen patches.

To evaluate the effectiveness of the Patchworks design, I ran three empirical studies. My preliminary user study and simulation study on the Patchworks for Java implementation found that programmers using

[1] https://www.tricentis.com/resource-assets/software-fail-watch-2016/

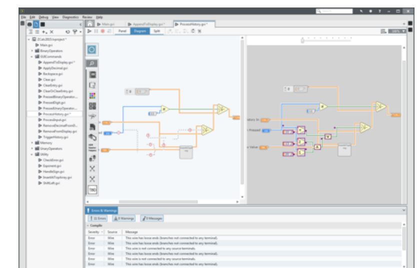
Patchworks navigated faster, spent less time arranging their code, and made fewer navigation mistakes than with other code editors [1, 2]. I then built Patchworks into an industrial code editor product, LabVIEW, and ran a user study with 32 professional developers. This summative evaluation found that compared to a tab-based editor, Patchworks has a much smaller cost to navigate, developers juxtapose far more often, and make far fewer navigation mistakes with Patchworks [7].

By studying the behaviors of how developers seek information, I can better design tools to support these behaviors. In collaboration with Oregon State University and IBM Research, I studied the foraging behaviors of developers by running three empirical studies and then analyzing predictive models of their navigations. A key finding from these studies is developers vastly over estimated the outcomes of their navigations and often navigated to irrelevant code [6]. Furthermore, developers' goal, whether learning about code or fixing a bug, significantly impacted their information seeking behavior [3]. In the first ever study of how developers forage for information on mobile devices, we found that mobile code editors are greatly lacking in supporting developers' information needs and found a number of challenges for future tools [8]. Based on these empirical findings, we also analyzed predictive models of developers' navigation such that researchers can better study navigation behaviors and design better navigation tools, such as recommender systems [5].

2. Tools to Support Code-Change

After navigating to the relevant code, another fundamental challenge is making changes to the code. A particularly common type of code change is refactoring, which consists of restructuring code without changing its behavior. Although most code editors provide numerous features for refactoring, they are very rarely used. This issue is made even more complex in visual programming environments, such as LabVIEW. My formative investigations of these developers found that they often "throw away code" and rewrite it from scratch; a very inefficient means of making progress.

Based on interviews with 54 engineers and a controlled lab study, I designed Yestercode, a LabVIEW-extended tool to support manually refactoring of visual dataflow code. Toward this goal, Yestercode transparently records all code edits and provides a side-by-side comparison of a previous version of the code with the current version with annotations. A timeline allows the developer to quickly scan through their recent history. In my evaluation study, this enabled programmers to introduce far fewer bugs and report a significantly lower cognitive load [4].



The Yestercode-extended LabVIEW development environment.

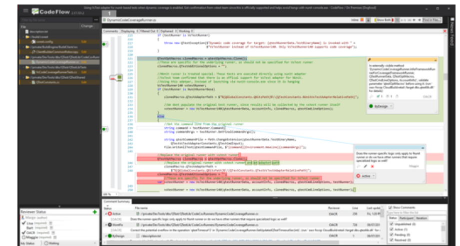
My formative work also found that developers do not adequately test their code and often introduce bugs, even when making minor changes. This led to the design of CodeDeviant, a tool that assists in detecting unintended behavior-altering changes, such as a buggy refactoring. An initial evaluation found that programmers were more likely to detect bugs using CodeDeviant and did so in significantly less time [9].

4. Tools to Enhance Collaboration during Code Reviews

Once code changes have been made, the changes are often reviewed. While doing so, it is particularly challenging for developers to effectively communicate to one another. These

developers spend considerable time performing code reviews to ensure that their peers are committing high-quality code to projects. In fact, one developer at Microsoft reported to us that he regularly performs over 100 code reviews in a week. However, much of this time is spent providing low-level feedback about code (e.g., the naming of a variable or the most efficient way to structure an *If* statement) rather than more insightful high-level design feedback (e.g., the relationship between classes).

I designed and implemented CFar, an extension to a code reviewing tool at Microsoft, that strives to enhance communication among developers while also placing their attention on more important aspects of the code review. CFar automatically annotates code with program analysis warnings and provides features that allow developers to discuss the warnings with other reviewers.



The CodeFlow code reviewing tool with my CFar extension.

To understand the impact CFar has on code reviews, I ran a small lab study as well as deployed the tool to over 90 developers at Microsoft to use in their daily job for 3 months. The results indicated that developers found CFar to enhance communication, to improve the code quality since the developers were fixing the program analysis warnings, and to save time by freeing them from commenting on low-level issues [10].

5. Research Agenda

In future research, I will make progress on three overarching projects that aim to address substantial gaps in developer support, and should bring considerable value to both the research community as well as industry:.

Collaborative development tools for distributed teams. As development teams are growing in size and becoming more distributed, maintaining their efficiency is of paramount concern. In fact, top software companies such as Google, Facebook, and Microsoft all have teams dedicated to building cloud-based tools to improve the productivity of their developers across teams. My internship at Microsoft Research was implementing and studying such a system that could be scaled to 30,000 developers. However, there has not been sufficient research performed to address questions such as: What aspects of distributed teams are not currently being supported by tools? How do these tools integrate into developers' workflows? How can novel designs leverage the potential of cloud-based applications to enhance collaboration among distributed teams? Working toward answering these questions will be of tremendous interest to researchers and industry practitioners.

Learnability of professional development tools. Developers spend an inordinate amount of time learning how to use the development tools needed for a given project. My dissertation work has been focused on the usability of development tools, however, very little work has examined the learnability of development tools. For example, how do software engineers obtain skills and knowledge to effectively operate their development tools? By better understanding this process, not only can we design easier to learn tools but we can also reduce barriers in migrating to new tooling, and thus saving developers' time and effort.

Machine learning for software engineers. Software that incorporates machine learning systems is ever growing. But who builds such systems, ML experts or software developers? As an intern

at IBM Research, my research group observed the disconnect that occurs when a software developer without machine learning knowledge has to build these systems. Given the fact that every developer cannot become a machine learning expert, how can we help these developers in such an endeavor? What knowledge do they need? What barriers do they face? What tools can help them understand and debug this software?

With my human-centric approach to research, I aspire to make progress on the trillion-dollar problem of software maintenance. Through my industry-research collaborations, I plan to study the problems that software developers have in practice and apply my research to make a consequential impact.

References

1. **Austin Z. Henley** and Scott D. Fleming, "The Patchworks Code Editor: Toward Faster Navigation with Less Code Arranging and Fewer Navigation Mistakes," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI'14), 2511-2520.
2. **Austin Z. Henley**, Alka Singh, Scott D. Fleming and Maria V. Luong, "Helping Programmers Navigate Code Faster with Patchworks: A Simulation Study," in *Proceedings of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'14)*, Melbourne, Australia, 2014, 77-80.
3. David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Margaret Burnett, Irwin Kwan, **Austin Z. Henley**, Jamie Macbeth, Charles Hill, and Amber Horvath, "To Fix or to Learn? How Production Bias Affects Developers' Information Foraging During Debugging," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution* (ICSME '15), 11-20.
4. **Austin Z. Henley** and Scott D. Fleming, "Yestercode: Improving Code-change Support in Visual Dataflow Programming Environments," *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'16)*, 106-114. (Best Paper Award)
5. Alka Singh, **Austin Z. Henley**, Scott D. Fleming and Maria V. Luong, "An Empirical Evaluation of Models of Programmer Navigation," *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME'16)*, 9-19.
6. David Piorkowski, **Austin Z. Henley**, Tahmid Nabi, Scott D. Fleming, Christopher Scaffidi, and Margaret Burnett, "Foraging and Navigations, Fundamentally: Developers' Predictions of Value and Cost," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE'16), 97-108. (Distinguished Paper Award)
7. **Austin Z. Henley**, Scott D. Fleming, and Maria V. Luong, "Toward Principles for the Design of Navigation Affordances in Code Editors: An Empirical Investigation," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (CHI '17), 5690-5702.
8. David Piorkowski, Sean Penney, **Austin Z. Henley**, Marco Pistoia, Margaret Burnett, Omer Tripp, and Pietro Ferrara, "Foraging Goes Mobile: Foraging While on Mobile Devices," *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'17)*, 9-18. (Honorable Mention Award)
9. **Austin Z. Henley** and Scott D. Fleming, "CodeDeviant: Helping Programmers Detect Edits That Accidentally Alter Program Behavior," *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'17)*, 65-73.
10. **Austin Z. Henley**, Kıvanç Muşlu, Maria Christakis, Scott D. Fleming, and Christian Bird, "CFar: A Tool to Increase Communication, Productivity, and Review Quality in Collaborative Code Reviews," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (CHI '18), 157:1-157:13.