Teaching Statement

Matthew Might

Fall 2007

Over the past four years, my experiences as instructor, recitation lecturer, teaching assistant and undergraduate advisor have shaped my approach to teaching and learning. Because these experiences provide insight into the evolution of my teaching style, I've provided a sampling of them below. In brief, my teaching style emphasizes interactive synthesis of computational artifacts in the classroom and constructivist projects outside of it.

Instructor for introduction to object-oriented programming As one of the instructors for a course on object-oriented programming, my role was to deliver three one-hour lectures each week on GUI programming. As a prerequisite for several majors, this was a large course divided into six sections of fifty students each. (I gave the same lecture to three different sections each week.) While the topic for each week was chosen according to an established syllabus, the format of every lecture was at my complete discretion.

My time as a student had already led me to distrust impersonal, slide-driven teaching. What had once been an engaging dialogue has too often become a high-speed, 55-minute monologue. In response, lecture in my class was an interactive discussion of the GUI component of the week (*e.g.* buttons, check boxes, layouts). Each class began with a chalk board drawing of a GUI containing the week's component. After this came code on the board that generated the GUI. By then modifying the code to achieve some event-based behavior (*e.g.* clicking a button to change the text in a label), students gained an understanding of the link between the code and the result.

With the basic principles behind rendering and events established, the class moved into free-style mode, where students were elicited to add more components to the drawing and to request new event-driven behaviors. After adding a few stubs to the existing code, I asked a sequence of questions to the class that would cause them to fill in the required code. Even if the incorrect answer was given, I would still illustrate its effect. This teacher-as-compiler approach proved effective at getting students to interact.

Following this free-styling, the GUI drawing was erased, and the code on the board was replaced with new code. Afterward, students were called to come to the board to draw what they thought was created by this code, with other students offering feedback on their correctness as they drew.

In the last five minutes of every class, I turned on the projector and opened up about five different applets that I had created specifically for that lecture. Each of these applets was a micro-demo (50-100 lines of code) for some aspect of what had just been taught. All of this heavily commented code was made available on the web, and students were encouraged to use these micro-demos to piece together solutions their homework assignments.

With six weeks left in the semester, I created a 2D space-based combat game to demonstrate animation and sprites in Java, and released the code to the class with a promise: if you modify the game in fun or interesting ways, then I'll demo your changes in front of the class. No extra credit was offered for this. For an assignment with no bearing on their final grade, the response was uplifting. Twelve students added a range of features—from cheat codes, smarter AI, particle explosions and new weapons to networked multi-player modes and frame-rate-doubling efficiency modifications.

Most encouraging from this experience has been the student feedback. Over a year later, students still drop by my office, stop me on their way to class and approach me in public to shake my hand and tell me how much they enjoyed taking my class.

Lecturer and teaching assistant for languages and translation Before becoming an instructor, I spent a semester as a recitation lecturer for the first major-specific course in Computer Science at Georgia Tech, entitled *Languages and Translation*. Fortunately, the syllabus for this course was less entrenched, and this gave me the opportunity to actively shape the material covered. By designing and grading the three-phase final project, I successfully guided over 200 students through a challenge that the course instructor had considered beyond the difficulty level of the class.

It has long been my assertion that creating an interpreter or a compiler is one of the essential ingredients in demystifying the nature of computing for students. With this in mind, the final project consisted of constructing a lexer, a parser and a hand-coded library for a small, higher-order, Lisp-like language in C. To make this task easier, compiled object files for a lexer, a parser, and an interpreter back end were also released. By also providing the auto-grading test harness, students could develop in a very tight, interactive code-test loop. Finally, I offered three optional but widely attended lectures, one for each project phase.

The student performance exceeded my expectations: three-fourths of the class ended up with 90% or above, and most of the remainder received 80% or above on the project. Less than a percent of the class failed.

As with my experience as an instructor, I still receive positive feedback from students on both the class and the project. In fact, two of these students have since come under my research supervision, and both have said that the Lisp interpreter project was the reason they chose to do research in programming languages and compilers. But the outcome that is most gratifying is that several of my students went on to become TAs themselves; and inspired by my passion for the subject, they invited me to their own classes as a guest lecturer on topics like the λ calculus and regular languages.

Advising undergraduate research Two of my former students, Ben Chambers and Daniel Harvey, have since signed up to work with me on a research project. Under my supervision, they have added a new intermediate translation phase and adapted one of my static analyses (ΓCFA) to the MLton compiler for Standard ML. Based in part on results from this implementation, Ben and I published a paper in the 2007 International Conference on Verification, Model Checking and Abstract Interpretation. Ben has also joined the Ph.D. program at Northeastern University. Daniel Harvey is now graduated and at Microsoft's compiler group.