A New Approach to Teaching Programming

Daniel Jackson and Rob Miller
Version of February 2, 2009

We have developed a new undergraduate programming course, which has so far been taught three times. This paper explains the motivations of the course, describes its content and organization, and reports on our experiences to date. The key innovations of the course are: an emphasis on design and the transition from problem to solution; separation of concepts into three programming paradigms; the use of non-trivial case studies to motivate concepts; and early introduction of current technologies. The result is a course that differs both from traditional programming courses (because of its emphasis on design and project work) and from many software engineering courses (because of its emphasis on product over process).

1 Context

The undergraduate curriculum in computer science and electrical engineering at MIT has been completely redesigned in the last few years. As part of this redesign, we developed a new 'foundation course' in programming, *Elements of Software Construction*. This course, which is referred to in MIT style by its number, 6.005, is one of several elective 'foundation courses' that students take, usually as sophomores, after completing at least one of the new 'core courses' (6.01 and 6.02) that introduce fundamental concepts in both electrical engineering and computer science. The other foundation courses in computer science are introductions to computer architecture (6.004) and algorithms (6.006).

The design of 6.005 posed many challenges. First and foremost, it follows in the footsteps of two remarkable courses, *6.001: Structure and Interpretation of Computer Programs* [2] and *6.170: Laboratory in Software Engineering* [10], both of which were mandatory for computer science students in the old curriculum. Any material in those courses that does not appear in 6.005 is thus perceived as dropped from the curriculum. Moreover, both courses have inspired advocates and admirers over many years who are suspicious of any new course with a different intellectual agenda.

Second, 6.005 serves multiple audiences with different backgrounds, interests and aspirations. Although the course is designed for students in Electrical Engineering and Computer Science, about a third of the class comes from other departments. Some incoming students have hardly programmed at all; their exposure is limited to some small Python programs they wrote in 6.01. Others have programmed extensively in multiple languages. Our emphasis on design models requires the ability to think abstractly, but most students enter the course with minimal mathematical sophistication. The institute imposes a requirement on all freshmen of two calculus courses, which makes it hard for a departmental curriculum to find space for other areas of mathematics. Consequently,

our department's own math courses (6.041, 6.042) must serve multiple areas, and the basic concepts that are most useful in a software engineering course – propositional and predicate logic, structural induction, graphs and relations, etc – are covered only in one (6.042), and even then not every time the course is offered. As a foundation course, 6.005 must inculcate big ideas of lasting value, but must also, as the main programming course, teach the basic skills that students need for summer jobs, subsequent courses, and, for many, a career in software engineering.

Third, the very nature of programming presents an educational dilemma. There seems to be a mismatch between theory and practice. Our most mature and most elegant theories (of data abstraction, specification, verification, types, etc) seem to be disconnected from the daily practice of programming, where the difficulties lie less with small semantic details and more with the challenges of understanding and piecing together large, unwieldy (and poorly understood) libraries. Students, moreover, are eager to become familiar with common technologies (such as user interface toolkits, networking libraries, web languages, etc), which might be seen as taking space – both temporal and intellectual – from more fundamental topics.

2 Course Philosophy

Our course design rests on three fundamental premises. These premises comprise our response to the challenges just described, and our vision of what a foundational programming course should be.

Theory and practice together. We believe that the dichotomy between theory and practice is a false one. Teaching practical tools and techniques without a sound conceptual basis is building on sand. Students who learn how to use software technologies in a vacuum rarely come to grips with them in any fundamental way. They become mired in irrelevant details, and are unable to identify the key differentiators between technologies or designs. They are prone to develop poor habits, building exclusively by trial-and-error, and never fully understanding the limitations of their own inventions. On the other hand, theory without practice is likely to be misdirected. Unless concepts are applied to realistic examples, students find it hard to see their relevance, and emphasis will tend be placed not on the ideas that are most valuable but on those that are most readily formalized, or that provide the best fodder for exam questions.

Design as the focus. We believe that the central intellectual challenge of programming is *design*: grappling with the vagueness and complexity of engineering by articulating a clear problem, inventing the key elements of a solution, and assembling and evaluating an embodiment. Design problems, unlike algorithmic problems, do not tend to have easily comparable solutions, so recognizing different metrics and making appropriate tradeoffs between them is essential. In industrial practice, design is central; the best software systems are distinguished by their designs, and development failures are inevitably due to design-level problems (whether in formulating requirements, structuring the implementation, or specifying components). In addition to being the most important skill for students to acquire, design is also perhaps the hardest. The ability to complete the code of a specified procedure comes easily to most students, and seems

to need little tutoring. But the ability to articulate the essence of a problem, to invent robust and appropriate abstractions, to recognize tradeoffs, and – perhaps most importantly – the recognition of the value of simplicity (and the extraordinary effort needed to achieve it) come far less easily. Focusing on design is attractive for another reason too: while some of the skills imparted in a programming course are rather specialized, design skills are applicable in many fields.

Software is multifaceted. To handle the variety of problems that arise in practice, an engineer needs to be comfortable with multiple paradigms – and with the idea that different problems sometimes need very different approaches. In many curricula, the introductory programming course presents a single view of programming that tries to unify all programs in a single paradigm. Classic examples are formal-methods-inspired courses in which programs take an imperative form and are obtained by refining specifications; functional programming courses (in languages such as ML, Scheme and Haskell) that emphasize type structure and control of side-effects, and courses (such as 6170) that view programming as the invention and elaboration of abstract types. These approaches have the great advantage of focus and clarity, and, can be balanced by other courses with different perspectives. But in a curriculum that requires just one programming course, we believe it is important to expose students to multiple paradigms. 6.005 is structured into three segments, each covering a single paradigm, and each basic concept is introduced in one of the segments. An alternative is to present a similar collection of concepts in a mix-and-match approach (using design patterns, for example). In our opinion, this is an opportunity missed; the world views that underlie the concepts go by unnoticed, and the concepts themselves are harder to explain in the larger context. For example, in 6.005 the idea of inductive invariants is introduced in the context of state machine programs, and revisited later when discussing datatype representations and object models (where in other courses it may be introduced for the first time, in a far more complicated setting). As another example, immutability is introduced in the context of recursive datatypes and functions, where it is much easier to understand and appreciate than in the more typical context of full blown object-oriented programs.

3 Course Structure

The course is organized into three segments, each corresponding to a paradigm. For each paradigm, we teach students how to articulate the problem to be solved (in a suitable lightweight modeling notation) and then how to implement a solution, using a collection of design patterns that capture the typical expertise of developers versed in that paradigm. Each paradigm has the following components:

- A series of six 80-minute lectures, organized around a single substantive case study (discussed below) and smaller examples, starting from problem modeling and moving to design patterns and finally implementation details. There were also supplementary lectures on topics that did not fit neatly into a paradigm: a lecture on testing and one on debugging; a lecture on usability; a lecture on concurrency; and 3 lectures introducing Java and illustrating how to work with the Eclipse environment.

- A case study illustrating the key ideas, notations and patterns. The students attempt these by themselves in advance of their presentation in class. We call these assignments 'explorations'; their purpose is to help students appreciate each problem before we show a solution, and to give them at least one other solution (namely their own) to compare with ours.
- A project in which students work in groups of three on a software development from problem formulation through design to implementation and testing. The projects are loosely specified; in our experience, the disadvantage of not being able to test student solutions automatically is far outweighed by the advantage of giving them the experience of more open-ended problem solving. The last project was slightly more ambitious than the others, and prizes were awarded to the best solutions.
- One or more project work sessions, in which teams met in the lecture room and worked on their projects with teaching assistants and faculty at hand to provide help if needed. In the first two offerings of the course, we found that teams had difficulty finding time to get together and projects were often delayed. By using standard class times for these sessions, we were able to relieve teams of some of their scheduling burden and encourage early starts on projects.
- A laboratory, in which students worked individually on a tightly scripted series of small tasks to learn the basic tools and technologies they would require in their project work – for example, how to use network sockets, or how to set up and execute unit tests with JUnit. The purpose of these laboratories was to give students the skills and basic familiarity they needed as efficiently as possible, minimizing the amount of time and effort devoted to ideas that have little long-term value. Undergraduates (many of whom had taken the course previously) were hired as laboratory assistants to help students complete their tasks.
- Weekly 50-minute recitations led by graduate student teaching assistants. Typically, a few concrete problems were presented and worked on by the students and then discussed. The teaching assistants were also available for office hours and project work sessions (see below).
- A problem set consisting of five or ten small problems, each involving constructing a small model or writing some code. These, along with the recitations, were an innovation of the third offering of the course, after we realized that students needed some warming up to become comfortable with new ideas, techniques and notations before applying them in projects.

4 Lecturing Strategies

Although most of the learning in the class happens in the projects, we believe that lectures are important, as a means of conveying basic notions and illustrating how to tackle problems. We attempt to motivate all concepts and techniques with concrete examples. We start with a practical problem; explain why it's hard and interesting; show a solution; and only then attempt to set the solution in a more general and abstract context. The idea of structuring the course around substantive case studies reflects in part the influence of our predecessor 6.001, which showed students components of a runtime system

(such as a metacircular evaluator, assembler, and garbage collector). Our case studies are taken from applications further afield from computer science, because we felt this would engage a wider group of students, and because it offered the opportunity to show how to grapple with less well-defined and more open-ended problems. Our use of explorations reflects our pedagogical realization that the easiest way to convince students that a problem is hard and interesting is to give them a chance to solve it themselves first; we found that otherwise it is very hard to convey the excitement of a challenging problem.

Beyond the case studies, we use a variety of smaller examples. Even with these, we have tried to adhere to some basic principles to ensure that they are motivating and educational:

- *Clarity*. Each example should have a clear purpose (usually in illustrating one or more difficulties, concepts or techniques), and should be – as much as possible – free of distractions and unnecessary complexity.
- *Realism*. It is necessary but not sufficient that an example reflect real problems that arise in engineering practice. It should be entirely free of contrivances, so that the student can *recognize* it as a real problem.
- *Pragmatics*. The solution should be developed pragmatically, so that students can see what an engineer might do in real work. It should not be idealized, with more effort and polish than is achievable at reasonable cost in practice, nor should it fail to meet typical standards. In short, it should be capable of presentation *without apologies* of any sort.

This last point raises a dilemma with respect to 'clever' solutions. Obviously, we strive to show our students solutions that are not only practical but also elegant and surprising. In this way, an example can also be a vehicle to illustrate a clever idea or pattern that is incidental to the problem being solved. But if the solution is too clever, students may be distracted, or discouraged by the thought that they could never have invented such a solution themselves.

In addition to showing examples of good solutions, we try to show bad solutions for contrast, but we have not included as many of these as we would have liked. Most often, they are fragmentary solutions that are presented as strawmen along the path to a final solution. We realize that it would be preferable to show more than one complete solution, and explain why one (or more) is better than the others.

Presenting code in lecture is challenging. Like many of our colleagues, we have found that showing slides (made in Powerpoint or Keynote) has mostly detrimental effects, making the lecture less interactive, and encouraging the lecturer to speed through material without explaining it carefully. On the other hand, we find the challenge of organizing material into slides (and especially choosing good titles) very helpful. We therefore generally prepare slides in advance of lectures, deliver the material on a traditional blackboard, and issue the slides online in lieu of lecture notes. When a case study calls for showing code samples longer than a few lines, we have found it too tedious to write the code on the board. In these situations, we either project the code on a screen above the blackboard, or we hand it out on paper. Students like to have the slides on paper during the lecture so they can write on them directly; this then allows the code to be

included in the paper slides. We are not sure if this is a good idea, and have sometimes used instead a separate handout for code that allows a program to be shown in full on several sheets of paper.

Engaging students in a large lecture class is hard. We try to avoid long monologues; we ask frequent questions and we have students work with partners on small problems during lecture. We have found that just introducing one or two such problems into each lecture, each taking less than ten minutes (including ensuing discussion), is very productive. Not only does it seem to keep students alert and interested, but it also has on several occasions revealed to us our own gross misapprehensions of the students' level of understanding.

Lecture attendance in our department is quite low: typically 30-60% for a class in which attendance is not mandatory. We have made some effort to meet with groups of students to try and understand why this is so. We suspected that we were not succeeding in making lectures as effective as they might be, and that students find they learn better by other means. Although we could undoubtedly improve our lecturing, this appears not to be the main issue. Students who don't attend lecture invariably tell us that they know it damages their understanding, and that their attempts to learn the material from notes are not very productive. The students who attend lectures are usually the stronger ones, and they are the same students who attend recitation and take advantage of teaching assistant office hours. In the end of term course evaluation, we asked the students about their lecture attendance: amusingly, we got no usable data from this because it seems that the students who don't attend lecture likewise don't complete surveys.

In one of our meetings, some students told us that this didn't surprise them: that students either are motivated (and will take advantage of every learning opportunity the course offers) or are not (and will skip anything that is not mandatory). They felt that our efforts to increase lecture attendance were misguided, and that we should focus all our attention on the students who attend, not the ones who do not. This seems to be a harsh attitude, but it may be the most productive. At the very least, this interchange made us wonder whether some of our challenges lie beyond the particular course being taught.

5 The Paradigms

The course teaches three paradigms, which correspond very roughly to traditional imperative programming, functional programming, and object-oriented programming. Each paradigm, however, is intended to be more focused and conceptually coherent than a traditional course. The first paradigm, for example, ignores data structures entirely and focuses on states and transitions; the second covers algebraic datatypes and functions, and never considers mutation; the third treats the essence of object-oriented programming as computing over a *relational* heap.

5.1 State Machine Programming

Programs are viewed as state machines that interact with the outside world through discrete events. Machines are modeled using a simple transition diagram notation with a semantics based on CSP (machines synchronize by sharing events, and may perform non-shared events independently, and the behavior of a machine is represented as a trace set, ignoring the complexities of refusals) and a syntax based on Statecharts (with hierarchical nesting of states and parallel combination). Basic patterns for implementing state machines in Java are named and explained – such as *Object as Machine* in which a Java heap object represents a state machine, with a method for each event – as well as some more complex patterns – such as *Object as State* [the Gang of Four *State* pattern6] in which each state is represented by an object.

An alternative view of machines as grammars is then presented, and we teach an elementary form of the JSP method [9, 4] for synthesizing code from regular grammars. Aside from its benefit in software engineering, this material is also important because no other required course teaches basic notions of grammars, regular languages, regular expressions, parsing, and so on.

The case study is a 'midi-piano' that allows tunes to be played on a computer keyboard using Java's built-in MIDI library, and recorded and played back. A small but instructive complication is that an understanding of how the keyboard generates key repeats is needed; we model these and design a simple machine to absorb them. A bigger complication arises from the requirement that it be possible to record keys pressed during playback, so that recordings can be layered. Our solution uses an event queue to merge key presses and playback events, which is read by a single thread thus sequentializing the main functionality of the program.

Other lecture examples included a client that downloads and parses weather information (to illustrate Java basics), a toy traffic light system (to illustrate inductive invariants), a stock quoting widget with RTF and HTML output (to illustrate modularity and module dependences), and a console program that poses quiz questions and tallies the score (to illustrate JSP).

The project is to build a downloading client for a BitTorrent-like system, in which segments of a file are obtained (via HTTP) from multiple servers, with locations given by manifest files that are themselves stored in a distributed fashion. The manifests give alternative servers to be used if a request times out. We required that the client offer an API that satisfies the Java *Stream* interface, which causes a nasty complication since it requires that (in JSP terminology) the machine be 'inverted' with respect to the input. One might argue that this complication is self-inflicted, since it would be far less painful in a language other than Java that offered iterators (such as CLU or Ruby) or coroutines. It is nevertheless an instructive problem to solve.

5.2 Symbolic Programming

Computation is viewed as the application of functions to symbolic values. We start with the notion of recursive, algebraic datatypes, and model tuples, options, variants, lists and trees as simple ML-style datatype declarations. Recursive functions are defined by

cases on the datatypes. Creation of corresponding Java classes is then explained using patterns – which we give names such as *Variant as Class* – based on Felleisen and Friedman's book [5]. Standard Gang of Four patterns for implementing functions over datatypes (such as *Interpreter* and *Visitor*) are shown. The idea of data abstraction is approached through representation independence, by remarking on the significance of the equals sign in datatype declarations. Representation invariants and abstraction functions are introduced, which students are encouraged to 'implement' using *repCheck* and *toString* methods. Structural induction is motivated by the need to reason about rep invariants and the problem of defining a good unit test suite.

The case study is a Sudoku player that works by translation to SAT. Fortuitously, a very basic DPLL solver using only unit propagation and case splitting is sufficient to solve standard Sudoku puzzles in a few seconds. The basic clause and literal datatypes gave nice opportunities to discuss questions of representation and encapsulation, and to illustrate more complex patterns (such as *Factory Method* and *Facade*). This example demonstrates well the advantages of immutability; writing the solver using mutable clauses is far harder, and the obvious implementations will perform far worse (because they cannot exploit sharing and thus perform unnecessary copying). Students found this case study the hardest; many were not familiar with the idea of backtracking search and had trouble grasping it.

In this segment of the course, we also give a lecture on how to avoid debugging (using assertions, modular development with unit testing, code reviews, etc) and strategies for doing it if all else fails. A lecture on 'little languages' (inspired by 6.001) shows how to represent (and play!) Pachelbel's canon using a simple language of musical constructors and combinators. Students find this exciting, despite the ugliness of representing higher-order functions in Java.

The project is to build a player for ABC [1], an ASCII notation for music that is almost as expressive as traditional staff notation. We identified a subset of the ABC language that was sufficient to allow many online files to be played, especially of folk melodies (although unfortunately not the second movement of Beethoven's 7th [3] which required a number of small features missing from our subset, such as ties, slurs, grace notes, and broken rhythms). Building this program amounts to building a compiler for a rather rich expression language, and reinforced students' skills not only in datatypes and functions, but also in JSP and state machines (for the lexing and parsing).

5.3 Relational Programming

Computation is viewed as reading and writing a data store represented as a collection of relations, either implemented as a relational database, or as the heap of an object-oriented program in which each field of each class is viewed as a binary relation. This perspective more honestly reflects the reality of most object-oriented programs, in which objects are not values of a mutable datatype but are nodes in a graph, and in which methods are not simple mutators of local state but rather may cause changes to ripple through the graph. Although layering is desirable, dependences are often cyclic, and objects form patterns with complex interactions.

The idea of relational state and relational invariants is introduced at the problem level with object models, using the semantics of Alloy [8] and the diagrammatic syntax of OMT [11], to represent the underlying conceptual structure of a program. Implementation is presented as transformations from object models to Java code, using patterns that are widely used but not usually named, such as *Relation as Field* (which maps a relation in the object model to a field of a class) and *Relation as Hashmap* (which maps a relation to a hash map in a singleton object). The *Model View Controller* pattern is explained, along with the concurrency model of user interface toolkits such as Swing, and supporting patterns such as *Composite* (for the view hierarchy) and *Observer* (for widget listeners). Implementation of an object model as a relational database is shown, with queries and updates expressed in SQL.

The special concerns of mutable types are explained, such as representation exposure and abstract aliasing, and how to implement equality; we also explain how storage leaks arise in Java. We also give a special lecture on usability, covering basic design principles (such as learnability, visibility, efficiency and simplicity), iterative design, sketching and paper prototyping, and user testing. Although we did not require that the students use these techniques in their projects, some of them did so voluntarily. The subject of user interfaces also gives an opportunity to return to the subject of concurrency, where we give a brief comparison of shared-memory and message-passing paradigms, and again advocate the use of message-passing with blocking queues as the preferred pattern.

The case study is the design and implementation of a simple photo organizer that offers the ability to display photos as an array of thumbnails and organize them into collections (as found in Apple iPhoto, Google Picasa, Adobe Lightroom, Microsoft Expression Media, etc). This problem illustrates the value (and subtlety) of conceptual modeling – how to determine what a collection is, for example, and what should happen when a photo appearing in multiple, nested collections is deleted from one – and presents non-trivial challenges in user interface implementation. In fact, in our own solution we struggled to get a particular Swing component to refresh at the expected time; Swing in general is not an easy thing to master. The backing store for this application is easy to design – perhaps too easy – so it puts little pressure on the object model transformation patterns.

The project is to build an instant messaging system. Students design the features (such as buddy lists, chatrooms, conversations and status indicators), the user interface of the client, and the protocol between the client and server. Since this is the final project, students are given a little more time and are encouraged to compete for prizes for best design, best new feature, etc. Despite having barely three weeks from start to end, this project is quite challenging, and involves networking, graphical user interfaces, and concurrency (both in client and server). All teams achieve a basic working system, many incorporate whacky features (such as the ability to type Latex at one end and see formatted PDF at the other), and the user interfaces are often quite polished. One team even finished a beta version early enough to conduct user testing before making a final round of changes. As a frivolous rejoinder to our efforts to encourage code reuse, several teams added the ability to play the midi-piano from the first case study over an IM channel.

6 Discussion and Reflections

Overall, our impressions of how well the course has fared are positive. We believe that we have achieved our goal of making the course design-centric while still ensuring that students get adequate practice in nuts-and-bolts programming. The inclusion of an array of practical technologies seems to have given the students an essential 'cultural literacy' without diluting the conceptual content of the course.

Whether we have succeeded in our goal of fusing theory and practice is not clear. On the one hand, students report (in anonymous course evaluations) that they enjoyed the projects and explorations and learned a lot from them, and several told us that the course had changed the way they approach problems. On the other hand, problem sets were unpopular, and many students complained that they were too open-ended. Moreover, the teaching assistants reported that, despite the many examples in class and recitations, and practice in problem sets, many students were still unable even in the final project to construct well-formed models (eg, of state machines) that captured essential properties at an appropriate level of abstraction. We suspect that the kind of abstract thinking and focusing that modeling requires cannot be taught without a much greater investment of time and effort than is possible in a course such as 6.005. Perhaps what is needed is a discrete math course as a prerequisite that focuses on modeling (in the style, for example, of CMU's *Models of Software Systems* [7]).

The teamwork experience seems to have been a good one for most students. In the third offering of the courses, we introduced a team-building exercise for the final project. Some students were skeptical but others reported that it was useful. We plan to move the exercise earlier in the term so that it precedes the first team project.

In the first offering of the course, we had students work in pairs on six projects. This worked well, perhaps because of a small class with a pioneering spirit. In the second offering, which was much larger – 160 students rather than 25 – we paired the students up randomly (but without repetitions). This was a disaster: two-person teams are far too fragile, and many teams failed because the partners were not compatible or because one was uncooperative. We also made the mistake of thinking that all the classwork could be project-based. In the third offering, we reduced the number of projects by half and introduced individual problem sets and explorations. We augmented team mentoring with conventional weekly recitations, which not only helped the students but also gave the teaching assistants a more substantive pedagogical role.

Diversity of programming experience in incoming students remains a problem. About 80% of the incoming students have already programmed in Java in high school; the remaining 20% find the class very hard. In three versions of the course, we have tried three ways to bring these students up to speed: special labs, self-study with graded exercises, and a series of special lectures at the start. None of these solve the problem, perhaps because they all address knowledge of the programming language rather than the fundamentals of programming, which cannot be acquired overnight.

In contrast, our perennial concern at MIT that we will fail to get through to hackers (who come in thinking that software engineering has nothing to offer, and that sheer

willpower can overcome any programming problem) was assuaged, as suggested by this (not atypical) comment:

> *I had experience in C++, Python, Objective-C, Java, and various web-related languages and mark-up languages. I started a software company over the summer... The amazing aspect of this course is that it made me realize how much of programming I didn't know–there is a great difference between knowing languages and knowing how to program. So, I plan to go back and rewrite much of the code written over the summer to increase modularity and maintainability.*

We are generally happy with the multi-paradigm nature of the course, especially in the way it allows us to focus more deeply on fundamental concepts when they are more cleanly separated. The state machine paradigm is the least popular with students, perhaps because it seems least novel to them, or because they are more familiar (and motivated by) information-centric than control-centric applications. Of the projects, the instant messaging system is by far the most popular; many liked the ABC player, but some disliked it for the parsing component. Contrary to expectation, not a single student complained about the lack of a game as a project, which had been a mainstay of earlier software engineering courses at MIT.

Until this point, we have not discussed our choice of Java as a programming language, in part because we believe that the choice of language in programming courses is a less important issue than often claimed. We chose Java for its libraries (Swing, collections, networking, etc), tool support (Eclipse, JUnit, Javadoc, etc) and industrial use. Since our students use Python in their core courses, we thought it important for them to see a different language in 6.005, and in particular to be exposed to static typing and interfaces. We are of the opinion (along, it seems, with many in the object-oriented programming community) that inheritance is usually a mistake, so we emphasize interfaces over subclassing. Java is a powerful language and thus in a sense well suited to presenting multiple paradigms, but at the same time it has become complicated and unwieldy, and we are cognizant of the fact that using Java eliminates some of the aesthetic satisfaction that is experienced when programming in a more succinct and elegant language. Nothing, it seems, is easy in Java (consider what you need to know just to read a file), and unfortunately some aspects of the language design (such as the lack of iterators as found in Ruby or CLU, and the awkwardness of exceptions) militate against good programming practice. Amusingly (but sadly) students sometimes revel in complexity; in the course evaluations, several students mentioned the *Visitor* pattern as one of the highlights of the course, blissfully unaware that visitors are needed only because Java lacks pattern-matching.

Acknowledgments

Thanks to Hal Abelson, Srini Devadas, Mike Ernst, Matthias Felleisen, Eric Grimson, John Guttag, Nancy Lynch, Tomas Lozano-Perez, Ron Rivest, Peter Szolovits, Seth Teller, George Verghese and Steve Ward for their advice, suggestions, encouragement and criticism; to Albert Meyer, for helping to coordinate 6.005 and 6.042; to Saman

Amarasinghe, who co-taught the second offering of the course; to Sivan Toledo, who developed some Java self-study material; to Carlos Pacheco, who developed the initial versions of several of the course projects; to our teaching assistants – Jongmin Baek, Erdong Chen, Vicky Chou, Harold Cooper, Max Goldman, Saba Gul, Eunsuk Kang, Aseem Kishore, Alice Oh, Scott Ostler, Clayton Sims, PJ Steiner and Kuat Yessenov – who helped with developing projects and infrastructure as well as running the course; to our lab assistants and graders; to Maria Rebelo and Lisa Bella who helped with administrative aspects; and of course to our students, for tolerating a course in flux, and for their enthusiasm, engagement and constructive evaluations.

References

[1] ABC Music Notation. http://abcnotation.org.uk.

[2] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[3] Steve Allen. *Beethoven's Symphony No. 7, Movement 2 (in ABC notation).* http://www.ucolick.org/~sla/abcmusic/sym7mov2.html.

[4] John Cameron. *JSP & JSD: The Jackson Approach to Software Development.* IEEE Computer Society Press, 1989.

[5] Matthias Felleisen and Daniel P. Friedman. *A Little Java, A Few Patterns.* MIT Press, 1998.

[6] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[7] David Garlan, Alan Brown, Daniel Jackson, Jim Tomayko and Jeannette Wing. The CMU master of Software Engineering core curriculum. in *Software Engineering Education*, Lecture Notes in Computer Science, Vol. 895. Springer, 1995.

[8] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2006.

[9] Michael Jackson. *Principles of Program Design*. Academic Press, 1975.

[10] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[11] James Rumbaugh and Micheal Blaha. *Object-oriented modeling and design.* Prentice Hall IPE, 1991.

paradigm	**state machines**	**functional**	**relational**
models	statecharts grammars	recursive datatypes	object models
sample patterns	Machine as Object State as Object Stream Synthesis	Variant as Class Interpreter Visitor	Relation as Map Subset as Subclass Observer
case study	Midi Piano	SAT-based Sudoku	Photo Organizer
team project	Peer-to-peer Downloader	ABC Music Player	IM Client & Server
sample ideas	atomic events interleaving invariants	inductive types data abstraction rep invariants	mutable objects heap invariants interface decoupling

Course structure with sample patterns and ideas.