1 Introduction

6.005 is a course being developed as part of the new EECS curriculum at MIT. As a ‘foundation course,’ it sits between the pair of (required) introductory courses that integrate material from EE and CS, and the (elective) ‘header’ courses that mark the start of more specialized streams. The course is expected to serve, and be a prerequisite for, the headers for systems, software engineering, artificial intelligence, and perhaps others. Students will take a minimum of 3 foundation courses; 6.005 is expected to be taken by all students majoring in computer science.

These notes reflect the current state of my thinking about the course. Many of the ideas are due to my colleagues. I’m grateful in particular to those who have taken time to talk to me and have sent suggestions about the course in the last two months (Nancy Lynch, Frans Kaashoek, Saman Amarasinghe, Martin Rinard, Mike Ernst, Ron Rivest, Seth Teller, Matthias Felleisen, John Guttag, Hal Abelson, Butler Lampson, Derek Rayside, Greg Dennis, Robert Seater, Emina Torlak, Felix Chang, Mana Taghdiri).

2 Aims

The course has two aims. The first is to teach students fundamental programming skills that will enable them to engage successfully in programming projects in coursework and research projects, and to perform basic programming tasks in their subsequent careers. For students intending to follow a stream involving software development (such as systems or software engineering), these skills will be the base on which more advanced skills are built. The second aim is to engender in the students some important engineering sensibilities, so that they appreciate, for example, how to assess and mitigate risks (from the engineer’s own limitations, from the environment, and from the underlying technology), how to recognize and tame complexity, and how to formulate problems and evaluate the appropriateness of different solutions. This aspect of the course is intended to reinforce similar lessons learnt in other courses and educational contexts, and to be of wider application than the more concrete programming skills.

3 Themes

Courses on programming and software development range enormously in their content and style. Traditional programming courses focus on the syntax and semantics of a particular programming language and the mechanics of constructing, testing and debugging programs; or on the selection and expression of algorithms and data structures; or on mathematical specification and proofs of correctness; or on the variety of programming languages and their paradigms. Traditional software engineering courses tend to focus more on large-scale issues, emphasize lifecycle concerns (such as requirements elicitation, architecture, and maintainability) and often take a process or managerial
This course does not follow any of these conventional paths. Like traditional programming courses, it focuses on small scale system issues; but like software engineering courses, it emphasizes engineering concepts rather than algorithmic or linguistic issues. Three engineering themes run through the course:

- **Models.** Programming novices typically find it very hard to bridge the gap between a vague formulation of a problem and a precise solution in the form of an implementation. This difficulty arises in large part because they have no language for describing the problem or exploring design ideas, and cannot see the underlying abstractions in the code because of irrelevant complexities of the programming language. The course will therefore teach some basic modelling concepts, with appropriate lightweight notations, and show how to capture essential abstractions in a more succinct and perspicuous form prior to implementation. Familiarity with basic modelling notions is also an important element of software development literacy, since developers are nowadays expected to comprehend a wide variety of models. The design pattern literature, for example, makes extensive use of object models and sequence diagrams, and state diagrams are widely used to describe protocols and embedded systems.

- **Interfaces.** Most of our students rapidly acquire the ability to write short pieces of code – indeed, many will enter the course with this ability – and are thus able to implement modules, given their specifications, without much trouble. What they find much harder is the design of the interfaces between modules. Interface design is the lynchpin of software implementation. Poor interfaces result in needless complexity and coupling, and make a system unmaintainable. Good interfaces, on the other hand, simplify both the implementations of the interface and the implementation of its clients, make reuse possible, and allow independence arguments to be made. A pervasive theme of the course is therefore the design of interfaces, and their articulation in specifications. This includes conveying an understanding of the meaning of an interface (namely the abstractions that distinguish observable behaviour from internal implementation), as well as heuristic issues in what distinguishes good and bad interfaces.

- **Analysis.** Testing is an integral part of software development, and the value of testing should be inculcated from the start, and not treated as a secondary activity to be delegated to engineers with less talent. Mature engineers are not infallible; what distinguishes them from novices, however, is a keen awareness of their fallibility, an ability to assess the quality of an artifact and identify the most likely sources of error, and well-established habits to catch mistakes early. In addition to teaching fundamental notions of testing, the course will emphasize the role of invariants in reasoning about programs, and their conversion to runtime assertions.

## 4 Pedagogical Principles

The following pedagogical principles will guide the selection of material covered and the way in which students are engaged and assessed.
- **Demonstrable ideas.** Many useful and important ideas in software engineering cannot be demonstrated in the context of the small projects that students can complete within a term. An idea cannot be taught effectively simply by dictating to students; they must grapple with the idea and experience its power and limitations themselves. Moreover, attempting to teach ideas at too small a scale can be counterproductive, because students will not be convinced of their utility, and their skepticism may even spill over into rejecting ideas that have demonstrable value in the work they do. The course will therefore focus on ideas that can be demonstrated in programs of a few thousand lines of code, and will not include topics such as requirements or reverse engineering that are most valuable for large systems.

- **Technical, not managerial, concepts.** A course must inevitably reflect the intellectual tastes of its creators and teachers. Our own particular bias as computer scientists is that we favour technical ideas (related to language, structure, meaning, etc) over human and social ideas (such as management strategy, process, economics, team dynamics, contracts, marketing, etc).

- **Real problems, real tools.** Throughout the course, students will be challenged with simplified versions of realistic problems, rather than contrived toy problems. Real problems have a very different flavour from toy problems, in particular because they have untidy edges and are less amenable to simple formalization; handling them also gives students a greater sense of pride in the value and relevance of their work. Although acquiring skills with tools is not a focus of the course, students will make use of some standard industrial tools, not only because familiarity with these tools is an important element of software development literacy, but also to give us an opportunity to teach students how to handle their often gratuitous complexities. This consideration influences the choice of programming language, favoring Java, for example, over more elegant but less widely used languages (such as ML).

- **Examples as exemplars.** In some subjects – mathematics, for example – examples need do no more than convey the idea at hand, so they can be reduced to a form that is unrealistic and contrived. In software engineering, however, students treat examples for much more than the particular idea the teacher is attempting to convey. They develop their taste and judgment in design, implementation and analysis in part by observing the teacher’s work, even in trivially small things. Examples must therefore be exemplars of good practice. An example can simplify (especially by omission) but it should always be representative of the highest engineering standards. It is a disservice to students to show an example that requires an apology (“of course, in practice, you’d never do this”), especially if it illustrates the use of a technique that is more elaborate or burdensome than would be appropriate in practice. Students need frequent illustrations of how to use techniques judiciously; they need a model of applying techniques in a cost-effective manner that doesn’t require perfection and is not dogmatic. I learnt from Bob Harper that the best examples, on the contrary, contain hidden gems – that, in addition to conveying the idea at hand, they demonstrate an elegant pattern or exploit a clever insight. The best examples merit intense study, and can stand by themselves: Butler Lampson’s notes for 6.826 (Principles of Computer Systems) consist mainly of a series of subtle examples that each represent a wide class of systems.
· **Doing it twice.** Even the best students cannot acquire a basic engineering skill in one lecture and one problem set. Repetition is essential; at the very least, the student needs to do something twice to apply in the second attempt the lessons of the first. Being given an opportunity to revisit a problem (and gain lost points) also seems to bring satisfaction and closure to students, and relieves the stress that some feel to achieve perfect results the first time.

· **Fun for novices and experts.** Software engineering can involve a lot of drudgery: dealing with over-complex APIs, debugging messy code, reverse engineering tangled programs, etc. It is critical that students are convinced that drudgery is not inevitable, that it often arises from needless complexity, and that keeping complexity under control can be deeply satisfying. It is therefore essential that a first course in software engineering should be a pleasurable experience for students and staff alike, and that the exercises should be regarded as fun intellectual puzzles. This can be achieved by careful selection of examples, by minimizing the burden of learning how to use tools, by dramatically reducing the length of problem set handouts, by giving students very open problems that give them opportunities to be creative, by deemphasizing syntax and programming language complexities, and, most importantly, by giving students the sense that they are acquiring valuable skills. A particular challenge of a course with a large programming component is the diversity of the student body, some with extensive coding experience and some with none. To keep the novices happy, it is important to make everything explicit, and to make sure that no knowledge is required to complete problem sets beyond what is explicitly taught in class. To keep the more experience students engaged, it is important to design problem sets that will challenge them too, and to cover topics and approaches early on that will be new to them.

· **Ideas in their simplest context.** A major motivation for the structure of the course is to teach concepts in their simplest context, so that students can see their essence without being confused by extraneous details. For example, in 6.170 (Lab in Software Engineering), we have taught the idea of invariants and structural induction in the context of mutable datatypes. But invariants are much more easily explained considering safety properties of simple state machines; and structural induction is more easily explained for immutable data types for which the notion of a constructor is far more straightforward.

· **Getting it right: rewarding high quality work.** “The beginning of wisdom for a programmer is to recognize the difference between getting his program to work and getting it right.” [Opening sentence, Michael Jackson, *Principles of Program Design*, Academic Press, 1975]. Most software engineering courses attempt to convey this sensibility, but few manage to reward students who acquire it. Instead, students will often receive nearly full credit for a program that almost works, and which was constructed by a trial-and-error process that lacked a sense of direction or purpose. Their understandable conclusion from such an experience is that getting it right doesn’t matter, and that brute force of intellect counts for more than sound engineering principles and techniques. This course will adopt two strategies to counter this attitude, and to convey the message that “getting it right” is both achievable and satisfying. The first is a non-linear grading scheme that heavily penalizes work
containing sloppy errors; the second is cross-testing, in which students earn credit
for writing test cases that find bugs in other students’ programs (and lose credit for
bugs found in theirs).

• *Just-in time syntax*. Learning any notation, especially a programming language, can
be overwhelming, and it’s easy to waste class time on language details. This course
will teach notations as needed, in support of the conceptual ideas that will be the
focus of the lectures. For example, the basic control flow mechanism of Java, the
semantics of assignment, and a naive view of field update will be taught with state
machines; exceptions and runtime assertions will be taught with defensive pro-
gramming; subclassing will be taught with algebraic datatypes; etc.

5 Three Paradigms

Most software development and programming courses teach students a single para-
digm such as traditional imperative programming or functional programming. “Object-
oriented” programming, which many profess to teach, is not really a single paradigm,
and varies greatly according to whether you favour inheritance or delegation, mutable
or immutable types, threads over event queues, etc. The style of programming associ-
ated with the design patterns movement, for example, takes a position on most of these
dichotomies (favoring delegation over inheritance, for example, and making extensive
and often complex use of indirection), and is thus rather a distinctive (if complicated)
paradigm.

This course recognizes the diversity of paradigms, and teaches three distinct paradigms
explicitly. These paradigms are not independent, but are often combined in a single
program.

• *State machine programming* views a program as a finite state machine that responds
to inputs by performing transitions. In a complex system, the states represent con-
trol modes, each of which consists of a large number of constituent states repre-
senting data that is largely orthogonal to the control behaviour. This is the paradigm
on which embedded systems and protocols are built, and is often a good paradigm
to use to understand interactions between a user and an application.

• *Symbolic programming* views a program as a collection of side-effect-free functions
that act on abstract data values. Trees are pervasive, both in the structure of the
computation and the structure of the data; sharing is invisible because there is no
mutation. This is the paradigm that underlies language processing tools such as in-
terpreters and compilers, and embodies the classic abstract-datatype view in which
a programmer extends the built-in repertoire of types with user-defined types, es-
pecially extending the programming language with new primitives, so that design
can be viewed largely as language definition. This paradigm is often called “func-
tional programming”, but I prefer the term “symbolic” to focus on the structure of
the data, and because the course will emphasize first-order and not higher-order
programs, maintaining a clear distinction between functions and data.

• *Relational programming* views the state of the executing program as a relational
heap – each field of a class representing a relation that maps objects to objects.
Mutation is pervasive, and the boundaries between objects are no longer clear. Most “design patterns” (e.g., from the Gang of Four book) require this view, and it is essential for understanding user interface frameworks and data intensive systems in which a group of related classes form a database that is queried by traversals that cross class boundaries.