PREFACE

TO THE STUDENT

Each of you brings to the table a year or more of study in a particular object-oriented programming language. Some of the subtleties of those languages you have already encountered, probably enough of them so that you have already experienced the difficulty of mastering some of the intricate concepts of object-oriented programming languages. Mixed in with the concepts of inheritance, polymorphism, generic programming, and several variants of information hiding and access control, there are hundreds of smaller design decisions which impact these larger issues and impact each other as well. Most of these design decisions are treading in well-worn paths, because imbedded in these newer languages are myriad concepts which have been used in languages of the past.

Your instructor has carefully selected those chapters on which you are going to concentrate, but I encourage you to read and study the other chapters as time permits. There is great beauty in a consistent programming language design, and there is much to be learned even from some of the apparently poor design decisions which were made. All programming languages have flaws, but usually those flaws were deliberately accepted by the designer in order to achieve a specific goal. You may agree or disagree with the goal, but it is important to grasp the reasons why a design decision was made.

Event-driven programming can be frustrating and challenging, and certainly great fun. It is included here because it is an important tool for producing more useful and creative software, but also because it is a showcase for the advantages of object-oriented design. From the beginning graphical user interfaces were a major motivation for the design of object-oriented languages, because the hierarchy of types of objects seen on a display device (and interacted with using a mouse and keyboard) strongly suggests a hierarchy of software types. We dip into several EDP environments in this text, and our purpose is always to not only give the reader the flavor of how it feels to develop applications in that environment, but to actually cover enough of the details so that medium-sized projects can be undertaken and completed by the student within the framework of a semester of college.

Omitted from the historical context presented in this text are a great variety of languages with unique and creative designs which have motivated today's

designs. A short list of missing languages includes APL, LISP, PROLOG, FORTRAN, COBOL, Snobol, Pascal, and Ada. Contrary to popular opinion, *none* of these languages is "dead." Variants of them are still in use, and it is likely that they will continue to grow and evolve and be of continuing usefulness indefinitely. The reason for their longevity lies in their inventiveness. At the core of each lies an idea that motivated the design, and in all cases where the idea was clear and communicated well, there was staying power.

If you enjoy this text, you will also enjoy studying the design and implementation issues encountered in other languages, object-oriented or otherwise, and you are strongly encouraged to do so. It is likely that during that study some new types of design solutions will take shape in your mind that would otherwise never have occurred to you. Since the invention of the wheel, our tools have shaped our thinking. For this reason, programming language design lies at the heart of the discipline of computer science. I hope this text motivates you to continue your study in this very important area.

TO THE INSTRUCTOR

The last twenty years have seen a dramatic change in computer science curricula, engendered in part by the introduction of object-oriented languages into those curricula and by the near-universal use of such languages as a tool for teaching software design and construction principles to Computer Science majors. The large majority of departments use either Java or C++ as the major teaching language in their curricula, and object-oriented (OO) programming languages (OOPLs) as a whole have become very important in our curricula. Because of the breadth and density of concepts in this set of languages, and the fact that each is accompanied by a sophisticated library of container classes, classes for graphics and event-driven programming, and other concept-heavy classes, there is now room within the OO paradigm to discuss a large part of what we have always taught in the Programming Languages course, while building on a broad, established base of familiarity on the part of our students. This text is meant to be a resource for instructors wishing to teach programming language design and implementation from that perspective.

The student using this text should have completed a year of study in an object-oriented programming language. The text generalizes from that year of experience in order to present the foundational principles of programming languages, drawing on the students' OO background while widening their perspectives. It uses a historical approach to link the older procedural languages to current

object-oriented languages, tracing the roots of the latter to their origins in Simula 67. The main body of the text teaches a selected set of object-oriented languages from the standpoint of language design and implementation, beginning with the historically significant and highly original Smalltalk language. Enough information is presented to allow the students to see the depth and power of each language, and to be able to write some interesting programs.

Recognizing the growing importance of language-integrated libraries, the author presents in the context of each language a sampling of library types, including the standard types of container classes and associated iterators. In the same way, in keeping with the historical and current symbiosis between OO languages and event-driven programming (EDP), the text includes an early chapter on the basics of EDP, and follows up on that theme by including a detailed introduction to an EDP library for each of the languages covered.

The historical theme is reinforced in a pivotal chapter on C++ and Java similarities, tracing those languages back to their roots in C. The student is made to see how dramatically two languages can diverge from a common base, given two different sets of design goals. Further chapters complete the study of C++ and Java and provide a detailed and critical introduction to C# and Python. Common EDP themes such as graphics, animation, and user controls are introduced for each language, using as vehicles the MFC library (C++), Swing (Java), Windows Forms (C#), and Tkinter (Python).

The object-oriented paradigm isn't perfect, and it is certainly true that a design is not necessarily good just because it is object oriented. But the object-oriented languages taken as a group are a very rich and rewarding study. The sheer number of concepts involved in learning Java or C++ or any other object-oriented language, and the requirement we now place on our students to learn those concepts early in the curriculum, has enriched our curricula. The accompanying demand on our students for a higher degree of sophistication has changed the way we teach programming and software engineering.

A course based on this text makes a natural alternative or companion to the traditional Programming Languages course. A variety of ways suggest themselves to accommodate such a course into the curriculum. One idea is to offer a two-course sequence in Programming Languages, using this as either the first or the second course. Or two alternative courses could be offered, perhaps during alternate years. On the other hand, if this course is used as the only Programming Languages course at the undergraduate level, there is ample material in it to satisfy the ACM curriculum requirements. Along with a good theory course, it should prepare the student well for the study of the theory, design, and implementation of programming languages at the graduate level.