Preface

My nine-year-old niece is a computer programmer: Using her father's PC and Disney's "Print Studio" software, she constructs programs that print greeting cards for her friends. My niece has no training in art, graphic design, or computer programming, yet she programs greeting cards to her satisfaction—she practices "naive programming."

In similar and ever increasing fashion, naive programmers use visual, "drag and drop" languages to program applications for home, school, and office use. Naive programming will play a crucial role in satisfying the exploding demand for software, but there will always be limits—just as no hospital patient would submit to a surgery conducted by a naive surgeon, no customer of a complex or safety-critical system will settle for software written by anyone but a properly trained, professional programmer.

This textbook is meant for a first course for future professional programmers and computing scientists.

What makes "professional programming" different from naive programming? The answer lies in use of *structures*—control structures (sequencing, conditionals, iteration, parallelism), data structures (arrays, stacks, trees), and component structures (modules, classes, objects, packages). Professional programmers understand how to employ properly all three forms of structure; amateurs do not. Further, patterns of structures-within-structures define *architectures* that are learned and regularly imitated by professionals. As indicated by its title, this text presents standard architectures for component construction and patterns of control construction and data construction.

The text takes a "modern" approach by emphasizing component structures over the other two forms. Computing has matured into a distributed, component-based activity, where both computer hardware and software are assembled from standardized components and connected together by means of standardized interfaces. By no means does this text ignore the classic developments of control and data structures—they appear in due course at an appropriate level of detail. But component-level issues drive the software design process, and the text emphasizes this fact.

Java is used as the programming language in this text because it supplies solid support for component-structure-driven design; its control and data structuring mechanisms are adequate as well. Because Java and its support library are huge, only a carefully selected subset that promotes sound programming techniques is presented.

To the Student

As the previous narrative indicates, learning to program requires more than merely learning to write in a particular computer language—you must understand the structures within programs and how these structures behave. To do this, you must pore over the book's programming examples, copy them to your computer, test them, try to "break" or "trick" them, modify them in small ways, and try them again. In many ways, computer programs are like toys or appliances that can be examined, played with, disassembled, and reassembled in different ways. Experiences like these help you develop programming intuitions.

Most sections in the text end with a short exercises section that suggests simple ways to apply and modify the programs in the section. *Work at least one or two of the exercises before you proceed to the next section,* and if you have difficulty with an exercise, do not hesitate to reread the section. It is rare for anyone to understand a new concept after just one reading, and a technical topic like programming requires careful, thoughtful effort for deep understanding. Remember that progress is measured by the number of concepts and structures that you can use well and not by the number of pages you have read or number of programs you have typed.

Each Chapter concludes with a collection of projects that test your abilities to design and build complete programs. The projects are roughly ordered in terms of difficulty, and many are written so that you can "customize" them into a product that you or others would enjoy using—take time to make your final result one you would be proud to demonstrate to others.

Like video recorders and microwave ovens, programs come with instructions. When you study the example programs in the text, pay attention to the "instructions" (documentation) that are included. Because programs and program components are useless without documentation, you must develop the habit of documenting the programs you write. This activity pays huge dividends when you start building large, complex programs.

The Java programming language is used as the working language in this text. Java is by no means perfect, but it supports the crucial structures for programming, and it provides mechanisms for "fun" activities like graphics and animation. Chapter 2 states partial instructions for installing Java on your computer, but if you are inexperienced at installing software, you should seek help.

The programming examples from the text and other supporting materials can be found at the URL `http://www.cis.ksu.edu/santos/schmidt/ppj`.

To the Instructor

My experiences, plus the extensive feedback I have received from the Scott/Jones reviewers and my colleagues, have caused the text to evolve into an implementation of the following algorithm:

1. Convince the students that programs have architectures, like houses do. Tell them programming is a learned discipline, like house design and construction.

2. Start students sending messages to objects immediately. Amuse and motivate them with a bit of graphics, but don't overwhelm them with Java trivia.

3. Teach the students class design and component assembly via interfaces before the students get lost in loops.

4. Use control structures and array data structures to build "smarter" objects.

5. Finish with interesting applications—GUI-driven programs, database systems, interactive games, animations, and applets.

The ordering of Step 3 before Step 4 is crucial, because it encourages the component-driven approach to programming.

Here are some pragmatic issues and how they are handled by the text:

- *Design:* Beginners learn by imitation. For this reason, the text uses simplified versions of the Smalltalk Model-View-Controller (MVC) software architecture for its programs. (The MVC architecture structures a program so that its model component handles the computational duties of a program, the view components(s) handle input-output transmission, and the controller controls the transfer of data from view to model and back.) This architecture helps a beginner see how a program is assembled from components and how components are designed so that they can be easily unconnected, reconnected, and replaced. The beginner can readily imitate this architecture in her own projects.

  As part of the design process, components are first specified with UML/Java-style interfaces before any code is written. (An interface lists the names of the public methods and private attributes a class needs to do its job.) For complex applications, UML class diagrams are drawn.

- *Documentation:* All programs are documented in Sun's "javadoc" style, and the reader is shown how to use Sun's `javadoc` tool to automatically generate Web-based documentation pages (the so-called "Application Programming Interface"—"API") for her programs. UML class diagrams document the program's overall architecture.

- *Pedagogy:* Chapters are organized into Essentials-Projects-Supplement components. The Essentials sections present the central concepts of that chapter; included with these sections are exercise sets that guide the student through basic applications of the topic. The chapter concludes with a section of programming projects and multiple Supplement sections, which provide technical details that a student or instructor can skip the first time through the text.

- *Applications:* When using Java for programming examples, it is tempting to emphasize graphics, animations, applets, and networking, for which Java provides ample support. But not all programming fundamentals are best taught with these applications, so a middle ground is taken: The text presents a mixture of information processing examples and graphics examples. Animations and applets appear in due course. Networking is not a beginner's topic and is not covered.

The text does *not* use any specially written classes or packages supplied by the author—only the standard Java packages are used. This prevents a beginner from becoming dependent on nonstandard variants of Java and relieves the instructor of the headache of installing custom packages on classroom computers and students' personal computers.

Although the choice of Java as the text's programming language is basically sound, the language possesses several annoying features. One that simply cannot be avoided is using the static method, `main`, to start an application's execution. To avoid tedious explanations of static methods and classes from which no objects are ever created, the text naively claims in Chapter 2 that an application's start-up class (that is, the one that contains `main`) generates a "start-up" object. Technically, this is incorrect, but it allows a beginner to stick with the axiom, "Classes Generate Objects," which is promoted from Chapter 1 onwards. The remainder of the text presents the syntax and semantics of Java in technically correct detail.

The programming examples from the text and other supporting materials can be found at the URL `http://www.cis.ksu.edu/santos/schmidt/ppj`.

Acknowledgements

First and foremost, I thank Gudmund Skovbjerg and his students at Aarhus University, Denmark, who used several earlier drafts of this text. Their comments led to huge improvements in the book's organization and pedagogy. Vladimiro Sassone and his students at Catania University, Italy, and Peter Thiemann and his students at Freiburg University, Germany, also used early drafts of the text, and they thanked as well. I've also received useful comments from Thore Husfeldt and his students at the University of Lund University, Sweden, Aake Wikstro"m and his students at the University of Gothenburg, Sweden, and Sebastian Hunt at City University, London. I also thank my co-instructor, William Shea, my graduate teaching assistants, and my students at Kansas State University for tolerating numerous revisions over a multi-year period. Bonnie Braendgaard of Aarhus University, Carolyn Schauble of Colorado State University, and Bernhard Steffen and Volker Braun of Dortmund University are thanked for their insightful suggestions. I also appreciate the comments and criticisms of my departmental colleagues, Michael Huth and Stefan Sokolowski.

Richard Jones and Robert Horan of Scott/Jones Press deserve special thanks for their initial interest in the text, their tolerance of my rewritings, and their recruitment of the following review team, whose commentaries led the text into its final form: REVIEWERS' NAMES HERE.

The book's first draft was written while I spent part of a sabbatical year at Aarhus University, Denmark; I thank Olivier Danvy for hosting my visit. Subsequent drafts were written during periods when I was supported by the National Science Foundation, the Defense Advanced Research Projects Agency, and the National Aeronautics and Space Administration; I thank my project managers, Frank Anger (NSF), Helen Gill (DARPA), and Mike Lowry (NASA) for tolerating what I hope will be judged a substantial contribution to programming.

Finally, during the period of time this book was written, my mother, Frances Louise Walters Schmidt, died; I dedicate the text to her.