To the Instructor

Thank you for considering this textbook. This section is intended to help you use it effectively for students at the following levels:

- juniors and seniors in Computer Science taking a one-term cram course in Internet application design (the MIT way)
- juniors and seniors in Computer Science taking a one-year "capstone" course in software engineering
- seniors in Computer Science doing a capstone independent study project or bachelor's thesis
- sophomores in Computer Science or non-majors spending a semester learning about building modern information systems

With respect to these goals, we will treat the following issues: (1) what to do during lectures, (2) how to find clients for your students, (3) what to put on exams, (4) how to find and use alumni mentors, and (5) evaluation and grading.

Before plunging into these issues, let's take a step back and reflect on the rationale for teaching this material at all.

A Step Back

Why is software engineering part of the undergraduate computer science curriculum? There are enough mathematical and theoretical aspects of computer science to occupy students through a bachelor's degree. Yet most schools have always included at least some hands-on programming. Why? Perhaps there is a belief that someone with an engineering degree ought to be able to engineer the sorts of systems that society demands. In the 1980s, users wanted desktop

applications. Universities adapted by teaching students how to build a computer program that interacted with a single user at a time, processing input from the mouse and keyboard and displaying results graphically. Starting in the early 1990s, however, demand shifted toward server-based Internet applications. With 1,000 users potentially attempting the same action at the same instant, the technical challenge shifts to managing concurrency and transactions. Given stateless protocols such as HTTP, software engineers must learn to develop stateful user experiences. Given the ubiquitous network and evolving standards for remote procedure calls, students can learn practical ways of implementing distributed computing.

Once we've taught students how to build Internet applications, it is gratifying to observe their enormous potential. A computer science graduate in 1980 was, by his or her efforts alone, able to reach only a handful of users. Thanks to the ubiquitous Internet, a computer science student today is able to write a program that hundreds of thousands of people will use before that student ever graduates. One of our student teams, for example, built a photo-sharing service launched to the users of photo.net. Through November 2005, the software built by the students is holding more than one million photographs on behalf of roughly 87,000 users.

What Deep Principles Do They Need to Learn?

To contribute to the information systems of the next twenty years, in addition to teaching the material in the core computer science curriculum, we have to teach students:

- object-oriented design where each object is a Web service (distributed computing, demonstrating the old adage that "The exciting thing in computer science is always whatever we tried twenty years ago that didn't work.")
- about concurrency and transactions
- how to build a stateful user experience on top of stateless protocols
- about the relational database management system
- that they're only as good as their last user test

Universities have long taught theoretical methods for dealing with concurrency and transactions. The Internet raises new challenges in these areas. A dozen users may simultaneously ask for the same airline seat. Twenty responses to

a discussion forum question may come in simultaneously. The radio or hard-wired connection to a user may be interrupted halfway through an attempt to register at a site. Starting in 1994 there has been a convergence of solutions to these problems, with the fundamental element of the solution being the relational database management system (RDBMS). At a school like MIT, where the RDBMS has not been taught, this textbook gives an opportunity to introduce SQL and data modeling with tables. At a school with an existing database course, this textbook can be used to get students excited about using the RDBMS as a black box before they embark on a more formal course where the underpinnings are explained.

Scientists measure their results against nature. Engineers measure their results against human needs. Programmers ... don't measure their results. As a final overarching deep principle, we need to teach students to measure their results against the end-user experience. Anyone can build an Internet application. The applications that are successful and have impact are those whose data model and page flow permit the users to accomplish their tasks with a minimum of time and confusion.

What Skills Do They Need to Learn?

In a world where it seems that every villager in India has learned Java, we want our graduates to be more than mere coders. A graduate who can do nothing more than sit in a corner and code Java classes from specs is doing a job that is certain to be sent to a low-wage country eventually.

We'd like our students to be able to take vague and ambitious specifications and turn them into a system design that can be built and launched within a few months, with the features most important to users and easy to develop built first, and the difficult bells and whistles deferred to a second version. We'd like our students to know how to test prototypes with end-users and refine their application design once or twice within even a three-month project. We'd like our students to be able to think on their feet and speak up with constructive criticism at design reviews.

These desires translate into some aspects of how we use this textbook at MIT: real clients so that students are exposed to the vagueness and confusion of real-world problems; user testing built into the homework problems; "lecture" time primarily devoted to student-student interaction, with the instructors moderating the discussion.

Survey Courses Considered Helpful?

Suppose that one were convinced that the foregoing are the correct topics to teach a computer science undergraduate. Should we teach them one at a time, in-depth? Or should we start with a survey course that teaches all the concepts simultaneously in the context of building actual applications (this book)?

Students in a traditional computer science curriculum will

- spend a term learning the syntax of a language
- spend a term learning how to implement lists, stacks, hash tables
- spend a term learning that sorting is O(n log n)
- spend a term learning how to interpret a high-level language
- spend a term learning how to build a time-sharing operating system
- spend a term learning about the underpinnings of several different kinds of database management systems
- spend a term learning about AI algorithms

Students in MIT course 6.001 (Structure and Interpretation of Computer Programs, based on the Abelson/Sussman textbook of the same name) learn all of the above in one semester, albeit not very thoroughly. By the end of the semester, they're either really excited about the challenges in computer science or ... they've wised up and switched to biology.

Survey courses have been similarly successful on the electrical engineering side of our department. In the good old days, MIT offered 6.01, a linear networks course. Students learned RLC networks in detail. But they forgot why they'd wanted to major in electrical engineering. Today the first hardware course is 6.002, where students play with op-amps before learning about the transistor!

One of the most celebrated courses at MIT is the Aeronautics and Astronautics department's Unified Engineering. Here is the first semester's description from the course catalog:

Presents the principles and methods of engineering, as well as their interrelationships and applications, through lectures, recitations, design problems, and labs. Disciplines introduced include: statics, materials and structures, dynamics, fluid dynamics, thermodynamics, materials, propulsion, signal and system analysis, and circuits. Topics: mechanics of solids and fluids; statics and dynamics for bodies systems and networks; conservation of mass and momentum; properties of solids and fluids; temperature, conservation of energy;

stability and response of static and dynamic systems. Applications include particle and rigid body dynamics; stress and deformations in truss members; airfoils and nozzles in high-speed flow; passive and active circuits. Laboratory exposure to empirical methods in engineering; illustration of principles and practice. Design of typical aircraft or spacecraft elements.

Note that this is all presented in one semester, albeit with double the standard credit hours. For almost every topic in the course description, MIT has one or more full-semester courses exclusively devoted to that topic.

Experiences like these led us to develop *Software Engineering for Internet Applications* and the corresponding survey course in building computer systems for collaboration.

Using This Book for a Thesis Project

Most computer science programs require bachelor's candidates to engage in an open-ended development project, either as a "capstone" project or a thesis. Oftentimes the freedom inherent in this requirement serves as a quantity of rope sufficient for a student to hang him or herself. The student might choose to build anything from a graphics system to a compiler. A faculty member supervising the project might have to do a fair amount of work merely to determine what standards are appropriate in the student's chosen area. For example, if it is a compiler project, is it reasonable to expect the student to develop a complete Ada compiler in Lisp in one year? The core of an ML type-inferencer? A simple optimizing modification to gcc?

If you agree with the student to work within the framework of *Software Engineering for Internet Applications*, the project has enough structure that risk is minimized, yet enough flexibility that the student's creativity can flower. For example, using this book means that the student will be using a relational database management system. All of the code that you have to review will be in SQL. Yet the student is free to experiment with the operating system and HTML glue environment of his or her choice. The student will be building an Internet application that has user registration, content management, a discussion forum, and full-text search, and a combination of the book and the public Internet provide a good context for evaluating the student's achievement in these areas. Yet almost any client will put before the student idiosyncratic challenges that should give the student an opportunity to build something unusual.

We consider Mozart to have been creative although he did not develop new musical forms, relying instead on the structure laid down by Haydn. A student will accomplish more if he or she can spend the first months of a project working rather than figuring out what field in which to work, roughly what the scope of the project should be, what tools to choose from an unlimited palette, and so forth.

The One-Term Cram Course

When teaching this material in one semester, it is important that students set up their environments before the first class meeting. A student might have to reinstall operating systems and relational database management systems, contact technical support, or abandon an initial choice of tools.

The One-Year Thorough Course

There are several possible reasons for spreading this material over a full year:

- students who don't appreciate or can't handle a gung-ho pace
- students working individually rather than in teams (more coding per student)
- opportunity to go deeper into some of the underlying concepts and systems
- opportunity to launch services to real users mid-way through the course

If we had an extra semester, we would devote more attention to the inner workings of the relational database management system, demystifying the SQL parser and the various methods for handling concurrency. We would have the students look more carefully at the HTTP standard, possibly building their own simple Web server. We would cover some of the more exotic Web Consortium work, such as semantic Web and RDF and multi-modal interfaces. We would devote more time to performance measurement and engineering. We would push the teams and clients to launch their sites to real users as quickly as possible so that the students could learn from user activity and user feedback.

It would be nice to include a section on high-level formal specification of page flow and data model. Unfortunately, as of 2005, there are no tools available for this that compile into standard executable languages such as SQL and

Java. A quick glance at Unified Modeling Language (UML) might make one think that this is a useful nod in the direction of formal specification of Internet applications. However, UML cannot be compiled into a working system nor can it be verified against a system built in executable languages such as SQL and Java. Even if students mastered the 150 primitives of UML, the only thing that they would learn is that people in the IT industry can get paid high salaries, despite never having learned to write clear English prose. Object Role Modeling (ORM), however, is a high-level formal specification language that looks promising for automatic code generation in the coming years.

A Course for Sophomores

Less mature engineers are going to have more difficulty choosing an appropriate set of tools, more difficulty with tool installation and administration, and are going to be less resourceful in seeking assistance when appropriate. Thus if you are using this textbook with sophomores, it is probably a good idea to reduce flexibility and increase the physical rootedness of the students and the amount of hands-on assistance available.

Juniors and seniors might have had summer jobs working with Oracle and PHP on Debian Linux or with Microsoft .NET and SQL Server. They will probably be most productive if they can continue using their familiar tools. Furthermore, having a variety of tools in use during the semester provides all the students with an opportunity to learn a little bit about other development styles. The main risk to having students choose their tools is that some get sucked in by software vendor hype and elect to use, for example, three-tiered architectures and application servers. At MIT the students have three weeks before the start of the semester in which to install their chosen tools. All of the MIT students who decided to go the application server route were unable to get their systems up and running in time to do the ''Basics'' problem set and hence were forced to drop the class.

For sophomores, it is less likely that students will have extensive development experience with a particular set of tools and the risk of a student choosing an inappropriate set of tools is increased. It may be best to standardize on one set of tools so that everyone in the class is using the same systems.

Universities spend hundreds of millions of dollars on dormitories so that students can drink beer together, but are seemingly reluctant to spend a dime on

shared workspaces for students. This is a shame because for learning most technical material it is much more effective for students to work together and live separately. A student working in a common laboratory with teaching assistants and fellow students nearby won't get stuck on something simple, such as "how do I launch SQL*Plus?" If you can possibly arrange a room with a bunch of desks and PCs and make that the center of your class, this will be an enormous help to less experienced students.

What to Do during Lectures

We try to keep our mouths shut during class meeting times (two 80-minute sessions per week). Students in 6.171 are learning to present their work to other engineers and to offer on-the-fly constructive criticism in response to an engineering presentation by others. If we're talking, they're not learning these skills. At various times in the semester, notably at the beginning of the course, the students won't have anything to present. We might fill a meeting time with a 25-minute lecture on RDBMS fundamentals, followed by a collaborative project in which students break up into teams to solve a data modeling problem.

At a minimum, the meeting room must have one Web browser connected to a video projector. Ideally the room will also have extra Web browsers and keyboards distributed around the room, one for every 3–6 students, and blackboards or whiteboards for collaborative work by small teams.

Here is a sample schedule, the goal of which is to drive the student projects to public launch as quickly as possible:

- *Three weeks before the first meeting* Students informed that they are accepted into the class, thus giving them time to prepare their computing environments. Inform students that they ought to make sure their environment works by building at least one Web page that returns data queried from the RDBMS. They may simply wish to do "Basics" problems 1 through 6.
- *Week 1, Meeting 1* Schedule, grading standards, and other bureaucracy relegated to handouts and a URL reference; we establish a precedent that class time is devoted to engineering. After a 5-minute "welcome to the course" in which we explain what we want them to learn, we give a 15-minute lecture on why online learning communities are important and what are the required elements for a sustainable online community. To get the students accustomed

to the idea that they are going to be speaking up in class, we pick a few examples of online communities from the public Internet and ask students to criticize the features and user interface. We follow this with a 20-minute introduction of the RDBMS. Remind students that they must turn in the "Basics" problems in one week or be dropped from the class.

- *Week 1, Meeting 2* In grappling with the "Basics" problem set, the students have now had a chance to work with SQL. We give a 20-minute lecture on serialization and concurrency control in the RDBMS, pointing out the practical differences between optimistic and pessimistic locking. The rest of the class time is devoted to pitches by prospective clients. The clients introduce themselves and explain what they want to accomplish with their Internet application. Each client should get about 5 minutes. For those projects where the client is unable to present in person, an instructor gives the pitch on behalf of the client.
- *Week 2, Meeting 1* Students turn in the "Basics" problems. Today is the day that you assign teams to clients, and hence today is the day that you decide who is staying in the class. Drop anyone who did not turn in the problem set. They are not capable of building database-backed Web pages and hence are very unlikely to catch up. Most of the class time is devoted to code review on the "Basics" problems. You have secretly been surfing around before class looking at source code from various students. You're looking to get a discussion going on at least the following issues: (a) lack of commenting or identified authorship, (b) error handling in the comparative shopping problem, (c) different approaches to generating unique keys in the face of concurrency, (d) escaping single-quote characters in the search pages, (e) user interface design for the quote personalization system (tables versus bulleted lists, "kill" buttons versus checkboxes and a submit button), (f) different ways of parsing XML. Spend the last 5–10 minutes of class with some hints on working with the client. Students often have the most trouble contacting their client. They'll say "I sent him email a week ago, but he hasn't responded." Remind them to pick up the phone twice per day until they get a phone or in-person meeting with their client.

  (Giving students one week to do the "Basics" problem set seemed harsh to us, and hence we decided one term to give them two weeks to do it. Rather than spreading the work out, the result was that most students did nothing until two or three days before the due date and ended up staying up all night.)

- *Week 2, Meeting 2* Students break up into groups and work on a data modeling problem, e.g., "design an airline reservation system." The specification is open-ended, but you supply English-language queries that they'll have to translate into SQL against their tables and columns. A group can be one project team or two project teams working together. Ideally the classroom will have many separate blackboards. The instructors walk around answering questions and coaching the groups. After 30–40 minutes, you ask two or three of the best groups to present their work. After each presentation you moderate a discussion of the merits of the data model and how much work the RDBMS will have to do in answering the queries. You close the meeting time by introducing the B-tree index and explaining how to add indices to a data model to improve query performance.
- *Week 3, Meeting 1* Students turn in their work on "User Registration and Management" exercises. Class time is devoted to presentation and discussion of different teams' approaches to the "User Registration" chapter problems. At least a couple of teams will have been successful in meeting with their clients and drafting solutions to the "Planning" chapter. Devote 5–10 minutes of class time to discussing the work of the farthest-along teams in this area as a way of inspiring the rest of the class.
- *Week 3, Meeting 2* Students turn in their work on "Planning" and Exercises 1 through 3 in "Content Management" (up to but not including the skeletal implementation). Class time is devoted to presentation and discussion of teams' approaches to content management data models. Consider breaking up into teams to take a single-table data model and put it into Third Normal Form.
- *Week 4, Meeting 1* Devoted to look-and-feel criticism of public Internet applications and the more advanced teams' projects.
- *Week 4, Meeting 2* Students complete all exercises in "Content Management," including client sign-off. Class time devoted to team presentations of work so far and plans for immediate future.
- *Week 5, Meeting 1* Students complete all exercises in "Software Modularity." Class time devoted to team presentations of their design decisions and documentation.
- *Week 5, Meeting 2* Students complete exercises in the "Discussion" chapter up to, but not including the usability test.

- *Week 6, Meeting 1* Students complete all exercises in the "Discussion" chapter except execution of the refinement plan. Class time devoted to discussion of usability test results and whether the numbers could have been predicted from the page flow and HTML designs.
- *Week 6, Meeting 2* Students present their refined discussion forum systems. Class time devoted to presentation of the refined systems. Close with an exhortation that students spend the weekend starting the "Mobile" and "VoiceXML" problems in parallel so that if they are stuck with the tools they'll have an early warning.
- *Week 7, Meeting 1* Students complete all exercises in the "Mobile" chapter. Class time devoted to presentations and discussion of the wireless interfaces to the applications.
- *Week 7, Meeting 2* Students complete all exercises in the "VoiceXML" chapter. Class devoted to presentations and discussion. It would be very helpful to have an amplified telephone system so that the entire class can hear interactions between a team's system and a user.
- *Week 8, Meeting 1* Students complete all exercises in the "Scaling Gracefully" chapter. Take-home mid-term exam handed out (an individual rather than a team project). Class discussion of scaling exercises, ideally starting with each answer being presented by a separate team.
- *Week 8, Meeting 2* Exercises 1 and 2 from "Search" due. Discussion of team designs for full-text search.
- *Week 9, Meeting 1* Mid-term exam due. All exercises from the "Search" chapter due. Class time devoted to discussion of exam questions, answers, and implications.
- *Week 9, Meeting 2* "Planning Redux" exercises due. Note that the instructors must interview the clients as part of this chapter. Team presentations of their work and plans for public launch.

What to Put on Exams

You might think that exams are unnecessary in a project-oriented course such as this one. We give exams for the following reasons:

- we want to make sure that a student isn't being carried by his or her teammates

- we want to make sure that students are reading and re-reading the principles outlined in this textbook
- we want to make sure that students understand data modeling and concurrency
- we want to see if a student is capable of writing good analyses of Internet applications and compelling justifications of his or her design work
- by giving take-home exams rather than in-class quizzes we are able to create an experience that will add to the students' skills

A good style of question involves asking the students to try out a particular public Internet service and then build a data model that would support what they've just seen. The students should then load their data model and try to solve some SQL puzzles against them.

Another good question asks the students to visit a public Internet application, try it out, and write a critique of the user experience. In our exam we include the following admonition: "Your critique should be clear concerning what is wrong with the current system. Your critique should be explicit about what to change, such that a junior programmer could implement your improvements without depending on his or her own taste and judgment."

You might also want to ask the students to propose and justify a hardware and software architecture to handle a specific service and user load.

Note that all of these questions are sufficiently open-ended to lead to interesting classroom discussion. Note further that these exams must be graded by someone experienced with software engineering and data modeling.

Finding Clients

A real-world client has much to offer your students. A real-world client will phrase problems in vague and general terms. A real-world client will bring content and users to flesh out what would otherwise be a purely academic exercise. A real-world client can provide students with performance feedback. A real-world client forces students to confront the challenge of demonstrating their achievement to a non-technical audience.

What can your students offer real-world clients? In some cases, a student team will build a launchable, documented, maintainable, high-performance system that the client can run for years. This happy result, however, is not neces-

sary in order for a client to get value from participating in a course based on this textbook. Oftentimes working with a student team will enable a client to make decisions and formulate precise specifications. Most people are unable to make good decisions about information systems without seeing a prototype. We don't promise clients that their student team will solve their problem, but we do promise clients that the experience will clarify their goals and, whatever else, will be over in 3.5 months.

Working groups within your own university can be a good source of clients. Groups that need to work with off-campus people, such as alumni, parents, or colleagues at other institutions, are especially logical candidates for online community support. Non-profit organizations can also be good sources of projects because they are usually much more patient than for-profit corporations and can afford to (a) wait for your semester to start, and (b) start over if necessary at the end of your semester in the event that the student team does not produce a launchable system. For-profit organizations can provide well-organized and highly motivated clients. Both cash-starved startups and small neglected departments within larger companies may be attracted to working with a student team. With any potential client, however, try to make sure that they have enough resources to gather content and users.

A bit of diversity among the client projects is nice, but at their cores all of the client projects should be online communities. At the very least, a project needs to have a discussion forum where User A can ask a question that User B will answer. Much of the value in this course comes from student teams comparing their differing approaches to the similar challenges of user registration, content management, and discussion support. If a client wants a 100-percent voice interface, their team won't be able to learn from other teams very effectively nor will other teams building primarily Web browser sites be able to learn from the voice-browser-only team. If a client says "I want an online store," just respond "no." If a client says "I want an online store where the customers talk to each other," respond with "Okay, but the students aren't going to build the checkout pages until the end of the term, and you'll have to offer them summer jobs if you want e-commerce admin pages."

Here are some criteria for selecting among clients:

- spirit of the project; does it look like an online learning community in which the users share a common purpose and the more experienced will teach the less experienced?

- availability of magnet content and users; is the client dreaming or does he or she have compelling unique content that will draw users or some other way of bringing users to the application?
- availability of the client; the university calendar is unforgiving and the client needs to be able to respond within 24 hours to a request for a critique
- long-term resources; it is great if students can go into a job interview and say "point your Web browser at http://www.foobar.org to see what I built," but this won't happen unless the client has the long-term wherewithal to host and maintain an Internet application

Alumni Mentors

In 1950 tuition at Ivy League schools was about $500 and the average new car cost nearly $2,000 (4 times tuition). In 2003 tuition is approaching $30,000 per year and a beautiful Honda Accord can be had for $15,000 (1/2 of tuition). Thanks to improvements in design and manufacturing engineering, the relative price of an automobile has fallen by a factor of 8 while its quality has improved dramatically. Why has the cost of a university education soared relative to automobiles and other manufactured goods? Consider the classroom circa 1950: 25 students, 1 teacher, 1 blackboard, 25 chairs. Compare to the classroom experience circa 2005: 25 students, 1 teacher, 1 blackboard, 25 chairs. Even if universities were to exercise restraint in the hiring of administrative staff, the cost of tuition is doomed to outstrip inflation because education is the only industry in America where there are no productivity improvements.

This problem is not too severe for teaching Physics 101. The school pays one instructor and fills a room with 300 tuition-paying students. But teaching software engineering effectively requires that students be given an apprenticeship. No school will want to pay the army of instructors that would represent an optimum-sized teaching staff for a software engineering project course like this one. Even if a school had an infinite amount of money, professors and graduate students are probably the wrong people for the job. How much experience does the average academic computer scientist have in comparing a collection of software source code to a statement of user requirements and suggesting improvements?

We can solve the staffing and expertise problems in one stroke by bringing in alumni volunteers. A typical school has 10 or 20 times as many alumni as

current students. If students are broken up into teams of 3 and each volunteer can assist two teams, we only need to convince approximately 1 percent of our alumni to volunteer each semester. As working software engineers, our graduates will likely do a much better job of assisting students than a fresh graduate student would and perhaps even a better job in some areas than a seasoned professor.

A course based on *Software Engineering for Internet Applications* is uniquely amenable to alumni mentoring because all of the students' work is accessible from any Web browser anywhere on the Internet. Between the plans and the /doc directory and the mandated "View Source" links at the bottom of every student-authored page, an alumnus 3,000 miles away ought to be able to contribute almost as effectively as someone who is willing to come down to campus two nights per week.

Evaluation and Grading

The daily cost of attending a top university these days is about the same as the daily rate to stay at the Four Seasons hotel in Boston, living on room-service lobster and champagne. It is no wonder, then, that the student feels entitled to have a pleasant experience. Suppose that you tell a student that his work is substandard. He may be angry with you for adversely affecting his self-esteem. He may complain to a dean, who will send you email and invite you to a meeting. You've upheld the standards of the institution, but what favor have you done yourself? Remember that the A students will probably go on to graduate school, get Ph.Ds., and settle into $35,000/year post-docs. The mediocre students are the ones who are likely to rise to high positions in Corporate America, and these are the ones from whom you'll be asking for funding, donations of computer systems, and so on. Why alienate paying customers and future executives merely because they aren't willing to put effort into software engineering?

In teaching with *Software Engineering for Internet Applications*, you have a natural opportunity to separate evaluation from teaching. The quality of the user experience and the solution engineered by a team is best evaluated by their client and the end-users. If the client responds to the questionnaire in Exercise 3 of the "Planning Redux" chapter by saying "Our team has solved all of our problems and we love working with them," what does your opinion matter?

Similarly if a usability study shows that test users are able to accomplish tasks quickly and reliably, what does your opinion of the page flow matter? During most of this course we try to act as coaches to help our students achieve high performance as perceived by their clients and end-users. We use every opportunity to arrange for students to get real-world feedback rather than letter grades from us.

The principal area where we must retain the role of evaluator is in looking at a team's documentation. The main question here is "How easy would it be for a new team of programmers, with access only to what is in the /doc directory on a team's server, to take over the project?"