A Systematic Approach To Teaching

Daniel Patterson

Every student creates their own understanding of the material presented to them. My goal as a teacher is thus two-fold: first, to **create an environment that has the support and structure** to carry out that creation and second, to provide them with **engaging and challenging material** that they will use to build up that understanding. Running parallel to that learning process, and more opaque to the student, is assessment of how consistent their understanding is with what I hope for them to learn. For each, there are *student-facing* aspects and *instructor-only* aspects.

An Environment for Learning: Student-facing At Northeastern University, I designed and taught an upper level course on verified compilers, in which students both learned how to use a proof assistant and then proved a series of small compiler transformations correct. I was very conscious that learning how to use a proof assistant is quite difficult, even for advanced students, all of whom had been programming for several years and some for much longer. This difficulty stems from the fact that, despite veneer (tactics, automation, etc), what underlies the type of proof assistants that I was using are dependently typed core languages, and building up huge dependently typed terms (as any non-trivial proof will be) is very hard, even for experienced functional programmers. Indeed, it can be as if learning to program again.

Understanding the challenge that students would have and the possible frustration that could result informed various aspects of the course design: first, that the vast majority of class time would be spent working in pairs on proofs, with me floating around to get people unstuck. Working in pairs made it a more dynamic experience, but also more productive, as both members of the pair would have to run out of ideas before the pair got stuck. At the end of each class, one group would present their work, allowing others to see an alternative approach. This class structure allowed students to practice by doing but also see a possible solution, all while having expert assistance available. In a pattern appropriate for advanced classes, I relegated traditional lecture material to readings that accompanied problem sets, with traditional office hours available. The second intervention was that I encouraged splitting out parts of proofs and continuing on, both modeling that during in-class demonstrations and structuring grading to encourage it in homework assignments. While quantitative results from small classes are difficult, qualitative results—in the form of surveys—confirmed that the class work was strongly appreciated, with nearly all the students successfully tackling the difficult material.

An Environment for Learning: Instructor-only In any class, students attend lectures, participate in class, do assignments as they receive them, etc. They experience, necessarily, only what is visible to them, and in larger classes, they may interact with many different members of a course staff. In the accelerated version of the first semester class at Northeastern University, I oversaw two TAs (head TAs), five tutors (regular TAs), and two graders (TAs that do not hold office hours). A significant aspect of that job was ensuring consistency. I did this in grading of homework assignments, for which I both wrote rubrics and also, along with the two TAs, reviewed the grading of the other seven. But even less visibly, it involved ensuring that all the staff, during labs, office hours, and online, maintained the balance of being helpful without just giving answers, i.e., creating an environment in which students are challenged and can learn, not merely an environment where any question they have will be answered.

In my own class, which was small and had no supporting staff, I was careful to set up structures that would be conducive to students having support but working independently. For example, I created a student-only discussion forum where I explicitly declared I would not participate, to allow peer support without any expectation that responses would be correct or canonical. Supplementing that, I set up a broadcast-only mailing list, so that students could email me questions, and if they warranted correction to the whole class, I could share them, but that otherwise the presence of the question was invisible to others. This setup provided support without encouraging students to reach out until they had gotten truly stuck, an important goal given the challenging material.

Engaging and Challenging Material: Student-facing My second role is to provide material that by engaging with students build up their understanding. In the accelerated intro class at Northeastern, I design a semester long project in which students built a tile-based game, which began as a single player game and later became multiplayer by connecting their clients to a server. Unfortunately, features of the framework we used made aspects of the client-server interaction difficult to introspect and made debugging the second part frustrating. While successfully interacting with classmates in the open world game was fun and rewarding, in the future I would be sure that the experience was either easier to debug, or structured in a different way so as to avoid the necessary debugging.

As another example of challenging material, I also wrote and delivered a lecture for the same introductory class where, after having used an untyped language throughout the semester, I livecoded a simple system to enforce (at runtime) some of the types they had been writing in comments. The lecture was highly dynamic—all typing I did driven by their input—and the eventual payoff so rewarding that the students applauded.

Engaging and Challenging Material: Instructor-only While students view assignments as one-after-another, perhaps related or perhaps not to previous material, the reality is that both the sequencing and what goes into them is of critical importance. While lectures or readings may provide students a surface level understanding, only by engaging with assignments will they actually build a deep understanding, and thus the assignments should be thought of as an entirely separate parallel presentation of the content of the class: they must encompass all of the important material, must build upon one another in reasonable increments, and must have enough of a balance between interest and difficulty to push students.

In the accelerated intro class at Northeastern, the tile-based game that I designed mirrored the students' increasing familiarity with data structures throughout the single player version. The conversion to the multiplayer version, while initially daunting, was actually an exercise in refactoring, where nearly all functionality could be re-used, with only small additions that distinguished between input from the keyboard and input from the network to identify which player was moving or acting.

In my course on verified compilers, providing engaging and challenging material primarily involved designing exercises, both for in class work and for homework. These exercises were the substance that students used to build their capacity both in theorem proving and compiler verification. For the course, we were using a common proof technique within the research literature called simulations, for which there were many complex examples, but nothing simple. What I came up with, which does constant folding only in the leaves of syntax trees (e.g., $2 + (2 + 2)$ compiles to $2 + 4$, not 6), included all the parts of a simulation proof while remaining very concise, thus could be used by the students as a mental model throughout the rest of the semester.

Assessment The final role that I have as a teacher is to **assess students**. This is usually taken as synonymous with exams and homework, but I take a slightly different view: courses must have some

core content that a successful participant in the course will be able to understand. The course may, and indeed most likely will, have plenty of additional content that students might learn, and each students' understanding of *that material* will vary. The purpose of assessment must be to determine whether students have sufficiently built an understanding of core content—and the possibility of the assessments being imperfect at capturing that must be acknowledged! This imperfection is more than just the possibility of inconsistency across course staff, but indeed that the very nature of the assessment may be flawed.

For example, while timed exams may be necessary for the sake of pragmatism (they can generally cover simpler material and still get some sense of an individuals' knowledge), even if graded perfectly there are reasons to doubt their quality: some students work poorly under time constraints, and even worse, they are often done under different contexts than any realistic (whether academic or industry) setting. For example, in the introductory programming classes at Northeastern, while assignments are done using an interactive development environment, in exams programs are usually written on paper. One reason for exams is to avoid plagiarism, but if we take plagiarism as simply an input that assessment should be designed with awareness of, rather than something that should be allowed to override other design constraints, we may come up with better alternatives.

Consider adapting the strategy of "code walks" used in some courses at Northeastern to exams as follows: an exam would take place over a set period of time, say 24 or 48 hours. At the end of the period, there is a several hour block (the normal "exam block") during which a random selection of students (say, 10%), on submission of their exam, will be asked to come and explain ("walk through") their code in person, with the grade dependent not on what they submitted but rather on their *explanation* of the submission. The long period to work on the exam allows students to take more time and use the programming environment they are familiar with while they develop their solutions. The code walk, by being applied stochastically, requires a comparatively small amount of staff time. It also should be relatively easy for students who did the exam: they explain the solution that they worked on over last day or two, but it provides a strong (psychological) check on academic code violations, as the randomness means that there is no way to predict if you will be in that pool. Finally, if a student is able to submit work they did not do and yet explain it with confidence, then they clearly understand the material, and thus the actual pedagogical loss to such plagiarism is minimal. One risk to this, of course, is that someone could get stressed out explaining their own code – this should be mitigated by having the "code walk" activity not be something that only shows up in the context of exams, as indeed, the process of explaining code is a critical skill that students should learn regardless. I explore this not only as a potential real idea, but as a demonstration of the type of systematic approach that I bring to teaching.

Courses I Can Teach I'm most drawn to introductory sequences, as that is where the interventions that we make have the strongest impact, both positive and negative. Much as in buildings, foundations matter. As interested as I am in teaching them, I am also interested in helping to evolve them in a systematic way. I can also teach discrete math, logic courses, theory of computation, and would be open to learning how to teach any other first or second year course as well. Of course, I can also teach courses from my research area, spanning across compilers, programming languages, formal verification, type systems, etc. Finally, a particular topic of interest is software development: before grad school, I started and spent several years running a company primarily working on legacy code bases, an experience not unlike what many aspiring software developers will encounter, and I'm interested in how to better prepare students for the reality of legacy code, debugging, etc.