Inside Smalltalk

Volume I

Wilf R. LaLonde
School of Computer Science
Carleton University

John R. Pugh
School of Computer Science
Carleton University

Prentice-Hall International, Inc.

Printed in the United States of America

10 9 8 7 6 5 4 3 2

ISBN 0-13-468430-3

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice-Hall, Inc., *Englewood Cliffs, New Jersey*

Table of Contents

Preface

INTRODUCTION

In the seventies, structured programming revolutionized the way programmers constructed software systems. Today, many are predicting that the object-oriented programming paradigm will be the second major revolution in software engineering and that object-oriented systems will become the predominant programming tools of the nineties. In the two volumes of **Inside Smalltalk**, we take an in-depth look at the Smalltalk-80 environment — the programming system that most consistently adheres to the object-oriented paradigm and that has served both as a model for object-oriented extensions to existing languages and as the basis for a new generation of languages supporting inheritance. It can be argued that Smalltalk has had more impact on software development in the last decade than any other programming language. Smalltalk fosters the notions of *programming in the large* and *programming by extension* rather than by *re-invention*. Smalltalk provided the foundation for window-based graphical user interfaces, for the development of truly reusable class libraries, and for the introduction of on-line tools such as code browsers. Our objective in **Inside Smalltalk** is to provide a comprehensive survey of the Smalltalk environment, the language, and the library. A secondary goal is to show how interactive graphical applications can be constructed using object-oriented programming techniques and the unique Smalltalk programming environment. Moreover, we show how Smalltalk's underlying philosophy of reusing and extending existing code permits the development of such applications with high productivity.

Programming in Smalltalk is different from programming in other languages such as Pascal, C, or Ada because of the major influence played by the object-oriented programming paradigm, the large class library, and the interactive programming environment. Developing programs in Smalltalk requires familiarity with all three of these components and the learning curve for programmers is therefore longer than for more traditional languages. Although there is no substitute for programming with the Smalltalk system itself, our

objective is to reduce this learning curve by providing a comprehensive description of the Smalltalk language, the class library and programming environment and by illustrating the use of object-oriented programming techniques to develop interactive graphical applications. The need for a Smalltalk guru to be close at hand when learning the system will then be minimized. In addition, **Inside Smalltalk** will be a valuable reference to accomplished Smalltalk programmers whenever they venture into uncharted territory in the class library.

Be forewarned that it will take you considerably longer to become an accomplished Smalltalk programmer than an accomplished Pascal programmer. However, the return on your investment will be an ability to develop interactive graphical applications with all the features of modern user interfaces; e.g., windows, menus, mouse interaction. Indeed, a major emphasis of the second volume is to describe the Smalltalk features that make this possible; namely, the model-view-controller paradigm for constructing user interfaces and the graphical and window classes in the library. At the time of this writing, and despite the fact that it is this material that gives Smalltalk much of its appeal, no in-depth presentation of the graphical and user interface classes was available in any other text.

Although the Smalltalk language is itself quite small, the Smalltalk system is large. Initially this limited its use to expensive, powerful workstations. However, efficient implementations of Smalltalk are now readily accessible to large numbers of users on the current generation of personal computers bringing the power of Smalltalk to the classroom and a mass audience.

ORGANIZATION OF THE BOOK

Inside Smalltalk consists of two volumes with the first volume divided into 4 major sections. The second volume concentrates on the window and user interface classes and describes how Smalltalk may be used to develop applications involving WIMP-based (Windows, Icons, Menu, and Pointer) user interfaces.

VOLUME ONE

The first section of Volume One introduces the fundamentals of object-oriented programming and Smalltalk, the second describes the Smalltalk programming environment, and the final two sections divide the class library into basic classes (objects, magnitudes, and collections), and graphical classes. A common thread throughout the latter two sections is to describe a set of related classes from the class library, to explain some of the rationale behind design decisions taken by the designers, and then to show how new classes may be added to extend the existing classes in some useful way. In addition, Chapter 10 is devoted entirely to extended case studies describing the implementation of graphics-based applications. Problem sets are included at the end of each chapter; these range from simple exercises, to extensions of examples presented in the text, and finally to major projects.

Fundamentals

In this section, we introduce the reader to the fundamental concepts of object-oriented programming. Using a language independent approach, Chapter 1 characterizes object-

oriented programming as programming with objects, programming by simulation, computation via message passing and programming in the presence of polymorphism, inheritance, and a large class library.

Chapter 2 describes how these fundamental notions manifest themselves in Smalltalk. Smalltalk is a language somewhat smaller in size than Pascal and based on a surprisingly small set of concepts; namely objects, messages, classes, subclassing, and inheritance. Our approach is to introduce these new concepts by relating them to their counterparts in traditional programming paradigms and programming languages. In particular, programming in Smalltalk is introduced by contrasting Smalltalk code with its Pascal equivalent.

The Programming Environment

Developing Smalltalk programs is characterized by a total integration of tools and an absence of modes. Editors, file managers, compilers, debuggers, and print utilities are all included within the Smalltalk environment. Chapters 3, 4, and 5 provide an introduction to the integrated collection of powerful and sophisticated tools that together form the Smalltalk programming environment. Chapter 3 provides an introduction to basic features of the user interface, in particular, windows and menu interaction and how to enter, edit, and evaluate Smalltalk code. Chapter 4 describes the central role played by browsers in the programming process both for navigating the class library and for editing and compiling additions to this library. Chapter 5 describes the use of inspectors to investigate the internal state of objects and the use of notifiers and debuggers to view and modify the state of a suspended computations.

Basic Classes

In this section, we describe the basic classes — those classes that form the core of the class library. Chapter 6 introduces the default behavior for operations such as copying, printing and comparing that are supported by class Object — the ultimate superclass of all classes. Chapter 7 describes the Magnitude classes including the numeric, character, date and time classes. Chapter 8 describes the Collection and Stream classes that are as fundamental to Smalltalk as lists are to Lisp. To provide a better understanding of the numerous and closely related collection classes, we consider the classes from a logical perspective partitioning them into four major logical groups.

Graphics

In this section, the classes supporting the interactive creation and manipulation of graphical images are surveyed and their use illustrated through three case studies. Chapter 9 explains the use of forms and the bitblt operations that serve as a base for the Smalltalk graphical model. Interaction with the mouse and keyboard is addressed together with a description of simple graphical interaction techniques. The chapter concludes with a review of the path or trajectory classes (arcs, circles, curves, lines, linear fits, and splines) and the use of pens.

Chapter 10 presents three extended graphical examples: film loops, a magnifying glass, and a simple video game. Film loops are never ending movies and show how simple animation sequences can be developed. Techniques for obtaining flicker-free displays and for

storage of graphical forms on disk are also introduced. The latter facility illustrates the use of object mutation — the ability for one object to mutate into another. The magnifying glass application allows a user to move a magnifier over the display while magnifying the image under the magnifying glass. This application illustrates advanced graphical programming techniques and, in particular, describes how circular rather than rectangular forms may be manipulated. Finally, the video game illustrates the evolutionary approach that characterizes the design and development of Smalltalk applications. The design decisions that took place during the development of the game are described in detail along with the use of notions such as reusability, specialization, and generalization that differentiate object-oriented design from traditional design methodologies.

VOLUME TWO

Windows

In Volume Two, we describe the Smalltalk classes that provide (1) the familiar overlapping windows, pop-up menus, and mouse interaction facility that characterize the Smalltalk user interface and (2) the model-view-controller framework for the construction of user interfaces. Chapter 1 provides an introduction to the model-view-controller paradigm, dependency maintenance, the distinction between process management and window management, and the window transformation protocol. Chapter 2 provides an overview of the existing window classes and provides a detailed description of the basic views and controllers that support the window classes described in subsequent chapters. Extensive examples are provided to show how views and controllers can be created and used. Chapters 3 through 7 describe menu, switch, text, form (graphics), and pop-up windows respectively. Each of these chapters describes the differences between the standard classes and pluggable classes and shows (1) how users can use the existing classes, (2) how they may be modified to provide extensions, and (3) how new classes based on the existing ones can be created for special applications. Finally, Chapter 8 provides an extended example to illustrate the construction of a large-scale window application. It deals with the construction of a window maker — an editor that helps users create user interfaces. In the process, a design for a library of switch forms and a library editor is developed. The existing window classes are extended to support the window maker application and more than a dozen subwindows are designed to support the window maker editor.

WHO SHOULD READ THIS BOOK?

Smalltalk provides a new programming paradigm and the two volumes are therefore aimed at readers who are receptive to new ways of thinking about problem solving and new programming language concepts. We expect that most readers will have some programming experience in a procedural language. Programmers familiar with Pascal, C, Ada, or Fortran will find the language easy to learn and will be pleasantly surprised at the extensive set of support tools in the environment.

To gain full benefit from the book, readers should have access to a Smalltalk system and be prepared to adopt an exploratory hands-on approach to programming and problem-solving. **Inside Smalltalk** is for the professional programmer and serious student who wish to use the Smalltalk system as a powerful, efficient prototyping and development environment. The book can be effectively used in undergraduate and graduate courses in object-oriented programming or software engineering where Smalltalk will be a language of instruction. The book will be particularly valuable for students carrying out extensive thesis and project work in Smalltalk.

SMALLTALK DIALECTS

Two releases of Smalltalk-80 have been licensed by the Xerox Corporation. These are known as Smalltalk-80 Version 1 and Smalltalk-80 Version 2 respectively. Version 2 includes several features, notably support for multiple inheritance, not supported by Version 1. ParcPlace Systems[1] now has exclusive worldwide ownership of the Smalltalk-80 system. The Smalltalk language[2] is available under royalty-free license from ParcPlace. Smalltalk-80 Version 2 is now accepted as the standard Smalltalk-80 system and it is this dialect of Smalltalk that is described in this book. Indeed, whenever we use the term Smalltalk in this text we are referring to Smalltalk-80. Smalltalk-80 for Sun, Macintosh, Apollo, DEC, Hewlett Packard, and 80386 MS-DOS systems is available from ParcPlace Systems. Smalltalk-80 code is almost entirely portable across different host platforms. The Smalltalk-80 system is now marketed by ParcPlace Systems under the name Objectworks for Smalltalk-80.

Digitalk[3] markets Smalltalk/V, a dialect of Smalltalk for Macintosh and IBM PC computers. Excluding the user interface classes, there is a great deal of commonality between the Smalltalk V and Smalltalk-80 class libraries. Similarly, the range of programming tools is similar, although there are distinct differences in the structure and functionality of specific tools such as the browser, in the method of interaction with the environment and in the degree of integration with the specific platform

ACKNOWLEDGMENTS

First and foremost, we would like to acknowledge the great contribution made to the software community by the group of researchers at the Xerox Palo Alto Research Center (PARC) who were responsible for the development of the Smalltalk system. In particular, we single out Alan Kay, Adele Goldberg, and Dan Ingalls, who in 1987 received formal recognition of their work with the 1987 ACM Software Systems Award. In recognition for the development of a software system that has had a lasting influence, that has reflected contributions to new and still evolving concepts, and that has resulted in commercial

---

[1]ParcPlace Systems, 1550 Plymouth Street, Mountain View, CA 94043.

[2]Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Implementation* (Reading, Mass.: Addison-Wesley, 1983).

[3]Digitalk, Inc. 9841 Airport Road Bvld. Los Angeles, CA 90045.

acceptance, the Xerox PARC group received the award for seminal contributions to object-oriented programming languages and related programming techniques. Smalltalk was cited as having provided the foundation for explorations in new software methodologies, graphical user interface designs, and forms of on-line assistance to the software development process. Our thanks also to ParcPlace Systems for continuing to develop and market the Smalltalk-80 system.

We also thank Dave Thomas, who many years ago foresaw the potential of object-oriented programming and motivated us to become involved in research in the area. To the many students at Carleton University in Ottawa and to others who attended our object-oriented programming and Smalltalk workshops, our sincere thanks for being such willing guinea pigs for much of the material that now appears in this book. Our thanks also to the reviewers and, in particular, Richard Bernat of the University of Texas at Austin and Bharot Jayaraman of the University of North Carolina at Chapel Hill, for their helpful comments. To Marcia Horton, Christina Burghard, and their colleagues at Prentice Hall, for their support and patience in the development of the book. Finally, on a more personal note, we thank our respective wives, Marla Doughty and Christine Pugh, for their support and understanding, and our children, Brannon, Robin, Chloé, and Gareth, who have yet to understand why their "daddies" were too often unavailable.

1

Object-Oriented Programming

1.1 INTRODUCTION

In terms of its influence on the programming community, object-oriented programming (OOP) is predicted to be to the nineties what structured programming was to the seventies. But what is it that makes a system or programming language object-oriented? What exactly is meant by the term object-oriented? In this chapter we try to answer these and related questions. We will introduce object-oriented concepts in a language independent manner. However, because terminology in the field has not been standardized and since we will be describing Smalltalk in the rest of this book, we will use the terminology adopted by Smalltalk.

1.2 OOP IS PROGRAMMING BY SIMULATION

Object-oriented programming is most easily described as **programming by simulation**. The programming metaphor is based on personifying the physical or conceptual objects from some real-world domain into objects in the program domain; e.g., objects are clients in a business, foods in a produce store, or parts in a factory. We try to reincarnate objects from the problem domain into our computer models, giving the objects in our program the same characteristics and capabilities as their real-world counterparts. This process is often referred to as **anthropomorphic programming** or **programming by personification**.

The power of simulation as a programming metaphor can be seen from the success of the window-based user interfaces now common in personal workstations. The Apple Macintosh™, for example, uses a ***desktop*** metaphor in which icons representing such

common office objects as documents, folders, and even trash cans appear on the desk. Interactively, a user can open documents, copy them, store a document with other documents in a folder, or place a document in the trash can. Operations on the desktop objects mimic the way their real-world counterparts are manipulated. When implementation domain objects have a direct mapping to problem domain objects, the resulting software is far easier to understand and use.

Consider the following problem specification for a simple video game.[1] A typical display for the game is shown in Fig. 1.1. The objective of the game is to remove all the bricks from the wall. When the ball strikes a brick, the brick disappears. The ball can be redirected using the paddle, which the player can move to the left or right using the mouse. The ball bounces off the sides, bricks, and paddle in a conventional fashion. A player is provided with at most three balls (one at a time) to remove the bricks. A ball is lost if it passes below the paddle; i.e., if the player misses it! Demolishing the bricks with the allotted three balls is a win — failure to do so is a loss.

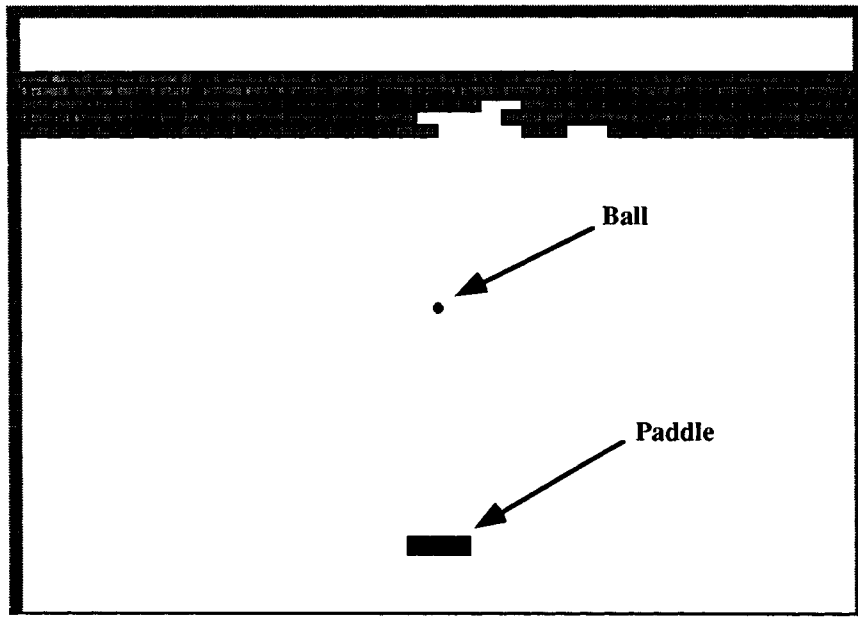

Figure 1.1 Simple video game.

An object-oriented solution to this problem would simulate the objects in the real video game. Software objects would be constructed to represent the paddle, the sides, the ball, the bricks in the wall, and so on. Furthermore, operations on these objects would represent problem-domain tasks such as moving the ball and the paddle, determining if the paddle has struck the ball or whether the ball is lost, removing bricks from the wall, and so on.

[1]Problem taken from D. H. Bell et al., *Software Engineering — A Programming Approach* (Englewood Cliffs, New Jersey: Prentice-Hall International, 1987).

1.3 TRADITIONAL VERSUS OBJECT-ORIENTED PROGRAMMING

Object-oriented programming is fundamentally different from traditional procedural or algorithmic approaches. Object-oriented programming describes a system in terms of the objects involved. Traditional programming approaches, on the other hand, describe systems in terms of their functionality. We will use the video game example to illustrate the differences between the traditional and object-oriented approaches.

1.3.1 A Traditional Approach

The classical top-down stepwise refinement approach to problem solving involves refining a problem solution into greater levels of detail based on functional decomposition. Taking a functional approach, we might first describe the solution to our video game in terms of the abstract statement:

```
Video Game
```

The next step in the solution might be to decompose this statement into the following:

```
WHILE Someone wants to play DO
    Set Initial Game Display
    Play a Single Game
ENDWHILE
```

The design could now be refined further by taking some of the abstract functions in the current solution such as **Set Initial Game Display** and **Play a Single Game** and decomposing them in a similar fashion.

```
Set Initial Game Display
    Draw Wall
    Draw Sides
    Initialize Paddle

Play a Single Game
    Set Score to 0
    Set Balls Left to 3
    WHILE Balls Left > 0 DO
        Play a Ball
        Decrement Balls Left
    ENDWHILE
```

The next step might be to refine the **Play a Ball** module.

```
Play a Ball
    Enter new Ball into Game
    WHILE Ball is in Play DO
        Check Ball Position
        Update Score & Display
        Move Ball
        Move Paddle
    ENDWHILE
    Remove Ball from Game
```

We are refining the solution to the problem algorithmically in a step-by-step manner, with each step in the process describing a solution to the problem at a certain level of abstraction. Systems refined in this way are most easily described using a diagram (see Fig. 1.2) where major modules are hierarchically organized and where each module represents a function or subproblem in the solution. A design produced using a functional decomposition approach fits very nicely with the procedural approach to programming encouraged by early languages such as Fortran, Pascal, or Cobol, where the subroutine, procedure, or subprogram is the predominant mechanism for structuring code. There is a direct mapping between functional modules in the design and procedures in the code.

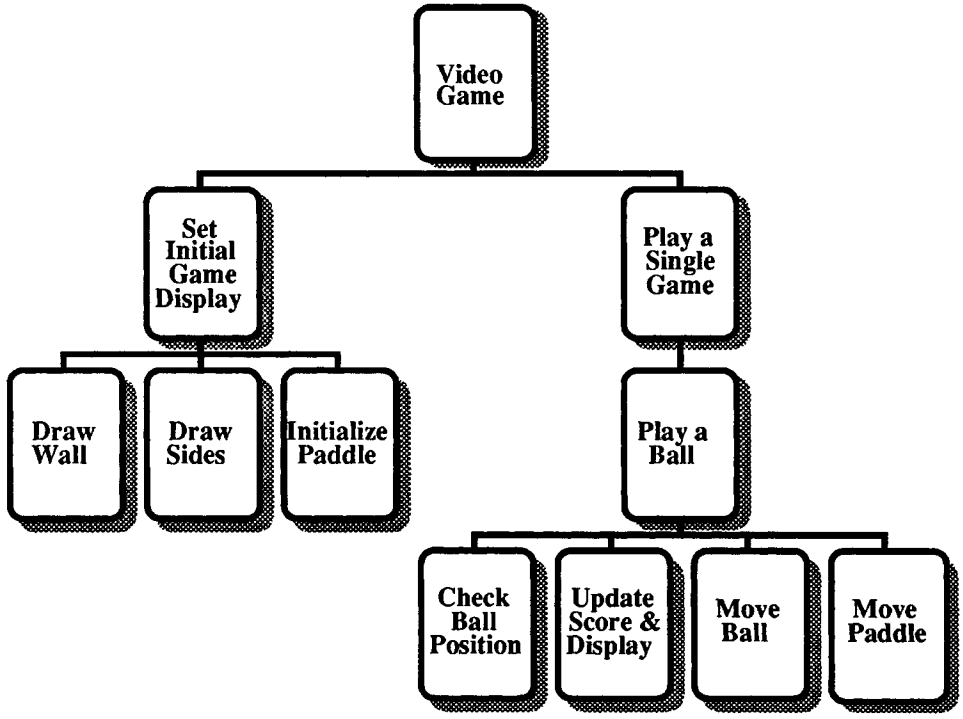

Figure 1.2 Functional decomposition of the video game.

1.3.2 An Object-Oriented Approach

If we take an object-oriented approach to this problem, our first concern is to try to identify not the functions but the objects that will be involved in the computation. The easiest objects to identify are those with real-world counterparts. In the case of our video game example, this leads us to think of objects such as the **bricks**, the **wall**, the **sides**, the **paddle**, the **ball**, and the **video game** itself, as shown in Fig. 1.3.