Lydia B. Chilton Teaching Statement

I teach because I love returning to the **fundamentals**. Being constantly reminded of the basics helps me focus on the important problems in computer science and human-computer interaction (HCI). Moreover, students become independent thinkers by first mastering the fundamentals, then bringing their own unique perspective. I have mentored student research at MIT, Stanford, and the University of Washington. I have helped students turn vague interests into research questions relating to fundamental underpinnings of computer science. My reward is getting access to their unique ideas and skillsets, thus doing projects I would never think to do on my own.

Creating Courses for Everyone

My first experience with mentoring student projects was at MIT during my MEng in 2007. MIT needed a class to teach web programming, so I started a month-long web programming class and competition called 6.470. Crafting this course from scratch was a team effort, but I was the chairperson in charge of developing the curriculum, teaching half of the lectures, raising prize money, and handling logistical arrangements such as coordinating free food.

In the first year, we raised $10,000 in prize money and had to cap enrollment at 60 students. I also lead 6.470 in 2008 and 2009 where we raised $20,000 and $40,000, respectively. Enrollment surged. Today, 6.470 is alive and well. New leaders have emerged and students are excited to put 6.470 on their resume. I have had numerous awkward conversations where students with 6.470 on their resume start describing what it is to me before I tell them I know what it is. I have seen people wearing 6.470 t-shirts 3,000 miles away from Cambridge on the Stanford campus.

The most important task in crafting 6.470 was finding a web programming challenge that the staff could judge student submissions on. At the time, MIT had two other student-led programming competitions: an AI programming competition and a robotics competition. Both were about robots killing robots. This did not appeal to me personally, thus my goal in 6.470 was to create a class where students could create something useful that solved a problem in their own lives. No robot killing required. The first year we ran 6.470, the course challenge was creating a website that used a large database of movie data. The student submissions were amazingly creative. Students created a movie-preference-based dating site, an app for coordinating movie-watching events, and sites to help dorms keep track of movie rentals. Once students learn the fundamentals of web programming, they can express their own interests.

Students enjoyed selecting their own real-world problem to solve in 6.470. To me it was important to have a class where students could follow their own interests. Classes about killing robots appeal to some people more than others. Classes that allow you to follow your own interests are for everyone.

Listening to Student Interests

I served as a teaching assistant for project-based HCI classes at both MIT and Stanford. In both cases I got to mentor group projects and help teams ideate and execute a project during the term. A common pattern is that groups come in with vague interests that are not well articulated. For example, one group in Stanford's Human-Computer Interaction Research class (CS 376) wanted to "study digital money." It would be easy to dismiss these ideas as vague, ill-formed, and not research, but my approach is to see if I can tease out one concrete research idea from their amorphous interest. One method I have for this is using the "5 Whys" described in *The Toyota*

Way. This is a common business and software engineering practice to getting to the root cause of a problem, or the root inspiration for a solution.

In the case of the group that wanted to "study digital money". I asked students why they were interested in digital money, what digital money means to them, what their experience with digital money was. At first I got circular answers such as "digital money is new and important." But as I dug deeper, I unearthed a single personal experience of that group that illustrates and motivates the problem concretely. For example, I helped the group establish that digital money concretely meant paying through the phone with apps like Uber or Venmo, and I got to the root of their simple insight through a concrete experience with Venmo: "Paying with Venmo is so simple, I already have my phone out and I don't have to reach for my wallet."

From this insight, we formulated a research goal: if paying with Venmo really is simpler than credit cards, then we should be able to measure the degree of simplicity in terms of how much more people are willing to pay for the same product if they get to pay with Venmo. To test this, the students set up a stand in the student center every day for a week and sold a desirable product, Bubble Tea, at two difference prices: one if you were paying with Venmo and one if you were paying with Credit Card. They kept track of which customers had both Venmo and credit cards and decided to use one over the other. Using this methodology they found that people were willing to pay more to use Venmo, which surprised me, and validated their theory.

Measuring the price elasticity of Venmo is not the kind of project I would ever do by myself. First, because measuring price elasticity has more to do with business than computer science. And second, because selling Bubble Tea at two different prices is outside my comfort zone. But this team of students had no problem with that whatsoever. They enjoyed it. As a mentor, I was able to be part of something that I would never have gotten to do because I encouraged them to follow their interests that used their unique skills. And it was all because instead of dismissing their project, I listened to their insights and helped them express them in a scientific manner.

Fostering Good Software and Research Practices

My teaching philosophy is to help make students independent thinkers by having them master fundamentals, then letting them apply these fundamentals to their own unique interests. However, part of teaching this is not just teaching the fundamentals, but also teaching an efficient process for applying them. One fundamental principle of software development is to reduce your cycle time. You do this by making process improvements to your development process. Although people all know this, we often do not do it because it requires some investment in creating tools. When I create teaching materials, I always demonstrate how good frameworks make the development process faster, better, more reliable, and less tedious.

As the TA for the graduate Machine Learning course at the University of Washington, I created the problem sets used in the class. The first problem set required students to implement the ID3 decision tree construction algorithm. Although the algorithm can be expressed simply in less than 20 lines of pseudo code, it is tricky to handle all the edge cases and the recursion properly. To help the students implement ID3, I provided test cases, as any good software development process would have. Additionally, to illustrate the power of a good development process, I also gave them tools for **visualizing their outputs** against the test cases. Visualization is a powerful part of the development cycle that is often overlooked. Although visualizations require an investment of development time, they pay rich dividends in the future. We process visual data faster than we process text data. We process visual data subconsciously in parallel, whereas we

process text data consciously in serial. Visualizations make debugging much easier by using fast, low-effort subconscious processing rather than slow and exhausting conscious processing.

Many students remarked that the visualization tool was a huge improvement to their debugging process. I do not know if providing this example encouraged them to make the visualizations in the future. Obviously, I cannot force students to adopt good practices, I can expose students to good practices and trust they eventually see the value and adopt them into their own practice.

Future Teaching Plans

In addition to teaching introductory courses in programming, software engineering, artificial intelligence, and algorithms, I can teach fundamental courses in human-computer interaction

- **Principles and Practice of Human-Computer Interaction**. Undergraduates preparing for industry jobs or start-ups need to have a firm grounding in HCI. This course centers on the practice of iterative design. It is both practical in application and fundamental in that the practices are firmly grounded in the psychological underpinnings of the human mind.
- **Web Programming for Fun and Profit.** Web programming is a fundamental skill for anyone who wants to easily get a message out to the world – either for research or business purposes. Web programming integrates database design, client-server architecture, usability, as well as programming models such as the Model View Controller and Observer patterns.
- **Crowdsourcing**. Crowdsourcing is a revolutionary way to collect large datasets and thus it is important to many computer science subfields such as graphics and machine learning. I have developed many of the current practices of crowdsourcing datasets, I am abreast of the best industry practices, and I am excited to help people from many fields master this tool.
- **Human-Centered Design for All**. Human-Centered Design is an essential approach to solving problems in HCI but is has general insights that apply to other disciplines, such as mechanical engineering, medicine, business, and civil engineering. A class teaching the shared methodology of human-centered design can bring together students from seemingly disparate fields and get them to work together towards the common goal of building systems that people love.

Conclusion

My mission is to create courses that instill the best practices of software development. Those practices are backed by the technical foundations of robust system-building and also the fundamental principles of human cognition. As long as software is written by people and used by people, principles of cognition are a central element of engineering. Although cognition is complicated and overwhelming, by focusing on the fundamentals we can engineer systems that are tuned to the requirements of human cognition.