Introduction to Interactive Programming
by Lynn Andrea Stein
A Rethinking CS101 Project

Preface

Interactive Programming is an introduction to computer programming intended for students in standard CS1 courses (or interested professionals) with no prior programming experience. It is the first textbook to rethink the traditional curriculum in light of the current interaction-based computer revolution. *Interactive Programming* shifts the foundation on which the teaching of Computer Science is based, treating computation as *interaction* rather than *calculation*, thus providing students with a solid grounding in the thought that underlies modern software practice. Students still learn the basic and necessary elements of computer programming and the Java language, but the context in which they learn it is more consistent both with Java's tools and philosophy and with the prevailing practice from which it arises.

Why Interactive Programming?

Traditionally, introductory programming teaches algorithmic problem-solving. In this view, a program is a sequence of instructions that describe the steps necessary to achieve a desired result. The 'pieces' of this program are these steps. They are combined by sequencing. The program produced is evaluated by means of its end result. Students trained in this way often have difficulty moving beyond the notion that there is a single thread of control over which they have complete control.

In contrast, most programs of interest today are made up of implicitly or explicitly concurrent components that interact to provide ongoing services. Buzzwords such as "client/server" and "event-driven" are part of the descriptive language of this new generation of programs. Embedded systems and software agents typify their incarnations. User interface design, distributed programming, and the world-wide web are logical extensions of a way of thinking that has interaction at its core.

When programming is taught from a traditional perspective, important topics like these are treated as advanced and inaccessible to the introductory student. It is unsurprising that senior software engineers report that today's undergraduates are ill-equipped to handle the realities of embedded interactive software. Most require on-the-job retraining to "think concurrently." Students trained in the traditional curriculum are often so indoctrinated in the "sequence of steps" mentality that they can no longer rely on the intuition common to every child coordinating a group of friends or trying to sneak a cookie behind her parent's back.

Interactive Programming provides an alternate entry into the computer science curriculum. It teaches problem decomposition, program design, construction, and evaluation, beginning with the following premises: A program is a community of interacting entities. Its "pieces" are these implicitly or explicitly concurrent entities: user interfaces, databases, network services, etc. They are combined by virtue of ongoing interactions which are constrained by interfaces and by protocols. A program is evaluated by its adherence to a set of invariants, constraints, and service guarantees -- timely response, no memory leaks, etc.

Because it begins from this alternate notion of what programming is about, *Interactive Programming* tells a rather different story from the traditional introductory programming book. By its end, students are empowered to write and read code for client-server chat programs, networked video games, web servers,

user interfaces, and remote interaction protocols. They build event-driven graphical user interfaces and spawn cooperating threads. Each of these programs -- all of which are beyond the scope of traditionally taught introductory courses -- is a natural extension of the community metaphor for computation.

Many computer science departments are contemplating a change to the Java programming language for introductory computer science courses. While it is possible to make this change without transforming the introductory curriculum, adopting Java without a corresponding curricular change amounts to sweeping more and more of what is important in today's computational world under the rug. Java embodies much of modern programming practice. Insisting on traditional approaches actually makes certain aspects of the language less accessible. Shifting to a curriculum in which concurrent interacting entities play a central role makes far more of modern computation theory, practice, and tools accessible to today's introductory student.

A more complete argument for rewriting the introductory computer science curriculum in this way is contained in "What We've Swept Under the Rug: Radically Rethinking CS1" (*Computer Science Education Journal*, to appear). See also http://www.ai.mit.edu/projects/cs101/.

Ramifications for Later Curriculum

Interactive Programming includes a number of topics not often taught to introductory students: networks, user interfaces, client/server architecture, and event-driven programming. At the same time, students will develop a basic facility for programming and for problem decomposition, the most crucial skills taught in most existing CS1 courses.

In all respects, this course is still an introductory programming course. Its thematic lesson concerns a model of computation as interaction, rather than calculation. But its pragmatic goals include most of the skills that are learned in standard introductory CS. The fundamental lesson of this course remains how to take a description of a problem and construct a program whose behavior solves that problem. It differs from traditional courses in its underlying assumptions, the kinds of descriptions that can be considered, and the corresponding conceptualizations that are used to build a program. The computational constructs and modeling tools have changed; the problem still remains the programming.

As a result, this new CS1 course requires little revision of the rest of the computational course sequence. Upper level courses can continue as they are, but are likely to find their task simplified somewhat by the new perspective that students bring to them.

The remainder of the curriculum which begins with an introduction to computation on these terms may thus look much like the existing computer science undergraduate curriculum. Nonetheless, there are subtle but significant improvements. Several important topics that are currently covered only in advanced undergraduate or graduate level classes can be introduced earlier in the curriculum. For example, topics in distributed algorithms and parallel complexity -- such as the time/processor tradeoff -- can be taught in the first course in computer science theory if the model of parallel computation is already familiar. Since modern algorithms increasingly makes use of such approaches, it seems only natural to expose our undergraduates to the fundamental ideas in these areas.

Other topics, already present at the undergraduate level, become much easier to explain when students come equipped with this world view. Much of operating systems becomes an exploration of different methods for implementing and ensuring appropriate behavior multiprocessing, rather than focusing on the

concept of parallel execution itself. Students seeing these ideas for the second time, now in depth, are more likely to appreciate some of the subtleties of the problem rather than being confused by the many levels at which operating system code must operate. Synchronization and interprocess communication can be introduced along with scheduling. Transaction-safety, remote procedure call, and shared memory models similarly follow smoothly from this approach.

Further, a whole host of issues that now fit into our curriculum poorly, if at all, now become sensible parts of the model of computation that we teach our students. For example, the traditional curriculum has a tremendously difficult time introducing the topic of user interfaces. In many schools, this "special case" is tacked on to the curriculum as an afterthought (or altogether ignored), largely because it just doesn't fit. To readers of this book, however, accounting for the role of the user becomes straightforward. The user is another member of the community of interacting processes that together constitute our computation. The programmer's job is to develop an acceptable interface that gives each participant -- program or person -- an appropriate set of responsibilities and services. Of course, a human has different skills and needs from a computer program, but this, too, is a natural part of our larger way of thinking -- and teaching -- about computational systems.

Teaching computation this way also has the potential to harness our students' natural instincts. Traditional introductory courses tell their students, "Forget all of your intuitions about how the world works. This is computation; it is nothing like the world in which you live." Instead, *Interactive Programming* teaches that computation is very much like the world in which we live. It harnesses our intuitions about that world---about simultaneity and ordering constraints, about when it is more useful to partition a task and when it is simpler not to, and about what information must be available to whom at what time and how to get it there---and teaches readers to use that intuition to become better programmers.

A Short History of the Rethinking CS101 Project

This book is a part of a larger project to reshape the ways in which introductory computer science is taught (and, indeed, the ways in which the field itself is conceptualized). The Rethinking CS101 Project grew out of work in a variety of computational fields -- artificial intelligence, robotics, software agents, human-computer interaction, as well as programming languages -- and their common difficulties with the conventional wisdom concerning how computation is constituted. For example, introductory computer science teaches that a program's job is to calculate some desired result and then to stop. When a robot stops, however, this is generally a sign that it has broken. (Further, there's not really a "result" that the robot "calculates"; instead, it is supposed to continually exhibit appropriate behavior.)

Research Roots

In the early 1990s, the author worked to bring intuitions about computation into the classroom through the use of simple, inexpensive robotics. The use of robots enabled a focus on software life cycle, non-repeatability, and pragmatic software engineering uncommon in traditional introductory classrooms. The curriculum that developed from this experimentation marked a radical departure from the traditional single-threaded, sequentialist story.

The use of robotics clearly forced a shift in perspective in the introductory programming curriculum. In the first half of the decade, this shift was echoed, if more subtly, in the popular software market through approaches such as event-driven programming, client-server architectures, and enterprise computing.

Those techniques -- increasingly important to industry -- were still not deemed suitable for an introductory computing classroom. Nonetheless, they were inescapably changing the face of the computing sciences. Computing-in-the-raw is no longer calculate-and-stop. Instead, it is made up of agents and services, communities of ongoing interacting entities. Yet today's introductory classrooms shed little light on these now-prevalent industry practices.

Courses taught during this period included MIT freshmen, MIT graduate students, and international researchers in artificial intelligence. Spin-offs of these efforts include robotics classes at a variety of universities and colleges as well as the now-annual Robot-Building Laboratory at the National Conference on Artificial Intelligence and the establishment of the KISS Institute for Practical Robotics (of which the author is an Institute Fellow).

With the advent of the world-wide web and the popular adoption of Java, a new avenue towards teaching these approaches has been opened. The current Rethinking CS101 Project has shifted its focus away from physical robots and towards the underlying principles of interactive computation as illustrated by purely software systems. (A side effort within the project continues to pursue the robot hook, both in software simulations and in the interests of capitalizing on the newly emerging commodity robot market. Although robots are not central to the curricular shift represented by this project, they are easily integrated into its methods and models.) *Interactive Programming* represents the codification of the underlying approach to computation in a form suitable for adoption in otherwise-traditional university computer science curricula, thereby bringing them closer to state-of-the-art practice.

Classroom Experience

The curriculum presented in *Interactive Programming* has been taught in a variety of venues. The first course taught with the current set of materials was held in the summer of 1996, in a one-week intensive minicourse using the Java 1.0 API and Sun's JDK, the only Java available at the time. Its students were executives, managers, and a few software engineers enrolled in MIT's Summer Professional Programs. The majority had no substantial prior programming experience.

The course was subsequently taught twice in MIT's regular curriculum. Students were largely first-semester freshmen and others with no prior programming experience. (The course is also popular among advanced students in non-computational fields who want a single semester of computational coursework.) Student feedback has been resoundingly positive. The MIT course has been adopted by the EECS Department as a regular offering and is listed in the catalog as subject number 6.030, Introduction to Interactive Programming.

Precursors to this textbook were also used in teaching several other minicourses to professional audiences. These include the 1997 and 1998 Professional Institutes at MIT and a tutorial offered at the ACM SIGPLAN's Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA '97). Students in these courses included software professionals, academics, and trainers. Generally versed in traditional programming, they attended the minicourses to learn a new way to think about computation.

Other instructors have used the beta release of the textbook. In the fall of 1998, the course materials was used at a handful of undergraduate institutions with student bodies substantially less sophisticated than MIT's, as well as an advanced class in a secondary school. Serious beta testing began in the fall of 1999,

when over a thousand students at more than a dozen colleges and universities around the world used *Interactive Programming* as their primary text. Additional non-traditional classroom tests are also underway. Ultimately, the textbook is intended for deployment in mainstream undergraduate classrooms as well as certain advanced secondary classes, perhaps AP.

The curriculum itself has attracted widespread attention. It has been presented at a variety of international meetings and its agenda is documented in a variety of publications (see enclosures). The Rethinking CS101 Project at MIT has recently received the donation of a 30-machine teaching laboratory from Microsoft Research/University Curriculum Programs. A strategic relationship with Sun Microsystems is also under negotiation, and the National Science Foundation has selected Rethinking CS101 for an Educational Innovation Award.

How to Use This Book

Interactive Programming is designed for use by students who have no prior programming experience (typically college freshmen). It ultimately teaches both the fundamentals of computer programming and the details of the Java programming language.

The book is divided into five parts. The first briefly overviews the idea of programs built out of communities of interacting entities. The second part introduces the mechanics of Java programming, from things, types, and names to objects and classes. It is essential to the book and is intended to be read in the order presented. Part three elaborates on these ideas, introducing threads as first-class citizens of the programming world and exploring inheritance, exception-handling, and design. Part four emphasizes a variety of issues in the design of an individual entity. It is not necessary to read this section in any particular order, and certain chapters can be omitted entirely without serious detriment. Part five similarly surveys a variety of interrelated topics, in this case concerning the ways in which communities are coupled together, and its chapters, too, can be taken out of order or omitted.

The five parts, taken together, constitute a single-semester introductory course in computer programming. In such a course, some of the supplementary material (described below) will not be used. For a one-quarter course, part five and selected earlier chapters should probably be omitted. Alternately, the complete book can be spread over two quarters or over a full year, augmented as necessary from the supplementary materials.

Part By Part

Part 1 is brief and introductory, providing an overview of the approach to computer programming taken. Part 2 begins with the basic syntax and semantics of programming constructs. At the same time, from the earliest examples, students are introduced to concurrent, interactive, embedded programs. For example, interfaces are introduced early as they specify a contract between two parts of a computer system. By the middle of part 3, students have learned to write what might in other contexts be called "stand-alone" programs -- complete programs including class definitions and a main routine. They have also learned that every program is a part of a system of interacting entities -- including the user, libraries and other software, hardware, etc. -- and that no program truly stands alone.

The remainder of the book addresses issues and alternatives that arise in the design of software communities. Part 4 focuses on ways to extend the basic entities that students build. The notion of a

dispatching control loop provokes an exploration of procedural abstraction, in which separate routines handle each possible case. This in turn leads to a de-emphasis of the central control loop and a shift to event-driven programming, in which individual "handler" procedures take center stage. In a typical event system, dispatch may be provided implicitly, i.e., by underlying hardware or software. A third model -- smart objects that handle their own behavior -- is also explored. Java's AWT is introduced as both a tool and an example of an event-based system.

Part 5 addresses the issue of how entities are tied together. A recurring theme -- throughout the book, but emphasized here -- concerns interface design. This refers both to the Java construct -- a signature specification, introduced in chapter 4 -- and to the more general concept, including human (user) interface design. In addition to learning how to specify an interface, students learn what the interface does not specify. In other chapters, students learn about streams, messages, and shared memory, about connecting to objects in the same name space and to those running under different processes or on different machines, and about how to communicate with them. They also learn the basic ideas of safety and liveness, that shared mutable state can lead to program failures, and some simple mechanisms for coping with them. They do not, of course, learn to build arbitrarily complex programs that avoid deadlock under all circumstances. This topic will be visited later in the computer science curriculum. Instead, they learn to recognize the general preconditions for the possibility of safety failures and the kinds of solutions that might be possible. The goal, throughout this course, is to give students the basic conceptual vocabulary that will allow them to ask the right questions as they meet more complex issues later in their education.

Interactive Programming ends with an overview of various patterns of large-scale systems architecture, reviewing tradeoffs among various approaches and providing a common language for software architects. The last chapter examines conventional patterns by which complex concurrent and distributed systems are constructed. The emphasis is on designing and understanding a variety of interactive communities. This chapter also leads naturally into final projects. In courses taught using this curriculum and preliminary drafts of the book, typical final projects have included client/server chat programs and networked video games. Not what you would generally expect from first semester freshmen!

Pedagogical Elements and Supplementary Materials

Although this book is primarily intended for an introduction to computer science course, it will include enough reference material to stand alone as a self-study course in Java, without requiring a language supplement. Three kinds of supplementary materials help provide this support: in-chapter sidebars, between-chapter interludes, and auxiliary case studies. Reference charts and a glossary are also included.

To avoid muddying the text with too many language-specific details, sidebars are used throughout to explain details of Java syntax and semantics. The text explicates the conceptual development of the ideas; the sidebars are intended to provide detailed information on technical aspects of the language or the programming process.

Sidebars come in two flavors. Syntax sidebars explain language-specific details and pragmatics in the form of a reference manual. Style sidebars explain good documentation and coding practice. The use of sidebars serves two purposes. First, it frees the main text of some of the details that confuse rather than elucidate the presentation of central concepts. Second, the sidebars, together with the reference charts in Appendix B, form a supplementary desktop reference for students while they are programming.

The narrative of the book is periodically interrupted for an extended example, called an interlude. Interludes are adapted from potential programming assignments. They are presented between chapters, rather than within them, and can be included or omitted at the instructor's preference. Interludes provide detailed illustrations for the student to study. They exemplify the themes of the course in terms of the material studied to that point. They also provide the basis for exercises allowing students to practice and assess their mastery of relevant skill sets. Complete code for each interlude is supplied on the textbook's web site.

Also supplementing the book is a set of case studies. These are not included within the bound text. Instead, they will be made available over the world-wide web. The case studies provide descriptions of current applications exemplifying the principles central to the course. For example, one case study is based on an article in the trade literature on constructing an http server. With only minor modification, this article is an excellent illustration of the relevant themes of the course as well as a concrete example of a real-world application that is accessible to students in the later chapters.

In addition to the materials described above, the supporting materials include a set of exercises, lecture notes, programming assignments, and sample quizzes. Some exercises appear chapter by chapter in the bound book. Other resources are available through the online supplement.

About the Author

Lynn Andrea Stein is Associate Professor of Computer Science and Engineering at the Massachusetts Institute of Technology, where she has been a member of the faculty since 1990. She is also a member of both MIT's Artificial Intelligence Laboratory and its Laboratory for Computer Science. Stein, an internationally recognized researcher and educator, has been teaching computer science since the early 1980s. Stein received her undergraduate degree (AB cum laude in Computer Science) from Harvard and Radcliffe Colleges and both ScM and PhD degrees from the Department of Computer Science at Brown University. While at Brown, she held an IBM Graduate Fellowship and received the Sigma Xi Graduate Student Award.

Stein's research work spans a variety of fields and her publications include seminal work in fields as diverse as the semantics of sharing in object-oriented programming languages, non-monotonic taxonomic reasoning, and cognitive architectures for robotics. Within the past year, Stein has given invited addresses at the National Conference on Artificial Intelligence, the European Meeting on Cybernetics and Systems Research (W. Ross Ashby Plenary Lecture of the International Federation for Systems Research), and the Consortium for Computing in Small Colleges Northeastern Conference (Keynote Address). She will present a keynote address at SIGCSE's European analog, the International Conference on Innovation and Technology in Computer Science Education, in Helsinki in July, 2000.

Stein is a 1993 recipient of the National Science Foundation Young Investigator Award. She recently ended a term of service as an Executive Councilor of the American Association for Artificial Intelligence and is the former Chair of that organization's Symposium Committee. She currently serves on several international advisory and steering committees.

Stein has recently returned from a on sabbatical leave from MIT as an Office of Naval Research Science Scholar at the Mary Ingraham Bunting Institute of Radcliffe College, a multidisciplinary think tank in Cambridge Massachusetts.

Acknowledgements

This document would not have been possible without the generous support of the National Science Foundation under Young Investigator Award No. IRI-9357761 and more recently under Educational Innovation Award EIA-9979859. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. Additional support was provided by the Massachusetts Institute of Technology's Classes of 1951 and 1955 Funds and by the Department of Electrical Engineering and Computer Science, by Microsoft Research, and by Sun Microsystems. The initial revision of these notes was supported by the Office of Naval Research through the Science Scholars Program at the Mary Ingraham Bunting Institute of Radcliffe College, where the author was on sabbatical leave as a 1997-98 and 1998-99 Fellow.

The earliest inklings of these ideas probably took root while I was teaching&emdash;and learning to teach&emdash;at Harvard. There is no doubt that that they were nourished in the dynamic environment of Brown's CS Department, which supported many parallel growths. Both my students and my teachers over the years have taught me more than I can say, and the debt I owe them cannot begin to be repayed. My more recent colleagues, at MIT and elsewhere, have been a source of inspiration and challenge, both of which have strengthened the work immeasurably. Feedback, especially concerning industrial relevance and especially from the networked community, has been invaluable.

A significant portion of the writing of this book was completed in the wonderfully nurturing environment that is Radcliffe's Mary Ingraham Bunting Institute. I could not possibly do justice to what that space and time meant to me. Suffice it to say that the Bunting Fellows of 1997-99 are a most remarkable cohort, in whose debt I will forever remain. I hope that this project will be one more testament to the power of Polly Bunting's campaign against the "climate of unexpectation."

Many people have contributed to the development of these ideas; here I can only single out a few individuals to whom I owe the greatest debts. Early versions of some of these ideas were developed jointly with Jim Hendler, and his continued support has been of tremendous benefit. Hal Abelson has been both mentor and inspiration throughout this project, and I'm sure I don't begin to appreciate the extent of his contributions. Kim Bruce was among the work's earliest (and greatest) supporters. Robert Duvall has been a fellow traveller along this road.

I have had tremendous assistance from the teaching staff who have helped me over the years with the development of this material: Ben Adida, Alfred C. Ashford, Joshua Reuben Brown, Duncan Bryce, Jennifer Chung, Daniele De Francesco, Matt Deeds, Robert Duvall, Mike Harder, Craig Henderson, Stephanie Hong, Pavel Langer, Emily Marcus, Paul Njoroge, Todd Parnell, Ben Pick, Salil Pitroda, Lydia Sandon, Luis F. G. Sarmenta, Emil Sit, Maciej Stachowiak, Ben Vandiver, Mike Wessler, Nathan Williams, and Henry Wong; also Matt Domsch, Carol Lee, Karsten Ulland, and Anne Wright, who helped me with an earlier but crazier experiment.

The students who have participated in the several versions of this course -- 6.80s, 6.75s, conference tutorials, the various versions of 6.096 and 6.030, and those at other institutions whom I have yet to meet -- have left their mark in untold ways upon this material. I am indebted to them for their insights as well as their patience.

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>