

6170: Laboratory in Software Engineering, Fall 2001

A New Approach to 6170

Daniel Jackson

1 Caveat & Acknowledgments

This note represents my opinions, and not necessarily those of the current teaching staff of 6170, nor of the larger group of faculty who are involved in the teaching of 6170 from term to term, nor of the faculty of the department as a whole. Nevertheless, my opinions have been reinforced by discussions with other faculty members who share my concerns. I am particularly grateful to Martin Rinard for his perspective, and to the many students who have spent time telling me about their experiences of MIT courses.

2 Problems

In recent offerings, it seems to me that 6170 has suffered primarily from two problems: one a problem of many courses in the department (the ‘firehose’ phenomenon), and one specific to 6170 (the need to accommodate the material of two courses in one). Both of these have ramifications in the broader context of the undergraduate degree. A problem more local to 6170 is a perceived gap between theory and practice.

2.1 The MIT Firehose

6170 has become a *burden*: too much work to teach, too much work to take. The ‘firehose’ course is a long tradition at MIT, and there is much to be said for the excitement and devotion that many MIT courses engender in the students who take them. But as a course becomes more and more elaborate, we run the risk that it is no longer *fun*, and that, with a loss of pleasure, comes a loss of enthusiasm and attention. If passing the course requires an extraordinary amount of work even for the best prepared and most talented students, the effect on the rest is demoralizing. Students who are more than clever and conscientious enough to master the material of the course find that just keeping their heads above water is a challenge, and have no time or energy left to mull ideas over, to experiment, to pursue a digression – in short, to learn deeply.

A large part of the problem, it seems to me, is the use of an infrastructure that grows monotonically from term to term. Problem sets get longer, projects expand, and specifications of problems to be solved become more elaborate. Ironically, infrastructure designed to help students and teaching staff, such as mechanisms to hand in and automatically check code, becomes itself a burden. Doing a problem set no longer involves just grabbing a pencil and a sheet of paper, and starting to think. Instead, the student must become familiar with an arcane environment of tools and protocols, like a programmer joining a new development group.

All this extra effort has some benefit, of course. The problems are challenging and even learning mundane details of an infrastructure can be useful to a student in a summer job. But the benefit does not nearly outweigh the cost. Students spend a large part of their time thinking about trivial and uninteresting things – time that could have been spent more imaginatively.

2.2 Two Courses in One

6170 is actually at least two courses in one. The central focus is software design and development, at a medium scale. Now a prerequisite for a software design and development course is a basic ability to program and a bit of software engineering sense – understanding, for example, the difference between getting a program to run the first time and getting it right.

Unfortunately, most students entering 6170 don’t have this background. Usually, 6001 is the only course they have taken; its focus is on ideas in programming languages and computation rather than on programming itself. So 6170 has had to play a role as a first programming course. There is no room in the schedule for lectures on basic programming notions, so students are forced to learn a new programming language (Java) in only a week or two, and have to

develop their proficiency in programming at the same time as they learn more advanced notions of software engineering.

The course therefore favours students who already have a strong programming background. This demoralizes the other students, perpetuating a culture that divides students into hackers and the rest. We miss the opportunity to give students a solid and sophisticated grounding in programming. Students without programming experience are made to feel inadequate and in need of remedial education; students with programming experience have to unlearn bad habits. This also probably reduces our chances of redressing the gender balance in computer science, since recent studies suggest that women react badly to being forced to compete in a such a culture.

Needless to say, the conflation of two courses contributes to the firehose problem. It means that students don't get a proper programming course, and it makes it harder to teach the central concepts of the course. Many students struggle so hard to get their programs to run at all that they learn little about design. To accommodate students' need to learn how to program, the problem sets of the first half of the term have traditionally had very little design content, with students coding modules to specifications rather than designing a system as a collection of modules. The final project became the first and last opportunity for practicing design skills.

2.3 Gap Between Theory and Practice

The fundamental ideas of 6170 – specification, data abstraction, decoupled design, invariants – are enormously valuable in practice. But many students don't discover this until they are out in the trenches; in a recent questionnaire given to alumni by a student group, 6170 and 6033 were cited as the most useful courses they had taken. While taking the course, however, many students regard the conceptual material as irrelevant, and there is a prevalent attitude, conveyed from one generation of students to another, that 6170 students learn all the useful stuff in their project work. Of course, we hope and expect that students will learn more from the final project than any other part of the course – it is, after all, by far the dominant activity time-wise.

Nevertheless, the complaints of some students that there is too large a gap between the theory presented in the lectures and the practice of software development is to be taken seriously. We must not capitulate and teach current fads such as UML in place of ideas of long-term value, but we should be able to convey to students more convincingly the value of the course's ideas in the context of industrial development.

3 Some Changes

This term, we will be experimenting with some radical changes to the course. The two-courses-in-one problem is of course a curricular problem, and cannot be solved withing 6170, so the best we can do is try to mitigate it. The other problems seem more tractable.

- *Reducing load.* The problem sets this term will be shorter and simpler; we are calling them 'exercises' to signal the change. There will be less code to write, and students will not be required to learn programming notions, complete an implementation, develop a test suite and document everything all at once. Instead, these will be separated: in one problem set, students will construct an abstract type; in another, they will apply techniques to ensure its correctness. Students will not be required to document their work in a uniform manner throughout the term. All new concepts will be presented in lecture; there will be no new material presented in recitations. There will be only a single, multiple-choice quiz.
- *Learning Java.* Students will be given more time to learn Java. Java tutorial material will be dropped from the lectures. Instead, students will study the Sun Java Tutorial at their own pace for the first two weeks of term. The first week's written work will be a very simple series of finger exercises. The exercise of the second week will be a scaled-back version of the first problem set from last term.
- *Reviews.* Recitations will be 'review sessions', in which student work is presented and critiqued by all the students in the section. The TA's role will be to direct discussion, encourage students to participate and draw connections between student work and the ideas presented in lecture. Reviews will let TAs give students more feedback than they have received in the past, and students will benefit by learning from their peers. Review sessions will not be compulsory, but significant credit will be given to students who contribute insightfully, which will offset credit lost in written work. As the term progresses, students will become more responsible for presenting their own work, so they will learn how to present their own ideas as well as critiquing others.
- *Creativity.* In the weekly exercises, students will be given free rein to solve a problem in their own way. The data

abstraction problem set will require them to design the API of the type, in addition to designing the representation. Students will be responsible for convincing their TA of the correctness of their code by their use of arguments or tests. TAs won't be made to pore over student code to find errors, but will instead evaluate the students' case for correctness – code, unlike a person, is guilty until proven innocent. Without a standardized specification, it will not be possible to construct an automatic grading mechanism for student code. But in my opinion, this has become a strait-jacket. In this looser approach, students will get more practice, earlier, in the design of APIs, and they will more likely appreciate the need to get code right the first time (since we won't be helping them find bugs).

- *Design Exercises.* Two of the exercises will be open-ended design exercises in which students will specify a small system, design it and build it. Students will learn, before the final project, how to work incrementally, minimize risk, and identify realistic feature subsets.
- *Practical Examples.* The trend, started a few years ago, of illustrating ideas in lectures with more realistic examples, will be continued. Coverage of design patterns will be increased, reinforced by the inclusion of design patterns in the new course text by Liskov. Additionally, several lectures will be spent discussing some case study programs, showing how they use ideas from the course, and critiquing their designs.

4 Comments?

Comments are very welcome, of course. Reactions from students would be especially helpful; I hope to hear extensively from students this term about their experience. Send mail to me at dnj@mit.edu, or drop by my office (NE43-530) and chat.