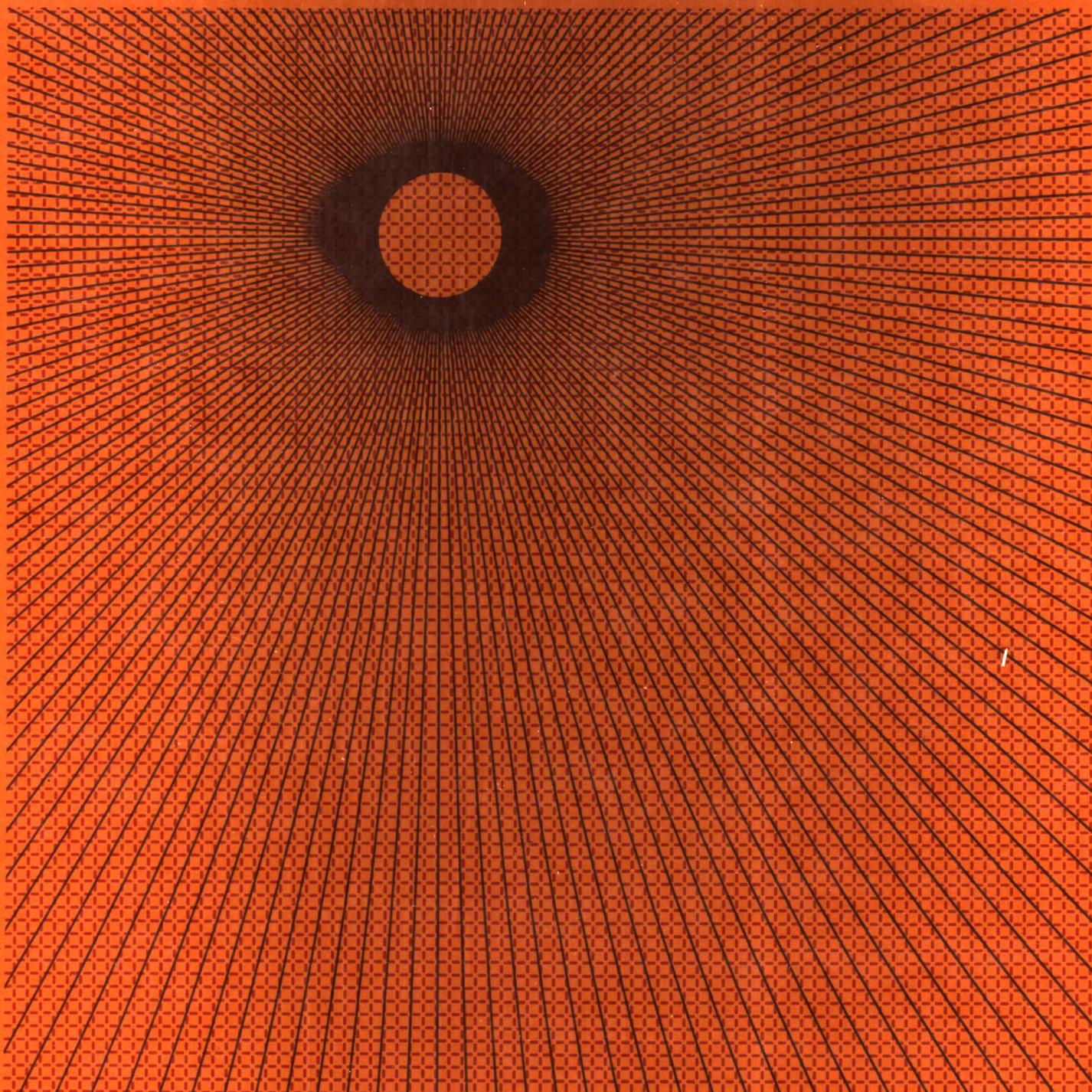


John R. Bourne

Object-Oriented Engineering

Building Engineering Systems
Using Smalltalk-80



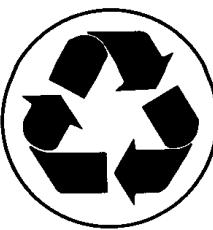
Object-Oriented Engineering

**Building Engineering
Systems
Using Smalltalk-80**



John R. Bourne
Vanderbilt University

IRWIN
Homewood, IL 60430
Boston, MA 02116



This symbol indicates that the paper in this book is made from recycled paper. Its fiber content exceeds the recommended minimum of 50% waste paper fibers as specified by the EPA.

©Richard D. Irwin, Inc., and Aksen Associates, Inc., 1992

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

The programs in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. The publisher has made every attempt to supply trademark information about manufacturers and their products mentioned in this book.

Cover and text designer: Harold Pattek
Composer: Science Typographers, Inc.
Typeface: Times Roman
Printer: R. R. Donnelley & Sons

Library of Congress Cataloging-in-Publication Data

Bourne, John R.

Object-oriented engineering: building engineering systems using smalltalk-80/John R. Bourne.

p. cm.

ISBN 0-256-11210-X

1. Smalltalk-80 (Computer program language) 2. Object-oriented programming. 3. Engineering—Data processing. I. Title.

QA76.73.S59B68 1992

620'.00285'5133—dc20

91-37875

Printed in the United States of America.

About the Author

John R. Bourne is a professor of electrical engineering and Director of the Center for Intelligent Systems at Vanderbilt University in Nashville. He received a BS degree from Vanderbilt University and MS and PhD degrees from the University of Florida. He spent 1982 as a visiting professor at Chalmers University in Gothenburg, Sweden, and during 1991 and 1992 he was on industrial leave at Northern Telecom, Inc. He has been editor of the *CRC Critical Reviews in Biomedical Engineering* since 1977 and was associate editor of the *IEEE Transactions on Biomedical Engineering* during 1981 and 1982. He has published extensively in professional journals and is the author of two books and several book chapters. During the last 10 years, Dr. Bourne's research interests have been in the area of intelligent systems and object-based methodologies.

The following
are registered
trademarks:

Actor	Open Look
Apple	OS/2
BOSS	PageMaker
Eiffel	ParcBench
Hewlett-Packard	ParcPlace
High C	PostScript
LaserJet	Presentation Manager
Macintosh	Simula 67
MS-DOS	Smalltalk-80
MS-Windows	Smalltalk/V
NeWS	SUN
Objective-C	UNIX
Objectworks	X Window System

Preface

Introduction

Engineering is defined as the application of scientific principles to the solving of real-world problems and the creation of artifacts for our society. The principal activities of engineering are analysis and design, that is, understanding and solving problems and designing new things. To support these activities, computers have become widely used during the last decades and are now employed throughout engineering. Furthermore, to program computers effectively, a succession of ever more complex computer languages have appeared throughout the last decades that have evolved into ever more sophisticated tools that engineers can use for problem solving. One of the most recent and innovative general computer language methodologies to appear is termed *object-oriented programming*. The name comes from the focus on describing problems in terms of objects, both physical and conceptual. Object-oriented programming methodologies appear to be superior to most other programming paradigms for solving many kinds of problems, especially problems in the domain of engineering in which problems are often readily decomposed.

Objectives and Organization

This text focuses on the understanding and use of object-oriented methodologies for engineering problem solving with a specific emphasis on analysis and design, two major topics taught in colleges of engineering and

applied in engineering practice. The concepts are covered and the techniques for analysis and design are supplemented with numerous practical examples. The material in the book is subdivided into three parts: concepts, tools, and engineering applications. The first part examines object-oriented methods broadly, starting with examples, giving basic definitions, and introducing the reader to methods for conducting object-oriented analysis and design. The second part reviews object-oriented languages and then turns to a specific discussion of the use of the Smalltalk-80 language and environment. The Smalltalk-80 language is utilized throughout for examples. Smalltalk-80¹ was one of the original object-oriented computer languages and remains as one of the most well-developed object-oriented languages. Furthermore, the Smalltalk environment is described in some detail because this environment remains prototypical of more advanced computer environments that have become essential productivity tools for industry. The final part of the text examines applications in engineering using object-oriented techniques. Specific attention is given to various methodologies for solving engineering problems.

Intended Audience

The audience for this text is expected to range from beginning college students to engineering practitioners. Little prerequisite knowledge is needed. Because object-oriented methodologies do not have a long history in the field of engineering, not much culture has grown up in this domain. Considerable attention has been given to explaining concepts starting with basic ideas, so that the text is appropriate for undergraduate engineering students. With appropriate explanation, the text could be used at the freshman level; however, the target level is approximately the junior undergraduate engineering student who has some background in computational methodologies.

How to Use This Text

The most effective way to learn object-oriented programming is to work problems on the computer while reading this and other textbooks. There is much to know; hence, Chapter 1 provides a supplementary reading list which the serious student will need to examine while learning about object-oriented methodologies. Perhaps more important is the need to have an object-oriented environment to work with during learning of these concepts. The author

¹Smalltalk-80 is a trademark of ParcPlace Systems.

recommends the Objectworks \ Smalltalk (Smalltalk-80) environment, marketed by ParPlace Systems, Inc. (1550 Plymouth St., Mountain View, CA 94043; Info@ParcPlace.{com,uucp}). This environment runs on many different types of computer systems including inexpensive PC-386 and Macintosh systems. These personal computer systems, when equipped with sufficient memory, are quite adequate for learning Smalltalk-80. Especially useful is the on-line tutorial provided with Smalltalk-80. Students who have access to this tutorial should go through it as soon as possible. Single-copy educational licenses and site licenses for Smalltalk-80 are available from ParcPlace Systems.

An instructor's manual for this textbook, available from the publisher, contains (1) notes about implementing an undergraduate course around the material contained in this textbook, (2) solutions for the exercises, and (3) information about Smalltalk-80 code that is available with the instructor's manual.

Smalltalk-80 Versions

Examples in the current edition of this text are coded in Objectworks \ Smalltalk, Release 4. During 1990 and 1991, ParcPlace Systems released Release 4 as a major change from the previous release (Version 2.5). The major new features of Release 4 were the addition of color and the enhanced portability of code between various hardware platforms and windowing systems. A significant part of the text is not dependent on which version of ST-80 is used; however, the second part of the text concerned with tools provides some examples that are release dependent. Almost all examples given are in Release 4. However, when it is useful to contrast methods used in the two releases, examples from the previous version are given and discussion provided about version differences. A preliminary edition of this text was written using only Version 2.5 code. This edition is available from the author.

Acknowledgments

The author thanks students who have, in one form or another, contributed to the material presented and have helped shape the character of this book. A special thanks is given to Zhihe Jiang who created the ViewBuilder program used in the text. Furthermore, colleagues who have worked with object-oriented methods are thanked, as well as the sponsors of many research projects that have supported the author's research during the last two decades. Thanks are given to Jacqueline Lusk, who prepared the figures, and Ellen P. Bourne, who edited and provided moral support during the writing of the text.

Thanks are also given to Howard Aksen, who was always attentive and worked hard to ensure the best possible product. Finally, thanks are given to the following reviewers of this book who carefully read and provided detailed constructive advice: Nader Bagherzadeh, University of California, Irvine; James L. Beug, Cal Poly, San Luis Obispo; Edward F. Gehringer, North Carolina State University; Howard Jachter, IBM; Fred Mowle, Purdue University.

ParcPlace Systems, Inc. grants permission to the author to use the copyrighted material, including source code excerpts and screen dumps from ParcPlace Systems' books and products, used in this book. Readers may copy such material, including all or any portions of the class hierarchy or message names, only under license from ParcPlace Systems.

J. Bourne
Nashville, Tennessee

Contents

	Preface	xiii
	Part One: The Concepts	1
1	Introduction	3
	1.1 Objectives	3
	1.2 Object-Oriented Analysis and Design	3
	1.3 Object-Oriented Methodologies for Engineering	6
	1.4 Tool Support for Practicing Object-Oriented Engineering	12
	1.5 Recommended Concurrent Reading	14
	1.6 Overview of Chapters	15
	References	16
	Exercises	17
2	Representing the World with Objects	19
	2.1 Introduction	19
	2.2 Defining Characteristics of Object-Oriented Methods	23
	2.3 Representing Objects	28
	2.4 The Causal Nature of Engineering Systems	33
	2.5 Representing Time	35
	2.6 Modeling and Simulation	36

2.7	Representation of Complex Systems: An Example of an Advice-Giving System	37
2.8	Summary	43
	References	43
	Exercises	43
3	Object-Oriented Analysis	45
3.1	Introduction	45
3.2	Analytical Methodologies for Building Object-Oriented Systems	46
3.3	Organizing Applications Using the Application/Class Organization Method (ACOM)	47
3.4	Analysis Maxims	56
3.5	Example Analysis of a Digital Circuit Simulator	59
	References	67
	Exercises	67
4	Introduction to Object-Oriented Design for Smalltalk-80	69
4.1	Design and Analysis	69
4.2	Major Design Aspects: The Model, the View, and the Controller	70
4.3	Model Design	73
4.4	View Design	76
4.5	Controller Design	78
4.6	Maxims for Design of Systems Using the MVC Triad	79
4.7	Example View Design Scenarios for Digital Circuits	80
4.8	Summary	83
	References	83
	Exercises	84
Part Two:	The Tools	85
5	Object-Oriented Languages	87
5.1	Procedural Languages and Object-Oriented Enhancements	87
5.2	Major Object-Oriented Languages	91
5.3	The Syntax of the Smalltalk Language	95
5.4	A Language Prospectus and Comparison	99
	References	105
	Exercises	107

6	Introduction to the ST-80 Environment	109
	6.1 Environment Concepts	109
	6.2 Characteristics of the ST-80 Graphics User Interface	112
	6.3 The Class Library	116
	6.4 Standard ST-80 Tools	117
	6.5 Creating Applications Using the Smalltalk Environment	125
	References	130
	Exercises	130
7	User Interface Design	133
	7.1 The User Interface	133
	7.2 Characteristics of Graphics Direct-Manipulation Interfaces	139
	7.3 Visual Display Methods	143
	7.4 Backward Compatibility for DisplayObjects: Forms, Pens, and Paths	148
	7.5 Windows and Views (Release 4)	154
	7.6 Basic Direct-Manipulation Interface Design Elements	159
	7.7 Graphically Tailoring the User View	165
	References	171
	Exercises	172
8	Tools for Building Applications: The Model-View-Controller Paradigm and View Components	175
	8.1 Model-View-Controller Concepts	175
	8.2 The View Framework	181
	8.3 Selection-in-List	184
	8.4 Buttons and Switches	186
	8.5 Transcripts and Text Editors	187
	8.6 Building Menus	188
	8.7 Using Forms	190
	8.8 Using Images	191
	8.9 Building and Using Icons	192
	8.10 Other View Components	193
	8.11 Putting the Components Together	194
	8.12 ViewBuilder	203

8.13	Comparison to ST/V Interface	213
	References	213
	Exercises	214
Part Three:	Applications in Engineering	217
9	Relating Engineering Problems to Object-Oriented Methodologies	219
9.1	Objective	219
9.2	The Nature of Engineering	219
9.3	Evaluation of Common Requirements in Engineering	223
9.4	Engineering and Object-Oriented Problem-Solving Methodologies	226
9.5	Characterizing Problems Using the Model-View-Controller Paradigm	233
9.6	Capitalizing on Abstraction and Reusability	245
9.7	Building Object-Oriented Solutions to Problems	247
9.8	Coupling to Traditional Solutions: Building an Integrated Problem-Solving Environment	248
9.9	Summary	253
	References	253
	Exercises	254
10	Constraint Methods	255
10.1	Basic Constraint Concepts	255
10.2	Using Constraints	257
10.3	Implementing Constraints: An Example for Digital Circuit Simulation (DCS)	260
10.4	Wiring and Testing Example Digital Circuit Simulation	269
10.5	Enhancing the Digital Circuit Simulation System: An Example in Tutoring Systems	273
10.6	A Petri Net Example	276
10.7	General Use of Constraint-Based Techniques	279
	References	280
	Exercises	281
11	Representing and Using Engineering Knowledge	283
11.1	Engineering Knowledge	283
11.2	Representing and Acquiring Knowledge	288

11.3	Frames, Semantic Nets, and Object Representations	290
11.4	Rules, Procedures, and Algorithms	303
11.5	Acquiring Knowledge	308
11.6	Representational Relationships	309
11.7	Identifying the Knowledge Organization Needed for Problem Solving	310
	References	311
	Exercises	312
12	Integrating Object-Based Environments with the Real World	315
12.1	Concepts	315
12.2	The External Environment	317
12.3	Interfacing Procedures and External Devices to the ST-80 Environment	323
12.4	Applications Using the External Environment	329
12.5	The Wrapper Concept	338
12.6	Object-Oriented Database Management	341
12.7	Summary	342
	References	343
	Exercises	343
13	Building an Application: Concepts in Rapid Prototyping	345
13.1	Introduction	345
13.2	The Digital Circuit Simulator Problem	346
13.3	Maxims for Rapid Prototyping	347
13.4	The Digital Circuit System: Teleological and Functional Assessment	350
13.5	DCS Architectural Organization	352
13.6	Organizing the Information to Store	354
13.7	Building the View	359
13.8	Interfacing to the Model	363
13.9	Operating the Simulator	365
13.10	The Digital Circuit Simulator Code	366
13.11	Additions to the DCS Application	377
13.12	Summary	378
	References	378
	Exercises	379

14	Engineering Simulation Methodologies and Simulation Frameworks	381
14.1	Introduction	381
14.2	The Kinds of Simulation	382
14.3	Multiple Independent Processes	391
14.4	Probability Distributions	393
14.5	A Simple Framework for Discrete-Event Simulation	395
14.6	A Discrete-Event-Simulation Framework	396
14.7	Simulation Using Petri Nets	398
14.8	Building an Example Simulation for Manufacturing	400
14.9	Summary	403
	References	404
	Exercises	406
15	Conclusions	409
15.1	Review	409
15.2	Prognosis for Object-Oriented Methodologies	409
15.3	Object-Oriented Methods in Engineering	410
15.4	The Importance of Environments	411
15.5	Reusability and Productivity	411
15.6	The Reduction of Complexity and Improvement in Rapid Prototyping	412
15.7	Graphics and Visualization	413
15.8	Language and Environment Trends	413
15.9	Near and Long-Term Trends for Object-Oriented Methodologies	414
15.10	Concluding Statement	415
	Exercises	415
	Appendix	417
	Index	419

Part One

The Concepts

The first part of the text, *the concepts*, describes basic ways of representing the world with objects. The first chapter gives an overview of the field of object-oriented engineering. Chapter 2 presents basic concepts associated with object-oriented methodologies, including comparison with several other types of representation methodologies. Chapter 3 examines the concept of object-oriented analysis with an emphasis on analytical techniques that will help the reader analyze problems that can be ultimately implemented in Smalltalk-80. The first section concludes with Chapter 4 on object-oriented design using Smalltalk-80.



Introduction

1.1 Objectives

The world is composed of objects. New objects and classes of objects are created by a process known as design. Existing objects and collections of objects can be understood through a collateral practice known as analysis, in which scientific principles are used to analyze the world. These two processes, *analysis* and *design*, make up most of the field of engineering, which is generally defined to be the application of scientific principles to practical problems. This textbook examines these two fundamental engineering practices in the context of object-oriented methodologies, loosely defined here as consisting of: (1) ways of thinking about and understanding the world as collections of objects and (2) programming methods for actually creating operable abstractions of objects in the world. The combination of these methodologies is defined here as *object-oriented engineering*.

The primary objectives of this text are (1) to create a framework for thinking about object-oriented analysis and design, (2) to describe and explicate tools and methodologies for building abstract program descriptions of objects and collections of objects, and (3) to show how object-oriented systems can be used for engineering problem solving.

1.2 Object-Oriented Analysis and Design

Consider Figure 1.1 in which the nature of engineering is schematically depicted. Different types of engineering practitioners are listed in the upper left part of the figure, including designers, analysts, and simulation-ori-

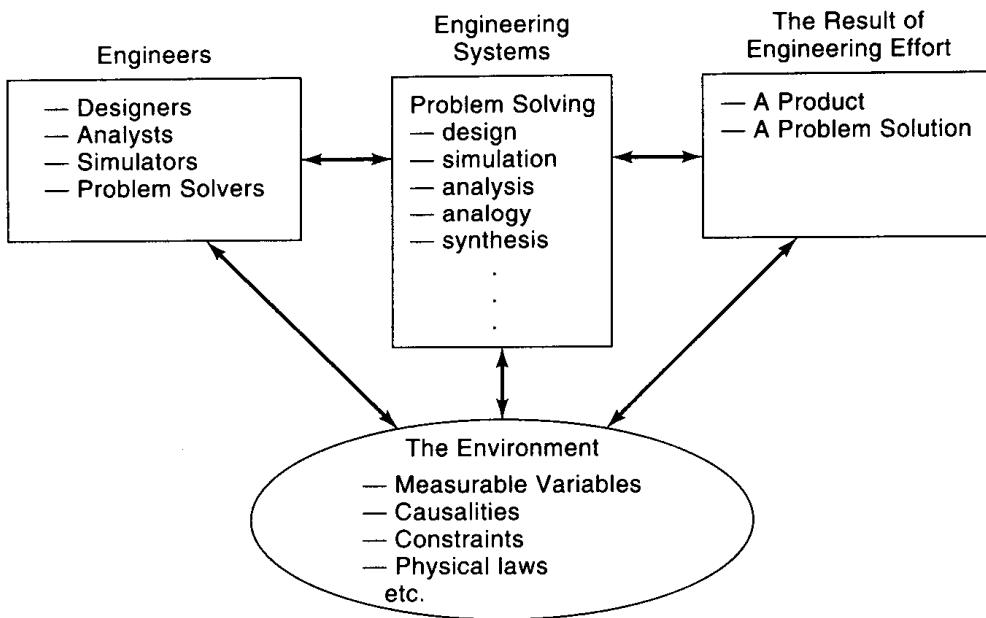


Figure 1.1 The Nature of Engineering.

ented engineers. Engineers typically utilize different approaches for problem solving; the methodologies used are actually most often composites of these stereotypic labels. Indeed, a typical problem-solving sequence may well consist of (1) analyzing a problem, (2) designing a new object, and (3) simulating the object using a computer. Engineering systems, as shown in the center of Figure 1.1, consist of a mixture of many problem-solving techniques which include those just discussed, as well as other more abstract approaches such as the use of analogy. The result of engineering activities, as portrayed in the rightmost box in Figure 1.1, consists of two main outcomes: (1) products and (2) solutions to problems. Analysis of problems and creation of new products revolve around the domain selected, that is, the environment that contains measurable variables, causal relationships between objects, constraints between entities, and physical laws that govern the behavior of objects. The complexities of describing the world mandate simple and descriptive mechanisms. This book will make the case that object-oriented techniques fill this need.

Next, consider Figure 1.2. This figure shows a framework for organization of a computer-based engineering problem-solving system. General-purpose problem-solving systems do not actually exist today because of the scope and difficulty of organizing knowledge about engineering. Discussion of this general problem-solving system is included here to indicate how such systems could be built to facilitate specific tests in engineering analysis and design and to serve as

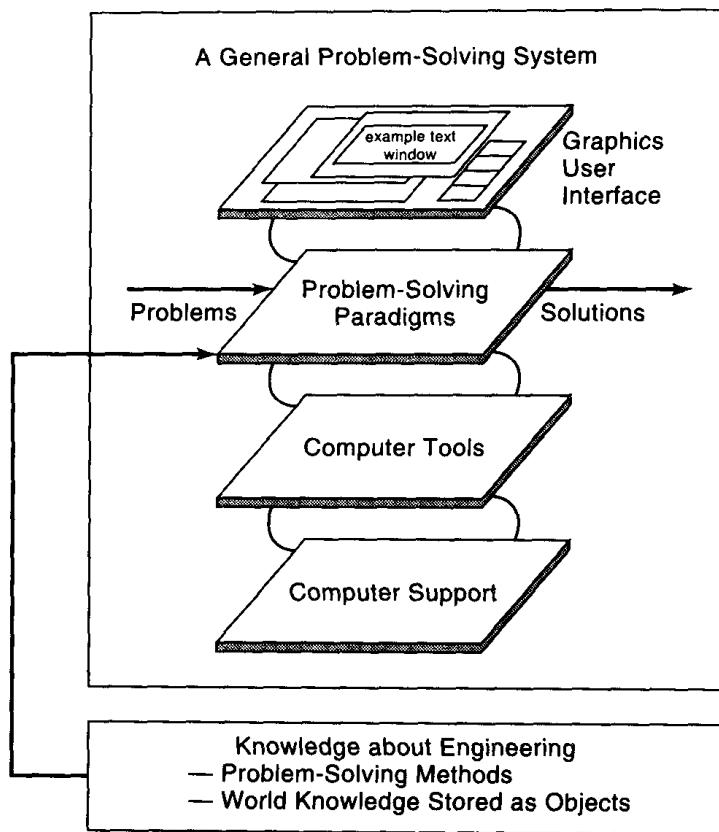


Figure 1.2 A Framework for a Problem-Solving System for Engineering.

a model for example systems described in later chapters. Furthermore, systems having this general framework can be constructed using the object-based technology discussed throughout this textbook. In the same way as in Figure 1.1, problems enter on the left side of the figure and solutions exit on the right. The problem-solving system is shown with four layers: (1) a graphics user interface (GUI) on the top layer, (2) a set of problem-solving paradigms (heuristic and algorithmic), (3) computer tools, and (4) computer hardware support. These facilities are arranged hierarchically to indicate how each lower layer supports the next higher layer. Shown also in the diagram is a knowledge source (e.g., a collection of information composed of data, methodologies, and algorithms) that is separated from the problem-solving system. The knowledge source is basically a knowledge-based management system that includes knowledge about a domain of interest as well as data. Knowledge types include knowledge about how to solve problems, as well as knowledge about the world and how things work.

The organization of the problem-solving system in layers is a typical methodology used to break down a system into smaller, less complex parts. A

graphics direct-manipulation interface (GDMI; see Chapter 7) provides facilities for interfacing the user to the system. Typical GDMS employ a “point-and-click” metaphor in which the user employs a mouse or similar pointing device to operate the system. Often consisting of multiple overlapping windows, lists from a user can select items, or buttons that a user can press. GDMS are a distinctly different interface from common “command line” interfaces in which users type in specific commands to cause computer operations to happen. The “Problem-Solving Paradigms” layer in the example in Figure 1.2 consists of various mechanisms for utilizing and accessing the tool layer. For example, problem-solving methodologies should be contained in layer 2, including techniques for organizing, inspecting, and understanding knowledge, as well as interfaces to specific tools that might be part of the system. The third layer from the top is basically a tool kit that provides the user with extended capabilities for problem solving. For example, tools in a tool kit might include inference systems (e.g., rule-based systems), interfaces to computer-aided-design (CAD) tools, analysis tools, simulation tools, and tutorial and information understanding assistants. Finally, at the lowest level of this example system is computer support, which consists of both hardware and software support. To be most useful, software should be runnable on and transportable across many types of hardware and basic software support should provide the same capabilities on hardware manufactured by different vendors.

The general-purpose problem-solving system described previously mirrors the problem-solving capabilities of engineers who conduct analysis and design. It is alleged that the cognitive skills that engineers use can be made explicit rather than implicit. As problem solvers, engineers can call on a wide range of inspection and manipulation tools (graphs, lists, references), different problem-solving methods, tools of various sorts, and computer support. These typical elements in the engineer’s tool kit are reflected in the layers shown in this hypothetical system. This text offers a look at many of the elements that can be used in building problem-solving systems that may ultimately become available for use in analysis and design tasks in engineering.

1.3 Object-Oriented Methodologies for Engineering

Object-oriented programming is a methodology that permits description of objects, including functional, behavioral, and declarative information. Function refers to what an object is designed to do, behavior refers to what it actually does, and declarative information consists of information about an object’s state. For example, the function of a car is to move, to turn, to transport people, and so on. The behavior of a car implements function; computer descriptions of behavior are often turned into simulations that can be used to

assist designers in observing changes in behavior as design specifications are varied. Declarative information simply provides a specification of the attributes of an object; for example, a car might have four doors, four wheels, and so on. The object-oriented paradigm provides a programming framework that can assist in such descriptions. Although there is some controversy about what makes a programming language or methodology object oriented, a typical definition is that an object: (1) has local state, (2) inherits characteristics from more abstract objects, and (3) responds to messages, decoding the meaning of messages in the object's local context. Such characteristics make objects very useful for describing how things work in the real world and ideal for use in problem-solving systems that facilitate the work of engineers.

Next, as an introduction to what one can do with object-oriented methods, four examples are examined that typify the use of object-based systems in engineering practice. The systems shown could be created in traditional programming environments; however, they fit more naturally to object-based methods due to the strong coupling of the paradigm to the real world. In each of the examples, this coupling will be discussed in an attempt to show the strength of object-oriented methods for use in engineering. The examples cover a range of applications: (1) a computer-aided-design (CAD) tool for electronics, (2) a signal analysis system, (3) an interface to instrumentation, and (4) an example of an object-oriented traffic simulator. These examples emphasize interactive graphics user interfaces which are easily built using the object-based methodologies described in this text.

1.3.1 Computer-Aided Design

New things used to be designed by hand. Engineers would work from requirements to generate designs which would be laboriously committed to detailed paper specifications and drawings, from which an actual new object could be fabricated. During recent decades this “design-by-hand” methodology has been gradually replaced with computer-aided design (CAD), in which computers assist engineers in (1) creating specifications for a new design and (2) testing a design using computer-based simulation. The advantages of the methodology lie chiefly in the ability to prototype designs rapidly, make changes quickly, and observe the effects of engineering decisions on the final product.

As an example, consider a simple CAD tool for design of electronic systems. The objective of designing an electronic system based on digital circuits could be to construct a new digital device or class of digital devices such as digital telephones, computers, or perhaps even microwave oven controllers. Decisions must be made about which electronic parts to use, how much the parts cost and their availability, and how the parts will work when connected. A

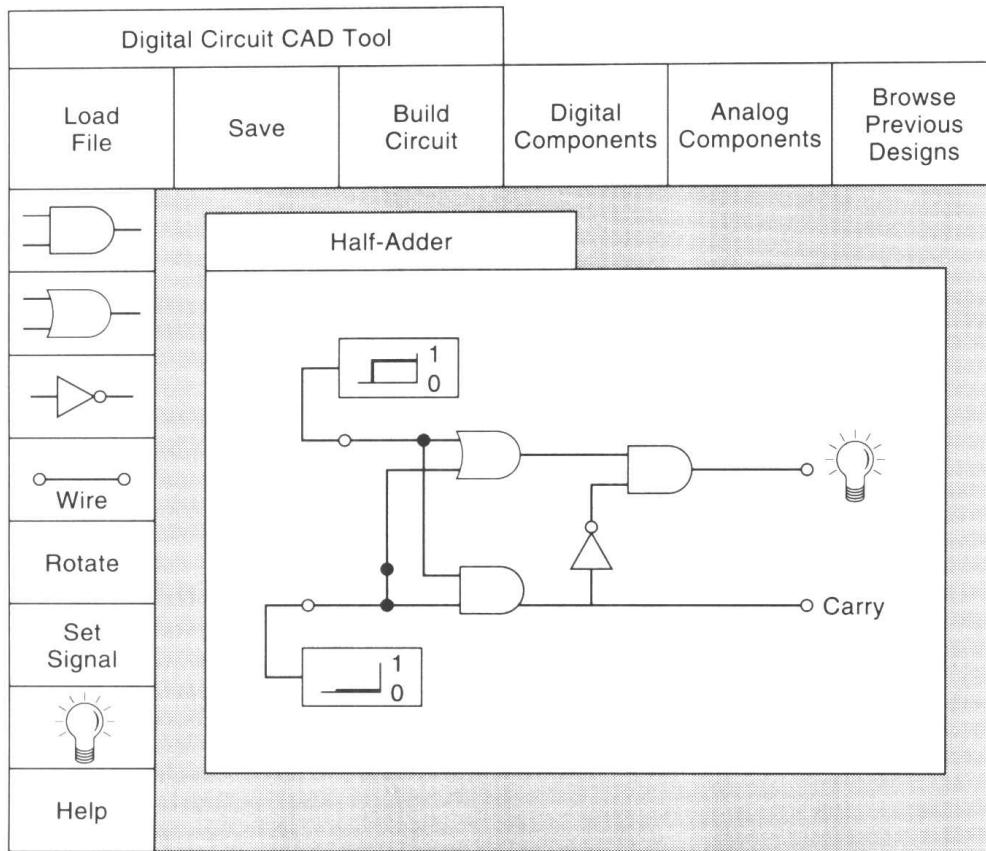


Figure 1.3 A Computer-Aided-Design Tool for Digital Electronics.

CAD tool will permit a design engineer to design a circuit, build a simulation of the circuit on the screen, and evaluate how the design will work prior to ever building an actual circuit. During design, the engineer may try different alternative circuit designs to attempt to achieve the same functionality with a reduced number of chips, for example. The capability for rapidly doing “what-ifs” (“what if I change this or that...”) coupled with the ability to simulate the resulting design makes CAD a powerful tool for designing most things. Perhaps the most powerful feature is simulation, because simulation helps the designer to understand how the designed system will work without actually having to build the system.

Figure 1.3 shows an example of how a computer screen might look for a CAD tool constructed for digital electronic design. The screen has a number of features: (1) at the top are several buttons that the user can press by moving a cursor inside a button and pressing a key on the mouse to activate the command on the button label. On the left side of the view are buttons which, when

activated, attach an icon¹ to the cursor and permit the user to place the icon on the graphics drawing pad in the center of the screen. By simply pointing and clicking, the designer can build a digital circuit by selecting components and wiring them together. Once wiring is completed, signals can be introduced at inputs in the simulated system and the behavior of the system can be observed. In this example, a signal could be added to an input wire using “Set Signal.” Automatically, the signal would propagate throughout the circuit. If a signal appeared at the output, the bulb shown would change from empty to filled to show that a signal had appeared at the output wire. Using this methodology, the designer could probe different points in the system to determine if the conceptual design and simulated implementation were correct prior to making a commitment to implementation in hardware.

Of course, industrial-strength CAD systems are tremendously more complex than the example just described [see, e.g., van der Meulen (1989)]. Most commercial systems have the capability of creating and simulating very complex systems and providing detailed analyses of problems in the design. However, the simple system shown in Figure 1.3 could be extended to handle more complex circuits and systems.

1.3.2 Signal Analysis

The term *analysis* may be applied to many things about which an understanding is sought. In engineering, one may wish to understand the nature of a physical system such as the stresses on a bridge or the meaning of signals secured from strain gauges measuring stresses in an earthquake fault zone. Signal analysis is a popular engineering activity in which signals obtained from many different kinds of physical processes are evaluated to secure information. Examples range from geological evaluation to analysis of biological signals.

In much the same spirit as the design example given previously, consider the view of a computer screen given in Figure 1.4 which shows a signal analysis system based on the same “point-and-click” methodology discussed previously. This example system permits sampling of external analog signals, displaying of signals, and viewing of fast Fourier transforms (FFT; Blahut, 1985) in a separate window. This set of activities is typical of many simple signal analysis systems in which the FFT is used to reveal a plot of the amplitude of the signal activity with respect to the frequency. The time-domain signal shown at the bottom of the figure and the frequency-domain signal at the top of the figure represent the same data from these two different perspectives. The

¹An icon is a graphic representation of an object.

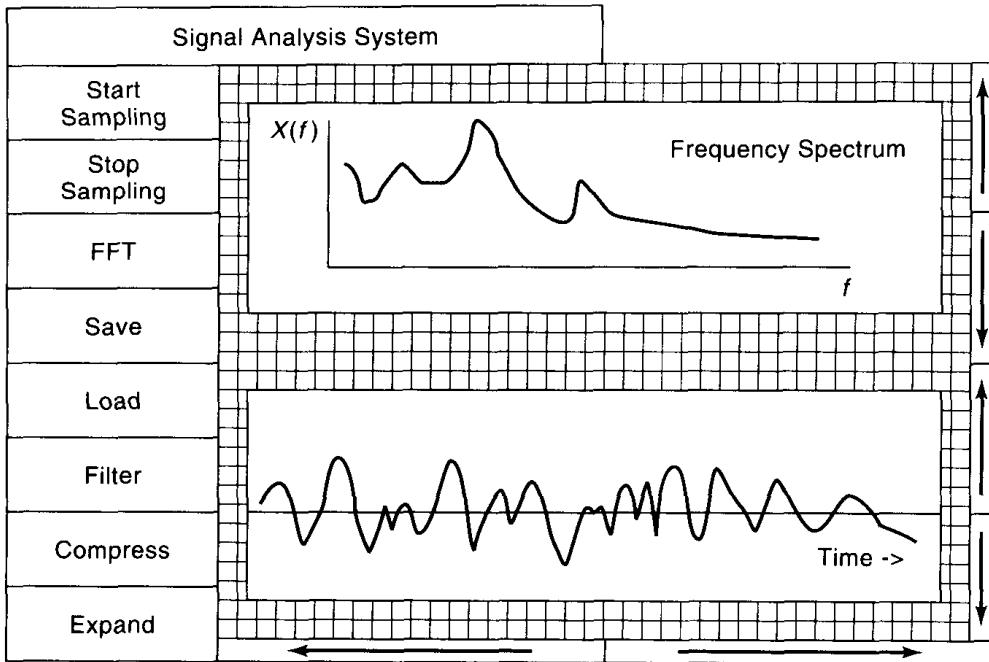


Figure 1.4 A Signal Analysis System User View.

capabilities of the example system, as reflected in the buttons to the left of the view include starting and stopping sampling of a signal, saving and loading files, filtering, compressing and expanding the signal, and finally producing an FFT. Arrows shown in the margins of the graphics views of the signals permit scrolling of the information up and down or left and right.

To clarify the use of an analysis system of the type shown in Figure 1.4, consider the task of a highway engineer who is attempting to determine if noise has been reduced sufficiently by noise barriers. The engineer would visit a highway site with a portable computer equipped with a microphone, sampling equipment, and the analysis system described previously. Placing a microphone at various locations in the environment, the engineer could use our example system to acquire noise data. By pressing "FFT," the engineer could then evaluate the range of frequencies that are present (e.g., the presence of low-frequency rumbling could be easily detected by examining the FFT plot in low-frequency ranges). Using this information, decisions about modifications to the site could be made.

1.3.3 Instrumentation Interface

Many engineers conduct experiments in the real world; that is, they interface measurement and control instruments to experiments or control

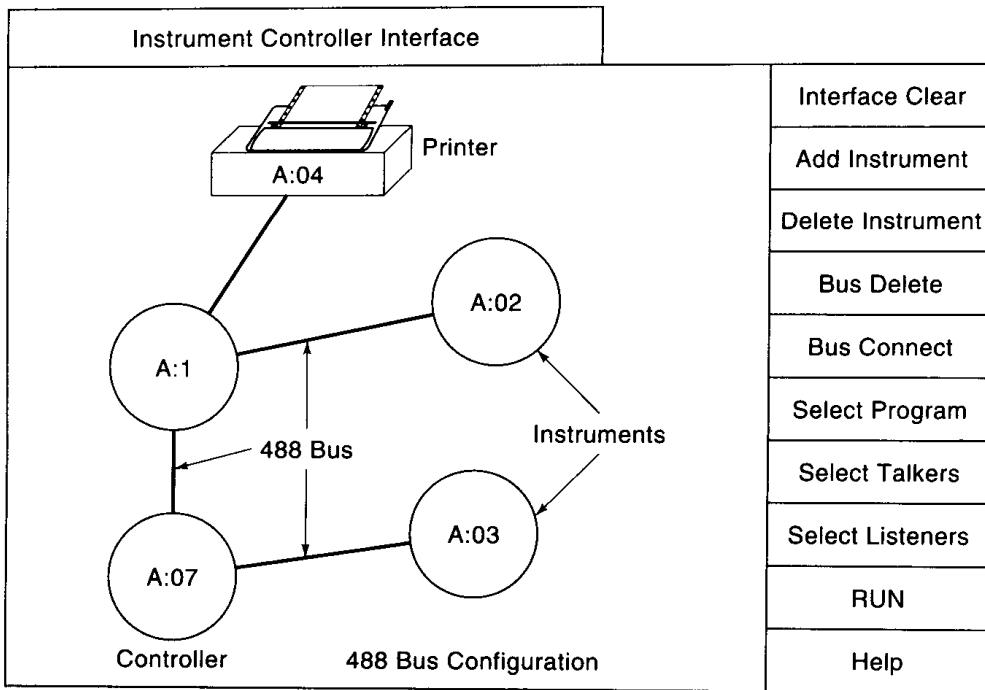


Figure 1.5 Control Panel for IEEE-488 Bus Instruments.

systems. In experimental laboratory situations, numerous instruments of various types are used to measure information from an experimental preparation. For example, oscilloscopes can secure electrical information, flow meters measure information about the flow of gases and liquids, and temperature gauges provide information about heat and cold. Often it is useful for engineers to employ collections of interconnected instruments that can be used in experimental laboratories. One popular interface to the world through instruments is the IEEE-488 bus [see, e.g., Gates (1989)], also known as the GPIB (general-purpose interface bus). Following the same paradigm as shown previously for design and analysis, object-oriented systems can be designed to facilitate the use of instruments in experimentation and control.

Consider Figure 1.5 in which a view of a control panel for an IEEE-488 instrumentation controller is shown. In this example, one could construct a laboratory containing different types of instruments that might need to be connected at various times to conduct different experiments. Each instrument would have a unique address (as shown here, A:02, A:03... are different addresses). The type of graphics interface shown would permit the users to "wire up" an instrument system by simply "pointing and clicking" to connect instruments on the screen. IEEE-488 bus technology provides the capability of making *electrical* connections, so that a complete custom instru-

mentation system could be rapidly constructed without the need to physically wire any instruments together. The panel on the right of the figure shows common operations such as “add an instrument” or “delete an instrument” from the configuration. The graph shown in the figure consists of nodes which are replaced with icons (e.g., the printer icon) when the type of instrument desired by the experimenter is selected.

The strength of object-oriented methodologies in the instrumentation example lies in the ability to encapsulate the behavior of instruments in objects and in the capability of graphically representing collections of instruments wired together in networks in an easily understandable way. Object-oriented languages provide excellent support for the building of the graphics interfaces that connect the user with the application.

1.3.4 Object-Oriented Simulation

The final example in this introductory material is concerned with using object-oriented methods for building simulation systems. Consider Figure 1.6 which shows a simulation of traffic at a traffic light. The view that the user sees is shown with connections to the computer objects that simulate the behavior of the objects displayed. In the simulation, each object described has its own individual behavior (e.g., the car objects). In the view, the user can observe the behavior of the cars operating in the constrained simulated environment around the traffic light. Object-oriented methodologies are particularly good for simulating these types of situations which include many different cases in which objects give and receive service. Other examples, including management of aircraft, manufacturing flow of material, personnel flow, decision making, hydrological systems, customers in a bank, cars in a car wash, and so on, operate using the same principles.

1.4 Tool Support for Practicing Object-Oriented Engineering

Table 1.1 displays introductory information about tool support for practicing object-oriented engineering. The first part of the table shows environments and languages that are commonly employed, and the second part of the table displays some of the myriad of tools that are currently commercially available.

The language and environment with the longest history is Smalltalk-80, originally developed at Xerox in the 1970s. Smalltalk/V, a similar system for

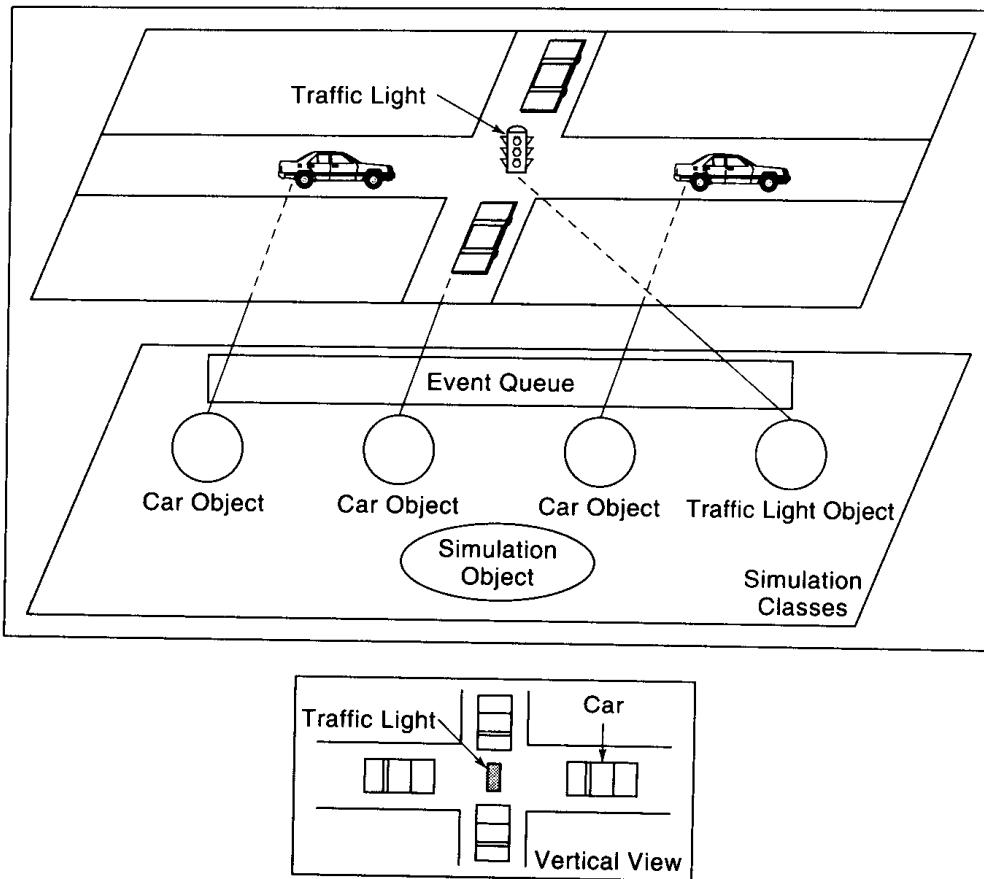


Figure 1.6 A Traffic Flow Simulation.

PCs and Macintoshes, is also shown along with C++, a Bell Laboratories-originated language, and Objective-C, a C-language-based language and environment with a Smalltalk-like flavor. These languages and environments will be described in detail in Chapter 5.

An environment is similar to an operating system in that an environment provides the user with tools in which to program applications. Although the term *operating system* is used to refer to the software that interfaces with the computer to manage the computer's resources for the benefit of users, *environment* is a term that has a higher-level meaning; in general, it describes the capabilities of the collection of tools that are available to service the needs of the user. Systems with the most useful and most user-friendly set of tools can permit users to achieve the highest levels of productivity. Due to the ease of constructing tools using object-oriented methods, systems such as Smalltalk-80 have evolved over the years to have very high levels of environmental convenience.

Table 1.1 Examples of Commercial Environments, Languages, and Tools Based on Object-Oriented Methods

Examples of Environments and Object-Oriented Languages		
Language System	Function	Source/Vendor
Smalltalk-80	Object-oriented environment and language; many hardware platforms	ParcPlace Systems
Smalltalk/V	Object-oriented environment and language; PC and Macintosh platforms	Digitalk
C++	C-based object-oriented language	AT&T and others
Objective-C	C-based object-oriented language	Stepstone
Common Lisp Object System (CLOS)	Object-oriented language built on Lisp	Xerox
Example Tools		
Humble	Inference engine	Xerox Special Information Systems
Analyst	Integrated office environment	Xerox Special Information Systems
Draw-80	Drawing package	Knowledge Systems Corporation
Pluggable Gauges	Gauges for Smalltalk	Knowledge Systems Corporation
GemStone	Object-oriented database management system	Servio
PERT	Business tools	Fuji Xerox
Petri net	CASE tools	Fuji Xerox

nience that permit users to construct systems for new applications in a minimal amount of time.

The second part of Table 1.1 shows a number of different tools that have been produced commercially for use in object-oriented systems. The intention here is not to be exhaustive in listing different tools, but to show the range and type of some common tools that are available.

1.5 Recommended Concurrent Reading

Table 1.2 lists a few textbooks and manuals that will assist the reader of this text in securing additional background information and ancillary knowl-

Table 1.2 Selected Concurrent Readings*

Title	Author
<i>Inside Smalltalk</i>	Pugh and LaLonde
<i>Smalltalk-80: The Interactive Programming Environment</i>	Goldberg
<i>Smalltalk-80: The Language and Its Implementation</i>	Goldberg and Robson
<i>The C++ Programming Language</i>	Stroustrup
<i>Object-Oriented Programming — An Evolutionary Approach</i>	Cox
<i>An Introduction to Object-Oriented Programming and Smalltalk</i>	Pinson and Wiener
<i>Object-Oriented Analysis</i>	Coad and Yourdon
<i>Object-Oriented Design with Applications</i>	Booch
<i>Journal of Object-Oriented Programming</i>	Wiener (Editor)
<i>Objectworks \ Smalltalk User's Guide (Release 4)</i>	ParcPlace Systems
<i>Objectworks \ Smalltalk Tutorial (Release 4)</i>	ParcPlace Systems

* See the references for complete citations.

edge. Six books and manuals on Smalltalk are shown, as well as one book on C++, one book on Objective-C, and other texts concerned with general object-oriented analysis and design methodologies. These texts will assist a serious student of object-oriented methodologies in securing viewpoints other than those given in this text. Also, later chapters in this text occasionally make reference to materials given in some of the texts listed in Table 1.2. Extensive information and examples concerned with the Smalltalk-80 environment and language are given throughout this textbook. Although an introduction to both the environment and language will be given, detailed descriptions will not be undertaken because there are excellent reference materials available that can be used in conjunction with this text. The *Objectworks \ Smalltalk User's Guide* and *Objectworks \ Smalltalk Tutorial* (1990) are recommended concurrent reading for students and others wanting to learn the details of this language and environment.

1.6 Overview of Chapters

This textbook is organized into three main parts: (1) the concepts, given in Chapters 1 through 4; (2) the tools, described in Chapters 5 through 8; and (3) the applications in engineering, given in Chapters 9 through 14. Chapter 15 provides a set of conclusions and a look at the future.

In the first part of the text, *the concepts*, basic ways of representing the world with objects are given, with an examination of methods for conducting

object-oriented analysis. Object-oriented design as an extension of object-oriented analysis is described along with an introduction to the methodologies for designing object-based applications.

The second part of the text, *the tools*, deals first with general object-oriented languages, second with a review of the characteristics of Smalltalk-80 graphics user interface design, and finally the tools for building applications in Smalltalk-80. This latter section takes a detailed look at the model-view-controller paradigm of Smalltalk-80 and shows how the paradigm can be used in building engineering applications.

The last part of the text deals with *engineering applications*. First, the general nature of the engineering methodology is examined and compared to the object-centered approach. Constraint methods, techniques for acquiring, modeling, and presenting knowledge, are given, as well as a description of interfacing object systems to the real world. Finally, a chapter on constructing object-oriented simulation systems is provided.

References

- Blahut, Richard E. (1985). *Fast Algorithms for Digital Signal Processing*. Addison-Wesley, Reading, MA.
- Bobrow, D. G. et al. (1987). *Common Lisp Object System Specification*. ANSI Report 87-001, September.
- Booch, Grady, (1991). *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, CA.
- Coad, P., and E. Yourdon (1990). *Object-Oriented Analysis*. Prentice-Hall, Englewood Cliffs, NJ.
- Cox, B. J. (1986). *Object Oriented Programming—An Evolutionary Approach*. Addison-Wesley, Reading, MA.
- Gates, S. C. (1989). *Laboratory Automation Using the IBM PC*. Prentice-Hall, Englewood Cliffs, NJ.
- Goldberg, Adele (1983). *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Menlo Park, CA.
- Goldberg, Adele, and David Robson (1983). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Menlo Park, CA.
- Journal of Object-Oriented Programming, P.O. Box 6338, 733 Woodland West Drive, Woodland Park, CO 80866.
- Meyer, Bertrand (1988). *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ.
- Objectworks \ Smalltalk User's Guide and Objectworks \ Smalltalk Tutorial*, Release 4 (1990). ParcPlace Systems, Mountain View, CA.

- Pinson, Lewis J., and Richard S. Wiener (1988). *An Introduction to Object-Oriented Programming and Smalltalk*. Addison-Wesley, Reading, MA.
- Pugh, John R., and W. R. LaLonde (1990). *Inside Smalltalk*, Volumes 1 and 2. Prentice-Hall, Englewood Cliffs, NJ.
- Smalltalk/V*. Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045.
- Stroustrup, B. (1986). *The C++ Programming Language*. Addison-Wesley, Reading, MA.
- van der Meulen, Pieter S. (1989). Development of an Interactive Simulator in Smalltalk. *Journal of Object-Oriented Programming*, Vol. 1, No. 5, January/February, 28–51.

Exercises

- 1.1 Secure recent promotional literature for Smalltalk-80 and Smalltalk/V. Familiarize yourself with the characteristics of the two systems and compare and contrast the two systems based on the information you obtain.
- 1.2 Conduct the same investigation as described in Exercise 1.1 for C++ and Objective-C.
- 1.3 Visit the library and review information in the concurrent readings listed in Table 1.2. Prepare a short prospectus about the contents of each of the references that you find. Look for other references in this same topic area and prepare a brief abstract of your findings.
- 1.4 Smalltalk-80 is delivered with an on-line tutorial and tutorial text. If you are reading this book to gain an in-depth understanding of Smalltalk and to learn how to build applications using Smalltalk, you should begin to study the ParcPlace tutorial and to work through the accompanying on-line tutorial.
- 1.5 List the reasons given in this chapter that object-oriented methodologies are useful for building engineering applications.

Representing the World with Objects

2.1 Introduction

The world is composed of objects, both animate and inanimate. We easily understand objects in nature, such as trees and rocks, that have well-known characteristics and synthetic objects, such as measuring instruments, automobiles, water heaters, and a host of other engineered artifacts. Less well understood are objects that represent abstractions, such as time, mental relationships, or even complex engineered systems composed of many interacting objects. In this chapter, we will examine the basic paradigms that are useful for representing different types of objects. Illustrations given include physical objects that have structural and behavioral characteristics and complex sets of objects that are examples of engineering systems that can be described with object-oriented techniques.

To describe an object or objects in object-oriented programming languages, four general terms are frequently employed: *encapsulation*, *abstraction*, *inheritance*, and *polymorphism*. Encapsulation refers to the hiding of the implementation of an object from the outside world. By hiding the internal structure of an object, understanding the behavior of an object is reduced to understanding the messages used to communicate with the object. Abstraction deals with separating the inherent qualities or properties of an object from an actual physical object to which they belong. For example, an abstract object might represent all automobiles with their accompanying variables and behaviors, such as the number of wheels or general information about computing automobile speed and direction. More specific objects might be specific brands of automobiles such as Toyota or Mercedes, for example. Inheritance permits more specific objects to inherit the specifications and behaviors of more abstract

objects. For the car example, a Mercedes could inherit all the behaviors and specifications that are common to all automobiles. Finally, polymorphism refers to the ability of different objects to respond in an appropriate manner to the same message. For example, the message “drive” sent to objects such as automobiles, bicycles, or ships in a vehicle simulation should cause these objects to respond in somewhat different but appropriate ways.

Consider Figure 2.1. Shown at the top of the figure is a template for describing classes and objects. The term *class* refers to a description of one or more similar objects and the term *instance* is used to designate specific objects that are members of the class. For example, a class **Car**¹ would contain a description of cars in general. Objects representing specific cars would then be instances of the class **Car**. The term *instantiation* refers to the process of creating a new object that is a specific instance of a class. The template shown in Figure 2.1 is basically a structure that provides an organization for naming a class and indicating if the class inherits information from other classes. Also, the template provides places to describe the internal parts of the class: class variables, instance variables, and methods. Class variables are variables that are associated with an entire class, including all instances of the class. Instance variables are variables that contain new values for each instantiation of a class. The term *method*² refers to the code inside an object that responds to a message sent to an object. A collection of methods defines the internal behavior of an object. Sending a *message* to an object causes a method to be selected and the code within the method to be executed. Messages provide the only interface between an object and the outside world.

As a concrete example, consider a class called **Toyota**. The class **Toyota** describes Toyotas in general. A class description can be built from the template discussed previously; as shown in the figure, class **Toyota** inherits from class **Car** and has several class variables and methods. Instances of the **Toyota** class would be specific Toyotas—*samsToyota* or *amysToyota*, for example. When one instantiates a class, a new specific instance of that class is created. The bottom of the figure shows how a class and several example instances are created from the template. The basic template is filled in to create both class and instance descriptions. A description of class **Toyota** is shown labeled as “Class Description Example.” The **Toyota** class inherits information from the class **Car**, meaning that variables associated with class **Car** and the methods that define the behavior of **Car** are inherited by **Toyota**. In class **Toyota**, the number of wheels (i.e., 4) and other variables can be set and will appear in all instances of the class. Shown below this class example are three examples of instances of

¹As a convention, the names of classes will be printed in bold throughout the text.

²Method names embedded in the text will be shown in italics.

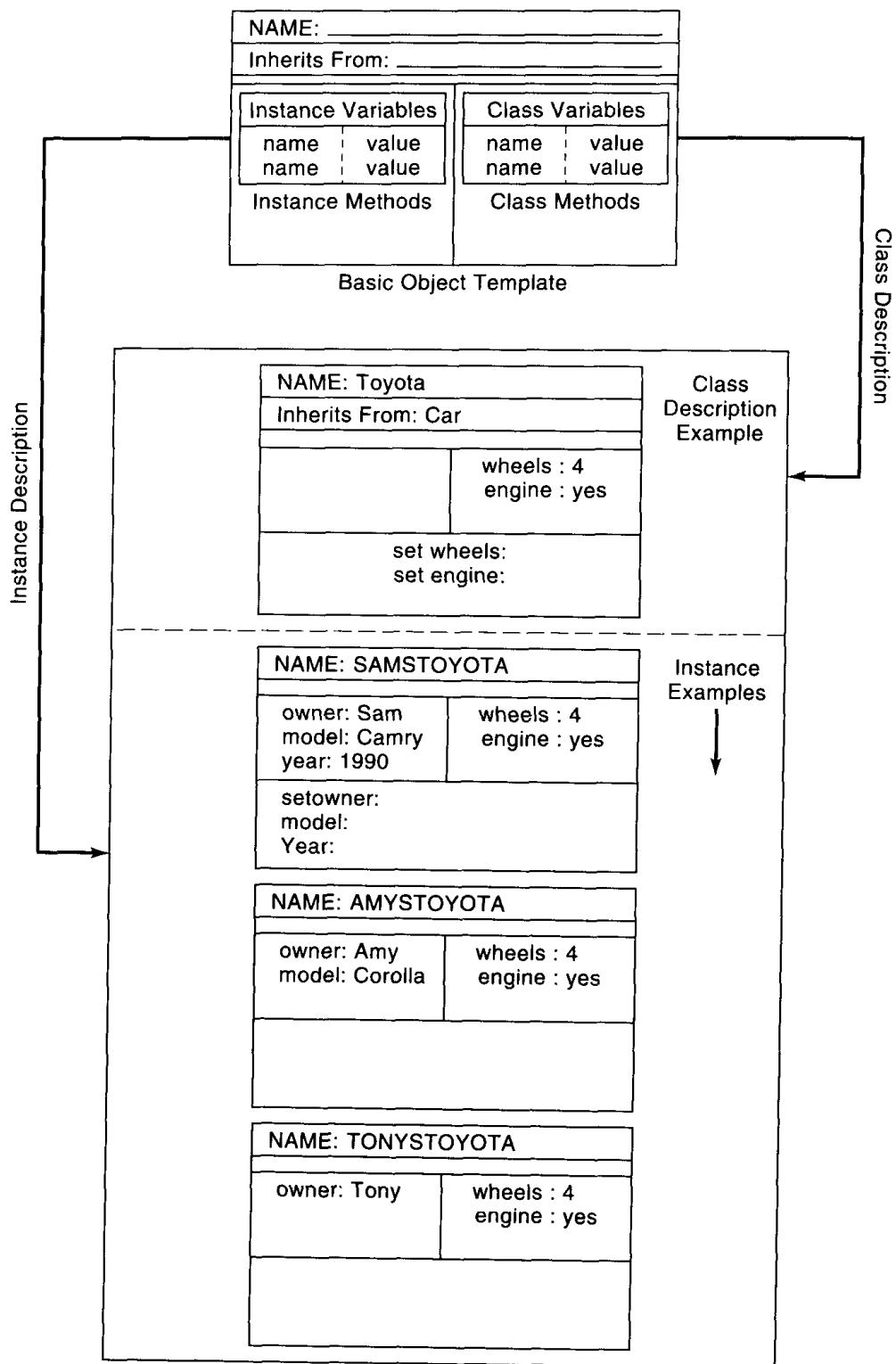


Figure 2.1 Creating Objects: Class and Instance Variables. A basic template is shown at the top. An example class and three instance examples are shown at the bottom of the figure.

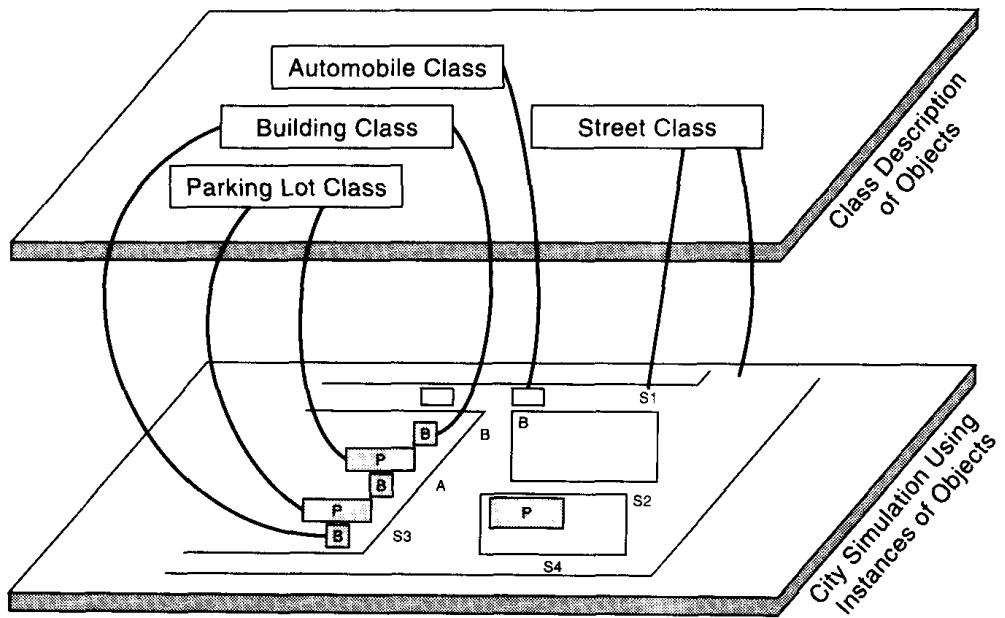


Figure 2.2 Describing the World with Instances of Objects. (P = instances of parking lots, B = buildings, A = automobiles, Sn = street number.)

class **Toyota**: samsToyota, amysToyota, and tonysToyota,³ all individual Toyota automobiles with different owners and models. To sum up, a class description permits describing information that is common to all instances, and the instance description permits describing individual objects that are specific instantiations of a class.

To describe the world using object-oriented methods, one can imagine defining all objects in the domain in which one is working according to the characteristics given previously. As an example, consider Figure 2.2 which shows how one might go about simulating a city block by defining objects. Such a simulation might be used for traffic planning or zoning problems. There are two layers in this diagram: On the bottom is a schematic visualization of a city block that includes parking lots, vehicles, streets, and building objects. Of course, there might be many more objects in this representation; these are simply illustrative. The top layer shows the different classes that are used to represent objects in this cityBlockWorld. The links between the class and instances of the class in the city block model are shown. This representation of a mental

³The convention for naming objects in this text will be to use descriptive names or to concatenate several words together with each concatenated word capitalized. Class names will start with a capital letter and names of instances will be uncapitalized.

organization of classes to support a problem organization is common in object-oriented problem solving. In this example, once the basic functionality of cityBlockWorld is determined, more objects with additional and realistic behaviors can be added providing, for example, the ability to study the influence of adding traffic lights or pedestrian walkways.

2.2 Defining Characteristics of Object-Oriented Methods

This section describes in more detail the characteristics of object-oriented methodologies. The terms employed are commonly used in describing methodologies in Smalltalk-80, yet are general enough to understand without first learning the syntax of Smalltalk-80. The various object-oriented languages and environments in existence differ in various ways in terminology, although the basic concepts are virtually identical. The Smalltalk-80 terminology used here is widely accepted, descriptive, easily understood, and will be used throughout the text.

Object Definition

One dictionary definition of an object is: "Anything perceptible by one or more of the senses, especially something that can be seen and felt; a material thing" (Morris, 1981). This definition is a familiar commonsense meaning; however, in the object-oriented programming world, objects also can represent abstractions of material things or even abstract concepts.

Figure 2.3 displays the anatomy of an object. Contained within an object are variables and methods. Variables are of two types: class and instance

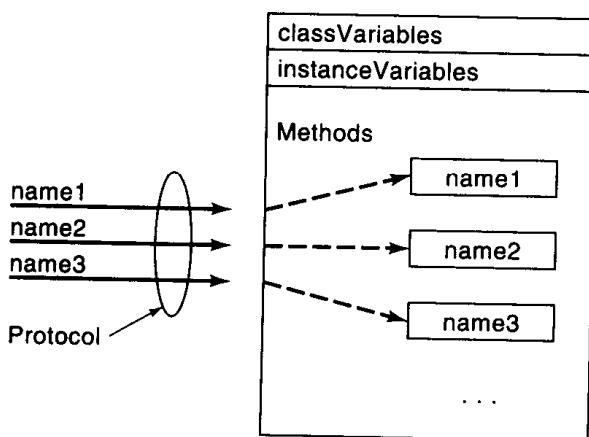


Figure 2.3 The Anatomy of an Object.

variables. Class variables are variables that are common to all instances of a class; instance variables may be different for each instance of a class. For example, as described previously, one might have defined a class called **Toyota** and several instances of this class—myToyota, yourToyota, and so on. Values of class variables would be shared among all the instances (e.g., wheels = 4), whereas values of instance variables could be different for each instance of **Toyota** (color = blue, color = red, etc. would be examples).

Messages are sent to objects to elicit different behaviors. Sending messages to objects is the only way to communicate with objects. Sending a specific message to an object triggers execution of the code for a *method* that has the same name as the message sent to the object. A typical object may contain many different methods that have been created by instantiating a specific class. Methods may reside in either the specific class or in class descriptions that are inherited by the instantiated class. Methods may also be grouped for understandability according to functionality into units called *protocols*. A protocol is merely a label that identifies a group of methods that perform related kinds of operations, for example, accessing, initializing, activating, terminating, and so forth. The use of protocols is primarily for assisting users in understanding the behavior of objects.

Object Attribute Implementation

Table 2.1 shows the basic terms associated with the characteristics of object-oriented programming. Each term will be described in detail next.

Abstraction

Abstraction is defined as: “The act or process of separating the inherent qualities or properties of something from the actual physical object or

Table 2.1 Basic Object-Oriented Characteristics

Abstraction: Separating the inherent qualities or properties of an object from an actual physical object to which they belong.

Encapsulation: The hiding of the implementation of an object from the outside world.

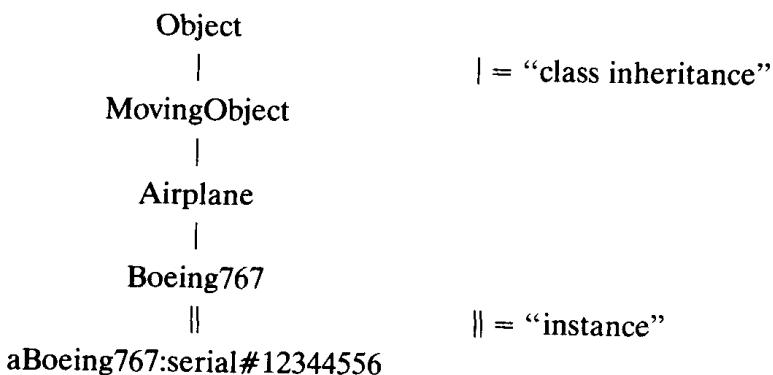
Inheritance: The ability of a class to use local state (variables) and behaviors (methods) from more abstract classes.

Polymorphism: The capability for many different objects to respond to the same message.

concept to which they belong" (Morris, 1981). The concrete manifestation of this definition in the object-oriented programming methodology is the collection of as many as possible inherent qualities or properties into abstract classes which can then be specialized by less abstract classes that inherit the characteristics of the abstract classes (see the following discussion on inheritance). Thus, rather than write descriptions for each specific type of automobile that exists, one attempts to create an abstraction of an automobile that contains all the qualities and properties that all automobiles have. Once this abstract class is described, each type of automobile can be described by specifying only changes or additions to the abstract class (this process is known as specialization).

There is often some confusion about the difference between abstraction and generalization. Generalization is defined as: "To reduce to a general form, class, or law" (Morris, 1981). Hence, generalization is law related, seeking principles that describe classes of things. In contrast, abstraction looks for concrete ways of amalgamating characteristics of class that are *common* to more specialized classes. Object-oriented programming methodologies use abstraction as a standard technique, not generalization.

In creating classes for representing things, a programmer should consider the following methodology. First, look for the most abstract characteristics of the object that is to be represented. For example, consider the following simple representation hierarchy:



Starting from the bottom, the task is to represent a specific Boeing 767 aircraft. A specific object is called an *instance*. The specific Boeing 767 is an instance of a class called **Boeing767** from which many instances of the aircraft can be fabricated. Note that the **Boeing767** class inherits information from **Airplane**, which, in turn, inherits information from **MovingObject**. The utility of this methodology lies chiefly in the strong capability of reducing the amount of information that must be repeated in describing new things. The characteristics of all moving objects can be inherited not only by airplanes but by automobiles,

animals, astronomical objects, ships, and so forth. Likewise, the characteristics of all airplanes can be inherited by all different types of aircraft that are represented.

Encapsulation

Encapsulation refers to capturing the state and behavior of an object entirely within the object. As indicated previously, object may contain several different types of variables: class variables, instance variables, and even local variables, which are known only to the methods within which they are defined. All of these variables are *encapsulated* within the object and define the state of the object. In addition, the behavior of an object is encapsulated within the object as a collection of methods that define how an object will behave when messages are sent to it.

Inheritance

Inheritance is the ability of objects to obtain information about their internal states and methods from more abstract ancestors. Consider the example given in Figure 2.4 in which examples of objects such as ships, wheeled vehicles, and planes inherit information from the abstract class **MovingObject**. As shown, **MovingObject** is the superclass of **Ship**, **WheeledVehicle**, and **Plane** and, as such, should contain all variables and methods that would be associated with ships, vehicles, and planes. Superclass is a term that refers to the next most abstract object in the ancestral hierarchy. Conversely, subclass is a term that refers to the class that inherits from the more abstract class. **MovingObject** might contain variables concerned with direction, speed, and weight and methods that contain code for computing speed, direction, and so on. These variables and methods would be equally applicable for any subclasses constructed using **MovingObject** as a superclass. **WheeledVehicle** is also an abstract class in that it serves as a superclass for all types of vehicles that have wheels. **Car** would inherit all information from its superclass and, perhaps, add variables such as ‘wheels’ or ‘turnSignalType.’ Note the additional two abstract classes: **Power-WheeledVehicle** and **UnpoweredVehicle**. The continual breaking down of the classification hierarchy in this way permits a quite specific way of defining an object taxonomy (i.e., a classification) of the world.

The major point to be made in this discussion is that inheritance permits specific objects to be created by using a hierarchy of classes. This technique enables designers to select classes that are already built and to specialize these classes to instantiate objects that are needed for a design.

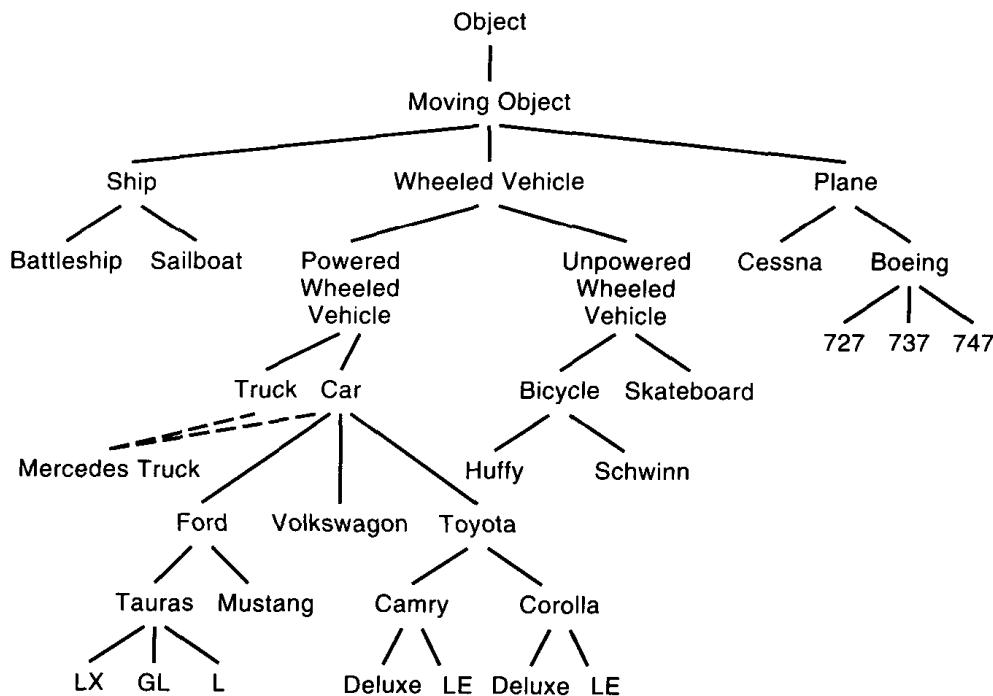


Figure 2.4 Hierarchical Class Description. Dotted lines indicate multiple inheritance.

Hence, two advantages are readily apparent: (1) reducing the amount of code by sharing code in abstraction hierarchies and (2) simplifying the creation of new objects by specializing classes.

Inheritance may be strictly hierarchical or tangled, as shown in Figure 2.4. Tangled hierarchies, also known as multiple inheritance hierarchies, permit inheritance of information from multiple ancestors. Figure 2.4 is primarily a hierarchical representation; however, consider the portion of the inheritance lattice at the bottom left of the figure. A composite class called **MercedesTruck** is shown that inherits characteristics from both **Car** and **Truck**. This definition would make sense in cases, for example, in which a truck may have been created using engines or other parts from the Mercedes automobile line.

Polymorphism

Polymorphism, in general, refers to having many parts or forms. In object-oriented programming, polymorphism is the capability for different objects to respond to the same meaning of a message. Figure 2.5 gives an example of polymorphism. Four objects are shown in this example: (1) aValve,

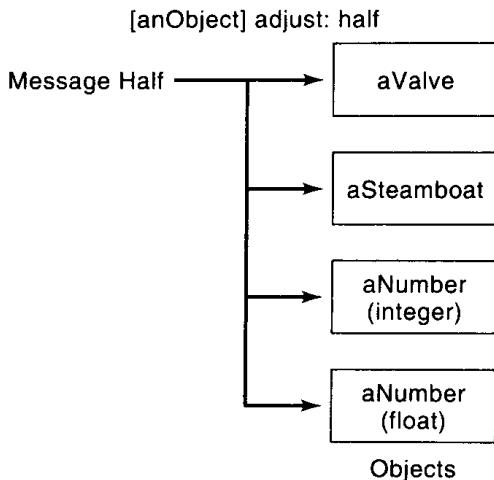


Figure 2.5 An Example of Polymorphism.

(2) aSteamBoat, (3) an integer number, and (4) a floating-point number. The same message is sent to each object:

anObject adjust: half

where *anObject* is the object, *adjust:* is the message sent to the object that activates the method *adjust:*, and *half* is the argument (itself an object!) passed along with the message to the method. In Smalltalk-80, the colon after the message name indicates that there should be an argument following the message name. The message *adjust:* has a different implementation for each of the four objects. The valve should be turned to half-open, the steamboat's speed might be adjusted to half of full speed, and each of the two different kinds of numbers would be halved, each according to the type of number it is.

The point to this discussion is that defining polymorphic behavior of objects permits a more natural way of understanding the behavior of objects. In the preceding example, the message *half* results in different behaviors in each of the examples, even though the same message name is sent to each object. Rather than having to define different messages for each object such as *halfSpeed* or *halfTurn*, the capability of using the same message name simplifies the understanding of the behavior of the objects. Without this capability, individualized message names could proliferate, thus increasing the number and complexity of messages that need to be understood.

2.3 Representing Objects

There are many perspectives on representing objects in the world. Some objects may require only representation of structure, whereas others may

require representing both structure and behavior. A rock or picture could be examples in which the behavior of the object is incidental to its structure. Many objects require both structural and behavioral specifications, such as an automobile or television, for example. Still other viewpoints might stress *function*, that is, what the object is supposed to do, *teleology*, the design intent for the object, or *causality*, what other objects an object may influence. Representing the structural and behavioral characteristics of objects is discussed next.

The Structural Representation of Objects

To analyze problems and design new material objects using computer-assisted systems, a key problem is the representation of objects in the world and in the computer environment. Many different styles of representation methodologies are used. The most common representational structure found in actual programs is often ad hoc, that is, most systems are constructed using no formal representation structure other than programming language constructs such as constants, procedures, queues, structures, and so forth. The problem with using standard language features for description of knowledge is that frequently more robust representation mechanisms are needed. Consider a part of Figure 2.4 described in textual form as shown in Table 2.2. This textual representation of an object hierarchy is clearly more difficult to read and understand than the graphical form of the hierarchy. Once the hierarchy grows beyond only a few elements, graphic representations are essential to understanding the relationships in the hierarchy. Various kinds of textual representation mechanisms have evolved in different programming languages; perhaps the most active field conducting research on representation mechanisms is artificial intelligence.

Research in the field of artificial intelligence (AI) during the last two decades has looked intensely at the representation problem, finding that rules

Table 2.2 Textual Form of a Representation Hierarchy

(Object
(MovingObject
(Ship
(Battleship)
(Sailboat))
(WheeledVehicle)
(PoweredWheeledVehicle
(Truck)
(Car
(Mercedes)
(Volkswagen)
(Toyota
(Camry)
(Corolla))))))

for representing inferential knowledge and frames for representing facts and structures in the world can be employed to solve a wide range of problems. Frames (Fikes and Kehler, 1985) are often used to describe objects in the world and can also be used to specify behavioral characteristics of objects as well. Frames are often said to be object-oriented; however, frame-based object descriptions differ in several ways from the object-oriented mechanisms described previously. The differences and similarities of frame and object-oriented representations will be described next. Frames are important from a historical perspective because object-oriented methodologies are derived, in part, from the study of representing knowledge with frames.

Frames are a representation mechanism that permit a structured representation of objects. Table 2.2 showed how part of the hierarchy shown in Figure 2.4 could be represented as a simple definition. The syntax of Table 2.2 is that of the Lisp programming language, in which parentheses are used to group items. Other syntactic styles are used in other languages, as shown in Table 2.3, but all have the same essential meaning. Frames are generally described as having slots and values in the slots. In the example shown, slots are the names of the entries shown, that is, occupants, color, and so on. Note that the first three languages used for representation in this example define a structure and the *types* of values that go in the slot. In contrast, Smalltalk-80 employs dynamic variable binding and thus requires no initial definition of the type of value that is associated with its instance variables. Frames also permit representation of behavioral information in the form of *predication* on slots and/or structures. Predication means that a procedure can be activated when information is put into a slot or retrieved from a slot. The technique is also known by the term *active value*. Although frames allow a combination of both structural and behavioral descriptions in the same declaration, the implementation and use of frames is considered by some to be a bit cumbersome.

Object-oriented languages permit representing things in much the same manner as frames. The last entry in Table 2.3 shows the Smalltalk-80 style for representing the same information as the other languages in the table.

In this example, there appears to be no advantage to the use of an object to capture framelike information. Yet, as will be illustrated, the major difference between objects and frames is in the explicit capturing of object behavior as methods encapsulated in an object, a task that is much simpler in objects than in frame-based or conventional languages.

Whole-Part Hierarchies

Whole-part hierarchies differ from inheritance hierarchies. Inheritance hierarchies provide a way of describing more specialized objects by

Table 2.3 Syntactic Styles in Several Computer Languages for Representing Information in Frames**Lisp Style**

```
(defFrame WheeledVehicle
  (wheels int)
  (engineSize int)
  (occupants symbol)
  (numberOfSeats int)
  (color symbol)
)
```

C Style

```
struct WheeledVehicle {
    int wheels;
    int engineSize;
    symbol occupants;
    int numberOfSeats;
    symbol color;
};
```

Pascal Style

```
type WheeledVehicle record
    wheels : symbol;
    engineSize : int;
    occupants : symbol;
    numberOfSeats : int;
    color : symbol;
    end;
```

Smalltalk Style

```
MovingObject subClass: #WheeledVehicle
instanceVariables: 'wheels
    engineSize
    occupants
    numberOfSeats
    color'
classVariables: ''
```

inheriting state and behavior from more abstract objects. Whole-part hierarchies have an entirely different representation role. Consider Figure 2.6 which describes a decomposition of an automobile. On the left of the figure is shown the collection of variables that describe all the things that make up the car. In this example, only a number of the parts are shown. Each of the parts has attached to it a description of the parts as shown in the middle column in the figure. For each of the parts illustrated, further decomposition is possible as well, as shown on the right side of the figure. This decomposition permits describing all the objects in a complex object in a hierarchical way with increasing specificity. The *whole* is described in terms of its *parts*, in a hierarchical way—hence, the term *whole-part hierarchy*.

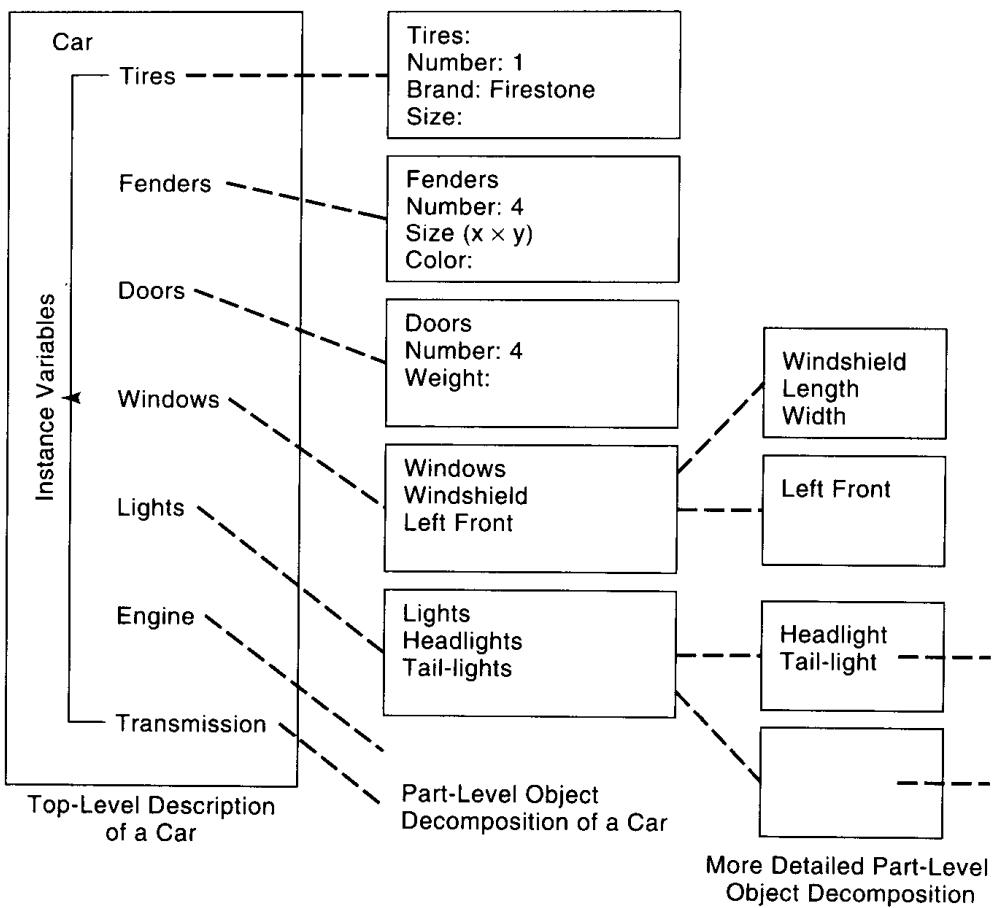


Figure 2.6 Whole-Part Object Decomposition of an Automobile.

The Behavioral Representation of Objects

In addition to representing the structure of objects, one needs to represent the behavior of objects. In objects, behavior is represented by methods, as discussed earlier in this chapter. In frames, behavior is represented by active values or predicates, that is, code attached to a slot which runs when a value is retrieved or stored in a slot. This same behavior can be easily captured in objects, as shown next.

Consider the frame given in Table 2.4. A frame called **Car** is defined and an instance of **Car** called **myToyota** is created. In the slot for “**“numberOfDoors”**” is found a predicate called **doortest**, which is a function that is run whenever a value is inserted into this slot. Hence, if the number of doors inserted into the slot exceeds 5, a warning message is printed. This same type of

Table 2.4 Implementing Active Values in Frames and Objects*

Frame Definition	
(defFrame Car	; define a car frame
(model symbol)	; a car model
(numberOfDoors integer doortest)	; check the number of doors
)	
(defInstance Car myToyota	; myToyota is an instance of Car
(model Toyota)	
(numberOfDoors 19)	
)	
(defun doortest (number)	
(if (number > 5) then (print 'Too Many Doors'))	
Object Definition	
Object subclass: #Car	
instanceVariables: 'model numberOfDoors'	
classVariables: ''	
Method Names	
defineModel: aModel	"a method to name the model"
numberOfDoors: aNumber	"a method to set the number of doors"
self doorTest: aNumber	"send the number of doors to doorTest"
doorTest: aNumber	"a method to test if too many doors"
(number > 5) ifTrue:	
[Print 'Too Many Doors']	

* The details of object description syntax will be given in subsequent chapters.

behavior can easily be duplicated using objects by creating an object also called **Car** and defining methods internal to the object that permit defining the car's model name and the `numberOfDoors`. Testing how many doors are allowed requires the running of another method called `doorTest` whenever the instance variable "numberOfDoors" is set.

The mechanism of active values is particularly useful for implementing systems in which values are displayed on gauges. For example, a frame may contain information about a panel of gauges that needs to be displayed on the screen; when a new value is inserted into a slot, the gauge should be updated and displayed. Similarly, objects can contain methods that drive gauges or other visual representations that display the state of an object.

2.4 The Causal Nature of Engineering Systems

Causality refers to the relationship between cause and effect. In engineering, systems are frequently composed of many parts which have cause-and-effect relationships between the parts of a system. Simple examples abound in the real world. For example, push-button radios or traffic lights exhibit the

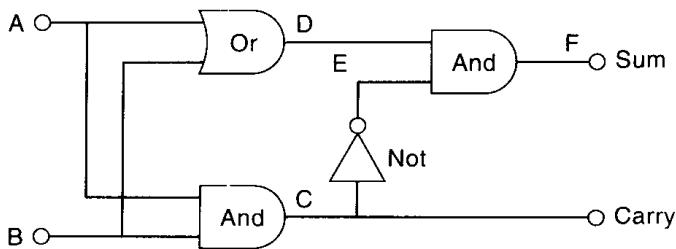
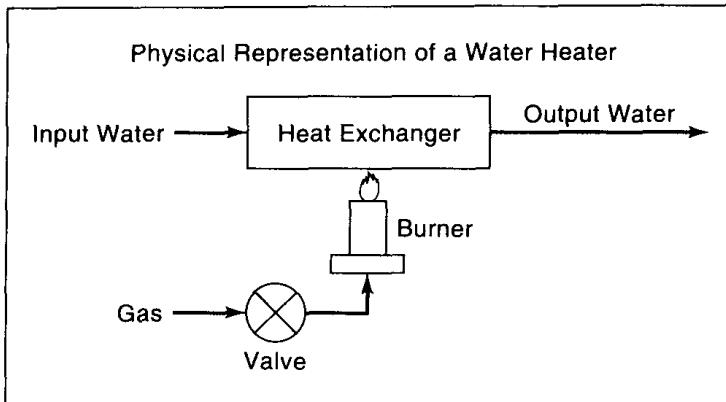


Figure 2.7 A Simple Digital Circuit: A Half-Adder.

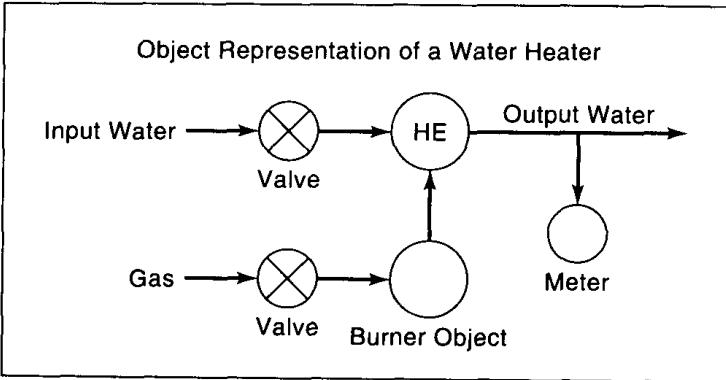
behavior that causing one button to be on or one light to be on causes the remaining buttons or lights to be off. Simple commonsense causality exists such as the knowledge that applying pressure to the brakes of a car causes the car to stop; in reality there are several causal linkages in this chain, yet we abstract the overall causality to understand the functioning of the mechanism.

Figure 2.7 shows a simple digital circuit (the same one shown in Figure 1.3). If we view each element in this circuit as an object and each wire as an object that connects circuit element objects together, we can obtain an idea of how causality works for digital circuits. Imagine that a signal is sent to the ‘OR’ gate input marked ‘A’ in the diagram. The ‘OR’ gate contains a *constraint*, that is, a rule or condition that must always be true which describes the behavior of the ‘OR’ gate. When either input of the ‘OR’ gate goes high, the output of the gate will become high. The wire at the top of the diagram between the ‘OR’ gate and the ‘AND’ gate at the output of the circuit propagates the signal to the rightmost ‘AND’ gate. At this gate, the constraint propagates a 1 to the output and lights the light bulb, if the remaining input to the ‘AND’ gate is high. This movement of information between objects is a simulation of physical causality and is easily modeled using object-oriented methods. Furthermore, the modeling approach described here is a rudimentary technique that could be used for implementing computer-aided-design systems.

As a second example in a separate domain, consider the hot-water heater schematic shown in Figure 2.8. The physical representation of the water heater shown at the top of the figure shows a schematic of the physical arrangement of the water heater components. Water flows from an input through a heat exchanger to produce hot water at the output. Heat is produced by a burner that heats the heat exchanger. At the bottom of the figure is shown a representation of the same water heater using objects. Each object has internalized its behavior, that is, how it will react to different kinds of inputs. For example, sending more gas to the burner will cause the burner to produce more heat and hence (causally) the heat exchanger to impart more heat to the water.



(a)



(b)

Figure 2.8 Representing a Hot-Water Heater.

2.5 Representing Time

In the examples given about causality, the influence of time has been omitted. To provide more complete representations of the real world, it is necessary to include time as part of the description of real systems. In the examples just given, time would be a factor in that there would be small delays in each of the digital circuit gates described and there would be a lag between application of heat to the heat exchanger and the water reaching a certain temperature. In other design examples, such as designing a traffic system or a flush toilet, in which events occur sequentially, time needs to be explicitly represented.

To represent time, two methods are frequently used for simulation. One can keep track of the time at which things occur by posting information on

a central blackboard or queue. As events occur, information about future events can be posted on a queue. These events can be sorted according to time and removed from the queue one by one to simulate the activity of a system. In another, but similar technique, incoming information (e.g., a signal) can be sent to an object, and the object can delay forwarding information for an amount of time to the objects to which it is connected. In the digital circuit example, after receiving a signal at its input, a gate object might wait for a predetermined amount of time before forwarding information on to the objects to which it is connected. These techniques will be described in detail in Chapters 10 and 13.

2.6 Modeling and Simulation

A model is normally defined to be a representation of a physical object. However, models of mental processes may also be represented—(so-called “mental models” [see, e.g., Gentner and Stevens (1983)]. In the context of the discussions of representation in this chapter, models can be defined as structured declarative representations that capture the physical attributes of real things. Using this definition, frames can capture the representation of things as can objects arranged in class hierarchies. Once behavior is added to a model, in the case of frames using active values or by defining methods in objects, the models become runnable and thus can stimulate the behavior of systems.

Modeling using objects is as simple as defining the characteristics of the objects that are to be modeled. Following the methodologies discussed earlier in this chapter, class variables, instance variables, protocols, and methods are defined for classes. These classes typically inherit variables and methods from classes that have already been created. New models of abstract objects can be quickly created by recognizing that the model of an object differs only in minor ways from a more abstract object that already exists. A complete methodology for modeling using objects will be presented in the next chapter.

Simulation of any system can be accomplished using procedural code. For decades, many different computer languages have been used for simulating various physical and abstract processes. The advantage to object-oriented programming methods for simulation lies in the modularity, understandability, and ease of specialization capabilities inherent to the methodology. Of special interest in using simulation in Smalltalk-80 is the existence of the Smalltalk-80 virtual machine which has the capacity to schedule and control multiple processes that appear to run simultaneously. There is no actual multiple-processor machine—hence, the term *virtual*, meaning existing in essence but not in actual fact. Objects that have self-contained behaviors and operate either independently or share resources can be readily modeled by making use of the facilities

for controlling multiple processes in Smalltalk-80. This topic will be discussed in Chapter 14.

2.7 Representation of Complex Systems: An Example of an Advice-Giving System

Complex object-oriented systems created for use in engineering or other domains commonly consist of many interacting objects. A typical Smalltalk-80 system will consist of objects that describe a *model* of the domain, a *controller* that interfaces the user with the system, and a *view* that displays what the user sees. Each of these subparts of a system will, in turn, contain many interacting objects. The virtue of object-oriented methodologies for creating complex systems is that complexity can often be reduced and existing classes may be specialized to achieve a certain degree of parsimony. To provide a flavor for how to represent a complex system, the architecture of a common advice-giving system will be described next.

The term *advice-giving system* describes what many types of engineering systems do. Most computer systems created to serve the needs of engineers who analyze problems and design entities require a strong component of advice giving. These systems contain sufficient domain knowledge, algorithms, and heuristics to advise users about problems encountered. Rudimentary advice givers include expert systems (Hayes-Roth, Waterman, and Lenat, 1983) that capture the knowledge of the experts in a domain, CAD tools that provide simulations of the real world, and signal analysis systems that transform signals from one domain to another. More complex advice-giving systems would provide help, tutoring, advice, and even adjust the presentation of a system to the cognitive level of the user.

As an example of an advice-giving system, consider the architecture of the problem-solving system that was shown in Figure 1.2. This general model is suitable for use in many different engineering domains. Next, the general requirements for casting this model into sets of interacting objects will be considered and a specific example provided for the signal analysis system top-level view that was given in Figure 1.4.

Requirements

Systems that give advice need (1) to contain the knowledge to support the advice given and (2) to understand the needs of users and how to present knowledge to users. Systems that display some degree of intelligence should not only contain the knowledge but also provide the facilities to assist the user in

reasoning about the problem. To factor the problem into several parts, the object characteristics described next are divided into the three parts identified previously: (1) objects that deal with knowledge and data (the "model"), (2) objects concerned with control (the "controller," i.e., the interaction of the user with the system through the mouse and keyboard), and (3) objects that implement the "view," which consists of how information is presented on the screen. This factoring of information about construction of a relatively complex system permits conceptually separating parts of the system into logical groups, and, as we shall see in Chapter 4, fits directly into the standard Smalltalk-80 paradigm for description of systems. The terms used in describing the elements of the advice-giving example system are cast in Smalltalk-80 terminology which will be presented in increasing levels of detail in subsequent chapters.

The Model

The model may be considered to be a collection of objects that represents knowledge and data in a system. In a very simple system, the model might be only a data structure of some type such as a "dictionary" or an array of numbers. In larger systems, the model often includes a wide range of data and knowledge types. Next, several common objects found in a model for an advice-giving system are listed.

Dictionaries. A simple data structure that is frequently used is a dictionary, which consists of a set of associations of key and value pairs. For example, a dictionary would be a useful representation to keep track of information about things that the system knows about. For example,

SignalAnalysisMethodsDictionary:	
Key	Value
sineWave	anObject that contains information about sine waves; e.g., amplitude and frequency
squareWave	anObject that contains information about square waves
	etc.

In this example, each key in the dictionary would contain an item, which, when assessed, would return an object that provides more information about the item.

Collections. Collections are a popular way of capturing information. Many kinds of collections might be used for information storage; most popular are ordered collections that behave like queues and sorted collections in which the order of

the collection is maintained according to some predetermined metric such as time. When representation of two-dimensional or higher-dimensional knowledge or data is unnecessary, collections of various types provide a simple way to implement a model. In our example, a collection could be used to store the data points that represent the ordered samples from an input signal and also store the results of the fast Fourier transformation of the signal.

Networks. For more complex information, network representations are the most useful representation mechanism for static knowledge. Frames, as discussed previously, may be represented in network fashion. Nesting of dictionaries can be used to implement a network (i.e., the value in a dictionary contains another dictionary, etc.). A network could contain information that assists the user in understanding how to use an advice-giving system. Chapter 11, in fact, provides an example of using a network for this purpose.

User Models. In an advice-giving system, one object that is needed is a user model to keep track of what the user does while using the system. In order to make a system appear intelligent, this model would need to contain sufficient information to permit inferences to be made about what the user does and does not know. A dictionary would be useful for storing user models. Keys in the dictionary could be the name of each user and the value associated with each key would contain information about each user's level of knowledge.

Algorithms. Algorithms should also be included in the model of an advice-giving system. Algorithms are an essential element of the model because procedural evaluations are frequently needed to interpret and evaluate data. Algorithms are best implemented as methods in Smalltalk-80 or in procedures written in traditional languages that are coupled to Smalltalk-80 (see Chapter 12).

Heuristics. To give advice, heuristics may be a required part of the model. A heuristic is a rule of thumb, often used by humans to make inferences about a situation. Popularized in the field of expert systems (Hayes-Roth, Waterman, and Lenat, 1983), rule systems that code heuristics in the form of IF → THEN rules have become a standard part of many traditional engineering systems. The role of heuristics in an advice-giving system is to capture human reasoning about a domain. Heuristic reasoning is particularly useful in engineering design in which it may be difficult to specify all decisions algorithmically.

Control Objects

Controller. A controller coordinates a view (i.e., what the user sees), a model (i.e., the data or information), and user actions. In Smalltalk-80 there is a

systemwide controller known as the **ControlManager** that coordinates all views that are presented on the screen. In addition, there are other controller classes that provide the capability of associating pop-up menus with the pressing of mouse buttons and permit the sensing of the location of the cursor on the screen.

For creating controllers for the example advice-giving system, the primary responsibility of the systems designer is to decide what actions should be taken in any view that is shown on the screen. The typical control methodology is to provide a pop-up menu in each view which, when activated by the mouse, permits the user to make selections, which subsequently causes some action to be taken that influences what is shown on the screen. An alternative to this method is to always display on the screen all possible control actions by using buttons, which when pressed cause the required action.

View Objects

In a graphics user interface, most interactions of the system with the user are handled graphically. Smalltalk-80 views are displayed in windows of the host windowing system that permit opening, closing, resizing, moving, and collapsing of windows. Rectangular areas within a window, known as "views," may contain many different types of user interface mechanisms such as lists, buttons, settable dials, text, and so forth.

Considering the preceding description of objects in terms of the architecture given in Figure 1.2, objects implementing views and controllers for the views are contained in the top-level layer of the system labeled "graphics user interface," whereas the model is contained in the second layer. Knowledge attendant to the models is contained separately in a data/knowledge base that is conceptualized as a separate block at the bottom of this figure.

Architecture of the Signal Analysis Advisory System

Figure 2.9 displays an architecture for a signal analysis advisory system constructed along the same lines discussed previously. The architecture is quite similar to the architecture of Figure 1.2 except that it contains the objects that are required for the specific application.

The goal of the signal analysis advice-giving system is to implement a system that provides advice to users about signals that they acquire. Advice in this simple example is fairly limited; for example, the advice might be limited to the interpretation of a signal that has been acquired.

The user interface shown in Figure 1.4 for a signal analysis system contained no advice feature. If a button were added on the right side of the

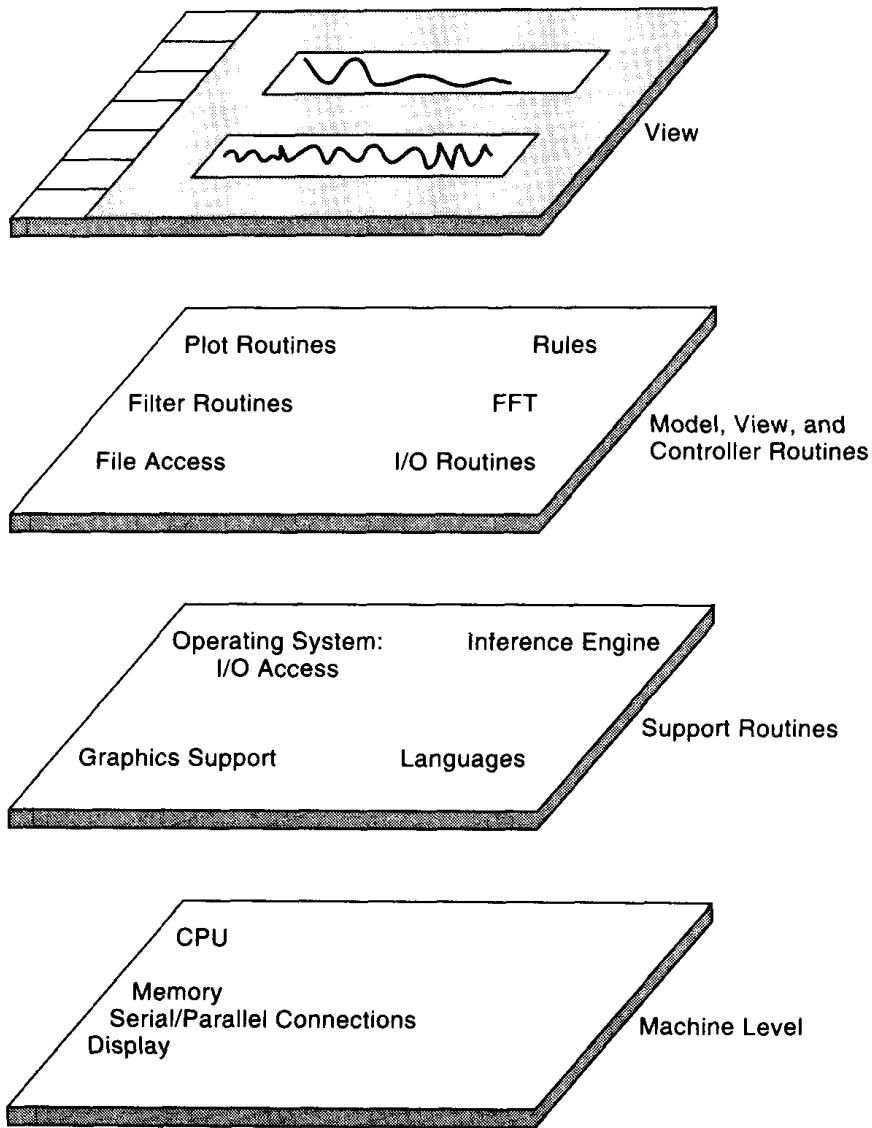


Figure 2.9 An Architecture for an Advice-Giving Signal Processing System.

view which, when pressed, would give an interpretation of the signal, the screen might look like the rendition shown in Figure 2.10. The user would employ the system to acquire signals, as described in Chapter 1; when done, the interpret button would be pressed and a textual discussion of the signal on the screen would be given.

Implementation of such an advice-giving system could be accomplished using the kinds of classes shown in Table 2.5. These classes are commonly used for implementing the tasks described. In subsequent chapters,

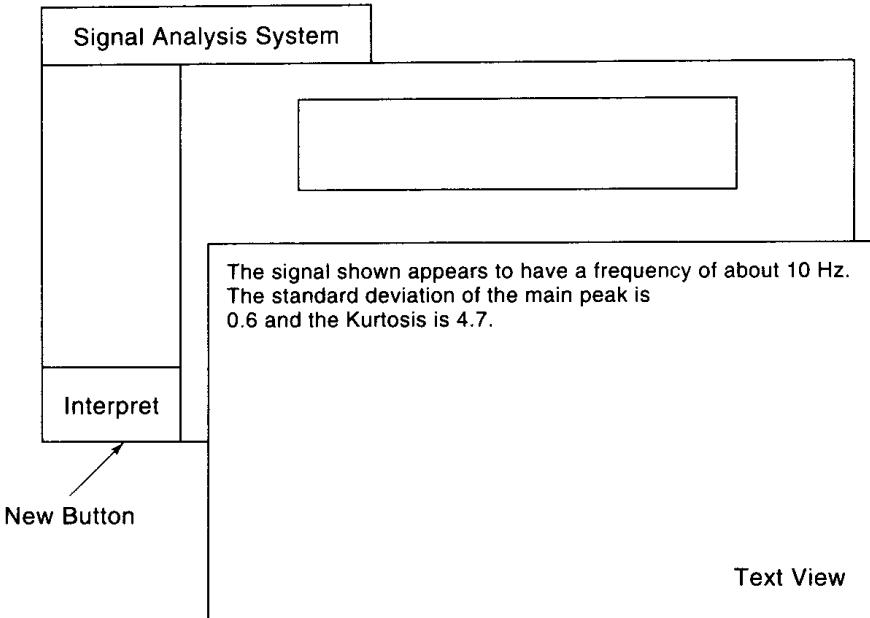


Figure 2.10 Addition of an Analysis View to the Signal Analysis System Shown in Figure 1.4.

Table 2.5 Classes Used for Creating an Advice-Giving Signal Analysis System

Model Classes

OrderedCollection	Stores the signal
OrderedCollection	Stores the FFT
Dictionary	User model actions stored
Dictionary	Rules

Controller Classes

ControllerWithMenu	Provides mouse and menu control
--------------------	---------------------------------

View Classes

SignalView	Shows the signal
FFT View	Shows the FFT
Buttons	Initiates actions when the user presses a button

methods for implementing all the features of this type of user interface, along with the associated controller and the model characteristics, will be discussed.

2.8 Summary

This chapter has given a basic introduction to common terms associated with objects and explained how to represent both structural information and behavioral information using objects. Comparisons were made with similar ways of representing information in different computer languages and in the frame representation paradigm. Representing time and causality were described and a concluding example presented that conceptually explained how to build an object-based engineering system for analyzing signals.

References

- Fikes, R., and T. Kehler (1985). The Role of Frame-Based Representation in Reasoning. *Communications of the ACM*, Vol. 28, No. 9, September, 904–920.
- Gentner, D., and A. L. Stevens (Eds.) (1983). *Mental Models*. Lawrence Erlbaum Associates, Hillsdale, NJ.
- Hayes-Roth, F., D. A. Waterman, and D. B. Lenat (Eds.) (1983). *Building Expert Systems*. Addison-Wesley, Reading, MA.
- Morris, William (Ed.) (1981). *The American Heritage Dictionary of the English Language*. Houghton-Mifflin, Boston, MA.

Exercises

2.1

Pick a common object with which you are familiar and draw a class inheritance hierarchy which could be used to instantiate the object. Try to show the breadth of the tree created and be sure to pick an example with several levels of abstraction. The key to successful completion of this problem is the selection of an object that you completely understand. Common objects in the world are acceptable such as kinds of machines, computer systems, dishwashers, and so on.

2.2

For the same physical object considered in the previous problem, create a whole-part hierarchy.

2.3

Next, add instance variables to the hierarchy of Exercise 2.1 at each level created, showing how the more specialized classes can inherit and use the variables defined at more abstract levels.

- 2.4 Give an example, different from those presented in the chapter, of the basic object-oriented characteristics listed in Table 2.1. You may wish to use the same example that you picked for the preceding exercises.
- 2.5 In the architecture given for implementing a signal-processing advice-giving system, a hierarchical representation scheme was used. Pick another example and sketch the same type of hierarchy for your example. Other appropriate domains to consider for this task would be (1) a CAD system or (2) a control system.
- 2.6 How does modeling differ from simulation?
- 2.7 Explain the distinction between classes and instances of classes.

Object-Oriented Analysis

3.1 Introduction

The dictionary definition of “analysis” is an appropriate starting point for understanding object-oriented analysis. *Analysis* is defined to mean “the separating of a material or abstract entity into its constituent elements” (Webster, 1989). Analysis is a key component of design, normally preceding the creation of new designs. In order to build a new entity or object, designers analyze various requirements for building a new object such as the technology needed, feasibility, cost, and analogy to previous designs. Object-oriented or not, design requires analysis as a precursor. In this chapter and the next, an attempt will be made to show that traditional engineering analysis and design can be closely matched to object-oriented analysis and design. This chapter will discuss analysis methodologies and provide an analytical methodology that can be used for object-oriented analysis. The analysis techniques discussed here are designed for use with Smalltalk-80, but have applicability for other object-oriented languages as well. There are many other approaches to object-oriented analysis that employ different but complementary approaches [see, e.g., Coad and Yourdon (1990) and Shlaer and Mellor (1988)]. Chapter 4 will map the analytical process described into a design methodology.

Analytical thinking about decomposition of both material and abstract entities is quite similar. For example, consider the following engineering systems: analyzing the requirements for building a car, simulating a traffic system, building a CAD system, and creating a signal analysis system. The process of analysis is similar for each case. Both material and conceptual entities can be usually subdivided into components. Indeed, the engineering methodology of breaking a problem into parts normally permits problem solving to follow this

paradigm. Separation of a physical object into parts is easily understood. For example, an automobile may be subdivided into its constituent elements: engine, body, transmission, interior accommodations, tires, and so on. Likewise, conceptual decomposition can be found in simulation and analysis systems. A traffic simulation system may be decomposed into various objects, such as simulated cars, simulated time, and so forth; a signal analysis system may be subdivided into algorithms, acquisition objects, and presentation objects.

The goal of this chapter is to describe and demonstrate analytical techniques that can be used for object-oriented problem solving. The techniques presented should work equally well with a variety of different problem types, for example: (1) object decomposition for information organization, (2) analysis of objects required for building an analysis system, (3) creation of simulation systems, and (4) organizing objects in a design.

The context in which the information in this chapter will be presented utilizes the same generic descriptive terms employed in Chapter 2 and in the Smalltalk-80 environment. The terms class, instance, class variables, instance variables, protocols, and methods, as explained in Chapter 2, will be used as a basis for describing the analytical methods.

3.2 Analytical Methodologies for Building Object-Oriented Systems

General Concept

The purpose of an initial analysis of a problem is to provide an organizing structure or framework in which the problem can be further understood and decomposed. In general, the process is to identify all objects in the problem and their behaviors. At the outset, the global problem should be considered by describing the overall conceptualization of the problem. It is helpful to prepare a natural-language outline of the problem, roughly identifying all the possible objects that might “live” in the final system. This preliminary work will help organize one’s initial thoughts about how to approach the problem. For example, if one is creating a system to simulate digital circuits, a name could be selected to identify the system, such as *DigitalCircuitSimulation*, which can be used subsequently to create a class named **DigitalCircuitSimulation**. This class ultimately would be used to create a *DigitalCircuitSimulation* object that would coordinate all objects in the application.

Analyzing the Problem

Given an idea for creating a system, it is a short step to naming the organizing object. Preparing for the next task, understanding the constituent

Table 3.1 Definition of System Description Terms

Concept	Explanation
Teleology	The purpose of a design The design intent Specifications
Function	The designed activity of an object The way something should work Algorithmic or heuristic definitions
Behavior	Manner of behaving or acting Relationship between input and output of a system determined from measurement The way something works
Structure	The way things are constructed or connected Pertains to both material and abstract things

parts of the problem, requires more in-depth understanding of the problem domain and alternative solutions. Precisely “*what is it that is to be accomplished?*” with the system being created is a typical question that can be asked at the outset of designing any system. Most often, the answer to this question requires a consideration of a number of factors prior to proceeding with detailed design. Teleological, functional, behavioral, and structural specifications must be considered to provide a mental framework for a design. Table 3.1 summarizes some common definitions of these terms. Teleology refers to design intent, that is, what the design is supposed to accomplish. Although it is usually clear to a designer what the intention is in designing a system for some purpose, the system that is designed does not always exhibit the intended functionality. Function refers to the specification of what a system is intended to do according to design specifications. In principle, behavior should exactly mirror function; that is, the behavior of a system should implement the functionality of a system. In fact, this is often not the case. Hence, it may well be better to define behavior as the relationship between the inputs and outputs of a system; that is, what the system actually does. Physical specifications, in the case of physical entities, specify the structure of an object and how parts fit together. In the case of software, as in the specification of a user interface, for example, a structural specification would include how the system appears to the user and how the user interacts with the system.

Table 3.2 provides an example of the use of these terms for a simple amplifier system. In examining the way these terms are used, one should note clearly that behavior is the way something actually works, whereas function specifies the way something is designed to work. One can extend these concepts

Table 3.2 Terminology Utilization for an Amplifier

Definition: An amplifier is an electronic device that changes (amplifies) small signals into large signals. A common example is an audio amplifier.

Teleology: The design intent for an amplifier is the specification of what the amplifier is supposed to do. A specific design will have a particular specification, e.g., amount of amplification, bandwidth, etc.

Function: The function of a particular amplifier is an algorithmic or heuristic definition of how the amplifier should work, i.e., what it should do to fulfill the design specifications.

Behavior: The behavior is the way the amplifier actually created works, which may differ somewhat from the functional specification. For any given input to an amplifier, there will be an output. This relationship, for all inputs and outputs, defines the behavior of the amplifier.

Structure: The structure of an amplifier is defined in terms of its constituent parts, e.g., resistors, integrated circuits, power supply, etc., which may be represented abstractly on a schematic.

to other domains as well. For example, consider designing an exit ramp for an interstate highway. The *teleology* (design intent) would consist of a set of specifications. *Function* could be represented in sets of equations that describe traffic flow. *Behavior* would be secured by taking actual data from the exit ramp (e.g., numbers and rate of cars using the ramp), and *structure* would be the physical location of guard rails, the width of the ramp, and so on. Note that behavior and function could be identically the same; however, in most real-world cases, behavior is normally somewhat different from what is predicted by the ideal function specification. Also, the design intent may define a function that does not cover all behaviors of an object.

Object Identification: Physical and Abstract Representations

Physical decomposition of objects into constituent parts is usually relatively straightforward because the subparts of an object are identifiable objects as well. The whole-part decomposition of an automobile given in the previous chapter is an example of a physical decomposition. Decomposition of abstract entities is more complex, however. The signal analysis system is a good example: A top-level object exists which coordinates all the remaining objects in the system, for example, objects that manage or implement algorithms, display objects, user interaction objects, data acquisition objects, and so forth. The conceptual mapping of real objects into object representations comes directly from an analysis of the teleology of the object. In this example, the teleology of a signal analysis system is to create a tool that provides the user the capability of understanding what is in a sampled signal. Functional specification provides a

listing of capabilities of the designed system. These capabilities often match the objects that are in the system. For example, one capability that a signal analysis system should have is to perform an FFT on a signal; this task will require an FFT analysis object. Similarly, if filtering is another task, a filter object will be needed. Hence, the analysis of which objects are needed proceeds directly from the evaluation of the function. Implementation of the objects includes a behavioral evaluation.

In the discussions that follow, the reader should remember (as discussed in the previous chapter) that classes define the characteristics of a general category (i.e., class) of object and that a specific object can be created by instantiating a class. For example, the specific objects *anAmplifier* or *amplifier-1-1992* could be created by instantiating the class **Amplifier**. The object *anAmplifier* is an instance of the class **Amplifier**.

As discussed in Chapter 2, inheritance plays a very significant role in object-oriented systems. Indeed, inheritance becomes a significant issue in analyzing a problem. In all object-oriented languages (see Chapter 5 for a discussion of different object-oriented languages), inheritance capabilities are provided. However, inheritance without preexisting classes that can be specialized provides only a small improvement above standard procedural programming practices. In other words, object-oriented languages that contain many preexisting classes provide a designer the most flexibility for rapidly building new applications because existing classes can be reused and specialized. Without these classes, problem analysis simply becomes a description of classes. The analysis problem is not very difficult in this case. In contrast, in object-oriented environments such as Smalltalk-80, which have many classes already in existence as part of the environment, the analytical task is much greater. The programmer must first evaluate the function of the object being designed and then determine if it is advantageous to create a subclass of any class that is already in the environment. Thus, to be effective, a programmer must understand, or at least have some acquaintance with, several hundred classes. This situation makes it extremely attractive for new object-oriented programmers to program without regard to existing classes because of the steep learning curve associated with understanding the class libraries. However, productivity suffers. Once the programmer learns many classes and creating subclasses of other existing classes begins to be a normal class definition modality, productivity markedly increases. Chapter 5 will address these considerations in detail.

Steps in Object-Oriented Analysis

Table 3.3 displays the steps for object-oriented analysis. The first step is to consider the design intent for the system being analyzed. This step basically consists of trying to understand what the system being created is supposed to do

Table 3.3 Steps in Object-Oriented Analysis

-
1. Define the design intent of the application.
 2. Determine the name of the organizing class. Write a definition, including the general categories of classes that the organizing class coordinates.
 3. Define and name constituent classes managed by the organizing class. Write a definition of the functionality of each class.
 4. For each class, determine the local state in terms of class and instance variables.
 5. For each class, define the functionality such that object functions can be coordinated by an instantiation of the organizing class.
 6. Create the specific objects in the application by instantiating the classes designed.
-

when it is completed. At this point, a set of specifications can be created that can assist in guiding the implementation of the system. Step 2 requires identifying the organizing class and providing a definition of the functionality of this class as well as identifying the types of classes in the application coordinated by the organizing class. Step 3 repeats the process for each specific class identified. Step 4 defines the state by identifying the class and instance variables needed to maintain information inside each class. In step 5, a description of the function of each class is created including producing descriptive names that identify what an object created from a class will do when sent various messages. Finally, the specific objects in the application can be created by instantiating the classes defined.

State Definition

An object's state can be characterized as the collection of the values of various internal variables at any particular time. When an object's behavior is influenced by its history, as manifested in these variables, then an object is said to have a state. To capture state information in an object, class and instance variables for each class used to create an object should be defined. Class variables are those variables that are shared by all instances of a class. Separate copies of instance variables exist in each instantiation of a class. Hence, when analyzing each object, the programmer must discern (1) if there are any variables that are common to all instances of the class and (2) what are the variables that define the state for the instances.

Behavioral Specification of Objects

For each object defined, the final task is specification of the behavior of each object. Consistent with the definitions given previously, *the behavior of*

an object is what the object does when a message is sent to it. The input to an object is a message; in response to a message, an object can change its internal state and/or it can send messages to other objects. Internal object behavior is best organized into protocols, or collections of methods that have similar or allied functions. Thus, it is useful to initially determine what the protocols are that an object should have. Examples of some types of protocol names are initializing, accessing, printing, drawing, activating, and so on. Names should be picked that have a clear meaning to the user so that there is no ambiguity about where methods will be located. Good selections for names of protocols will ultimately enhance understandability. Protocol names are not used in the actual code in the Smalltalk-80 examples to be given in upcoming chapters, but are simply employed as an organizing and self-documenting mechanism.

The overall behavior of a system created is defined by the collection of behaviors of the individual objects in the system. To make definition of system behavior more understandable, a top-level system object is normally defined that organizes the remaining objects in the system.

The information given previously is quite general; hence, a systematic methodology for analyzing applications and generating descriptions that can be used to produce actual object code for Smalltalk-80 is presented next.

3.3 Organizing Applications Using the Application / Class Organization Method (ACOM)

This section of the chapter presents a “pencil-and-paper” organizing technique for analyzing applications. The method is called the *application / class organization method* (ACOM). The purpose of the methodology is to provide a framework in which an initial set of classes, their variables, and methods can be specified for a new application. The utility of the method is the assistance it gives users in organizing their thoughts about how to structure an application. The “pencil-and-paper” technique is useful for initially learning how to analyze applications and for groups that are building large applications. Computerized organizing methods are actually available that accomplish the same aims as the “pencil-and-paper” methods. These organizational aids will be discussed in Part Two of this text.

ACOM is loosely based on an idea proposed by Beck and Cunningham (1989) and Cunningham and Beck (1986) that is known as the CRC (class-responsibility-collaborator) card method. Figure 3.1 shows an example of this method. The idea is to create 4-inch by 6-inch index cards with three main pieces of information on the card: (1) the name of the class, (2) the responsibilities of the class, and (3) the names of objects with which the class object communicates. As shown in the figure, there might be many CRC cards generated to describe an application; the list of collaborators would be the

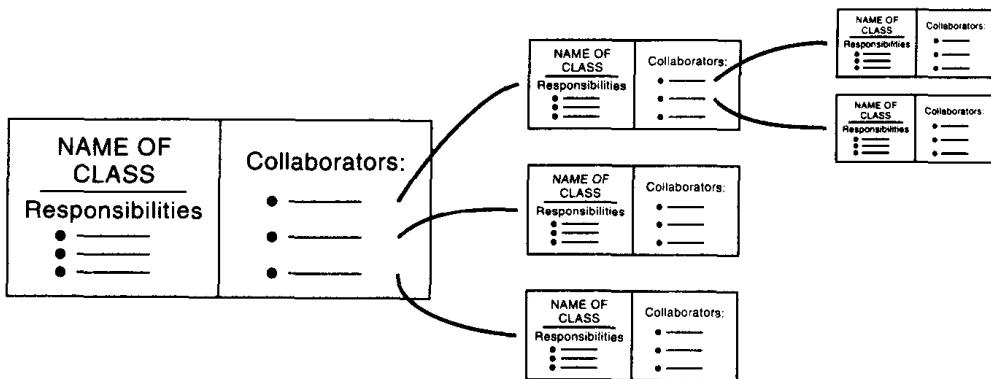


Figure 3.1 Class-Responsibility-Collaborator Card Concept. [After Beck and Cunningham (1989).]

names of objects that would be described by additional cards and so forth. The utility of the scheme is that one can analyze a problem by writing on cards the names of classes and how they relate to other classes. This kind of problem decomposition is a first step in understanding how a problem can be separated into parts. Typical use would be for an individual or team to create as many cards as possible. Team use of the technique can facilitate “brainstorming” and team members can even play the role of the objects created to try and discern their own responsibilities and collaborators.

ACOM differs from the CRC method in that there are four different classes of cards, each of which contains considerable additional information. ACOM cards were designed to permit a direct translation from analysis of an application to coding of the application, without losing any of the generality of the CRC method. Each ACOM card shown in this text can be reproduced and used for classroom or individual analysis sessions.

Figure 3.2 shows the first ACOM card—the *application organizer*. At the top of the card, a space for indicating the application name is provided; as indicated earlier in the chapter, this name will likely become a class name that describes the organization of the entire application. The box below the application name is provided to permit writing a description of the application. In the lines below the box, the names of objects in the application can be listed. At the end of each line is an oval in which one can jot down a cross reference to another card; sometimes it is useful to employ different colored pens to color code related class cards. Class cards and extension cards, to be discussed next, contain small boxes at the top left and right which can be used for color-coded coordination with the application organizer card. One benefit of the use of cards is that additional information can be put on the back of the card, as needed. In experiments conducted by the author in teaching students the use of the ACOM method, of 4-inch by 6-inch cards were found to be preferable to 3-inch by 5-inch cards.

Application Name: _____	Application Description
Objects in Application:	
_____	_____
_____	_____
_____	_____
_____	_____
(more)	
cross ref	

Figure 3.2 The Application Organizer Card.

Figure 3.3a shows the class organizer card and Figure 3.3b shows an extension card that can be used, as needed, to capture additional information. The format of the card is very deliberate; it was specifically designed to mirror the syntax of creating classes in Smalltalk-80, with the intention that if cards are created by hand using this technique it is a very short step to producing actual code. Currently, it is difficult to work as a team using Smalltalk-80 because Smalltalk-80 provides an environment only for an individual. Hence, ACOM cards should provide a useful start-up accessory for the initial analysis of a problem by a team. Teams can come together and agree on most of the classes that should be in an application, try different organizations, discuss alternatives, and define behaviors prior to any coding actually taking place! Without an initial agreement about the basic organization, it would be difficult to share a team's work among individuals using their own separate Smalltalk-80 environments.

At the top of Figure 3.3a, the names superclass, class, and subclass are listed. The organization for naming a superclass and a class is an essential step in defining the code for an object, as will be shown in Part Two of this text. Space for indicating subclasses is provided to allow the user to indicate if the class can be used as a superclass for other classes. The bracket under "class" should be filled with the name of the class being created and its superclass should be indicated to the left. If there is no specific known superclass, simply write down **Object**, or leave it blank. There is a box at the top right which can be used to color-code link the class with the object specified in the application organizer card. Next, a description box is found on the card, followed by a space in which class variables and instance variables can be listed to define the state of the object. On the bottom half of the card, protocols and their methods are to

superclass	class	subclass	[]
[]	[]	[]	
description			
			
classVariables: [] instanceVariables: []			
Protocols		Methods	
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
Class Organizer Card			

(a)

[]	[]	[]
extended description		
		
Protocols Methods		
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
Class Organizer Card Extension		

(b)

Figure 3.3 (a) The Class Organizer Card and (b) the Class Organizer Extension Card.

methodName: _____	className: _____	<input type="checkbox"/>
instance <input type="checkbox"/>	class <input type="checkbox"/>	
description 		
returns: _____ replaces SuperClass Method? yes <input type="checkbox"/> no <input type="checkbox"/> code: 		

Figure 3.4 The Method Description Card.

organizer card. Next, a description box is found on the card, followed by a space in which class variables and instance variables can be listed to define the state of the object. On the bottom half of the card, protocols and their methods are to be listed. So far, no code has been generated, only the names of things and linkages to things. The card shown in Figure 3.3b is a simple extension card to the main class card in Figure 3.3a.

Figure 3.4 displays the final ACOM card—a method description card. Methods are listed in the class card under protocols by name. In the methods card, each name listed is indicated at the top of the card, along with its class. A box is provided to check whether the method is a class or instance method. Class methods are associated only with the class, whereas instance methods are used only in objects that are instantiations of the class. Following the mandatory description box is a place to indicate what the method should return and whether the method supersedes a method of the same name in a superclass.¹ The bottom half of the card provides space for actually writing code for the method. Students using the ACOM method for learning to organize classes can, at this point, ignore the writing of any code. Hence, using the ACOM method, individuals or teams can proceed from a conceptual analysis of an application all the way to writing code. Figure 3.5 shows in diagrammatic form how the ACOM cards are linked to each other. The final section of this chapter will

¹Typically, methods defined in a subclass replace methods of the same name that are defined in a superclass.

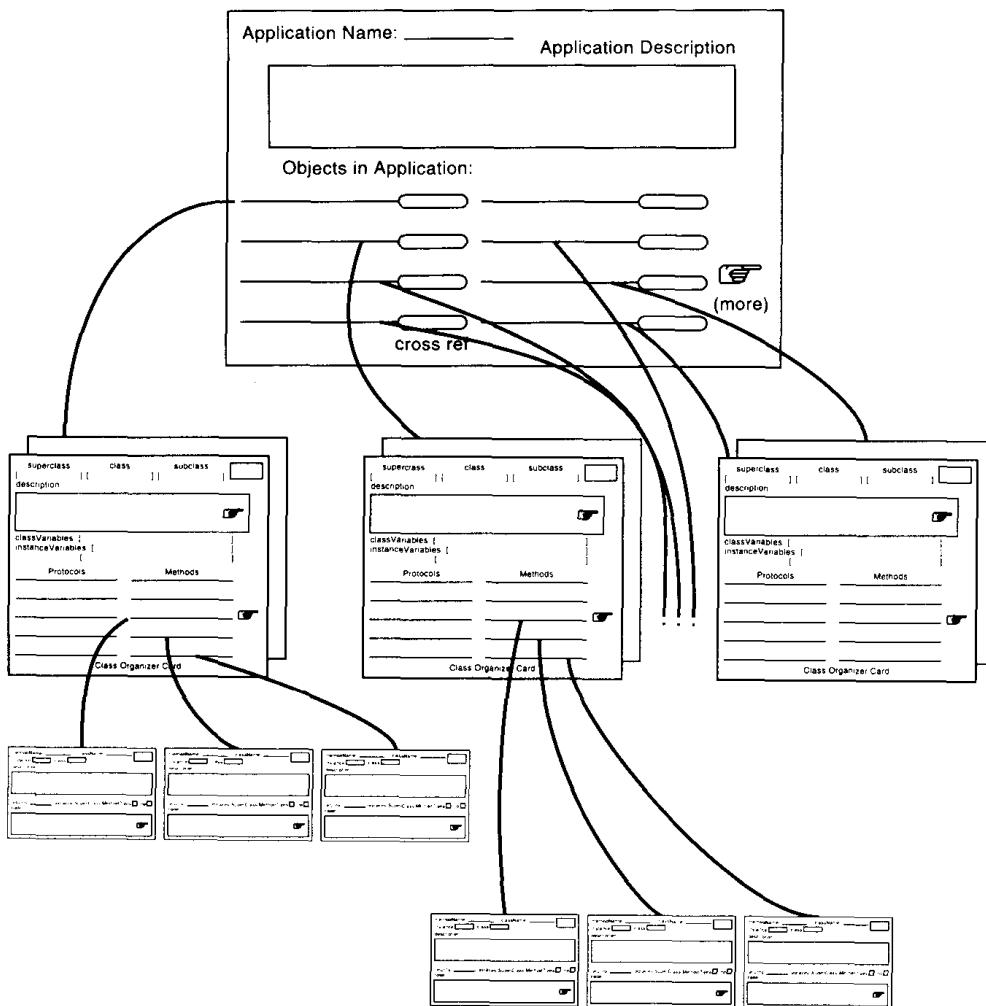


Figure 3.5 Illustration of ACOM Card Organization.

present a complete example of the use of the ACOM methodology for a simple application.

3.4 Analysis Maxims

Listed next are a few analysis maxims along with their explanations. These maxims are divided into four categories: understanding the problem framework, object identification, object behavior specification, and class association, coordination, and examples.

Understanding the Problem Framework

- Understand the general domain. If you are not expert in the domain of the application, work with someone who is. If you are creating a team to work on the application, be sure that the team includes knowledgeable people who represent different aspects of the problem.
- Secure as much background information as possible and organize the information.
- Analyze the nature of any constraints in the system being built; for example, does the application require representation of time, are there any direct or indirect causalities?
- Ascertain if the problem requires coordination between resources.
- Determine if the application requires any use of shared resources, for example, rule systems, databases, or other tools that might be shared between objects.
- Attempt to secure a sense of how large the problem is; for example, how many main objects might there be?
- If in a team, argue about the top-level concepts; if an individual, argue with yourself.
- Finally, complete the ACOM application organizer card.

Object Identification

- First, look for autonomous entities in the application. These should be both concrete and abstract entities. Time, for example, might be an abstract object.
- Make an ACOM class card for each object.
- Look for classes that can be created by specializing known classes in the class hierarchy. If designing as a team, ask team members to make suggestions about superclasses of the class that you have specified.

- Describe the prospective state of each object you wish to ultimately create. You will want to list all the names of all the possible variables that you might use on the class card.
- Look for class variables to see if there are any that are required to persist across all instances of a class.

Object Behavior Specification

- What are all the object behaviors? Consider object role playing. Think of yourself as an object and ask yourself questions about what “I” should be able to do?
- See if you can identify general protocols that can be used to organize the methods.
- Name the messages that cause an object to perform in each protocol identified.
- Finally, for each method associated with each message, write a description of the explicit things that each method should do.

Class Association, Coordination, and Examples

- Once a first pass of object identification and associated methods has been completed, organize all the cards produced and put the cards on a large table or on the floor. See if there are any obvious problems that require that cards be added or removed, such as multiple cards that define classes that permit description of the same object.
- Review the cards you have created and critique them. The review should include determining if you have identified cards that can be used to describe each object in the application. Then, for each object, communication issues should be examined. Look for and name all objects with which each object communicates. Is it clear in thinking through the object definitions that you have made that all have a purpose in the system? Are all defined objects needed?
- Draw a class hierarchy. To accomplish this task, consider the classes that you have created such as **FFTObject**. Determine if any

of these classes can share any functionality; if so, rewrite the card class-subclass information.

- Draw a whole-part hierarchy, if used. For this activity consider if the physical system that you are working with can be broken down into parts and described by a number of communicating, interacting objects.
- Determine if any coordination among objects is required. When coordination is required, create another ACOM class card that will explicitly handle coordination. For example, one could create an ACOM card called **ResourceCoordination**.
- Write examples for testing the behavior of objects created from the classes defined. Examples, at this point, can be English descriptions of how to test an object created.

3.5 Example Analysis of a Digital Circuit Simulator

This example presents a simple digital circuit simulator constructed to be able to observe the behavior of a collection of interconnected digital gates. To give a concrete example, suppose that we wish to simulate the circuit redisplayed in Figure 3.6 (previously shown in Figure 2.7) and would ultimately like to create a simulation system similar to the top-level view of the system shown in Figure 1.3. First, we need to specify the design intent (teleology). Briefly stated, the design intent could be

The purpose of this example is to create a digital circuit simulator that will permit users to observe how signals propagate in a simulated digital circuit.

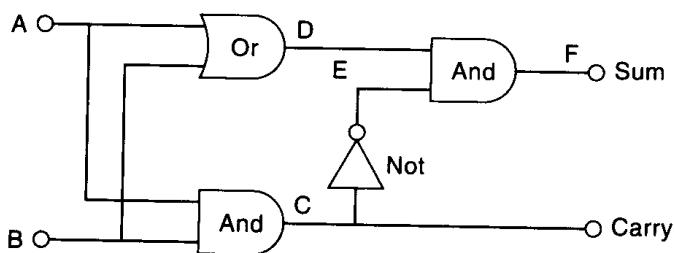


Figure 3.6 A Simple Digital Circuit: A Half-Adder.

Next, the simulator should be named and the types of constituent objects identified. For example, this information might be

*The name of the simulation is **DigitalCircuitSimulation** and the primary objects to be included in the simulation should be named the same as those found in physical digital circuits.*

Now, a specification of the function of the system and constituent objects is required. For example,

The functionality of the example Digital Circuit Simulator is (1) to permit creation of simulated digital circuits that behave in the same way as real digital circuits, (2) to permit users to interconnect circuits, and (3) to observe the behavior of interconnected components.

An initial decomposition of the figure shown would reveal that we need only four primary objects to describe the entire digital circuits world shown: **Wire**, **AndGate**, **OrGate**, and **NotGate**. In fact, we can immediately

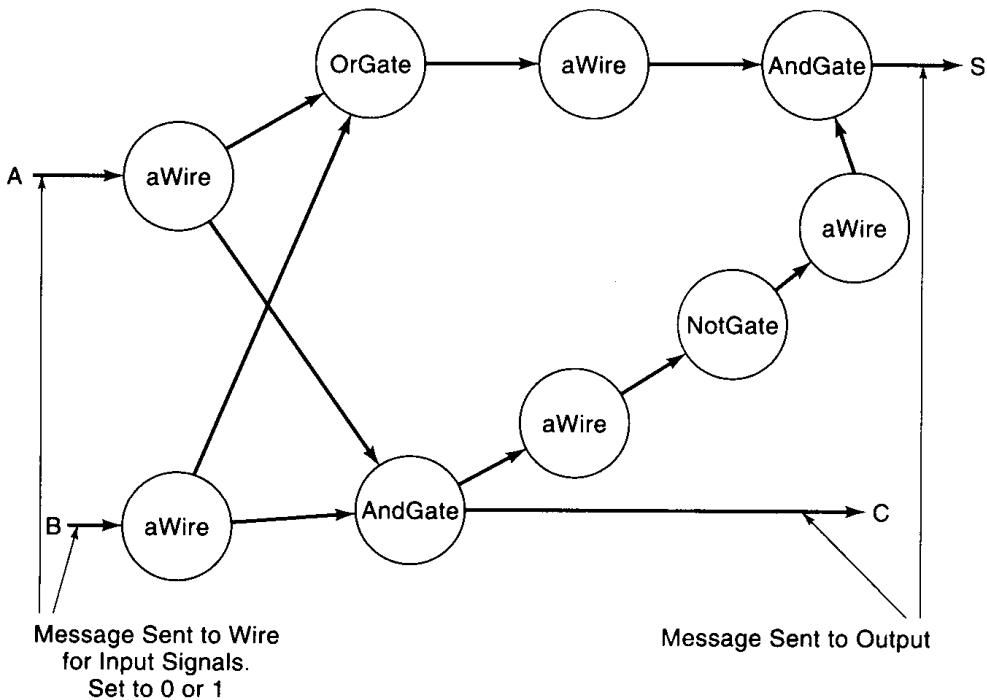


Figure 3.7 Example Digital Circuit in Figure 3.6 Represented as Objects and Messages. (C = carry, S = sum.)

translate the physical representation into a representation using objects as shown in Figure 3.7. In this representation, a wire is a full-fledged object, containing the local state of a value and its input and output connections. The task of a wire is to take an input and transfer that input directly to its output. Similarly, for each of the other objects, the task is to transfer the input to the output, modified by the constraint specified between the input and output. The term *constraint* refers to the constraining relationship that exists between the input and output terminals of a device. For an **AndGate**, the constraint is simply that the output is 1 whenever both inputs are 1, and 0 otherwise.

The figure shown in Figure 3.6 is called a “half-adder” and has the functionality shown in Table 3.4. This circuit does nothing more than add up the signals (0 or 1) attached to inputs A and B and produce a sum and carry on the two output leads. The reader is encouraged to trace signals through this circuit to verify its function and to consider the behavior of each object while conducting the trace.

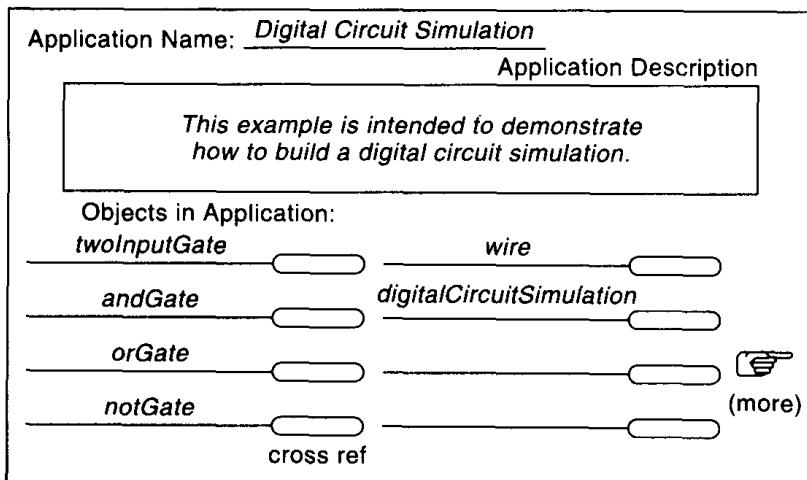
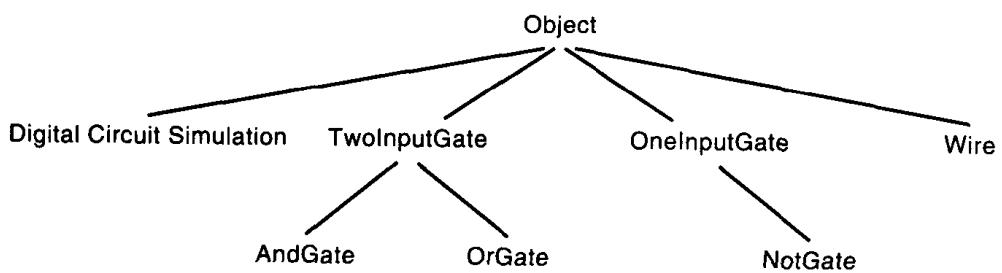
Figure 3.8 displays the ACOM application card created for this simulation system. As shown, there is a **DigitalCircuitSimulation** class, an **AndGate**, **OrGate**, **NotGate**, and a **Wire**. The first class in this list is simply an organizing class that contains a dictionary that contains all the objects in the simulation and a dictionary of all the wires. Each of these classes has its own separate class definition card.

To make the representation somewhat simpler, one can recognize that *and* gates and *or* gates are basically similar except that each contains a different constraint, or relationship that needs to be maintained between the input and output. The similarity between the two gate types is that both specify input and output connections. Hence, one can specify another, more abstract class—a **TwoInputGate**—to serve as a superclass for different types of gates with two inputs. Once the characteristics of this gate are defined, this gate can be subclassed to create both an **AndGate** and **OrGate** with only the addition of the specification of the constraining relationships between the inputs and output. Figure 3.9 shows the class hierarchy for this simple relationship as well as the remaining objects in the system. Constructing a diagram of this type is often quite useful when the number of objects becomes large.

Figure 3.10 displays the ACOM cards for the five basic classes created and Figure 3.11 shows an example method card for the **OrGate** constraint. Of course, the remaining cards need to be created to complete the ACOM description of the digital circuit simulator. The complete code for the digital circuit simulator is contained in Chapter 10, as well as a description of how to implement the constraint relationships. Additional implementation information using this example will be given in Chapter 10 and a user interface for the system will be constructed in Chapter 13.

Table 3.4 Functionality of Example Digital Circuit

Signal A	Signal B	Sum	Carry
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

**Figure 3.8** Application Organization Card for the Digital Circuit Simulation.**Figure 3.9** Object Hierarchy for Digital Circuit Simulation.

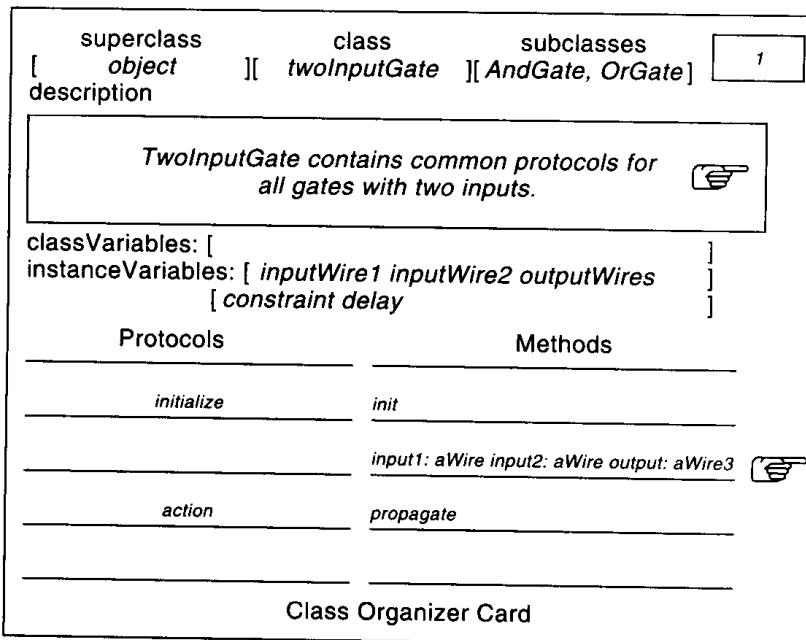
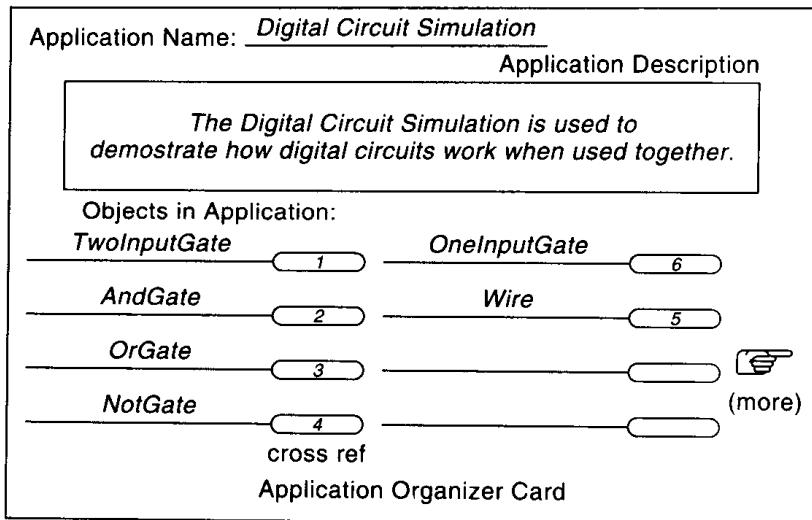


Figure 3.10 ACOM Card Examples for the Digital Circuit Simulation System.

superclass [<i>TwoInputGate</i>][class AndGate	subclasses]	<input type="text" value="2"/>
description			
<p>An AndGate produces the logical and of the input signals.</p> 			
<p>classVariables: [] instanceVariables: [] []</p>			
Protocols		Methods	
<i>initialize</i>		<i>init</i>	
<hr/>		<hr/>	
<hr/>		<hr/>	
<hr/>		<hr/>	
Class Organizer Card			

superclass [<i>TwoInputGate</i>][class OrGate	subclasses]	<input type="text" value="3"/>
description			
<p>The OrGate produces the logical or of two input signals.</p> 			
<p>classVariables: [] instanceVariables: [] []</p>			
Protocols		Methods	
<i>initialize</i>		<i>init</i>	
<hr/>		<hr/>	
<hr/>		<hr/>	
<hr/>		<hr/>	
Class Organizer Card			

Figure 3.10 *Continued*

superclass [<i>OneInputGate</i>]	class NotGate	subclasses]	4
description			
<p><i>The NotGate inverts the input signal. A '1' becomes a '0' and vice versa.</i></p>			
<p>classVariables: [] instanceVariables: [] []</p>			
Protocols		Methods	
<i>initialize</i>		<i>init</i>	
Class Organizer Card			

superclass [<i>Object</i>]	class Wire	subclasses]	5
description			
<p><i>Wires are used to connect gates together.</i></p>			
<p>classVariables: [] instanceVariables: [<i>inputconnection outputConnection value</i>] []</p>			
Protocols		Methods	
<i>access</i>		<i>inputconnection: aGate,</i>	
		<i>outputconnection: aGate,</i>	
		<i>set: aValue, value</i>	
<i>initialize</i>		<i>init</i>	
Class Organizer Card			

Figure 3.10 *Continued*

superclass [<i>Object</i>]	class [<i>OneInputGate</i>]	subclasses [<i>NotGate</i>]	6
description			
<p><i>OneInputGate</i> contains common protocols for all gates with one input.</p>			
<p>classVariables: [] instanceVariables: [<i>inputWire outputWire value</i>]</p>			
Protocols		Methods	
<i>initialize</i>	<i>init</i>		
	<i>input: aWire output: aWire</i>		
<i>action</i>	<i>propagate</i>		
Class Organizer Card			

Figure 3.10 *Continued*

methodName: <u>init</u>	className: <u>OrGate</u>	
instance <input checked="" type="checkbox"/> X	class <input type="checkbox"/>	
description		
<p>This method initializes the superclass and sets up the constraint block.</p>		
returns: <u>0 or 1</u> replaces SuperClass Method? yes <input type="checkbox"/> no <input checked="" type="checkbox"/> code: <pre>init super init. constraint := ':input1 :input2 (input1=1) (input2=1)ifTrue:[1] ifFalse:[0]]'</pre>		
Method Description Card		

Figure 3.11 Example Method Card for the OrGate.

References

- Beck, K., and W. Cunningham (1989). A Laboratory for Teaching Object-Oriented Thinking. *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, 1–6.
- Coad, Peter, and Edward Yourdon (1990). *Object-Oriented Analysis*. Prentice-Hall, Englewood Cliffs, NJ.
- Cunningham, W., and K. Beck (1986). A Diagram for Object-Oriented Programs. *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, 361–367.
- Schlaer, Sally, and Steve Mellor (1988). *Object-Oriented Systems Analysis*. Prentice-Hall, Englewood Cliffs, NJ.
- Webster's Unabridged Dictionary*, (1990). Dilitium Press, New York.

Exercises

3.1 Consider the concepts defined in Table 3.1. Select an example of your choosing and explain the four concepts in the context of your example.

3.2 The ACOM card problems:

- a. Make copies of the ACOM cards given or produce cards yourself in the same or similar format.
- b. Pick a physical object that you understand for analysis and follow the steps given in the chapter to produce a complete ACOM description of the object. Do not worry about the method code at this time; simply indicate what the method should do. The following is a list of possible objects to consider:

a hot water heater	gas heater	electric heater	wood stove
disk brakes	the heart	circulatory system	lung
myoelectric arm	nuclear reactor	vending machine	robotic device
communications network	touchtone phone	floppy disk	city simulation
chemical reactor	chemical reaction	flush toilet	hot tub
oscilloscope	voltmeter	car wash	parking meter
flying birds	seismograph	traffic light	traffic light system
antilock braking	cruise control	parking meter	ear
pencil sharpener	induction motor	television	radio

- c. Repeat the analysis procedure that you just conducted with a group of people. Use the ACOM method in this group and argue about the characteris-

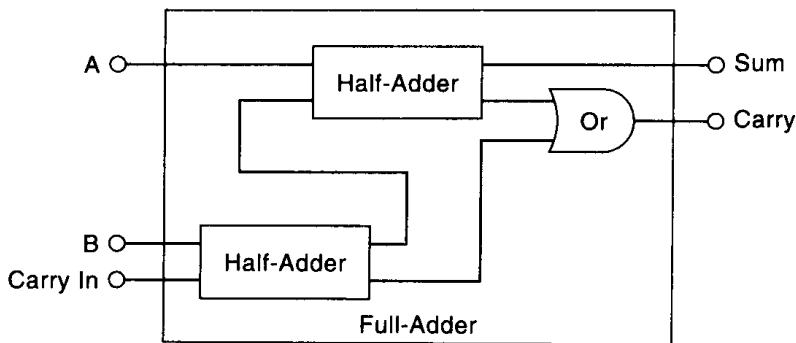


Figure 3.12 A Full-Adder.

tics of each card created. When you are done, compare what the group has done with the results you came up with in the previous problem. Did working in a group for this exercise help or hinder your efforts?

- 3.3 After completing the preceding exercise, throw the cards away (or just temporarily hide them) and start over. After completing a new set, compare your result with your first attempt. Were there any changes?
- 3.4 The **OrGate** initialization specification was provided in Figure 3.11. Write a similar method description for an **AndGate** and for a **NotGate**.
- 3.5 Extend the digital circuit example given in the chapter by creating ACOM cards for a full-adder circuit. A full-adder is shown in Figure 3.12.

Introduction to Object-Oriented Design for Smalltalk-80

4.1 Design and Analysis

Analysis is part of design. In Chapter 3, methodologies for analyzing problems in terms of object definitions were examined. Using the object-oriented methodology presented, analysis of a problem leads to the creation of object definitions that can be ultimately turned into code. Design can be cast in various flavors. For example, the terms *conceptual design*, *architectural design*, and *implementation design* all have somewhat different meanings. Conceptual design refers to a loose organization of ideas and concepts that ultimately leads to the creation of an object. Architectural design is more specific, referring to a framework for the design, and implementation design refers to the actual specifications required to fabricate something. For most designs, the overall classification of *design* includes analysis as well as these three different levels of design specificity. For physical systems, design includes the analysis phase and ultimately includes all specifications needed to actually create the artifact specified. For example, design of an automobile would be complete when all specifications were available to build the car. In contrast, for object-oriented design of software in Smalltalk-80, the distinction between definition and fabrication is somewhat blurred. When the design includes behavioral specifications of the objects, it actually becomes runnable, thus implementing the design itself. In this text, the position is taken that for design of software systems, this situation is a desirable one; that is, the designed system should be an actual implementation. For traditional procedurally coded software systems (i.e., software composed of a sequence of steps), this position would be unreasonable. For object-oriented systems implemented in Smalltalk-80, it is logical. The reason is that procedural coding requires many steps that are not very close to

the actual design. In contrast, objects in object-oriented software systems mimic the actual objects specified in the conceptual design.

In Chapter 3, analysis revealed the characteristics of objects needed to create a *model* of an object being designed. The model, however, was insufficient to describe a complete system. A method for providing the user a way to control the system (*controller*) and means of viewing the output of the system (*view*) are required, in addition to the basic representation of knowledge and data associated with the problem (*model*). Applications can be systematically factored into these three components to provide an organizational framework.

A framework is useful for organizing a design. Several different approaches for organizing components in a framework have been discussed in the literature¹; many are similar to the model-view-controller (MVC) methodology used in Smalltalk-80. The approach taken in this text follows the methodology utilized by Smalltalk-80 to organize these three aspects of design activity. Aspects of the MVC paradigm have been described elsewhere (Krasner and Pope, 1989; *Objectworks \ Smalltalk User's Guide*, 1990). This chapter attempts to cast the use of the MVC framework into a design activity; that is, how one designs what the user sees (the view), how control is implemented between the user and a system (the controller), and how these aspects of a system are linked to the model provided by analysis of the problem. The descriptions given here are incomplete and lacking in detail; the intention of this chapter is to provide concepts without the details, which will be supplied in subsequent chapters.

4.2 Major Design Aspects: The Model, the View, and the Controller

Figure 4.1 displays the basic model-view-controller (MVC) triad of Smalltalk-80. Considerable information about this framework will be given in this text, because the MVC provides unique cohesive facilities for creation of complete systems. This chapter will discuss concepts related to the MVC and subsequent chapters, especially Chapter 8, will provide additional details and discussion about actual utilization of the MVC. Table 4.1 presents a capsule summary of the specific features of the MVC triad that are important for understanding the functionality of the three main components of the MVC.

Model

The model is used to represent the data or knowledge about the application constructed. A very simple model might be a list of several items, a

¹See, for example, Barth (1986), MacApp (1987), and Serpent (1989).

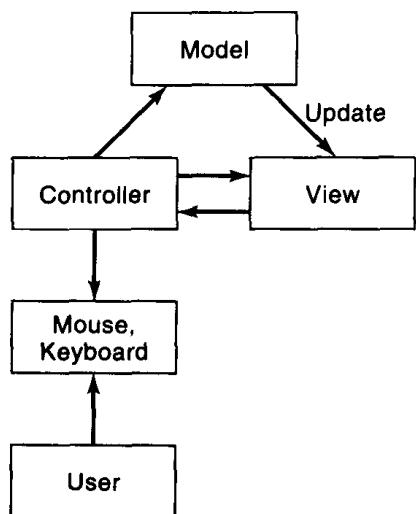


Figure 4.1 The Model-View-Controller.

Table 4.1 Features of the Model-View-Controller

Model	Represents the data or knowledge in an application
View	What the user sees on the screen
Controller	Provides the interface between the user and the system

dictionary, or a collection of some kind. More complex models might be network representations of information, rules, or pictorial information to display.

Controller

The controller serves as the connection between the user and the computer system. In most cases, the controller need only handle input from the keyboard and the mouse, although it is possible to consider control from other input modalities such as speech or signals.

View

The view is the display on the screen of the computer's monitor. In direct-manipulation interfaces (i.e., those interfaces that provide the user with the capability of directly manipulating objects on the screen), the view often

consists of collections of icons and windows that display information and are sensitive to a mouse-driven cursor. Subsequent chapters will describe the characteristics of such interfaces in depth.

A major idea in the MVC is the isolation of the model from the view and controller. Because the model normally represents a set of data or knowledge, it has no need to know how the controller or the view operates. Hence, the view can communicate information to the controller, and vice versa. However, the model is connected to the view in the following special way. When any change in the model is made, for example, updating a data value, the model should send itself a message that indicates that it has changed (*self-changed*). The model contains the ability to recognize that for every designated change it should send the view a message that indicates that the view should update (i.e., redisplay) itself. Hence, if the view is presenting some data contained in the model, it can modify the display on the screen each time information in the model is changed. Furthermore, the model can indicate to its dependents (i.e., those objects dependent on the model) what aspects of itself are changed. To give an example, imagine that a view contains graphs of five different sets of data points. Each graph might be a collection of numbers that are contained in a dictionary in the model. If a single value is changed in one of the graphs, the screen should be updated to reflect the change in data. One solution to updating what the user sees is to simply redraw the entire screen; however, this is inefficient. Hence, the MVC methodology permits indicating what aspect of the model has been changed, and by passing this information to the view, the view can differentially update itself. In the MVC paradigm, any model has associated with itself a collection of *dependents*, those objects that are dependent on the information in the model. Because the view is dependent on the information in the model, it is a dependent of the model and should be updated when the model is changed in some way. In Chapter 8, additional information about these ideas and examples will be given.

Alternatives to the MVC

As mentioned previously, there are various alternatives for factoring a design framework into logical parts, such as the model, view, and controller. For example, Serpent (1989) breaks a system into three parts: the presentation layer, the dialogue layer, and the application layer. The application layer is equivalent to the model of the MVC, the dialogue layer contains a mix of presentation and control functionality, and the presentation layer controls the screen layout (equivalent to the view). Serpent supports X-Windows (Schiefler, Gettys, and Newman, 1989) and may be used with applications written in C or Ada. Apple's MacApp implements Macintosh user interfaces. The names utilized for the

Table 4.2 Examples of Representation Alternatives for Use in Model

Class Name	Description
Array	Sequenceable (i.e., having sequence) collection whose elements are any object
OrderedCollection	Ordered objects, e.g., a queue is an example of an ordered collection of objects
SortedCollection	Objects sorted by a criteria, e.g., alphabetic or numeric
Bag	Unordered collection with duplicate elements allowed
Set	Unordered collection with no duplicates
Dictionary	Unordered, hashed, using keys and values to associate objects
String	Text

framework are different from the MVC, but the functionality is essentially the same. Finally, InterViews is a graphics user interface tool kit (Linton, Vlisides, and Calder, 1989) that consists of a library of C++ (Stroustrup, 1986) classes for implementing user interfaces. InterViews employs three classes for building interfaces: interactive objects, such as buttons and menus, structured graphics objects, and structured text objects. These classes are combined with application code to produce an executable application. The concepts employed are not the same as the MVC methodology and represent an alternative methodology. In this text, the focus will be on the MVC methodology because it represents the oldest and probably best understood method for implementing application frameworks.

4.3 Model Design

Designing a model requires knowledge about possible types of knowledge or data structures that can be used for a model. Models can be of almost arbitrary complexity, ranging from an object that holds only one number to complex networks or rule systems. Table 4.2 shows some typical classes that can be used for model representation. These classes are among the most commonly used classes in Smalltalk-80 employed for representing information. Possibly the most useful are the collection classes, which include such classes as **Array**, **OrderedCollection**, and **SortedCollection** as shown in the table. Class **SequenceableCollection** is an abstract superclass of these classes, which each have a well-defined ordering of their elements. The naming of these classes follows almost-commonsense conventions; that is, an array is simply a sequence of elements with integer keys, an ordered collection is a sequence of elements that maintains an order, and a sorted collection maintains the collection in a sorted

order according to some type of sorting criterion. The primary design decision for determining the kind of model (or models) to use is based on what it is that you wish to represent. Lists of items are quite easily represented as an ordered collection, which is nothing more than a list of objects that are maintained in order. Note that in the object-oriented paradigm an object can be of arbitrary complexity. Hence, an ordered collection of objects might actually contain a great deal of information in the form of complex objects stored in order. Sorted collections are useful when one needs to sort items according to some scheme—time, for example. Other types of collection classes are frequently useful such as **Bags**, **Sets**, and particularly **Dictionaries**, as defined in the table. Dictionaries are frequently used for storing keyed associations between items.

More complex information representation may be used as a model. For example, a semantic network could serve as a model. A *semantic network* is a classical artificial-intelligence-based representation paradigm used for representing declarative knowledge. Consisting of nodes and links, a semantic net-

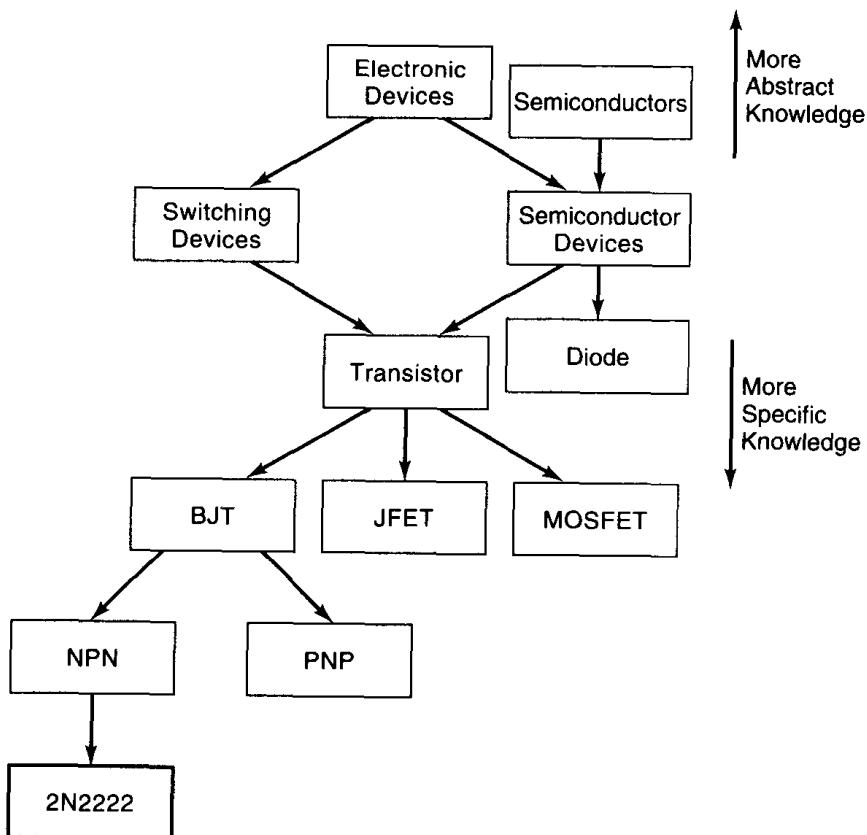


Figure 4.2 A Semantic Network as a Representation Model. [After Bourne et al. (1989).]

work is a labeled directed graph that attempts to capture meaning while describing objects. Consider Figure 4.2 as an example. This network represents inheritance information about electronic components. Information about electronic devices and semiconductors in general is stored at the top level. Arranged in a hierarchy, the objects in each node of this hierarchy inherit information from objects higher up in the hierarchy. At the lowest level, objects such as the BJT (bipolar junction transistor) inherit information from all the node above it. Circled with a bold line in this figure is 2N2222, which is a specific kind of transistor and is an instance of an NPN transistor. Networks of this type can be used to represent semantic information (i.e., meaning).

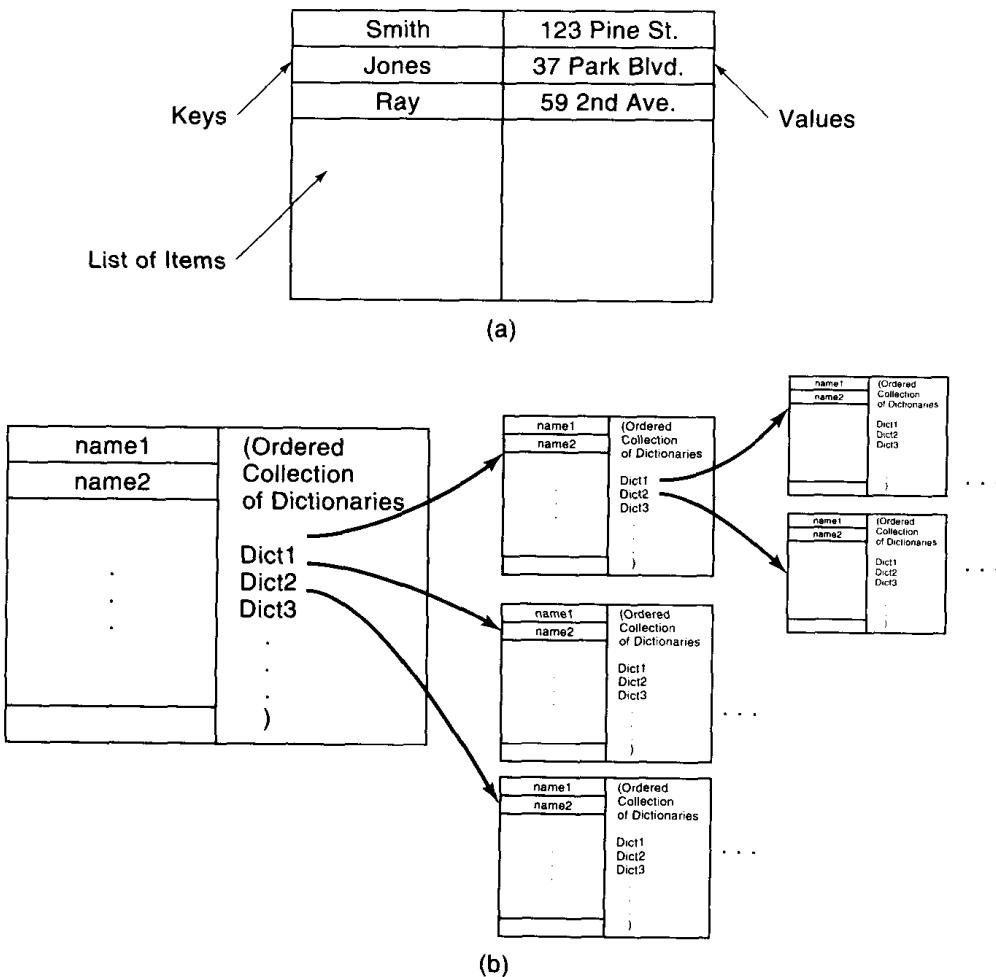


Figure 4.3 Using Dictionaries for Representation. (a) A Simple Dictionary with Keys and Text for Values and (b) Dictionary Inspectors Showing the Use of Dictionaries for Representing Tree Structures.

Figure 4.3 shows how dictionaries can be employed as a model for a complicated information storage problem. Figure 4.3a shows the simplest possible use of a dictionary, using the keys to track information that is stored in the values associated with the keys. An address book is an easy example to understand; the keys would be the names of people, for example, and each value associated with a key would contain addresses and phone numbers. Because values can contain any object, it is a short step to extend the representation power of dictionaries to multiple levels. Figure 4.3b shows that the value associated with a particular key might contain an ordered collection of objects which are each dictionaries. In turn, each of these dictionaries might contain keys which have values that are dictionaries. This progression could go on to any level and provides an easy way to represent large hierarchies of information.

4.4 View Design

Views are what the user sees on the screen within host windows. Various software components can be employed to construct views. The most

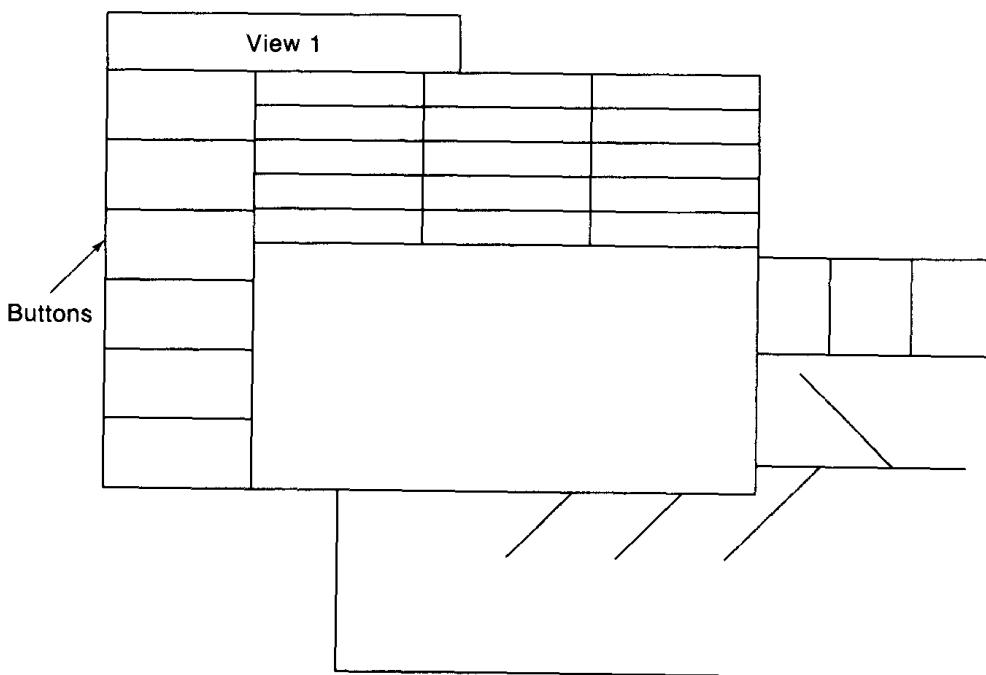


Figure 4.4 Tiled and Overlapping Views. View1 overlaps View2; the contents of each view is tiled.

common types of elements are lists, buttons, icons, text windows, graphic representations of things, gauges, and so forth. In a windowing system, a view may consist of "tiled" and/or "overlapping" views. A tiled view refers to a window that has smaller pieces of the view placed like tiles of different sizes. In Chapter 1, several examples of tiled views were shown. Overlapping views are allowed to overlap, the top view obscuring the one below it. Typically, information about hidden parts of a view are stored so that if the top view is moved, collapsed, or closed, the view below can be seen. Figure 4.4 shows the way two tiled and overlapping views might appear on a computer screen.

Figure 4.5 presents a first look at a concept which will be described in detail in subsequent chapters. The concept is that one can have many different classes that represent different kinds of views which can be put together using a "view builder" system to build any kind of view with the desired characteristics for an application. At the top of the figure is shown a "view library," basically a

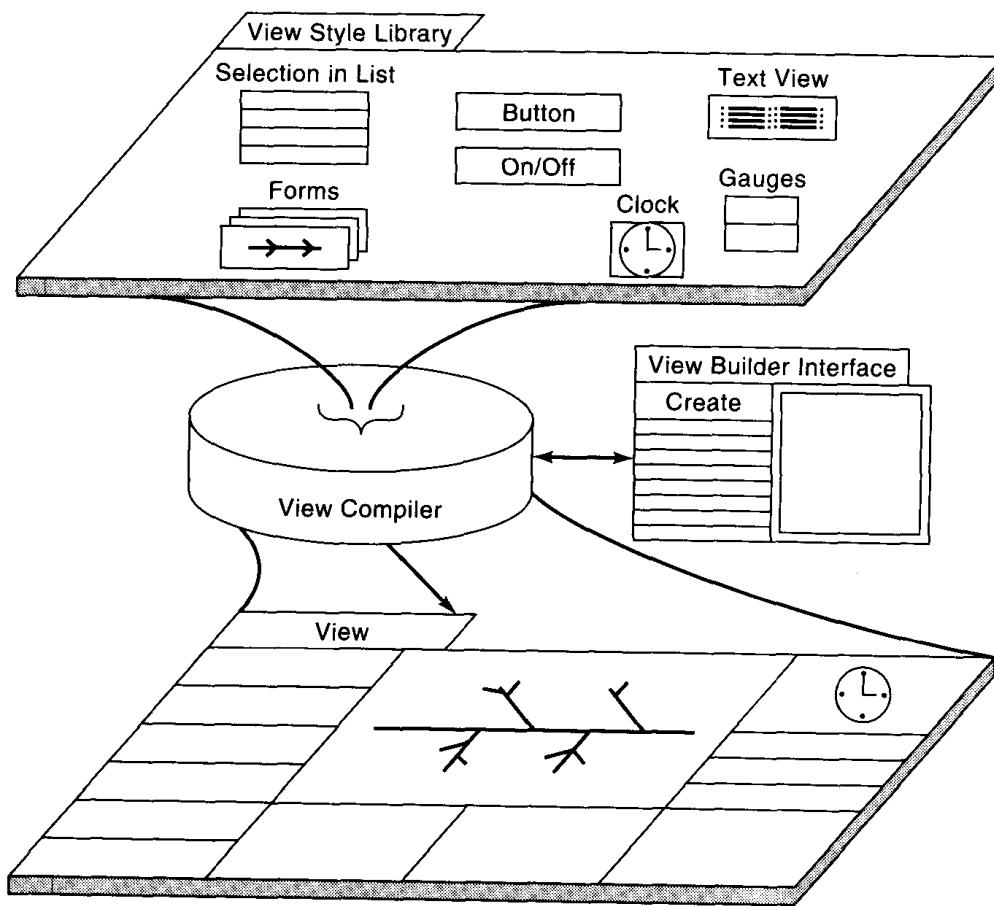


Figure 4.5 The ViewBuilder Concept.

visual representation of the classes that are in a system which can be used to build specific application views. These classes are shown funneling into a “view compiler,” with a graphics interface to the compiler shown to the right. Finally, at the bottom of the figure is shown the final completed tiled view, which includes several of the view types shown in the library at the top of the figure. The primary concept is to permit a user to build application views by simply “picking and placing” views at various parts of an empty canvas. The compiler collects information about where the user wants to put things and then, using this information, generates the code needed to actually create the view. This technique is explained in later chapters and an example is presented for building an interface to a model.

The issue of what to put in a view depends entirely on the needs of the user. For example, if items are to be selected from a list, a list view should be selected. If text is to be portrayed or images shown, views that support the display of such things will be required.

4.5 Controller Design

Controllers provide the interface between the user and the view contained in a window. By continuously interpreting the location of the cursor within a window and monitoring which of the buttons on the mouse or keys on the keyboard are pressed, information about the intentions of the user can be passed to model and view of the MVC triad. Figure 4.6 displays the controller

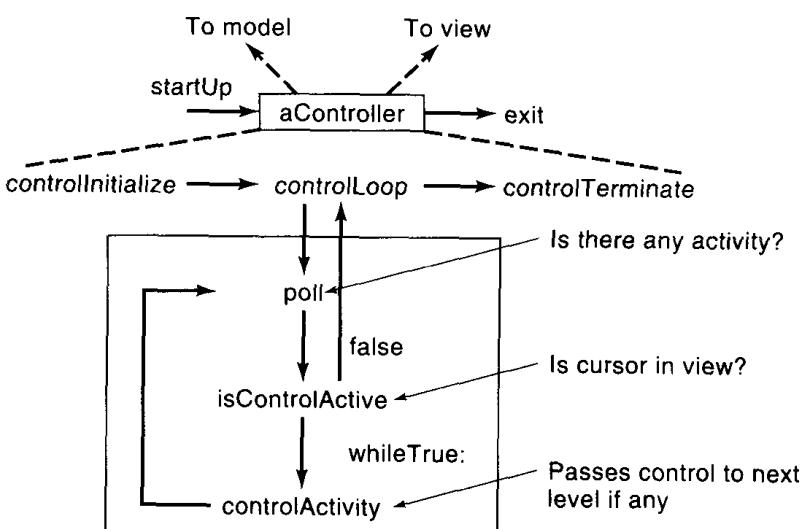


Figure 4.6 Control Methodology Outline.

methodology used in Smalltalk-80. In this figure only the controller is shown at the top of the figure and the dotted lines show links to the model and view of an application. Once a message called *startUp* is sent to a controller, the controller is initialized, goes into a control loop, and completes with a termination routine. In the *controlLoop*, polling is carried out to see if there has been any activity. If there has been no activity for a long time, a flag (semaphore) is set to indicate that polling can wait until a signal is received, indicating that there is activity. Next, when activity resumes, one default behavior is to check to see if the cursor is in a view and that a cursor button is not pressed. As long as these latter conditions are true, the controller associated with the view within which the cursor is located is active. When the cursor is moved to different views, the controller associated with the view that contains the cursor becomes active.

4.6 Maxims for Design of Systems Using the MVC Triad

Model Maxims

If possible, choose a representation for the model that is easy to understand, for example, an instance of the **Dictionary** or **OrderedCollection** classes. Many other, more complex representations can be built from the common representation building blocks (i.e., classes) that are available in Smalltalk-80. For example, a tree can be built by using multiple dictionaries.

To ease viewing objects, choose a representation that is easily inspectable. *Inspectors* permit you to view graphically the contents of an object on the screen, a clear benefit for debugging and understanding. All basic objects in Smalltalk-80 can be inspected; however, complex models that you create may require creating inspectors that permit simple viewing of the parts of the model that need to be observed. A number of the common collection classes provide special capabilities. For example, in Smalltalk-80, both the **Dictionary** and **OrderedCollection** classes have inspectors.

Remember that the model is not directly connected to the view, but communicates with the view by sending *update* messages (details will be given in Chapter 8). Hence, in specifying the model, one only has to be concerned with the structure of the knowledge and data that are to be used in the design.

Complex models may be needed for various representations; models of any complexity can be employed and different aspects of a model can be assessed by the view or different views. This idea permits using a complicated model for different purposes. For example, a model of an automobile might be used for different purposes, such as determining performance or cost. It is useful to capture such multiple perspectives in a model of an entity, while retaining the ability for the model to be viewed from different perspectives.

View Maxims

Views can be cluttered or uncluttered. For example, a view can have many buttons, lists, and pop-up or pull-down menus. Some applications may need to maximize drawing space on the screen (e.g., drawing schematics, etc.) and should use pop-up or pull-down menus.

A cluttered view with all the actions available to the user is easier to use than a view with pop-up or pull-down menus. An uncluttered view that employs pop-up or pull-down menus is more difficult for novices to use.

Pop-up and pull-down menus give more flexibility of control to the designer. This is especially true for hierarchical menus (i.e., menus that have selections which, when selected, produce other menus; see Chapter 8).

Views that scroll back and forth and up and down within a window or a part of a window are useful for presenting information in a small space.

Icons (small images that represent something) are useful for imparting visual information to the user of a system. The designer should consider if significant assistance in helping a user employ the system could be gained by using icons. See Chapter 7 for a discussion of user interface issues.

The size of the presentation for a screen view should be as large as needed to adequately represent what is to be shown. If the screen is too small, users will have a difficult time actually using the system. However, one solution is to permit windows to zoom in and out easily, a feature found in most windowing systems.

Controller Maxims

Use default controllers; they will normally work for most things. Most views specify a default controller. If you wish to include special control features, these can be added as your system is refined. However, remember that some views will not need a controller; for example, views that present something, such as text, which does not require interaction with the user.

4.7 Example View Design Scenarios for Digital Circuits

Figures 4.7a, b, and c show a few alternatives for designing views for a digital circuit simulator system. The view shown in Figure 4.7a is the same as the view shown in Figure 1.3. The primary design feature of this view is that all actions that the user can take are observable on the screen, presented by labels in buttons that can be pressed to initiate the desired action. For example, pressing “load” will pop up a message asking for a file name so that the

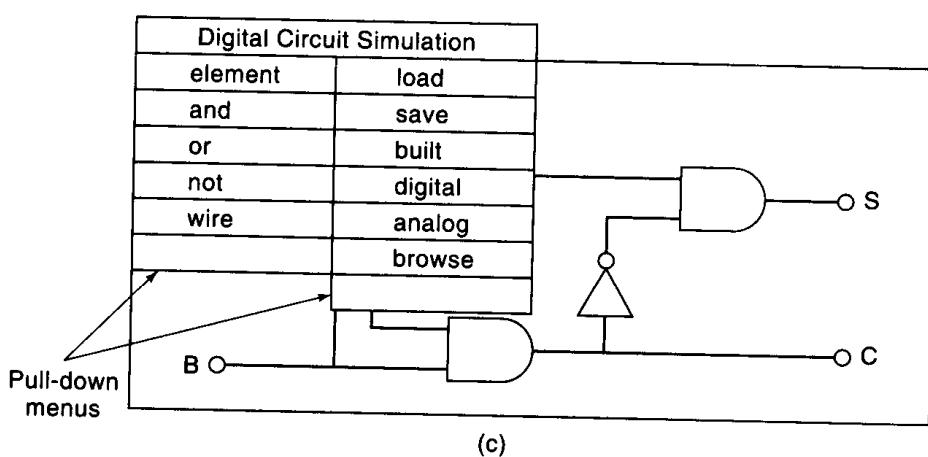
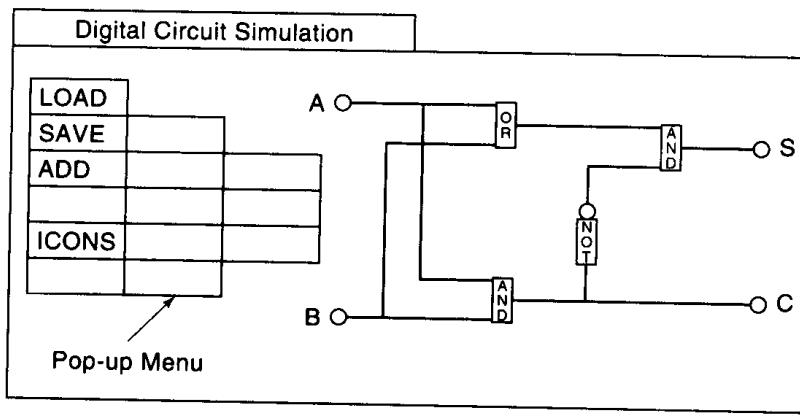
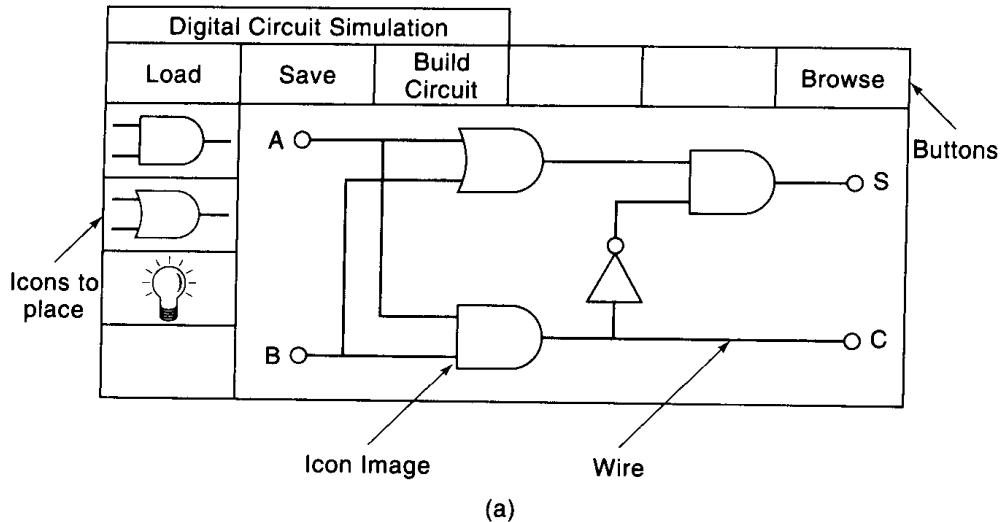


Figure 4.7 Different Interface methodologies for the Digital Circuit Simulator Example. (a) Design Using Push Buttons, (b) Design Using a Pop-up Menu, and (c) Design with Pull-Down Menus.

specified file can then be loaded. Icons that can be used to create circuits are shown on the left side of the screen, also in buttons. When the user clicks on one of these icons, an icon with the same image moves with the cursor to the drawing canvas. The use of the icon representation lends some degree of schematic reality to the construction of circuits in this view design.

Figure 4.7b shows an alternative to Figure 4.7a. The area on which the user can construct a circuit is much larger, due to the lack of control buttons. Control is provided via a pop-up menu. The pop-up menu is activated by pressing a mouse button; by moving the cursor the user can then select an item in the menu and the appropriate action attached to the menu item will be carried out. A so-called “walking menu” or hierarchical menu is shown. The hierarchical menu is the same as a normal list-style menu but each item in the list, if selected, can produce yet another list, which, in turn, can pop up additional menus. Because each list is positioned to the right of the calling menu item, the lists appear to “walk” across the screen. Chapter 8 will provide information on how to use such menus.

Figure 4.7c shows how a pull-down menu would be used for this same application. Arrayed across the top of the view are items, each of which have submenus that “pull down” when selected. The pull-down menu is essentially the same as the hierarchical menu, except it is spatially rearranged. This type of menu has been popularized by the Macintosh computer and is found in many end-user application programs such as Pagemaker (Aldus, 1990).

Table 4.3 summarizes the different design considerations for the examples shown in Figure 4.7. Much more detailed information about views and coupling views to the user will be given in Chapter 7.

Table 4.3 Design Decisions for View Interfaces Shown in Figure 4.7

Design A	
Push-button model	Easily observable actions Takes excessive screen space Relatively cluttered view
Design B	
Pop-up menu mode	Uncluttered view Gives maximum screen space User needs to recall how to pop up menu
Design C	
Pull-down menu model	Uncluttered view Covers view in fixed places

4.8 Summary

So, what is design? Design, as presented here, is the sequence of steps needed to produce a complete specification that can be used to create a system. Analysis is a precursor to and an integral part of design. Designs are facilitated by operating within a framework, such as the MVC triad, discussed previously. What are the steps that are taken to create a design? First, conceptual design organizes concepts about the object or class of objects being designed. Second, a rough idea about how to represent and display the designed object can be secured by casting the design requirements into an organizational framework. The MVC organizational framework is useful for this purpose, although other less specific techniques could work equally well. Third, data and knowledge structures are identified for the model and existing data structures selected or a new representation created. Fourth, control and presentation methods are specified and linked to the data/knowledge representation. These specific techniques represent the way that Smalltalk-80 is used for application software design. The ideas of factorization and the linking of a representation structure with what the user sees and how the user interacts with the system are general purpose. Indeed, these techniques can be used to understand many types of software interface design problems.

References

- Aldus (1990). *Pagemaker 3.01*. Aldus Corporation, Seattle, WA.
- Apple Programmer's and Developer's Assoc. (1987). *MacApp: The Expandable Macintosh Application*. Renton, WA.
- Barth, R. S. (1986). An Object-Oriented Approach to Graphical Interfaces. *ACM Transactions on Graphics*, Vol. 5, No. 2, April, 142–172.
- Bourne, John R., J. Cantwell, A. J. Brodersen, B. Antao, A. Koussis, and Y.-C. Huang (1989). Intelligent Hypertutoring in Engineering. *Academic Computing*, September, 18–47.
- Brown, J. R., and S. Cunningham (1989). *Programming the User Interface*. Wiley, New York.
- Krasner, G. E., and S. T. Pope (1988). A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, Vol. 1, No. 3, August-September, 26–49.
- Linton, M. A., J. M. Vlisides, and P. R. Calder (1989). Composing User Interfaces with InterViews. *Computer*, February, 8–22.

Objectworks \ Smalltalk User's Guide, Release 4 (1990). ParcPlace Systems, Mountain View, CA.

Serpent Overview (1989). Software Engineering Institute, Carnegie Mellon University, CMU/SEI-89-UG-2, Pittsburgh, PA.

Schiefler, R. W., J. Gettys, and R. Newman (1989). *X Window Systems: C Library and Protocol Reference*. Digital Press, Maynard, MA.

Stroustrup, B. (1986). *The C++ Programming Language*. Addison-Wesley, Reading, MA.

Exercises

- 4.1** The MVC paradigm appears to be a logical way of factoring the elements needed to create software systems of the type discussed in this and previous chapters. Are there any alternative ways to do the same thing? Investigate some other common methods and see if you agree or disagree that all problem factorization can be reduced to the MVC paradigm. (*Hint:* A good place to start looking for this information is in some of the references given; e.g., Schiefler, Gettys, and Newman and the Apple reference.)
- 4.2** A primary consideration in model design is the structure of the model. Explain how the collection classes allow programmers to concentrate more on the design of the model structure rather than the implementation code.
- 4.3** For each of the collection classes in Table 4.2, describe the particular features of each class and give an example of how each could be used in an application. (*Hint:* It will be helpful to examine the collection classes using the Smalltalk browser and to read examples in the tutorial manuals or other references given in Chapter 1.)
- 4.4** Given the description of a dictionary as a set of associations made up of keys and values, draw a diagram of how you could create a tree using dictionaries.
- 4.5** Select an example system to be created (or perhaps a system that you have seen somewhere else, e.g., Hypercard on the Macintosh). Discuss how to describe this problem in terms of the MVC.

Part Two

The Tools

Chapters 5 through 8 present tools that can be used for creating object-oriented systems. Although the focus of these chapters is on the use of the Smalltalk-80 language and environment, other alternative languages and environments will be mentioned. Chapter 5 briefly reviews conventional computer languages and examines object-oriented extensions to several languages, compares object-oriented languages, and briefly presents the syntax of Smalltalk. Chapter 6 examines the Smalltalk environment and classes. Chapter 7 provides a look at methods for user interface design, and finally Chapter 8 provides an in-depth examination of tools in Smalltalk-80 that can be used for building applications.

Object-Oriented Languages

This chapter begins by reviewing the characteristics of several common procedural languages that have been used in engineering during the last several decades. A number of these languages have been extended to include the object-oriented paradigm and new languages have emerged that support the object-oriented methodology.

5.1 Procedural Languages and Object-Oriented Enhancements

Table 5.1 presents a list of nine popular languages that have been used in engineering during the last three decades [see MacLennan (1983) for extensive descriptions of several of these languages]. The longevity of each language along with some notable features and a list of object-oriented extensions, if any, are given. Each language is discussed briefly next.

Fortran

John Backus of IBM and three associates are credited with creating the specification for Fortran in 1954. Fortran (FORmula TRANslating System) was initially design in the course of Backus' work on producing an efficient compiler. The language specification was almost a by-product of this work. Fortran is a compiled language in which source code is compiled, individual modules linked, and the linked code is loaded into memory and finally executed. With an emphasis on efficiency of implementation, Fortran has been very popular for production programs and is still widely used although its popularity

Table 5.1 Principal Computer Languages and Object-Oriented Extensions

Language	Year Initiated	Primary Features	Object Extensions
Fortran	1954	Scientific programming applications; efficient implementations	None
Algol	1958	Similar to Fortran notation; strong typing	None
Lisp	1958	Functional notation; rich, large environment	Flavors, CLOS, Common Loops
APL	1962	Mathematical notation; dynamic binding	None
Pascal	1968	Simple syntax; strong typing	Object Pascal, Turbo Pascal
Smalltalk	1971	Object-oriented; large environment	
C	1972	General purpose; weakly typed	C++, Objective-C
Prolog	1972	Logic-based; declarative	
Ada	1979	Strong typing; rich, complex, and large	InnovAda

has eroded significantly during the last decade. Fortran is decidedly nonobject oriented, exhibiting virtually none of the characteristics that make a language object oriented, as described in Chapter 2. However, one can argue that Fortran has a very primitive form of support for encapsulation, as manifested in subroutines that permit multiple entry points.

Algol

Algol was designed, beginning in 1957, in a quest to create a universal programming language. Simple, elegant, and general, Algol is a strongly typed¹ language that was designed for scientific computation. A statically bound² language, Algol uses declarative and imperative constructs similar to Fortran.

¹The term *type* refers to a group of expressions that share a common characteristic, for example, integer or floating-point numbers. *Strongly typed* means that the type of any expression in a language can be determined. Conversely, *untyped* means that there is no type. *Weakly typed* means that the type of an expression can sometimes be determined.

²Binding refers to whether the attributes of expressions are determined at compile time. A computer language is said to be *statically bound* if its expressions are determined at compile time whereas a language is said to be *dynamically bound* if its expressions are not determined until run time.

Although similar to Fortran, Algol never achieved widespread use in the United States and there are no object-oriented constructs in the language. Simula (Birtwistle et al., 1973), however, was designed to be an object-oriented extension of Algol.

APL

APL (A Programming Language) was created in 1962 (Iverson, 1962) and enjoyed some popularity. APL provided a concise mathematics-like notation, dynamic binding, and provided some early experience with an operating environment for the language.

Lisp

Lisp (List Processing) was designed in the late 1950s to support symbolic programming in artificial-intelligence applications. Lisp is an applicative language in contrast to the imperative languages described previously. Imperative languages rely on assignment and changeable memory to accomplish a programming task. In contrast, applicative languages work on the principle of applying a function to arguments. A principal goal of Lisp was to allow manipulation of symbols, often represented as lists. Originally, Lisp was an interpreted language; in the 1980s, most Lisp versions added compilation capabilities. Lisp is currently the language that is used most frequently for artificial-intelligence research and applications. There are numerous Lisp varieties and Lisp serves as the basis for many AI tools. Flavors (Moon, 1986) was one of the early object-oriented extensions to any language and still serves as a object-oriented programming vehicle in Lisp. Common Lisp Object System (CLOS; Bobrow et al., 1988) is an emerging standard for object-oriented methodologies in CommonLisp (Steele, 1984).

Pascal

Pascal, a successor to the ideas in Fortran and Algol, was explicitly designed to teach programming. Initiated in 1968, Pascal remains a principal language for teaching high school and college students how to program. Strongly typed with basic Algol and Fortran-style primitive data types (reals, integers, Booleans, etc.), Pascal has been extended to include object-oriented constructs. For example, Object Pascal (Snyder, 1986) and Clascal (Schmucker, 1986) have

added object-oriented constructs. Popular Pascal packages such as Turbo Pascal (Borland, 1990) have also added object-oriented features.

Smalltalk

Smalltalk grew from the vision of Alan Kay in the late 1960s (Kay, 1977). Based on concepts from Simula (Birtwistle, 1979) and Logo (Papert, 1980), Smalltalk began as a research effort about 1970. A wholly object-oriented language, Smalltalk has developed since the early 1970s into a combination programming language and environment. Smalltalk has served as a model for many modern object-oriented programming languages, yet still retains a significant position in the object-oriented programming environments market due, in part, to the robustness of its large class library. Smalltalk is suitable for use in many applications and is especially useful for rapid prototyping and creating graphics interfaces. Smalltalk is a “pure” object-oriented language exhibiting all four fundamental characteristics: abstraction, encapsulation, inheritance, and polymorphism.

Prolog

Prolog (Clocksin and Mellish, 1984) is based on logic programming. Developed in France in the early 1970s, Prolog (Programming in Logic) has experienced wide acceptance in Japan and France, as well as limited success in the United States. Although it was once thought by some that Prolog could supplant Lisp as a leader in artificial-intelligence (AI) work, this promise did not materialize. Prolog is a declarative language in which logic statements are made according to a specified syntax. The Prolog interpreter attempts to unify declarations, that is, to find variables that, when substituted into terms (i.e., data objects), make the terms become identical. Prolog is essentially a backward-chaining rule system. If declarative rules can be written as a problem description, the internal control of a Prolog system can then attempt to unify the statements. An advantage to Prolog is that it is entirely declarative; there is no need to encode control information. For this reason, Prolog has been popular among AI programming novices.

C

C is a programming language that is the native language used in implementing the UNIX operating system. C was developed in the early 1970s

by Dennis Ritchie at Bell Laboratories and has become in the following decades one of the most popular general-purpose programming languages. Some of the hallmarks of C are that it is a relatively small language, and it is terse, modular, and portable. It is a weakly typed language. Compilation of C produces machine code that approaches the efficiency of assembly-language coding. In applications in which time is a critical factor, C has become the language of choice in many industries. There are many flavors of C including object-oriented varieties such as Objective-C or C++, which will be discussed in the next section.

Ada

In the mid-1970s, the United States Department of Defense supported the development of a next-generation programming language that would support top-down software design. A primary concern was to provide modularity and information hiding in order to reduce the amount of time needed to create large software systems. The language created was Ada, named in honor of Augusta Ada, Countess of Lovelace, said to be the world's first programmer. In the context of the definitions of object-oriented programming given in Chapter 2, Ada is not a true object-oriented language because it does not support inheritance, yet it has support for data abstraction and information hiding. There has been some work to add additional object-oriented features [see, e.g., Simonian and Crone (1988)].

5.2 Major Object-Oriented Languages

There are many object-oriented languages. Saunders (1989) reviewed some 88 object-oriented languages, most having academic roots and 16 of which were commercially available. A review of all these languages will not be attempted here; instead, only a handful of the most important languages, divided into four categories will be discussed. These categories are the C-oriented systems, the Lisp-related systems, the Smalltalk systems, and other types. The primary purpose of these descriptions is to convey the types and capabilities of object-oriented languages. An explicit attempt is made to compare the systems to the capabilities of Smalltalk.

Table 5.2 presents a summary of eight object-oriented languages, two from each category mentioned previously. The first two items are two Smalltalk versions: (1) Objectworks \ Smalltalk (ST-80) by ParcPlace Systems, which is used for most examples in this text, and (2) Smalltalk/V created by Digitalk. The C-oriented languages examined are C++ and Objective-C. Lisp entries are

Table 5.2 Examples of Several Object-Oriented Programming Languages

Name	Description
Smalltalk-80	Commercial object-oriented system for workstations, PCs, Macintoshes. From ParcPlace Systems.
Smalltalk/V	Commercial object-oriented system for PCs and Macintoshes. From Digitalink.
Objective-C	Object-oriented extension of C with several class libraries. From Stepstone Corporation.
C++	Object-oriented enhancement to C. AT&T and others.
Flavors	Early MIT-developed object-oriented extension to Lisp Weinreb and Moon (1980).
CLOS	Common Lisp Object System — an extension to Common Lisp. Franz, Inc., Lucid, Inc., and others.
Eiffel	Object-oriented language emphasizing class structures. From Interactive Software Engineering, Goleta, CA.
Actor	A PC-based object-oriented system with strong similarities to Smalltalk. From The Whitewater Group.

Flavors and Common Lisp Object System (CLOS) and, in the miscellaneous language category, Eiffel and Actor are briefly reviewed.

C-Oriented Systems

C++ was designed by B. Stroustrup in the early 1980s at Bell Laboratories. C++ adds support to C for data abstraction and provides various improvements to the C language, such as strong type checking. C++ is based on the C language (Kernighan and Ritchie, 1978, 1988), but adds inheritance.

C++ has achieved a wide following in the programming community, presumably because of the large number of C programmers who exist in this country. Most current versions of C++, however, have no built-in integrated programming environment and must use the features of whatever operating system with which they are used. Furthermore, no class libraries are part of the original C++, although efforts have been made to create libraries for use with C++ [see, e.g., InterViews by Linton, Vlisides, and Calder (1989)].

Objective-C, created in 1983 by Productivity Products International (PPI), now the Stepstone Company, has achieved a significant level of prominence in the object-oriented marketplace. Several features contribute to Objective-C's success. First, class libraries that implement easily understandable classes (e.g., **Dictionary**, **Array**, **Set**, etc.) are provided which give the user some reusability capability. Sixteen classes are provided with the basic system and

Table 5.3 An Example of Flavors Programming [Concepts from Weinreb and Moon (1980)]

```
(defflavor moving-object (x y xVel yVel mass) ()  
(defflavor ship (enginePower numberPassengers name)  
    (moving-object))  
(defflavor meteor (ironPercentage)  
    (moving-object))  
(defmethod (ship :speed)() (sqrt (+ (^ xVel 2) (^ yVel 2))))  
(defmethod (ship :direction)() (atan yVel xVel))  
(setq myShip (make-instance 'ship ':engine-power 300 ':numberPassengers 2000 ':name Titanic))
```

more can be purchased for implementing user interface functions. The syntax of Objective-C is more easily understood than C++, and is perhaps closer on the phyloprogrammatic scale to Smalltalk than to C or C++.

Lisp-Related Systems

Flavors was reported in 1980 (Weinreb and Moon, 1980) as an extension to Lisp for the Symbolics Lisp Machine. The name is said to come from an analogy to the creation of different flavors of ice cream confections that could be created by “mixing in” additional flavor components to a base flavor. The “mixing in” idea is equivalent to inheritance. Flavors simply extends Lisp by adding additional functions that permit defining and manipulating objects. Table 5.3 gives a simple example of some definitions in Flavors. Like the enhancements to C, Flavors is a language extension, not an object-oriented language in its own right.

The syntax for the code shown in Table 5.3 is somewhat simplified from the actual Flavors code, but the intent is the same. In the same way as inheritance was discussed in Chapter 2, Flavors provides in this example for the description of a ship as a subclass of a moving object. The definition syntax defines the name of the object, followed by an instance variable, then by the name of the superclass that is to be inherited (“mixed in”). In the case of the moving object, there is no object to inherit from; in the remaining definitions, inheritance is from the moving object. Methods are defined separately from the structure definition. In this example, two methods are defined, one for finding the speed and the other for finding the direction of a ship. Finally, an example of defining myShip as an instance of ship with explicitly defined variables is shown.

The Common Lisp Object System (CLOS) (Keene, 1989) is similar to Flavors in that the system is an extension to a Lisp language. CLOS differs from Flavors in that it is much better developed, containing numerous features

that Flavors does not have. CLOS is emerging as an object-oriented standard for use with CommonLisp. There are a number of other Lisp extensions [see, e.g., Saunders (1989)].

Smalltalk Systems

ParcPlace Systems' Smalltalk-80 is the oldest and most complete object-oriented programming language and environment. ST-80 is available for a variety of different computer platforms including the PC, Macintosh, HP/Apollo workstation, Sun, and DEC workstations. An outstanding feature of this system is the ability to run transparently across the different platforms. In other words, code generated on one system will run without change on all the other hardware platforms. ST-80 was marketed by Xerox and other licensees (e.g., Tektronix, Fuji) until the late 1980s. ParcPlace Systems was created in 1986 as a separate business unit associated with Xerox Palo Alto Research Center (PARC). In 1988, ParcPlace Systems became an independent corporation and now distributes Smalltalk-80 as one of several software products.

Smalltalk/V (V stands for Virtual) is the principal competitor to Smalltalk-80. Marketed by Digitalk, ST/V runs on PC and Macintosh systems. ST/V has a smaller number of classes than ST-80, and requires less memory to run. In 1989, Digitalk introduced a version of ST/V that runs with OS/2 and can produce standalone applications, a feature that stirred some excitement in the object-oriented community at the time.

Other Object-Oriented Languages

There are a number of object-oriented languages that do not easily fit into either of the three categories listed previously. For example, Eiffel (Meyer, 1988) is a basic object-oriented language that features strong typing to help programmers ensure correctness and robustness of their software. Eiffel runs on various versions of UNIX and uses C as intermediate code, making Eiffel potentially portable to any machine that will run C. Actor (1989) is a software product that runs on PC-class machines and has many of the attributes of Smalltalk in a personal computer environment. Actor, however, should not be confused with research on "actor languages" which is concerned with the study of multiple independent computational agents.

General Observations and Comparisons

Object-oriented languages, having become increasingly popular during recent years, can be found in many different varieties. Naturally, there is

considerable controversy about which features are best and, of course, the answer to this normally is linked to one's perspective. For example, a person learning about object-oriented methods who has a strong C programming background will naturally gravitate to one of the object-oriented extensions to C. Similarly, Lisp aficionados will prefer Flavors or CLOS to other language varieties. People who have a preference for Smalltalk are likely to be less efficiency conscious but have a high interest in reusability and productivity.

In terms of size of memory usage and speed, the C-related languages have a distinct advantage because C code will normally produce compact and fast code—a clear advantage in many situations. However, almost none of the C-related languages have robust environments that support debugging, editing, and understanding programs, features that are supported much better by both the Lisp and Smalltalk environments.

In terms of facilities, both Lisp and Smalltalk have excellent environmental support, including excellent debugging facilities. Smalltalk-80 has an incremental compilation facility that compiles methods as they are defined; this capability makes it appear to the programmer that the compilation phase of programming is avoided. Both Lisp and Smalltalk have automatic garbage collection facilities, a capability not found as a standard feature in other languages surveyed. In brief, garbage collection refers to the system capability for automatic reclamation of memory space occupied by lists (Lisp) or objects (Smalltalk) which are no longer in use.

For reusability, the different Smalltalk systems are markedly better than any of the other languages that have been discussed. A large number of classes makes it possible for the programmer to reuse facilities that come with the system without reprogramming, an often huge savings in time.

5.3 The Syntax of the Smalltalk Language

The Smalltalk language is small, simple, and easy to use to write programs. This statement contrasts with a common perception that Smalltalk is difficult to understand and use. Both the first statement and the perception are correct. The explanation is that, indeed, the *language* part of Smalltalk is small, simple, and easy to use. However, the *environment* part of Smalltalk, including all the classes that exist, is large and requires a considerable effort to learn. This section will examine some features of the language and Chapter 6 will explore the environment, including a description of the tools and classes that are part of the ST-80 environment. The reader is referred to texts that review Smalltalk syntax in depth [see, e.g., LaLonde, Wilf, and Pugh (1990); Goldberg and Robson (1989), and Pinson and Wiener (1988) for more details].

Tables 5.4 and 5.5 summarize some of the main syntax features for Smalltalk-80. Language syntax for other varieties of Smalltalk, for example,

Table 5.4 The Syntax of Smalltalk-80: Literals

Types	Examples
Numbers	35 8r177 16rFE 2r0100010 1.5e6 3.14
Character	\$b
String	'This is a string'
Symbol	#book #update
Array	#(1 2 3) #(#(1 2 3) 'a string' #aSymbol)

Smalltalk/V, is little different. Different types of literals are shown in Table 5.4: numbers, characters, strings, symbols, and array representation. Numbers are quite simple. As shown in Table 5.4, any number from integer through floating point can be represented simply. The notation employed for using different radixes is: Rnnnnnnn, where R is the radix value, r is a placeholder, and nnnnnn is the value of the number in the radix indicated. Note that numbers in arithmetic expressions are automatically coerced, for example, 3.14*2 is equal to 6.28, because the number 2 is coerced to be of the same type as the number 3.14. Characters are preceded by a dollar sign and strings are delimited by single quotes. Symbols are designated by a preceding pound sign as are arrays. Arrays, however, have leading and trailing parentheses. Arrays may be nested, as shown, and may contain any object as an element.

Table 5.5 displays examples of several Smalltalk-80 syntactic constructions. Smalltalk follows the convention that instance variable names begin with lowercase and global and class variables names begin with uppercase. Global variables, such as the system dictionary “Smalltalk,” may be accessed by simply

Table 5.5 The Syntax of Smalltalk: Syntactic Constructions

Concepts	Examples
Variable names	numberHolder dictionaryOfLanguageTypes Smalltalk GlobalVariable
Assignment	year := 1993.
Messages	salary := 2000. <i>ClassObject</i> message <i>instanceObject</i> message <i>Smalltalk</i> inspect
Blocks	myToyota drive: north at: 55
Conditionals	[:r 3.14 * r * r] input < 65 ifTrue: [Transcript show: 'turn up the heat'] ifFalse: [Transcript show: 'have a good day'] [fileStreamPointer atEnd] whileFalse: [characterPointer increment]
Iteration	#(1 2 3) do: [:x x*x] 'The Quick Brown Fox' select: [:c c isUpperCase]

sending the dictionary the message “inspect” (i.e., *Smalltalk inspect*). Assignment is direct and simple, for example, assigning a variable to an expression as shown in Table 5.5. In the examples given, the expressions shown are simply numeric values. The “`:=`” operator is an assignment standard for Smalltalk; for many years, the assignment operator was a back arrow. The “`:=`” operator is, however, more compatible with standard keyboards than the back arrow.

Objects respond only to messages. As shown in Table 5.5, there may be class objects that respond to messages and instances of classes that respond to messages. The distinction is important! In subsequent chapters, it will be pointed out how class and instance messages are kept separate from each other. Novice Smalltalk programmers often confuse the two types of messages. In the example given, there is one class indicated by a capital letter: **ClassObject**, an instance object `instanceObject`, a global variable, and an instance of a class (`myToyota`). In each case, the object to which the message is sent is shown in italics. Note the last example: “`myToyota drive: north at: 55.`” `myToyota` is an instance of the Toyota class (recall the example in Chapter 2) and `drive:at:` is the message! `drive:at:` takes two arguments shown here as `north` and `55`. This syntax makes it possible to construct fairly readable messages.

Messages sent to objects can be categorized into four groups: unary, binary, keyword, and parenthesized expressions. In parsing a Smalltalk expression, parenthesized expressions take precedence over unary expressions, unary expressions take precedence over binary expressions, and binary expressions take precedence over keyword expressions. Binary and unary expressions parse left to right. Parenthesized expressions are those expressions enclosed by parentheses. Unary expressions are expressions that do not have arguments, for example: `Time now`. Binary expressions take a single argument, for example, `3 + 4` or `total - 2`. Binary expressions tend to be arithmetic. Keyword expressions contain messages with one or more arguments, for example, `index max: limit` or `HighwayContract spend: $1000 for: "testing."` Consider the parsing of the following expression:

5 factorial between: 52 + 45 and: ‘good morning’ size * 11

Because there are no parentheses in this expression, a unary reduction is made first, followed by a binary reduction, and finally the keyword reduction:

120 between: 52 + 45 and: 12 * 11	(unary reduction)
120 between: 97 and: 132	(binary reduction)
true	(keyword reduction)

In this example, sending `factorial` to the number 5 returns 120 and sending `size`

to the string ‘good morning’ returns 12. To evaluate the preceding expression, simply type it into a workspace in ST-80, highlight the expression, and select “printit” from the pop-up window. After execution, “true” will appear immediately after the expression.

Blocks are Smalltalk constructs that permit execution of a deferred sequence of actions. Delimited by a set of square brackets, blocks are used in many control structures in Smalltalk. In the example in Table 5.5, the block contains an argument *r*, indicated to be an argument by the preceding colon “::.” The block contains code to be executed at some later time. When sent the message *value: 10* the block would return, in this case, 314. If you try out this example, note that spacing between the elements in the block is important. The * is a message sent to 3.14; hence, there must be a space between the items to permit interpretation of messages and arguments. A block may have either no arguments passed or multiple arguments passed. In the first case, a block is simply sent the message *value* to cause it to execute. For two arguments, the message *value:value:* including appropriate arguments will cause execution (e.g., *aBlock value: 10 value: 20*). Blocks are also used in conditionals, as shown in the two examples in Table 5.5. In the first case, the expression “*input < 65*” is evaluated; ifTrue, one block is executed, and ifFalse, the alternative block is executed. Similarly, in the following example, a pointer to a fileStream is tested and although the file is not at the end of the stream, the code in the following block is executed to increment a character pointer.

Iteration appears in many ways in Smalltalk. Two ideas are shown in Table 5.5. One is the “do:” construct which simply iterates through collections of objects. In this example, three numbers in an array are passed to a block for execution. Each number, in turn, will be passed to the code inside the block, and the number will be squared. For the second example, a string is passed through the iterator “select” which tests each element in the string to see if it is uppercase or not. If the character tested is uppercase, it is retained, else it is discarded. There are a number of iterators of this type that are useful, for example, collect, reject, and detect that provide useful collection filtering capabilities.

The brief comments given previously about the syntax of the Smalltalk-80 language are not, in any way, intended to be a substitute for study of the language. The student is referred to the basic references on the Smalltalk-80 referenced in Chapter 1 and encouraged to become proficient in the language syntax before proceeding to study the material in subsequent chapters. A knowledge of the Smalltalk-80 language will be needed to understand the examples given. Perhaps the most useful material that the student can use to learn the details of the language is the introductory material presented in the *User’s Guide* that is delivered with Smalltalk-80 (*User’s Guide*, ParcPlace Systems, 1991).

5.4 A Language Prospectus and Comparison

Arguments about which language is best and why seemingly arise almost continuously. The next few paragraphs attempt to take a rational, if somewhat biased, look at why there are so many diverse opinions about languages. The apparent reason for this diversity is that different individuals have different needs and, in addition, they have backgrounds that predispose them toward one position or another. Outlined in the following discussion are a variety of concerns and issues and some opinions about the suitability of the various languages that have been presented for the problems indicated.

Language Familiarity

Probably the single most important reason for favoritism of one language over another is prior familiarity and experience with a language. Computer languages require considerable study to learn. After learning a language almost no one wants to give up a comfortable set of knowledge to learn a new language even if it provides some additional features. Familiarity with the C language is clearly the primary reason for the high degree of success of C++ as an object-oriented language. During recent decades, AT&T made UNIX widely available to universities. Several generations of students have learned to use this operating system and its native language, C. Engineers often have a predilection for efficiency and speed, attributes of C-language implementations. No wonder that C has become very popular and is used worldwide for creating systems and applications. Hence, with a high level of interest in and knowledge about C among programmers trained in using UNIX, it is plausible that C++ would attract the attention of these persons. It would be less likely that this group would move to using entirely different environments such as Smalltalk.

Task Suitability

A rationale methodology for selecting a language, apart from familiarity and experience, is the suitability of a language for achieving a task. Ultimately, the optimal computer language would be the one that solves the problems that one has with the least effort, most efficiency, and least time. Unfortunately, most problems that need to be solved all have different requirements and thus languages and environments with different characteristics will need to be employed. For example, tasks may require a high level of interactive use between the computer and human or may require large amounts of

computation. Obviously, slow but easy to use languages are unsuited for the latter task, and vice versa. Thus, it is clear that task suitability must be a driver in selecting languages and environments. Using this criterion, even languages such as Fortran and C have a distinct place, for example, for number crunching. Object-oriented languages have a niche in problem solving for tasks that permit problem factorization into objects with behaviors. The paradigm results in a reduction in complexity and simplicity in solving large problems.

Performance Issues

One potential criterion for choosing between languages and different implementations of the same language is performance. Although execution speed was once a criterion of paramount importance, this issue has almost vanished as computer performance has increased and costs for expanded computation power have decreased. Arguments for selecting languages for problem solving based on performance are no longer important except for the most intensive number-crunching requirements.

Memory Issues

In much the same way as the issue of performance, during recent years costs for memory have decreased so significantly that making language choices based on the amount of memory required is no longer considered except for special applications (e.g., mobile equipment, consumer systems, etc.). Some languages manage memory more efficiently and more automatically than others. For example, Lisp and Smalltalk automatically collect garbage; that is, both languages reclaim unused memory space made free by removing objects or pointers that are no longer used. C++ and Objective-C require that the user provide explicit memory management code.

Graphics Issues

Graphics have become increasingly important during recent decades. Inexpensive moderate-resolution graphics capabilities can now be found on most workstation and PC-level computers. Hence, languages that can make use of hardware capabilities are almost mandatory. Most languages have added graphics through the addition of graphics calls in the form of libraries. Examples are found in Fortran libraries for data presentation, libraries of C routines to drive graphics displays, and CAD packages that have built-in graphics drivers

that directly support application code. Other languages, such as Smalltalk, were designed around specific graphics user interface concepts that enabled every user interaction to be accomplished through the graphics interface.

User Interface Issues

The most common user interface since the 1960s has been the command line interface, in which users issue commands to an operating system after a prompt symbol displayed on a screen. Early efforts to move to graphics user interfaces met with cost problems due to the high expense of the graphics hardware required. Now, however, costs have declined to the point that it is likely that graphics "point-and-click" user interfaces will probably become the norm for the 1990s. Evidence for the growing influence of these types of interfaces is commonly observed in the Macintosh interface, Microsoft Windows, OS/2 PM, SunViews, the NeXT interface, and others [see, e.g., Brown and Cunningham (1989)]. Although some of these interfaces operate almost like a graphics translation of a command line interface, others capitalize on visual metaphors to provide additional capabilities. Computer languages can be an integral part of such capabilities, providing direct methods for manipulating user interfaces. If an application that is to be written requires a high degree of user interface support, creation of the application is likely to be much easier using a language that contains strong support for manipulating the interface.

Portability Issues

Portability of code from one machine to another has become an issue due to the many different types of hardware platforms that are available. Traditionally, to port software from one machine to another, a requirement has been to modify the source code that runs on one machine to the requirements of another machine. For example, a Fortran program that runs on one machine will not usually run on another manufacturer's hardware and compiler. C, however, was designed to be fairly portable. Most C code can be moved in source form from one hardware platform to another. The same is not true of libraries, of course, that require use of different input/output (I/O) ports, screen drivers, and so on. Smalltalk-80 has the outstanding capability of being able to move, intact, complete images between different hardware platforms. The way that this capability was secured was to produce different implementations of the basic parts of Smalltalk (the virtual machine) for a number of different hardware platforms. Small and simple, the virtual machine provides the capability for supporting the remainder of the Smalltalk system, written in

Smalltalk itself. Given this capability, Smalltalk can be characterized as the most portable language and environment.

Training Issues

In building applications, the tendency is to select, as indicated previously, the most familiar language available. However, the presence of skill in a language that is basically unsuitable for the required task may doom a project to failure. If one turns to more suitable languages to creating applications, the issue of training appears. How long does it take to learn a new language? What does the learning curve look like? How much material needs to be mastered before productive work can occur? The answers to these questions appear to be different for all the languages mentioned in this chapter but are directly proportional to a user's experience with a given language and with analogous languages. For example, a C programmer will have no difficulty learning Fortran or Pascal because the differences between the languages are relatively slight. Similarly, a Smalltalk-80 programmer can learn Smalltalk/V or Actor with little additional effort. However, the learning curve for a Fortran or C programmer to learn the substantial object-oriented languages remains flat for a long time. For languages that simply add basic object methodologies, such as C++ or Flavors, learning the basic object-oriented additions to the language occurs fairly rapidly. For Smalltalk, Actor, Objective-C, and languages that have class libraries and added facilities, the learning curve is much longer due to the requirement of learning how the classes work. What is the trade-off? The trade-off is that language syntax is usually fairly easy to learn and productive work can commence after a modest time investment for the basic object-oriented languages. The most complex object-oriented languages require a longer learning time but the payoff is a much higher level of productivity when training is complete. Figure 5.1 shows a *hypothesized* learning curve for some of the languages discussed in this chapter. It should be emphasized that this family of curves is not supported by any hard data and is based only on observation, anecdotal information, and experience. As shown in the figure, the traditional languages require a modest learning time to achieve moderate levels of productivity; the true object-oriented languages take longer to learn but result in higher levels of ultimate productivity. The time on the abscissa of the figure represents months of about half time devotion to study of the language. Of course, times vary considerably by individual. Testing to validate these hypothesized curves would require extensive efforts and, indeed, may not even be possible. However, it is believed that the general information shown has some comparative validity.

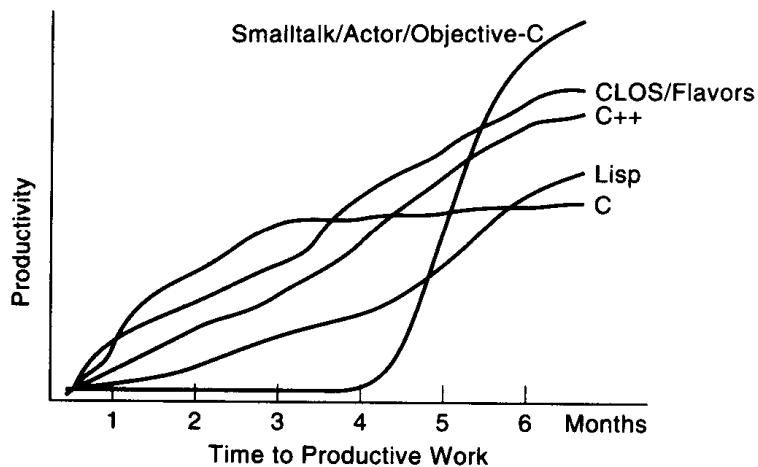


Figure 5.1 Time versus Productivity for Several Languages.

Reusability

The key to productivity is reusability. The easiest way to produce code is obviously to have it already! The next easiest way is to specialize slightly the code that you have. Having an extensive class library permits rapid creation of new classes by inheritance as discussed in Chapter 2. High productivity results in being able to create new classes without having to produce new code continuously. Of course, the major drawback of having a large class library is having to understand the contents of the library, a quite formidable task, particularly in systems with large libraries such as Smalltalk-80. Table 5.6 compares the number of classes that are currently provided with the four major commercial object-oriented languages: Smalltalk-80, Smalltalk/V, Objective-C, and C++. As is evident from Table 5.6, the number of "built-in" classes varies from over 330 for Smalltalk-80 to none for C++. Objective-C permits purchase of additional classes to support user interface creation.

Table 5.6 Classes Provided in Object-Oriented Languages (in 1991)

Name	Number of Classes
Smalltalk-80 (Release 4)	~ 330
Smalltalk/V (Windows version)	~ 175
Objective-C extension	> 20 > 60
C++	0

Ease of Programming

Language tools that assist in developing application programs are a consideration in selecting a language. Traditional programming languages have typically used the edit-compile-debug loop; that is, the user creates a source file of program text using a text editor of some type, compiles the source file, and then debugs the program by examining what errors occur when an attempt is made to run the program. Sophisticated compilers assist this process by doing extensive checking during compilation to assure the types are the same and that syntax is correctly followed. Furthermore, debuggers for some languages are available that permit single-stepping through a program to determine errors in logic. The most sophisticated tools provide this type of facility in a packaged way that assists the user in debugging program code. The more facilities that a debugger provides, the more rapidly application programs can be developed. Interpreted languages provide more flexibility than compiled languages because the code can be debugged on a line-by-line basis without having to go through the edit-compile-debug cycle. However, the drawback to interpreted languages are that they are much slower than compiled languages because additional time is taken up in the interpretation process. A good intermediate methodology is incremental compilation where the code is compiled in small increments. Hence, changes in any part of the program can be made easily and the behavior of the code traced easily, yet there is little speed penalty to making changes in the code. Smalltalk-80 uses this latter technique.

The Importance of an Environment

An environment is a collection of tools that provides support for the programmer. There are various examples of environments; for example, in Lisp one of the first environments was the InterLisp environment (Teitelman and Masinter, 1981), LOOPS is another, and Smalltalk-80 is a third. The next chapter will discuss the ST-80 environment in detail. The importance of a good environment is that the programmer is provided with all the needed tools to develop applications—everything from code inspection, editing, debugging, and graphics to interface building. The desktop metaphor is a good one; that is, everything needed to build an application is within easy reach on the desktop.

Issue Summary

Table 5.7 presents a summary of the features of the languages discussed in this chapter, segmented according to the characteristics just discussed.

Table 5.7 Language Feature Prospectus*

Feature	Extension to Languages	Object-Oriented Languages with Classes	Traditional Languages
Traditional programmer adaptability	High	Low	Very high
Performance	High	Moderate	High
Memory requirements	Low	High	Low
Training needs	Medium	High	Low
Productivity	Medium	Very high	Low
Reusability	Medium	Very high	Low
User interface support	Low	High	Low
Environmental support	Low	High	Low

* Extensions to languages: C++, Eiffel, Object Pascal, Flavors, CLOS; object-oriented languages with classes: Smalltalk, Actor, Objective-C; traditional languages: Fortran, C, Lisp, Pascal.

Language types are segmented into three different categories, as indicated in the table legend. An indication is given of the degree to which each group fulfills the criteria shown in the left column. Each characteristic is graded as low, medium, high, or very high. As one can see from this evaluation, the object-oriented languages containing class libraries are best in this type of evaluation. Doubtless the criteria used are flawed, but nonetheless should give an indication of how these different programming language categories differ in capability. If one were to assign numeric scores to the characteristics (e.g., low = 1 to very high = 4), one could obtain overall numeric scores. When this exercise is carried out, it is easy to see that language extensions provide some improvement over traditional languages, but languages with libraries of classes provide significant improvement.

References

- Actor Language Manual* (1989). The Whitewater Group, Evanston, IL.
- Birtwistle, G., O.-J. Dahl, B. Myhrlaug, and K. Hygaard (1973). *Simula Begin*. Petrocelli/Charter, New York.
- Birtwistle, Graham M. (1979). *A System for Discrete Event Modelling on Simula*. MacMillan, London.
- Bobrow, D. G., L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon (1988). Common Lisp Object System Specification, X3J13 Document 88-002R, June.

- Brown, J. R., and S. Cunningham (1982). *Programming the User Interface*. Wiley, New York.
- Clocksin, W. F., and C. S. Mellish (1984). *Programming in Prolog*, 2nd ed. Springer-Verlag, New York.
- CLOS. See Bobrow et al. (1988), and Keene (1989).
- Digitalk, Inc., 9841 Airport Road Blvd., Los Angeles, CA 90045.
- Eiffel, Interactive Software Engineering, Goleta, CA.
- Franz, Inc., Alameda, CA.
- Goldberg, Adele, and David Robson (1989). *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA.
- Iverson, K. (1962). *A Programming Language*. Wiley, New York.
- Kay, Alan C. (1977). Microelectronics and the Personal Computer. *Scientific American*, September, 231–244.
- Keene, Sonya E. (1989). *Object-Oriented Programming in Common LISP: A Programmer's Guide to CLOS*. Addison-Wesley, Reading, MA.
- Kernighan, B. W., and D. M. Ritchie (1978, 1988). *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ.
- LaLonde, Wilf R., and John R. Pugh (1990). *Inside Smalltalk*, Volumes 1 and 2. Prentice-Hall, Englewood Cliffs, NJ.
- Linton, M. A., J. M. Vlisides, and P. R. Calder (1989). Composing User Interfaces with InterViews. *Computer*, February, 8–22.
- Lucid, Inc., Menlo Park, CA.
- MacLennan, Bruce, J. (1983). *Principles of Programming Languages: Design, Evaluation and Implementation*. Holt, Rinehart and Winston, New York.
- Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ.
- Moon, David A. (1986). Object-Oriented Programming with Flavors. *Object-Oriented Programming Systems, Languages and Application Conference Proceedings*. Association for Computing Machinery, Portland, OR, September, 1–8.
- Objective-C, The Stepstone Corporation, 75 Glen Road, Sandy Hook, CT.
- Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York.
- Pinson, Lewis J., and Richard S. Wiener (1988). *An Introduction to Object-Oriented Programming and Smalltalk*. Addison-Wesley, Reading, MA.
- Saunders, J. (1989). A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming*, Vol. 1, No. 6.

- Simonian, R., and M. Crone (1988). InnovAda: True Object-Oriented Programming in Ada. *Journal of Object-Oriented Programming*, Vol. 1, No. 4, November/December, 14–21.
- Schmucker, K. (1986). *Object Oriented Programming on the Macintosh*. Hayden Books, Hasbrouck Heights, NJ.
- Snyder, Alan (1986). Common Objects: An Overview. *SIGPLAN Notices*. ACM, New York, 845–852.
- Steele, Guy L., Jr. (1984). *Common Lisp, The Language*. Digital Equipment Corporation, Maynard, MA.
- Stroustrup, B. (1986). *The C++ Programming Language*. Addison-Wesley, Reading, MA.
- Teitelman, W., and L. Masinter (1981). The Interlisp Programming Environment. *Computer*, Vol. 14, No. 4, April, 25–34.
- Turbo Pascal 5.5. Borland, Scotts Valley, CA.
- UNIX Operating System. Western Electric, Greensboro, NC.
- User's Guide, Objectworks \Smalltalk*, Release 4 (1991). ParcPlace Systems, Mountain View, CA.
- Weinreb, D., and D. Moon (1980). Flavors: Message Passing in the Lisp Machine. A.I. Memo No. 602, Massachusetts Institute of Technology, November.

Exercises

5.1

From the subset of programming languages that you know, compare and contrast the features of the languages that you like and dislike based on the kinds of analyses used to evaluate languages in the chapter. Next, form class groups and argue about your findings. Do all of you agree? Or do you disagree? Why?

5.2

Study the information in Table 5.7. Do you agree with this information. If not defend your position.

5.3

Smalltalk syntax review exercises. This is a good place to stop and spend a day or two doing the ST-80 tutorials that are delivered with the ST-80 system from ParcPlace to get a feel for the language, if you have not done so already. The knowledge that you gain by doing these tutorials will be helpful in upcoming chapters.

Introduction to the ST-80 Environment

6.1 Environment Concepts

In general, an environment is something that surrounds. When used in the phrase *computing environment*, the term means something that surrounds the code that one works with; even more generally, it denotes the set of tools that are available to the user to manipulate programmatic code, text, images, or other more complex computational entities.

Operating systems and environments are somewhat similar terms: Both deal with the interface between the computer and the user. Operating system software provides the interface to the machine hardware and environments are usually built on top of operating system facilities. The phrase *operating system environment* connotes collections of utilities that a user of an operating system can employ. Most generally, an operating system is considered to be an integrated set of software that defines a basic set of operations that can be conducted on the computer hardware. Typical components of an operating system include code to manage multiple processes, a command line interface, memory management routines, and device drivers. Some operating systems permit only one process to run at a time (e.g., DOS) and others permit multiple processes (e.g., UNIX). Operating systems have evolved during recent decades from primitive systems that directly handled the basic computer operations to complex multitasking, multi-cpu systems that can support hundreds of users on one large computer. Systems with names such as VMS, DOS, OS/2, MacOS, OS/MVS, Multics, OS/360, UNIX, CP/M, and so on have been created to be used with various types of hardware. Students of operating systems study the architecture of operating systems, services, file systems, scheduling, memory

Table 6.1 Components of the ST-80 Environment (Release 4)

Characteristic Name	Definition
System browser	Permits inspection and creation of code
File list	Access to the computer file system
Text editor	Creates text documents
*Terminal window	Connection to communications
*Form editor	Edits forms (bitmaps)
*Bit editor	Edits bits in forms
System workspace	Contains commonly used objects and messages for controlling system
Debugger	Used for correcting errors in code
System transcript	Displays messages from system

*Found only in Version 2.5.

management, concurrent processes, and examples of different types of operating systems.

Environments have some of the same flavor of operating systems, although it is more usual to discuss "higher" level kinds of programmatic components, when describing the environmental characteristics of a system. For example, an environment might contain many integrated tools for achieving a certain goal. For example, a desktop publishing environment might consist of a text editor, a page layout program, and facilities for printing to a printer or typesetter. This set of facilities is an example of a special-purpose environment, designed for the purpose of surrounding the application, in this case, desktop publishing. To extend this limited view to the general environment for programming, there can be many kinds of tools that facilitate the programmer's chore, for example, code browsers, debuggers, code tracing and inspection facilities, editors, graphics facilities, and communications. This listing of tool names closely mirrors the actual integrated set of tools that are found in the Smalltalk-80 environment which will be described in detail in this chapter.

Table 6.1 displays a summary of the characteristics of the Smalltalk-80 environment and Figure 6.1 shows an example of the visual representation of environmental tools on the computer screen "desktop." The most prominent of the main components of the environment is the system browser, which will be described in more detail. In brief, this browser permits organization of information about the entire system. All classes and their methods may be examined and new classes and methods may be easily added to the system. The desktop provides tools that interface to the external environment, for example, the terminal window and the file list, which respectively permit communication and

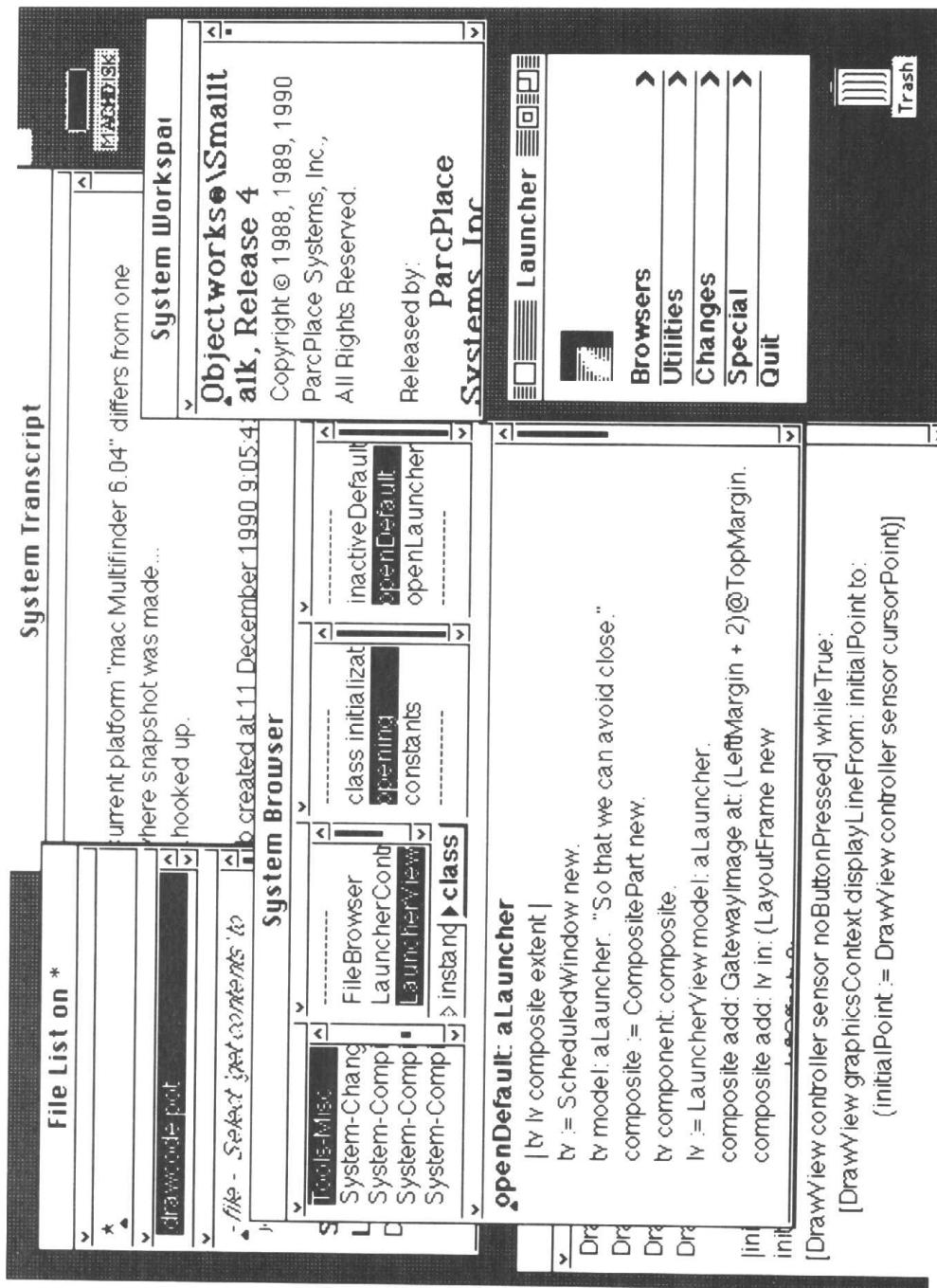


Figure 6.1 Screen Dump of Smalltalk-80 Screen.

listing of files on disk. Text editing is provided with a built-in editor. A debugger is provided that has many capabilities including single-stepping through code and examining the values of the variables. The system transcript provides a place for messages from the system to be displayed and the system workspace contains useful objects and methods for working with the system components.

During 1990 and 1991, a major revision to Smalltalk-80 was made by ParcPlace Systems which added several significant features to the environment. The capability for displaying in color was added, as was the ability to run in windows of the host operating system. In Table 6.1, the characteristics of the ST-80 environment that are found only in Version 2.5 (the ST-80 release preceding Release 4) and earlier versions are denoted with an asterisk. For example, the terminal window feature was removed from the standard Release 4 and added to a supplementary set of Smalltalk-80 code called Objectkit \ Smalltalk. Classes from earlier versions that dealt with bitmaps were also removed and graphics classes added that are compatible with each host platform. Release 4 provides a high level of integration with every host platform. Perhaps most interesting is the capability for moving application code from one host platform to another (e.g., from Macintosh to Sun, or from DEC to PC) without having to change the code. In each case, the application will run on each different platform, adopting its "look and feel" to the characteristics of the user interface of the given hardware platform.

6.2 Characteristics of the ST-80 Graphics User Interface

Figure 6.2 shows the relationship between the mouse and the display of graphics interface items on the screen. A cursor, represented as an arrow pointing to the northwest, is linked to the movement of the mouse. Pressing mouse buttons with different combinations of key presses will cause different operations on the screen. Depending on the host windowing system, button presses will result in different actions. For example, on the Macintosh, the cursor can be used to activate different icons in the window label that close, iconify, or zoom the window. On the PC version, Microsoft Windows provides essentially the same features but with a slightly different look and feel. Each host window has a slightly different way of dealing with the operation of the host window. Inside a Smalltalk-80 window, holding down the left mouse button (the "select" button on a two-button mouse; also the single Macintosh button) and moving the cursor over text will highlight the text (i.e., change the background color). This context sensitivity is a key to the operation of the cursor/mouse combination; that is, the mouse buttons will perform different activities depending on where the cursor is located on the screen. Hence, one requirement for the system is to keep track of where the cursor is, both inside a window

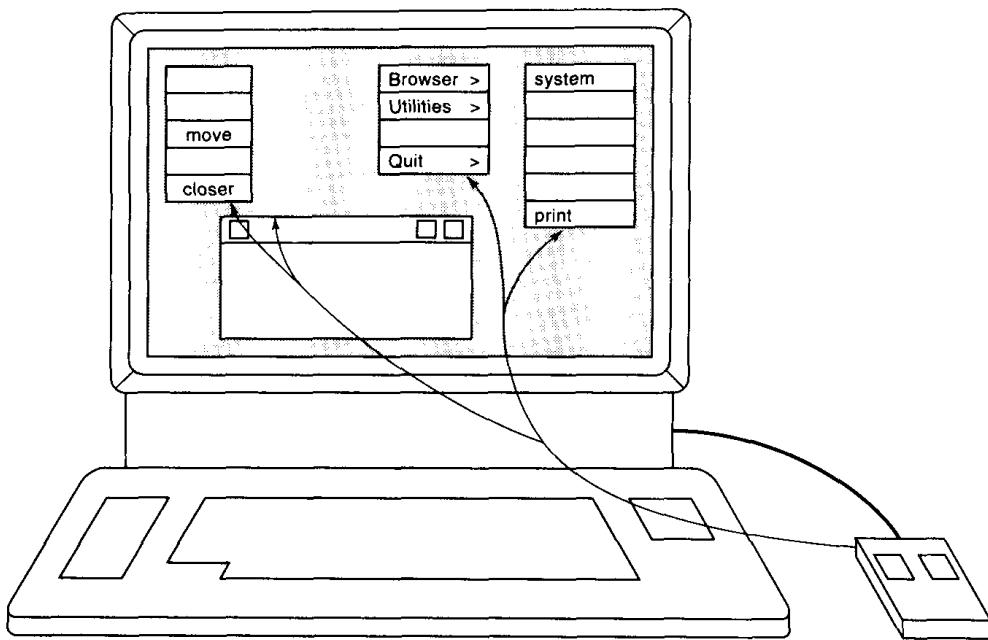


Figure 6.2 Context-Sensitive Menus: Different menus are activated by the mouse depending on the location of the cursor on the screen.

and with respect to icons on the window that control window moving, sizing, and closing.

Mouse pointing devices can have different numbers of buttons: one, two, or three. For three-button mice, the left button (known as the *select* button) is used to make selections from text, point to items to select, or choose windows. The middle button (the *operate* button) pops up specific menus related to the application. The right button (the *view* button) deals with standard operations such as sizing, moving, or closing a window. Two-button mice have *select* and *view* allocated to the left and right buttons; pressing both buttons at once yields the *operate* activity. One-button mice, such as on the Macintosh, require that either a keyboard key be pressed to obtain complete mouse functionality or that the user activate the desired menu by pressing an icon on the screen.

Table 6.2 shows some of the prominent characteristics of the user interface for Smalltalk-80. Several visual cues are provided to assist the user: Different icons for cursors are employed, for example, the pointer icon for pointing to things, the hourglass icon to indicate waiting for something to happen, and text highlighting which assists in editing and executing code. The interface to the system is via the mouse and the keyboard. Probably well over 90 percent of user interaction with the system can be via the mouse except for

Table 6.2 Characteristics of the ST-80 User Interface

Characteristic	Comment
Visual Cues	
Cursors	Various cursor icons are used
Text highlighting	Text can be highlighted for changing, deleting, moving, executing
User Interface	
Mouse	Multibuttoned/single-button mouse
Keyboard	Limited keyboard interaction needed and permitted
Windows	
Rectangular screen areas	Resizeable, moveable, collapsible windows
Views	Views of various types, e.g., lists, images
Text	Text operations with different styles, fonts
Menus	
Pop-up	Single lists to select from
Hierarchical	Treed lists
Buttons	When cursor is in button rectangle, pressing a mouse button causes an action
Graphics	Standard drawing methods: lines, polylines, wedges, rectangles, etc.

entering code as text via the keyboard. Windows are the basic medium for presenting different views. Windows can contain a view with a single thing, such as text or a list from which a selection can be made, but more commonly they contain multiple tiled views which represent all the elements needed for a complete application. The views that are displayed in windows are created through a process of specifying each view size, the view characteristics, and where it should be placed in the window. Standard menus consist of several types, for example, pop-up menus from which items can be selected and hierarchical menus that pop up and present multiple items in the form of a hierarchy. Buttons represented as checkboxes, radio buttons, rectangles, or words are mouse sensitive, and when pressed (i.e., moving the mouse cursor to the item and pressing the button on the mouse) can be used to initiate an action.

In the lower right corner of Figure 6.1 is a list called the Launcher which provides top-level control to the user of ST-80. In this example, all the tools on the desktop can be accessed by simply moving the mouse cursor to a selection and clicking the *select* mouse button on the line with the item name and the arrowhead. For each of the categories shown, another list of items

Table 6.3 Partial Class Hierarchy of Smalltalk-80 (Release 4)

Object	Magnitude
Behavior	Arithmetic Value
ClassDescription	Number
Class	Fraction
Metaclass	Integer
BlockClosure	
Boolean	
False	Character
True	Date
Collection	Time
Bag	Model
MappedCollection	Browser
Palette	Debugger
ColorPalette	PopUpMenu
CoveragePalette	StringHolder
SequenceableCollection	ProcessorScheduler
ArrayedCollection	Rectangle
Array	SharedQueue
CharacterArray	PeakableStream
String	Random
Text	VisualComponent
Interval	Icon
LinkedList	Image
Semaphore	Depth1Image
OrderedCollection	Depth2Image
Set	Depth4Image
Dictionary	Depth8Image
Controller	Depth16Image
ControllerWithMenu	Depth24Image
ListController	VisualPart
SectionInListController	CompositePart
ParagraphEditor	DependentCompositePart
TextController	BrowserView
CodeController	DependentPart
StandardSystemController	View
Delay	BooleanWidgetView
Filename	LabeledBooleanView
DosFilename	FractionalWidgetView
MacFilename	LauncherView
UnixFilename	NotifierView
InputSensor	ScrollingView
WindowSensor	ScrollingLinesView
InputState	ListView
LookPreferences	SelectionInListView
	EdgeWidget
	MenuBar
	Scroller
	Wrapper

appears from which the desired tool can be selected. Lists of selections are easily tailored to add new selections that can be used to launch (i.e., “run”) user applications.

6.3 The Class Library

One of the most important aspects of ST-80 is the existence of a class library consisting of more than 330 classes. Table 6.3 displays an abbreviated ST-80 class hierarchy from Smalltalk-80, Release 4. The classes shown are not the complete class library; only some of the more important classes are displayed here. This class hierarchy shows many of the classes that are discussed in

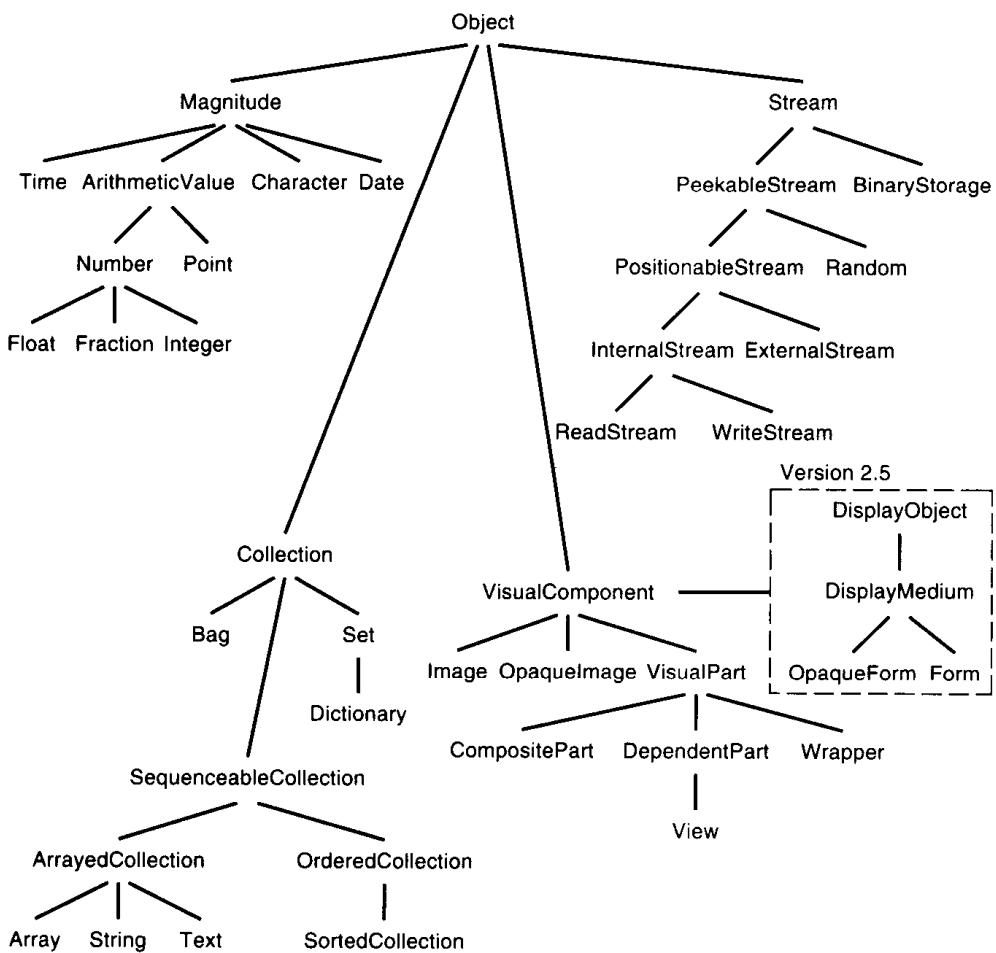


Figure 6.3 A Partial Class Hierarchy of the ST-80 System.

the remainder of this text. The reader is referred to the ST-80 system itself to browse the classes and read the accompanying comments. To secure an idea of the evolution of ST-80 classes, the book by Goldberg and Robson (1989) and the two-volume text by Pugh and LaLonde (1990) are recommended for further reading. The presentation methodology in this table is that successive indentations from the left side of the page indicate subclasses; for example, **Character** is a subclass of **Magnitude**, which is a subclass of **Object**. Figure 6.3 shows a partial presentation of a portion of an abbreviated ST-80 class hierarchy in a treelike form. In Release 4 of ST-80, the graphics presentation classes have changed; Figure 6.3 indicates where there have been significant class changes by showing a dotted box around classes that were present in ST-80, Version 2.5 and earlier. Note that for backward compatibility, the **DisplayObject** and subclasses from earlier (i.e., Version 2.5) releases were retained in initial releases of Objectworks \ Smalltalk, Release 4. The characteristics of classes used for display in Release 4 of ST-80 will be described in some detail in Chapter 7.

Magnitude, **Collection**, **Stream**, and **View** classes rank among the most used in the ST-80 environment, particularly the various subclasses of these abstract objects. Probably the best way for the novice ST-80 user to become acquainted with these classes is to use the browser to examine the class hierarchy and the methods that are in each class. The methodology for using the browser is presented next.

6.4 Standard ST-80 Tools

The Browser

The most important tool in the ST-80 environment is the system browser, which permits browsing the classes and methods in the class hierarchy. *Browsing*, in general, means to look through casually. In terms of browsing software, the intent of a browser is to permit the user to examine the code in an easy and straightforward manner. For ST-80, the system browser is really much more than a code examiner, providing capabilities for adding new classes and methods, finding classes, displaying hierarchies of classes, and generally organizing the user's work. Figure 6.4 displays a screen dump (i.e., an exact rendition of the way the screen looks) for the system browser. Shown across the top of the browser are four scrollable lists: the leftmost list contains *categories*, which give the user the capability of organizing classes into meaningful categories. Most often, users will use one or more categories for putting together an application. Associated with this view is a pop-up menu that permits adding, renaming, or removing categories. The second list from the left contains the name of the

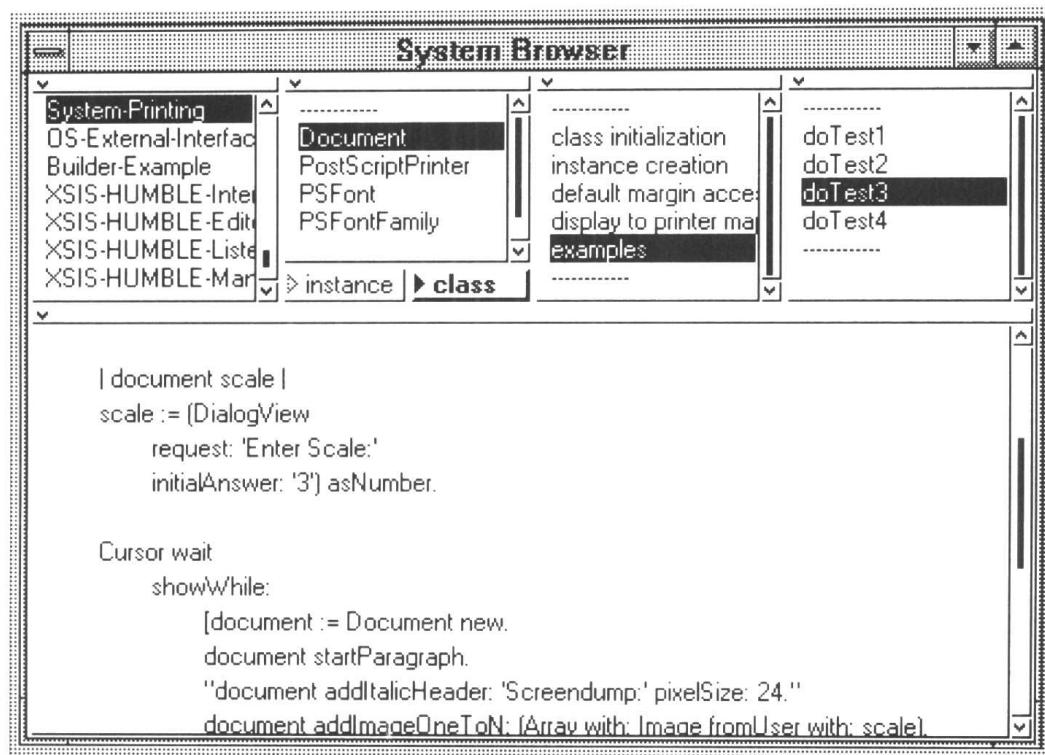


Figure 6.4 The System Browser.

classes that are in the category selected in the leftmost list. Separation of classes by category, in this way, permits easier inspection. The next list to the right is a list of *protocols*, which records logical groupings of names of methods associated with the class selected in the view to the left. Finally, in the rightmost view is found the *methods* list which includes the names of all the methods in the protocol selected. Shown in the large view at the bottom of the browser is the code of any method selected in the methods list. Also, displayed just below the class list are two buttons for selecting instances or classes. If the instance button is selected, then selections made in the protocol and methods lists refer to instance methods; if the class button is selected, then selections refer to class methods. This distinction is important and is frequently missed by novice Smalltalk users. The use of the browser to create an application will be covered in the following discussion.

The Text Editor

Text may be almost intuitively edited using the text editor. The text editor is invoked by simply selecting *file editor* from the system menu. The user

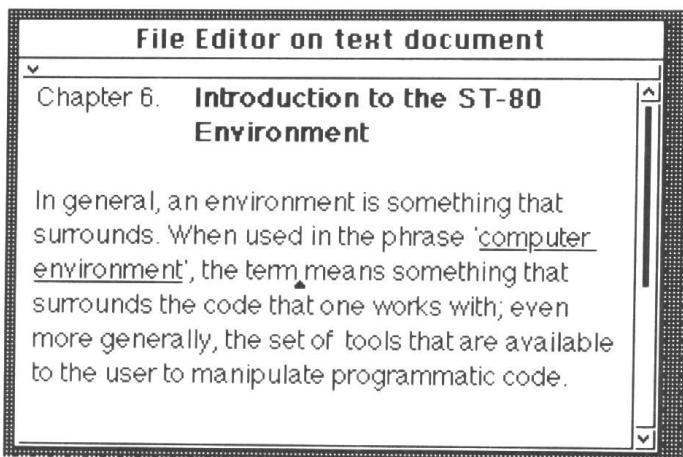


Figure 6.5 The Text Editor.

can size the window, type information into the window, and store the text in a file. Figure 6.5 shows a text editor window that contains different font sizes and styles. The pop-up menu that is invoked by the operate button is shown. Capabilities such as copy, cut and paste, underline, and save are the fundamental features of the editor. Highlighting text and then using these commands are sufficient for most needs, including rearranging words, sentences, and larger bodies of text.

Images

Images are pixel-based rectangular areas that display images on the screen. In ST-80, Release 4 and higher, images support a general protocol for color graphics. The number of bits per pixel, which determines the number of colors that can be displayed, can be 1, 2, 4, 8, 16, or 24. In earlier versions of ST-80, only bitmapped black-and-white images, called forms, were supported. Images from standard drawing packages on most platforms can be directly imported into Smalltalk.

The Form and Bit Editors

A form is a one-level image; that is, a form displays only black-and-white pixels in a rectangular area. Until the release of Release 4 of Objectworks \ Smalltalk by ParcPlace Systems in 1990/91, forms were the predominant medium on which Smalltalk programmers displayed images. A

description of forms and form manipulation capabilities is included here for backward compatibility. In Release 4 and higher, forms and bit editing capabilities have been removed in favor of using images. However, simple methods exist for translation between images and forms.

Figures 6.6 and 6.7 show examples of the form editor and the bite editor used in Version 2.5 and earlier releases of ST-80. The form editor had capabilities for creating simple drawings using the form tools that were iconically arrayed across the bottom of the form editor. The methodology employed was to use the bracket selection button, at the top left, to select a cursor from anywhere on the screen. The technique used was to bracket the area of the screen that would become the cursor with left and right brackets, activated by sequential mouse clicks. To create a small icon for drawing, a dot somewhere on the screen could be bracketed. Drawing was accomplished by using the single-point mode, multiple points, and straight or curved lines. The icon on the lower left button provided access to the bit editor to permit changes to be made in individual bits. The arrows at the right were used for input and output. Note that when the output arrow (lower right) was pressed, the form stored was the cursor selected by the angle bracket selection. This point was difficult for novice users to understand. Some users changed this feature of the form editor to store the entire image that was being edited. Although this change made logical sense, the original method of storing the cursor bracket was reasonable from the standpoint of being able to store forms of different parts of an image. A typical way to work with the form editor was to create several instances of the form editor and to paste information between editors. A favorite technique for annotating a picture was to type text in a workspace, bracket the text with the angle bracket to create a cursor of the text, and deposit the text in the target-image.

The bit editor, as shown in Figure 6.7, provided an expanded view of the bits in a small part of a form. In this example, the bit editor for the standard cursor, control was very simple. The user had only the choice of selecting a black or white pixel switch at the bottom of the view. Once one of these buttons was pressed, the color of each pixel could be changed by moving the cursor to an individual pixel and pressing the select button.

Creating forms by hand is, of course, not the only way to secure bitmapped images in Smalltalk. Perhaps the easiest methodology is to use a scanned image and directly import the image into ST-80. This capability in Release 4 has proved to be very popular and is a significant improvement over the Version 2.5 bitmap editing capabilities.

For Release 4 and higher, the comments made previously about bit editors and forms are obsolete. For these releases, images replace forms and either commercial drawing packages or the internal graphics methods of ST-80 can be used to create graphics. However, in 1991, no extensive color graphics editors existed for Objectworks \ Smalltalk.

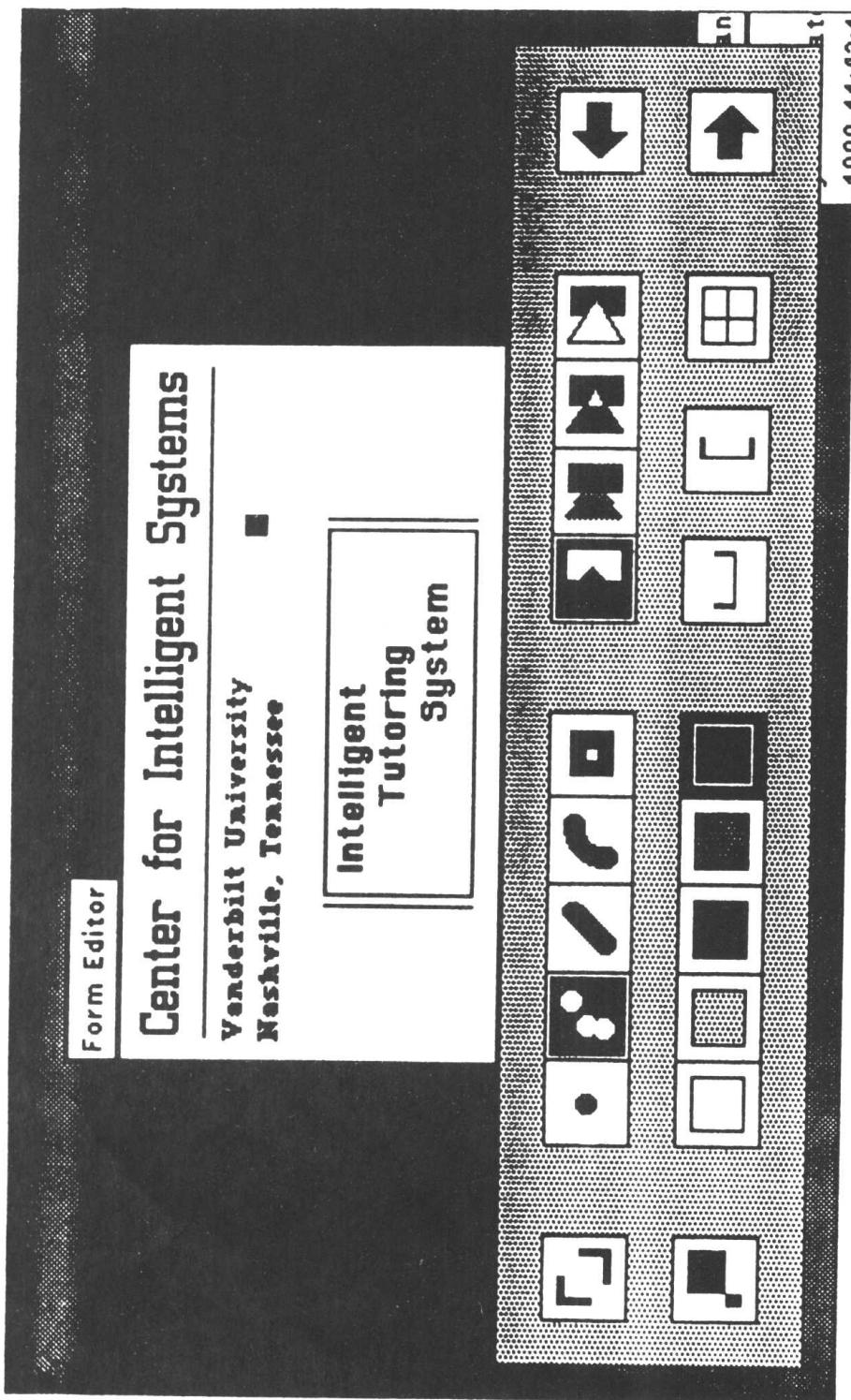


Figure 6.6 The Form Editor.

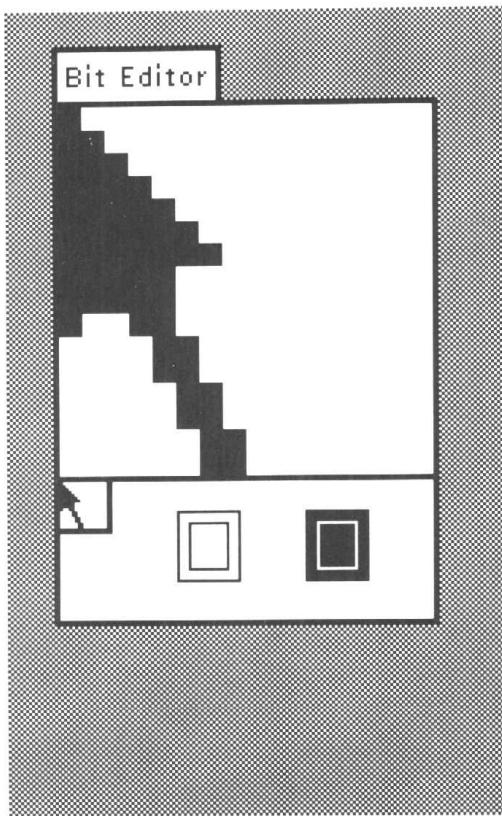


Figure 6.7 The Bit Editor.

Workspaces

A workspace is a window in which one can type in code and try out code prior to committing the code to a class. There are two workspace selections in the Launcher: the system workspace and general-purpose workspaces. The system workspace is a workspace containing many expressions involving objects and messages that are used to operate specific things in the ST-80 environment. Normally, users employ the system workspace for remembering how to do something associated with the ST-80 system. Figure 6.8 shows a small portion of the system workspace. Throughout the workspace there are example messages to classes that help the user remember how to use many of the common classes associated with operating the system. It is useful to add to the system workspace common things that you do, or need to remember. If building a large application, it is also useful to create special workspaces that can be used to remind you at a later time about some problem that has been left uncompleted or simply to help you or others understand the application. The



Figure 6.8 The System Workspace.

point of this discussion is that workspaces, in general, serve (1) as a kind of temporary work area or scratchpad and (2) as a cognitive aid in remembering what you are doing when programming. The system workspace serves the same purpose, but for remembering things associated with the system. The capability for easily accessing examples and making notes about applications created is particularly useful in a personal computing environment in which programming is done in sporadic sessions sometimes separated by long time intervals.

The Debugger

The ST-80 debugger is activated when an error occurs in executing code or when the user types control-C on the keyboard. The most usual mistake is sending a message to an object that is not one of the methods that the object understands. When such an error occurs, a window indicating that an error happened pops up; the user can then either "proceed" or "debug" (via a pop-up menu listing these two options). Selecting "debug" produces a window similar to the one shown in Figure 6.9. In the top part of the window is the message stack in a scrollable list, filled with successively earlier message calls. Selecting an item presents the code surrounding that message call in the text view just beneath the top view. This code is cast in the context of the running program with all variables set to the values in effect at the time of the message send. Shown in the two small lists at the bottom of the view are the variables and their values to the bottom right and the assignments of the current objects to the left. The outstanding capability of the debugging window is that the user can execute code with variables intact to trace very rapidly any problem that

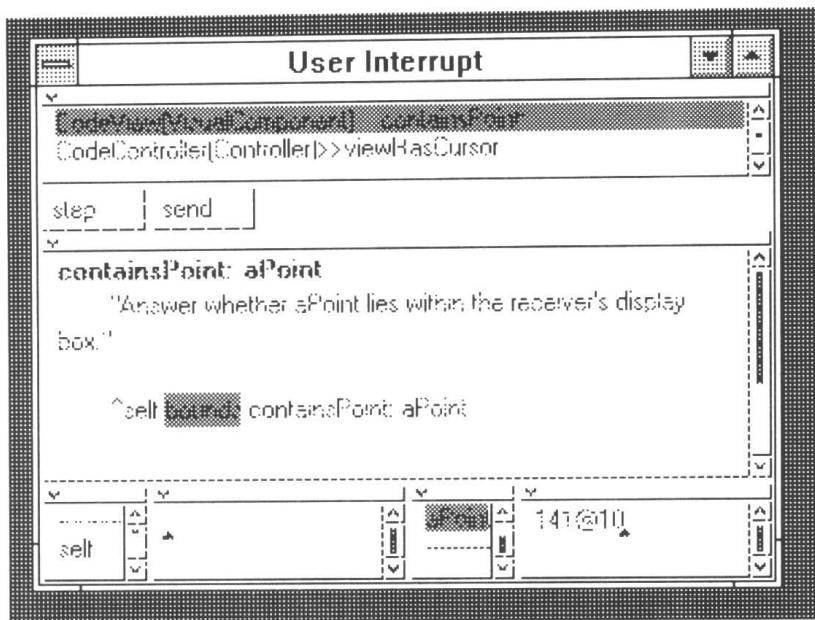


Figure 6.9 A Debugger Window.

occurs. Furthermore, corrections to the code can be made on the spot and the program restarted. These capabilities lead to very rapid program debugging.

The File List

Figure 6.10 shows the file list capabilities of the ST-80 desktop. The user enters the file directory of interest in the top view and selects “accept” from the attendant pop-up menu. Files in the directory appear in the file list in the middle view. Information in the bottom view about the file size and modification date appear. Operating the pop-up menu associated with the middle view and selecting “get contents” permits viewing of file contents in the bottom view; selecting “filein” files the Smalltalk code in the selected file into the image. Once the code is filed in, the user can examine the code by opening a browser or should select *update* in the category list of the browser to be able to browse newly filed-in classes in an already open browser.

Other Facilities

There are a number of other facilities on the ST-80 desktop not explicitly described in the preceding discussion. For example, the projects

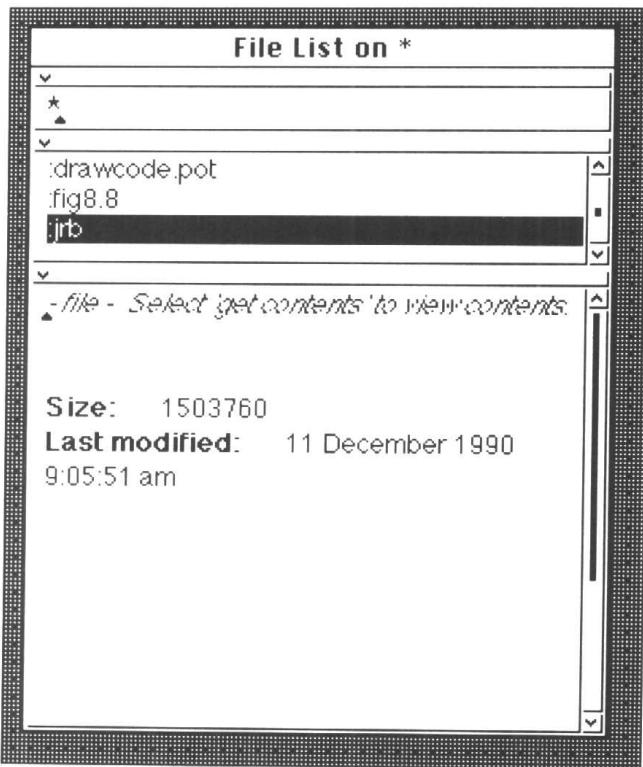


Figure 6.10 The File List.

facility allows a user to “clear his desktop” for another project, while retaining the desktop contents for a current project. In this way, the user can move between various different projects that are each tailored to a specific environment.

6.5 Creating Applications Using the Smalltalk Environment

The first step in creating a new application in the ST-80 environment is to create a system browser from the Launcher. In the browser, in order to add a new class, one selects a category in the category list of the browser to put the class in; or, alternatively, one creates a new category using that selection in the pop-up menu associated with the category list. At this point, the window will appear similar to the window shown in Figure 6.11. Using the text facility, the user defines the superclass name, class name, instance, and class variables, and selects “accept,” according to the format displayed in the bottom text view of

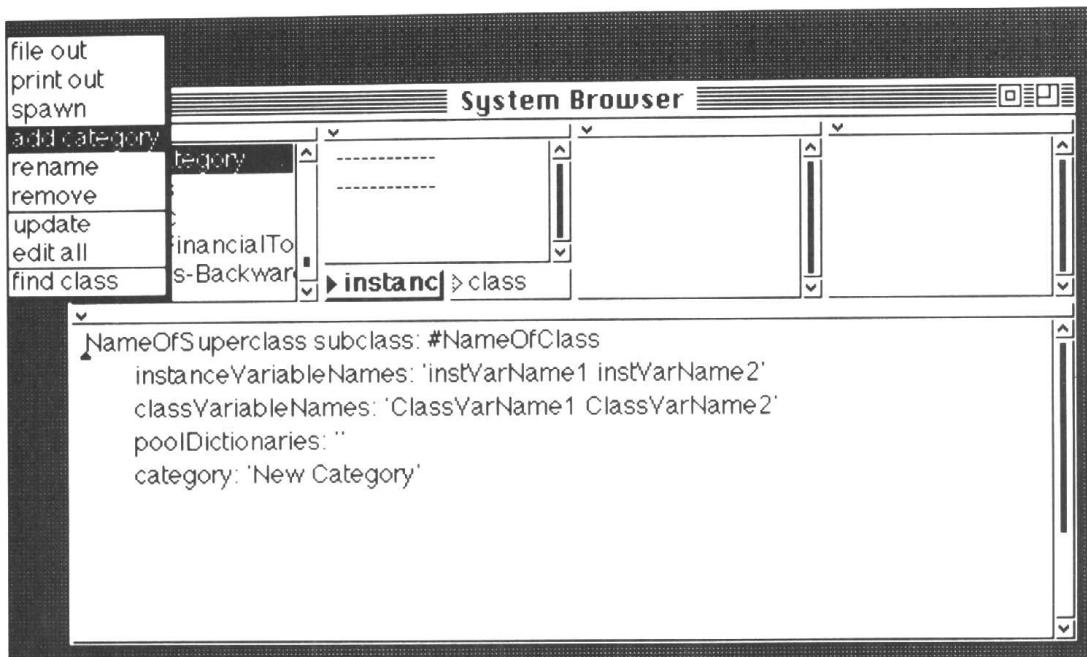


Figure 6.11 Creating a New Class.

the browser. In this format, placeholders are provided which should be replaced with the names of the class and variables to be created. For example, in Figure 6.11 “NameOfClass” should be replaced with the name of the class being created and “NameOfSuperclass” should be replaced with the name of the superclass of the class being created. Similarly, class and instance variable names should replace the placeholders “ClassVarName1 ClassVarName2” and “instVarName1 instVarName2.” If the class being created is not identified with any particular superclass, for example, **Dictionary**, **OrderedCollection**, and so fourth, then the class to use for “NameOfSuperclass” is **Object**. As an example, a new class named **Car** might be created using this technique; after editing, the lines shown in the text window of Figure 6.11 would look like:

```

Object subclass: Car
instanceVariableNames: 'wheels engine type horsepower style model'
classVariableNames: ''
poolDictionaries: ''
category: 'Moving Vehicle Category'

```

In this example, there are no class variables and no pooled dictionaries (dictionaries accessed by different objects) and the class has been entered in the category ‘Moving Vehicle Category.’ The final step in creating the new class description is to select “accept” from the pop-up list in the text window.

The next step in the class creation process is to add methods to the class being created. The first step in this process is to add the names of protocols used in the protocol list. For each protocol, there can be any number of methods associated with the protocol, the names of which are displayed in the rightmost list view in the system browser window. To create a method, the technique is, after the protocol is named and selected, to type the method code into the text view with the method name on the first line. After completing entry of the method, selecting “accept” in the text view will add the method to the class and the name of the method added will be displayed in the browser window. After “accepting” code, the syntax of the code is scrutinized by the compiler and the user notified if syntax problems are detected. Note carefully, that instance and class methods are entered separately according to the setting of the class-instance switches set in the browser.

Running and Debugging

Once code has been entered, either partially or completely, testing and debugging can begin. Normally, the user places in the *open* method in the application view a message on how to run the application. For example, a class called **VCircuitSimView** could have a class message called *open* which might appear as

```
open
    "VCircuitSimView open"
    ... the method code for opening the view ...
```

The quoted string is an executable comment with which the user can run the application. The way to use an executable comment of this type is to move the cursor to the beginning of the quoted character string and double click. This action highlights the text; popping up the action menu and selecting “doit” then sends the message *open* to the class **VCircuitSimView**. The application runs or else fails. In the latter case, a runtime failure will result in a debugger window popping up with a message across the top of the window and a text view containing the text of the code where execution ceased. Selection of two alternatives is possible at this junction: “proceed” or “debug.” Selecting the latter produces a window similar to the one shown in Figure 6.12, containing the error message at the top, the stack of message sends next, the text of a selected message send, and finally two lists of variables and classes present, if any. In this example, an error was introduced in the “open” code to illustrate one of the most common errors. The text string just below the debugger window in the text window of the file list *DrawingView new init* was executed after *open* was sent to **DrawingView**. The debugger appeared and indicated that the message was not

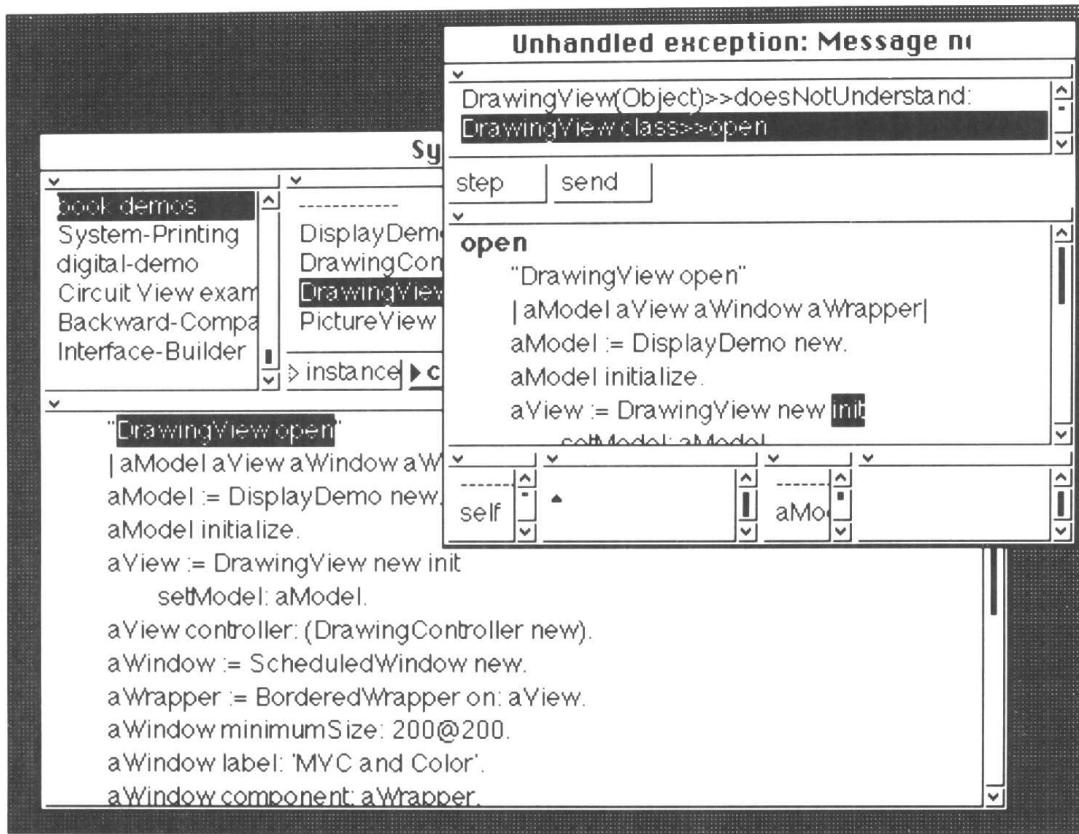


Figure 6.12 Example of Fixing a Bug in a Debugger Window.

understood. In this case, the error the user made was to use *init* rather than *initialize* as the message name for initialization; because the instance of class **DrawingView** that was created by sending *new* to this class did not have a method called *init*, the message send *init* failed. Probably this type of error is the most common, that is, trying to send a message to an object that the object does not understand. A second very common error is “subscript out of bounds” which is often found when the user attempts to access an element of a collection that does not exist.

System State /Changes

The ST-80 environment provides a single-user personal computing environment. After using the environment (i.e., editing, adding classes, running applications, etc.), the user can select “save” from the system menu to save the current state of work as an image file. Novices should note that the ST80.im file

that is first opened should be saved with a personalized name (e.g., "myImage"). Upon reentering the environment using this name, the user reenters with an exact copy of the system that was saved. One can leave notes in workspaces to remind oneself about what was being done, as needed.

There exists an audit trail facility, known as "changes." The ST-80 system keeps track of each change made to classes, editing, and so on in a file with the same name as the image file with a .cha extension instead of an .im extension. Naturally, because all changes are saved, this profile becomes quite large after some time and should be compressed periodically to save space (*Smalltalk condenseChanges*). There is a change management browser that can be used to reconstruct changes to a system, if a system crash occurs, for example. An excellent discussion of these facilities is found in Goldberg (1984).

Workspace Use

Workspaces are good for testing code. Quite often as one learns Smalltalk, code can be entered in a workspace and tested prior to entering a method in the system browser. It is sometimes convenient to name global variables in the workspace so that code can be executed and variables inspected while stepping through the code.

Transferring ACOM Cards to the Browser

In Chapter 3, ACOM cards were created for the digital circuit simulation example. In subsequent chapters, this example will be completed

Table 6.4 Steps in Transferring ACOM Cards to the System Browser

Step	Example or Comment
1. Select a name for the category	DigitalCircuitExample
2. Create classes:	Object subclass: AndGate
AndGate	
OrGate	
NotGate	
Wire	
3. Include the instance variables 'input output constraint' as shown on the cards	
4. Name the protocols	e.g., initialize, access
5. Add the methods to each protocol	e.g.; for wire: inputConnection: outputConnection: set:, etc.

along with all the methods. Given the information in Chapter 3, however, the information on the cards described can now be easily transcribed to the system browser. It is left to the reader to follow the steps given in Table 6.4.

Once the ACOM cards representing the model are transferred to the browser, the model-view-controller code for visually representing the model can be added to provide a visual interface to the code. The MVC methodology will be discussed in detail in Chapter 8 and a continuation of the digital circuit example will be given in Chapters 10 and 13.

References

- Goldberg, Adele (1984). *Smalltalk-80, The Interactive Programming Environment*. Addison-Wesley, Reading, MA.
- Goldberg, Adele, and David Robson (1989). *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA.
- Pugh, John R. and W. R. LaLonde (1990). *Inside Smalltalk*, Volumes 1 and 2. Prentice-Hall, Englewood Cliffs, NJ.
- Ritchie, D. M. (1979). *The Evolution of the Unix Time-Sharing System, Language Design and Programming Methodology. Lecture Notes in Computer Science*, Vol. 79, pp. 25–35. Springer-Verlag, Berlin.

Exercises

- 6.1 Conduct experiments with each of the major components of the ST-80 system to make sure you understand how they work.
- a. File list: Try examining code and filing in code. Specifically, make sure you can examine the contents of a file and that you can file Smalltalk code into the system.
 - b. System browser: Spend a lot of time browsing and examining classes. Try out the different options in the class menu. Try “inspect,” “explain,” “spawn hierarchy,” and so forth. Describe the results.
 - c. Try adding additional categories to the Launcher. Explore **LauncherView** by using the system browser and see if you can add additional functionality to the Launcher.
- 6.2 Investigate the debugger. Run the code (i.e., highlight the code in a workspace and select “doit” with the operate button of the mouse) just below this problem and explain how the debugger and the system browser can be used to find and correct the error. While you are using

the debugger, investigate and explain what each of the views in the debugger are used for.

|j k|
j := 0.

Transcript show: (j + k) printString; cr.

(*Hint:* Try assigning a value to k and repeating the experiment.)

- 6.3 Actually type the ACOM cards for Chapter 3 into the system browser. Make sure that you understand how to organize categories, classes, protocols, and methods.
- 6.4 Try out the editing facilities in the “file editor.” Type a paragraph, store the file, and retrieve it.

User Interface Design

This chapter discusses the general aspects of designing user interfaces. Moving from a broad viewpoint about the history and concepts of user interfaces to specific information about the implementation of displaying interface objects in ST-80, the information presented lays the groundwork for designing applications using the model-view-controller methods given in Chapter 8. Basic information relevant to ST-80, Release 4, display and view objects is presented. Also, Section 7.4 is devoted to compatibility between Release 4 and the previous version of ST-80 (Version 2.5). This information is presented for readers with prior knowledge of the display objects in the previous releases.

7.1 The User Interface

User Interfaces

The term *user interface* can mean a variety of things in different contexts. For the purposes of this text, discussion will be limited to explication of interfaces between humans and computers. Perhaps some of the discussion will be relevant to creating other types of user interfaces as well, such as interfaces to instruments, cars, and other human-engineered artifacts.

For communicating with computers, two important factors are (1) communicating information to the computer from the user and (2) communicating information from the computer to the user. Consider Figure 7.1 which illustrates a conceptual distance between the user and the computer. Shown as an arrow from the user to the screen is the “command interpretation distance” and shown as an arrow from the screen to the user is the “screen interpretation

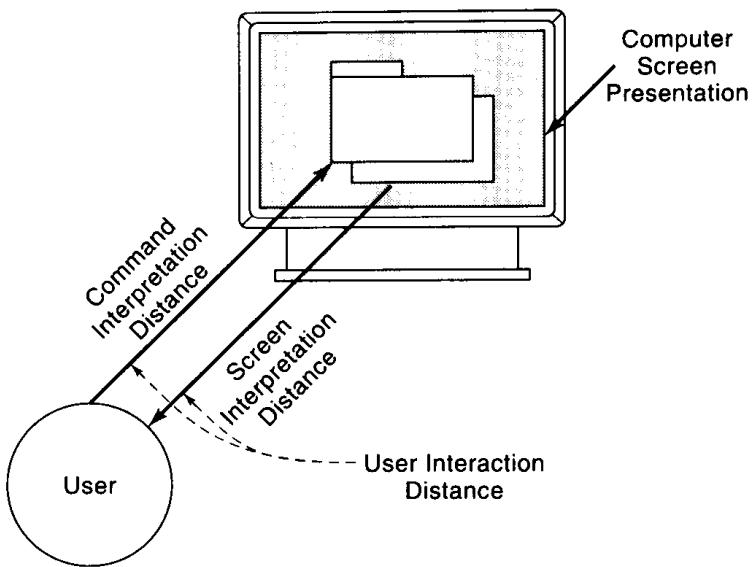


Figure 7.1 The Conceptual Distances between User and Computer.

distance.” These two distances conceptually illustrate the gap between the user and the interface. Ideally, if a user interface were wholly intuitive, the gap would be zero; that is, the interface could understand whatever the user “said” and the user could understand whatever the interface returned. Considerable effort has been made over the last two decades to close the conceptual gap between humans and computers. Since the early days of computing in which switches were directly manipulated on the console of a computer to input raw information, significant progress has been made to improve human-computer communication. The range of interfaces from command line interfaces to direct-manipulation interfaces will be examined next.

Primitive computers were programmed by setting switches on the front panel of the computer to load individual instructions into the machine. Then, to run a program, the operator would press a switch, perhaps labeled “run,” which would cause the program to be executed. This manual program entry and operation was, in fact, a kind of user interface, albeit very simple, primitive, and tedious to use. Subsequently, simple mnemonic interpreters were developed that could interpret simple commands, for example, “L” for load, “R” for run, and so on, which a user could enter on a keyboard or by feeding in paper tape. This progression toward more sophistication resulted in simple operating systems with direct and easy-to-understand commands that could be typed on a keyboard and observed when echoed on a screen or printer. Command line interfaces became the most popular interface for many years, consisting of sequences of typed characters that permitted the user to control the operating system by issuing commands such as “dir” or “ls.” The DOS and

Table 7.1 Examples of Commands Use in Command Line Interfaces

Command Line Example	Description
> DIR /P/W\BIN	A DOS command for listing a directory (/P = pause, /W = wide)
> DIR SORT > PRN	Sorts a DOS directory and prints out
> MODE COM1: 96,E,8,1,P	Sets the mode of a serial port (DOS)
%ls -al	Unix directory command
%stty cstopb ignbrk ixon evenp > /dev/tty02	Sets the terminal characteristics of tty02 to two stop bits, ignore break, input xon, and even parity

UNIX operating systems both employ command line interfaces. Table 7.1 shows several examples of command line interaction, to illustrate simple and complex styles of interaction. The first character in each line of the example is the prompt character issued by the operating system.

As shown in Table 7.1, typical commands in command line interfaces are only somewhat intuitive. For example, typing the DOS “dir” is a somewhat intuitive command for making a listing of a directory. The attendant “switches” “/P and /W” are not intuitive and their meanings must be obtained from a manual. UNIX commands are generally even more obscure, for example, the command “ls” for making a listing of a directory. Note that these examples from two common operating systems use forward slashes (/) and backslashes (\) in directory paths to produce the same meaning, a convention certain to confuse a novice. If one understands and remembers UNIX and DOS commands, the commands shown most likely have meaning; however, a novice is absolutely lost, having no notion about what commands to issue when confronted with the system prompt. Indeed, the mnemonics have meaning only when used and associated with the actions that they perform. Yet, once learned, these mnemonics form an easy and direct way to communicate with an operating system.

The difficulty of learning to use command line interfaces initiated the transition to modern user interfaces that are heavily graphics and pointing-device oriented. Graphics and direct-manipulation interfaces will be examined next.

Graphics and Direct-Manipulation Interfaces

Graphics User Interface

A graphics user interface (GUI) is most popularly thought to be the type of interface that one sees on a Macintosh computer or on a PC using

Microsoft Windows; that is, a windowed screen environment with menus, icons, and so on that can be accessed using a pointing device. If the term is taken literally, any interface that employs graphics could be associated with this term. For example, a computer system that simply presents graphical information would qualify under this literal interpretation. In fact, there is a significant range of capabilities that is contained in most GUIs. Typical GUIs use windows; for example, X-Windows (Schiefler, Gettys, and Newman, 1989) is an interface that primarily supports one style of windows. However, a windowing system may support only window activities and not the remaining characteristics of GUIs such as mouse support, menus, and so on.

A graphics user interface should have (1) some type of windowing mechanism, (2) an imaging model, that is, a definition of how graphics are displayed on the screen, and (3) an application builder interface (e.g., routines that can be used by the user to build the interface). Common examples of GUIs include names like NewWave from Hewlett-Packard, Microsoft Windows, Open Desktop, NextStep, OS/2 PM, and the Mac GUI. These interfaces all have different windowing systems and imaging models and run with different operating systems. The most familiar imaging model is Postscript (Holzgang, 1988) which can be used for printing or, in the screen version, for display. Perhaps some particular model and windowing system will become dominant in the future. All GUI systems look similar to the casual user. All have windows and permit resizing, moving, and collapsing. Most have pop-up and/or pull-down menus and buttons, provide text windows with editing features, and allow control via buttons. The use of color varies; some systems provide color, whereas others do not.

Direct-Manipulation Interfaces

The characteristics of GUIs become commingled with the definition of direct-manipulation interfaces. Shneiderman (1982) coined the term direct-manipulation interface and defined a DMI to have the following properties:

1. A DMI provides a continuous representation of objects.
2. Physical actions, rather than command line interfaces, are used to carry out operations.
3. Actions can be rapidly reversed with visual notification of the reversal.

This definition is more general than the rather specific attributes of GUIs listed previously, yet fits completely. A continuous representation of objects is manifested in a GUI as information in windows or icons which, when accessed or

activated, perform an action. The notion of physical actions rather than command line syntax is obviously implemented in the point-and-click metaphor commonly found in GUIs. Finally, the reversal of actions and visual notification are easily linked in the typical GUI format.

Hence, what is the difference between GUIs and DMIs? Really none, if the GUIs employed have the capability to conform to the Shneiderman definition. To make clear the intentions in this text, the term graphics direct-manipulation interface (GDMI) is used to reflect the need to provide DMI characteristics in the graphics interface.

Both hardware and software support are needed to implement graphics direct-manipulation interfaces. Each category will be examined in the following discussion.

Hardware Support for User Interfaces

Various types of pointing devices have been used to permit the user to move a cursor on a screen. Examples include trackballs, joysticks, touch pads, control panels, tablets, sensing of a pointing finger, light pen, and even sensors on a gloved hand. The intention is the same for all cases: to provide the user a natural and simple way to interface with the computer. How can one choose the best way? Like evolution, perhaps natural selection is the best. Certainly during recent decades each of these types of pointing devices has been promoted by various advocates. Currently the mouse appears to be the most popular. Let's see why.

Consider the use of a pointing finger or light pen as the pointing device. Sensed by devices placed around the periphery of the screen or from sensing the light on the end of a pen, the location on the screen at which the finger or light pen was pointed could be sensed. However, this methodology suffered from having the user actually point at the screen—a truly tiresome task. Hence, this methodology has virtually disappeared.

The use of joysticks has also become less prevalent, perhaps because of identification with video games, but more likely because of the cognitive dissonance of moving a cursor on the screen with a device that feels like a control for an airplane. The effect is that the use of the device does not match the task. If the task was simulating an airplane, then the joystick would be the most acceptable pointing device. The use of control panels separate from the keyboard, for example, a panel with left, right, up, down buttons has also largely disappeared due to the poor mating with the domain. Indeed, in current graphics user interfaces, even the standard cursor keys on the computer keyboard play a minor role.

Thus, almost by default, the most suitable device emerges—the mouse, and its cousin the trackball, which is essentially an upside-down mouse. The

trackball suffers from the problem that it is somewhat more difficult to use than a mouse. However, due to the increased popularity of portable computers and the requirement of having a flat surface on which to move a mouse, the trackball is becoming more popular. The primacy of the mouse/trackball probably came about because these types of pointing devices directly relate to the human pointing experience, are simple to operate, and are comfortable to use.

There still may be other modalities of interaction that may be useful which are yet to be invented. For example, sensors on gloves may provide a technique for pointing that would give multidimensional capabilities. This technique can be combined with binocular visual displays that would allow a user to become part of a virtual scene. There is currently little consensus on the utility of this display modality.

Software Support for User Interfaces

As discussed previously, the mouse/window/icon user interface is the most popular style of interface in use. Most of the qualities of the interface (apart from the mouse) reside in the software created to support the interface. Most window-based systems have a similar “look and feel”; however, there are other types of interfaces for special applications that are distinctly different. For example, consider a spreadsheet application in which rows and columns are presented to the user with constraints (mathematical relationships) at cell used to represent interactions between numbers. This interface is very compelling for applications involving accounting-related applications. An important point is that the user interface design mimics, in a way that the user understands, paper-and-pencil spreadsheets used prior to the introduction of computer spreadsheets. In the same way, desktop publishing interfaces mimic the same type of environments that were found in traditional publishing and layout environments. Hence, software design of user interfaces has been most successful in transferring metaphors from the real world to the computer interface for special applications. It is straightforward to mimic traditional application methods (e.g., spreadsheets, publishing) of doing things in designing a computer-based system to do a similar task; however, it is more difficult to create a general-purpose user interface for assisting programming tasks. The window/mouse/icon methodology comes currently closest to a universal interface that can assist programming tasks. An important feature of software for building user interfaces is that it should be open, inspectable, and easy to tailor to different application environments. Criticisms of user interface building tools include such factors as difficulty of use, difficulty in understanding, and nonportability between platforms, all problems that would argue against use of an interface

building software system. It is the contention of this text that the ST-80 user environment provides a rich user interface building tool that can be modified for many different applications and can create most user interface characteristics desired. The model-view-controller paradigm provides a rich mechanism for coordinating the application data, controlling the pointing device, and organizing what is seen on the screen. Use of this model will be described in detail in Chapter 8.

7.2 Characteristics of Graphics Direct-Manipulation Interfaces

The Graphics DMI

In this section, a framework for understanding issues surrounding design of graphics direct-manipulation interfaces will be built. The term GDMI is used in preference to either GUI or DMI because either by itself lacks the composite description; that is, a *graphics* direct-manipulation interface is the desired interface. Various researchers have studied direct-manipulation interfaces [see, e.g., Hutchins and Norman (1986) and Shneiderman (1982, 1983)]. Some of the ideas and terminology contained in these research efforts are used for the describing characteristics of GDMS presented next.

Closing the Gap between User and Machine

A fundamental principle in creating a user interface is to reduce the gap between the user and the machine. A desire is to make the operation of a computer system almost completely intuitive for the novice and, for the expert, to permit the system to be operated in efficient instinctive ways that do not hinder operation. An effective user interface should lower the cognitive overhead for naive users to the point that concrete actions result in concrete information being returned. Designing for both the novice and the expert is not trivial. Consider that a typical novice-oriented system must have information completely spelled out. Actions selected from button presses or multiple choices are needed to guide the novice from one step to another. An example would be the overly complex sequencing of steps in some word processors which require multiple sequences of operations to achieve a single conceptual step such as moving a paragraph from one location to another. Making the steps needed to perform an operation explicit is very useful for novices but is exceptionally annoying for the expert who desperately wants shortcuts to achieve actions

more efficiently. Ideally, an excellent user interface should adapt itself to the needs of the user so that such problems do not occur.

Two areas are considered next: the execution of commands and the interpretation of results. Commands are issued by the user and information distilled in some way is returned to the user. The combination of the execution gap and the interpretation gap represents the total distance between the user and computer. Closing the gap simply means making both the input of information simple and intuitive and the resulting information fed back to the user equally understandable. The gap is closed when all users, from novice to expert, feel that the interface is unobtrusive.

Execution Gap

Line-oriented commands are not particularly intuitive. To execute a command to make a directory listing, one might type to a command line interpreter such commands as "dir" or "ls." How can a novice learn about such commands? There are two ways: by reading a manual or through assistance menus that provide an interface alias for the actual commands. The popularity of simple command line interfaces (e.g., DOS) among casual users of computer systems and the popularity of complex command line interfaces (e.g., UNIX) among experts points toward the following conclusions. Novices need simplicity and concreteness. Experts need flexibility and conciseness. These conclusions seemingly promote building different-style interfaces. The GDMI has the potential for satisfying these different needs. A visual interface can close the execution gap significantly by providing a concrete representation to the users of what to do in the form of graphic representations. Pressing a button marked "list directory" is significantly easier than reading a manual to determine that one should type "ls" at the command line prompt.

The conjecture that GDIMIs narrow the execution gap would be more plausible if GDIMIs were intelligent enough to adapt to the needs of the user automatically. Without careful design or adaptability, operation of GDIMIs can become tedious for the expert.

Evaluation Gap

This gap is the information presentation portion of the overall gap between the user and the computer system. A simple example of this gap is the contrast between the presentation of a table of numbers and a graph that presents the numbers and their graphic interpretation. The results of operations must be exposed and observable. Graphic organization of data and knowledge

are essential. For example, a long list of rules (e.g., if-then rules) is hard for a user to understand. When presented in the form of a tree hierarchy, however, the logic in rules becomes more understandable. The same notion is true of many interfaces. For example, the desktop publishing interface and many word processing interfaces present text as it will be seen on the final printed page (WYSIWYG—"What you see is what you get"), a powerful and useful technique that has made packages that use such interfaces very popular. Closing the evaluation gap requires careful thought about what the user should see, how the information should be packaged, and what flexible features should be available to permit inspectability without confusion. These are tough issues that remain largely unresolved.

Concrete Representations

A technique for closing the execution/evaluation gap may be to move away from abstraction to concreteness in interface design. It is certain that command line interfaces are abstract. In other words, typing a specific command does not represent a concrete physical action; it is merely a sequence of characters that represents an abstraction for a physical operation. Pushing buttons with labels is also abstract, however. Yet, using labeled buttons on the screen has the advantage that potential actions are made explicit and thus concrete in the mind of the user. More concrete representations consist of icons that represent what one is intending to do in a concrete way. For example, grabbing a picture of a file and moving it to a picture of a garbage can might cause a user to think that the file was being discarded. There could be many kinds of graphics operations defined in this literal style. However, one must be careful to ensure that such literal graphics operations make sense so that in the context of the past experience of users, operations will make sense. For creating representations of the real world, iconified graphic representations make good sense; one can create, for example, simulations of factories, traffic flow, fluid flow, or electrical systems that have a high degree of conceptual correspondence to the real world. It is not clear if such clarity can be achieved for GDMIs for programming environments, however. The characteristics of GDMIs for programming environments have not yet been defined. Certain features are plausible candidates such as visually oriented debuggers, browsers, file inspectors, and editors. The interface capabilities of ST-80 point the way toward design of useful interface characteristics that can facilitate graphics system building and programming capabilities.

Figure 7.2 shows some of the common display objects used in the ST-80 interface. Shown in the figure are tiled and overlapping windows, icons, menus, and question boxes. Examine the icon that appears as a door; the

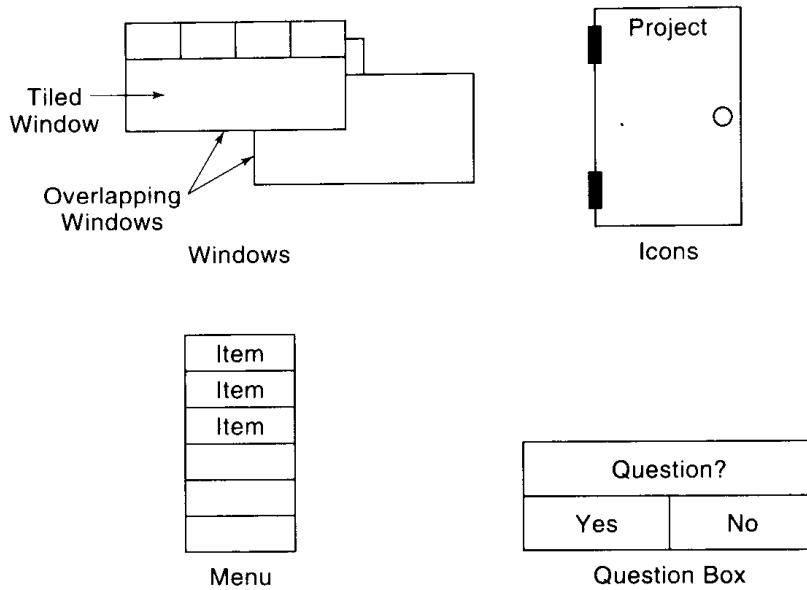


Figure 7.2 Four Common Display Objects Used in ST-80.

concept here is to provide an interface to various projects (i.e., different applications or work areas). By selecting “enter” from menus associated with different doors, various projects can be entered. Using multiple doors on the screen would provide a concrete cue to the user about what different applications are available in an environment. Tiled windows permit capturing in one unified window related and interacting information about an application. Similarly, separated windows provide some conceptual distance between activities that can be conducted on the computer desktop. The desktop metaphor relates to the analogy between multiple windows and multiple sheets or stacks of paper on a desktop. Lists provide a way of organizing alternatives to select from and in the case of hierarchical menus, permit organizing selections in tree form. Such hierarchical information imparts information to the user of the system simply by existing. Buttons and switches promote disclosure of operation information. Implicitly exclusive alternatives can be graphically delineated by so-called “push-buttons” or “radio” buttons which, when one presses one button “on,” the other buttons are turned “off.” Using this visual cue, the user is informed about the causal nature of information that exists between these alternative selections.

The next sections commence the description of how to display things graphically in ST-80. Beginning with simple display techniques, discussing windows, examining specific design elements, and finally considering how to

tailor the user interface, these sections lead toward explicit implementation information for the model-view-controller paradigm given in the next chapter.

7.3 Visual Display Methods

This section gives an overview of how visual components are displayed in ST-80. The information in this section is specific to ST-80, Release 4 and later releases. For compatibility with earlier versions, the next section presents information about displaying objects in Versions 2.5 and earlier. Comments about similarities and differences in the display techniques used in two versions are made where appropriate.

The three basic display surfaces defined for Release 4 are windows, pixmaps, and masks. Windows are the rectangular display ports supplied by the host platform and managed by the host platform's window manager. For example, windows may be MS-Windows on PC platforms, X-Windows on HP and Sun platforms, and Mac windows on the Macintosh. When a ST-80 image is moved between platforms, the look and feel of the window that appears is that of the host windowing system. The information displayed in the window, however, does not change. A pixmap is an off-screen graphics medium on which drawings can be constructed. In order to display in a window, drawings must be copied from the pixmap to a display surface in the window. Masks are used for stenciling operations, that is, for permitting color to be copied to selected parts of the display surface.

VisualComponent Hierarchy

Table 7.2 shows a partial class hierarchy for the class **VisualComponent** and some of its subclasses. **VisualComponent** is an abstract superclass, meaning that objects are never instantiated from it, but that it is used only as a class from which subclasses can inherit methods and variables. **VisualComponent** implements behaviors common to its subclasses including methods for moving images and determining whether a point is in the view. Some example subclasses include **Image** which captures multilevel bitmaps of sampled images, **Icon** which captures a bitmap to display when windows are collapsed, and an abstract class named **VisualPart** which has various subclasses for displaying visual information in views. **CompositePart**, **DependentPart**, **EdgeWidget**, and **Wrapper** are each subclasses of **VisualPart** that provide general information for concrete subclasses such as **View**, **Dialog View**, or different types of widgets. Widgets are simply additions to views that provide additional functionality, for

Table 7.2 Partial Class Hierarchy for VisualComponent (Release 4)

Visual Component	
CachedImage	
DisplayObject	"Backward Compatibility"
DisplayMedium	
Form	
InfiniteForm	
OpaqueForm	
Icon	
Image	
{6 subclasses}	
OpaqueImage	
TextLines	
VisualPart	
CompositePart	
DependentComposite	
BrowserView	
CompositeView	
DialogView	
DependentPart	
View	
{> 20 subclasses}	
EdgeWidget	
{5 subclasses}	
Wrapper	
{4 subclasses}	

example, scrollers along the edges of views that permit vertical or horizontal scrolling. Wrappers implement additional view functionality, for example, adding borders to a view. As indicated in Table 7.2, there are several subclasses for **View**, **Widget**, and **Wrapper**. The best way to become acquainted with these classes is to use the system browser to examine the classes and to review the comments contained under the comment selection of the pop-up menu associated with the list of class names in the browser.

Graphics Contexts

The class **GraphicsContext** holds information about how to display an object. For example, a graphics context would hold information about where to display, define the coordinate system, specify the rectangle within which to display, and give default values for paint selection, font type, line parameters, and so on. A graphics context can be obtained from a view by sending the message *graphicsContext* to a view. Messages within the class are available for accessing or setting the state of the graphics context. Hence, for example, one

Table 7.3 Example Methods for Displaying Graphics Objects*

Displaying strings	displayString:from:to: displayString:at:
Displaying lines	displayLineFrom:to:
Displaying polylines	displayPolyline: aCollectionOfPoints displayPolyline:at:
Displaying rectangles	displayRectangle:at: displayRectangle: aRectangle
Displaying polygons	displayPolygon:at: displayPolygon: aCollectionOfPoints
Displaying arcs	displayArcBoundedBy:startAngle:sweepAngle
Displaying wedges	displayWedgeBoundedBy:startAngle:sweepAngle
Displaying images	displayImage:at:

* Abstracted from ParcPlace Objectworks \ Smalltalk User's Guide, Release 4.

can change the color or font of an image to display simply by sending the desired color or font to the graphics context of the image.

Displaying Graphics Objects

Table 7.3 shows methods for displaying primitive graphics objects. Many of the messages shown are of the form *displayTHING: whatToDisplay* or *displayTHING: whatToDisplay at: whereToDisplay*. For example,

`displayPolyline: pointCollection at: aPoint`

is a method for displaying a line specified by a collection of points. The location at which to begin the display is to be specified in *aPoint*, which is relative to the window in which the polyline is to be displayed. All these display methods are to be sent to a graphics context which specifies the default display information as well as where to display the graphics information specified.

Paint

Paint and its subclasses provide the capability for coloring graphics objects. The **ColorValue** subclass of **Paint** permits specifying a color by name, for example, **ColorValue red**. An alternative to naming a color is to specify a mixture of RGB (red, green, blue) values, when mixed together yield the desired color; for example, **ColorValue red: 0.2 green: 0.8 blue: 0.0**. A subclass of **Paint**,

CoverageValue permits specification of the degree of coverage, from opaque to transparent, of a specified area.

Images

Images are pixel-based rectangular areas that are used to display graphics, both color and black and white. Most early versions of ST-80 were restricted to black-and-white images; ObjectWorks \ Smalltalk-80, Release 4, supports full color and integration with drawing platforms on host platforms. Images support color ranging from 1 bit up to 24 bits at each pixel, yielding color combinations from 2 to over 16 million. Of course, the proper graphics display device is needed to display color maps at high pixel depth. To understand how images are represented, consider the simplest case first. A rectangle of 100 by 100 pixels could be specified for displaying an image with only two colors, for example, black and white. An image of this size would consume only a small rectangle on a 640 by 480 screen. Each pixel would require one bit: a 0 to represent one color and a 1 to represent the other color. Consequently, to represent the entire image matrix, a total of $100 * 100 * 1 \text{ bit} = 10^4$ bits would be required. To increase the number of colors that could be displayed, the number of bits at each pixel can be increased; for example, to display 16 colors, 4 bits would be needed to uniquely specify each color, thus increasing the image matrix required to $4 * 10^4$.

For the beginner, experimentation with images is simple. For example, in a workplace one could type:

```
|anImage|
anImage := Image from User.
```

highlight this code and select “doit” from the operate menu. Sending the message *fromUser* to **Image** results in display of a cursor on the screen that permits the user to bracket any section of the screen. The resulting captured image from the screen can be saved in a variable and then displayed in a window. For example,

```
ScheduledControllers activeController view graphicsContext
displayImage: anImage at: 0@0.
```

will display the image stored in the variable ‘anImage’ in the current active window at the top left corner of the window (i.e., 0@0). To try out the previous example, type the three lines of code beginning with *|anImage| and ending with... 0@0* into a workspace, highlight the entire code, and select “doit” from the pop-up menu.

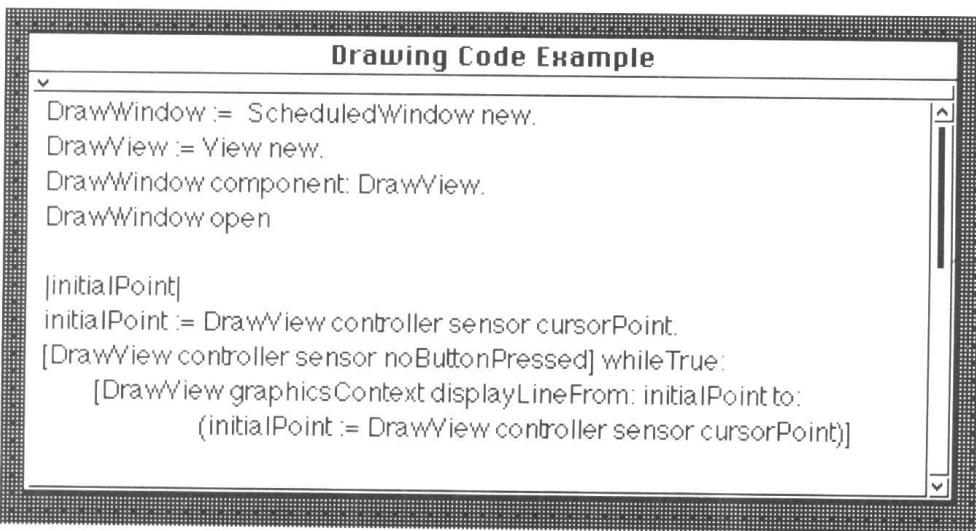
Furthermore, an image captured in this way can be stored in a file using the binary object storage facility of ST-80. Perhaps most interesting is that users can employ common drawing packages to draw images or capture clip art from packages on the host platform that can be directly imported into the Smalltalk environment.

Pallettes

Pallettes provide a way of interpreting the numerical pixel values in an image. Two main representations are used: a fixed palette in which the contributions of red, green, and blue to a color are specified and a mapped palette that stores a table of values corresponding to specific prespecified colors.

Animation

The class **VisualComponent** provides a useful method called *follow: aLocation while: true on: aGraphicsContext* which can be used for animation. This message can be sent to an image (because **Image** is a subclass of **VisualComponent**) and the image will track the mouse location specified in *aLocation* as long as the argument to “while:” is true. *Display* is on the graphics context specified. Figures 7.3 and 7.4 illustrate the use of the *follow:while:on:* method for drawing a line on a view in a window. Figure 7.3 shows how to set



The screenshot shows a Smalltalk window titled "Drawing Code Example". The code inside the window is as follows:

```
DrawWindow := ScheduledWindow new.  
DrawView := View new.  
DrawWindow component: DrawView.  
DrawWindow open  
  
|initialPoint|  
initialPoint := DrawView controller sensor cursorPoint.  
[DrawView controller sensor noButtonPressed] whileTrue:  
    [DrawView graphicsContext displayLineFrom: initialPoint to:  
        (initialPoint := DrawView controller sensor cursorPoint)]
```

Figure 7.3 Code for Drawing on a View.

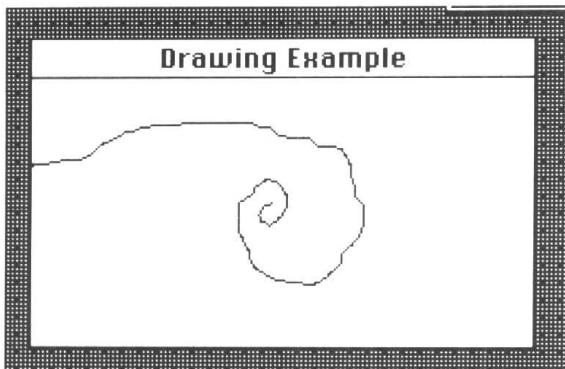


Figure 7.4 Example of Drawing a Line on a View Using Code in Figure 7.3.

up a window and view (DrawView) on which to draw a line and Figure 7.4 captures the drawing of a line on the view. Note the use of *displayLineFrom:to:* and detection of the location of the cursor in the window by using *DrawView controller sensor cursorPoint*.

A simple technique for animation is a “filmloop” which is simply a collection of images repeatedly displayed at the same location on the screen. Using a drawing package, one could, for example, create about 30 images of a flywheel with an arm. By continuously displaying these images in order, with a delay of perhaps 1/30 of a second or less between the presentation of each image, the result will be an animation of a flywheel on the screen. For any continuous repeatable animation, the filmloop idea is a useful technique. Details on implementation of filmloops is given in LaLonde and Pugh (1989).

7.4 Backward Compatibility for DisplayObjects: Forms, Pens, and Paths

This section presents an overview of how a few simple objects in ST-80, Version 2.5 and earlier, were displayed. This section can be skipped by readers who are not interested in comparing the display methods used in earlier versions of ST-80 with Release 4. This information is presented for those readers who still have Version 2.5 or those who wish to gain a historical perspective about the influential graphics models that were used in Smalltalk-80 for almost 20 years. The display model in previous versions was significantly different from the display model for Release 4 described in the previous section. General display notions will be examined as well as the hierarchy of display objects and a few specific useful objects that can be used for building displays in this version of the ST-80 display model. The interested reader is referred to

Table 7.4 Class Hierarchy for DisplayObjects (Version 2.5)

Object
DisplayObject
DisplayMedium
Form
Cursor
Display
DisplayText
Paragraph
InfiniteForm
OpaqueForm
Path
Arc
Curve
Line
LinearFit
Spline

Goldberg (1983) and Pinson and Wiener (1988) as well as the code and examples in the system browser of the ST-80 (Version 2.5) system.

Class Hierarchy for DisplayObjects

Table 7.4 displays a simplified version of the class hierarchy for **DisplayObjects**. The superclass of the objects is **DisplayObject** which contains most of the abstract protocol for many objects that can be displayed on the screen.

Table 7.5 provides more detail about the classes shown in the hierarchy displayed in Table 7.4 and adds additional classes that permit drawing on the screen such as the class **Pen**. The subdivision into three segments shown in Table 7.5 is similar to the segmentation in the ST-80 browser. The first category contains primitive display objects of simple drawing elements, the second category contains often-used display objects, and the third category provides brief definitions for several display objects based on paths. The reader should note that these graphics constructs are only available for backward compatibility in ST-80, Release 4 and higher versions.

Displaying Objects

Displaying objects on the screen is very simple. Forms are created as bitmaps, that is, simple rows of 1's and 0's. Forms are the ST-80, Version 2.5, equivalent of Release 4 images with a depth of 1 bit. The Smalltalk (ST-80)

Table 7.5 Commonly Used DisplayObjects in Version 2.5

Name	Utility
Primitive DisplayObjects	
Pen, point, quadrangle, rectangle	Drawing lines, graphs, rectangles / quadrangles (rectangles with an inside color)
Common DisplayObjects	
Cursor	Pointer representation
DisplayMedium	Abstract class for form and cursor
DisplayObject	Protocol for most displayed objects
DisplayScreen	The computer screen
DisplayText	Text for display
Form	A bitmap
InfiniteForm	An unbounded form
OpaqueForm	Parts of the bitmap are opaque and parts are transparent
Paragraph	Text for editing
Path DisplayObjects	
Arc	A quarter of a circle
Circle	Four arcs
Curve	A conic section determined by three points
Line	Straight line
LinearFit	Piecewise linear approximation

screen with 1's and 0's representing forms can, of course, display only black and white. To display near white and near black as well as shades in between using the form model, it was necessary to display varying numbers of black-and-white dots in an area.

Forms

In Version 2.5, **Form** was likely to be one of the most important classes that the beginner would first learn to display on the screen. The display screen of the computer is a global variable called "Display." It was straightforward to display a form anywhere on the screen; for example,

```
aForm displayAt: 0@0
```

would place "aForm" at the upper right-hand corner of the screen. In this example, "aForm" is any instance of a form created by using the form editor or

obtained from other sources (e.g., a scanned image). The most direct way to test this code is to grab a form from the screen and display it. For example, one could enter the following code in a workspace, highlight the code, and select “doit” from the operate menu:

```
|aForm|  
aForm := Form new fromUser  
aForm displayAt: 0@0
```

This example is essentially the same as the example given in a previous example using images. However, note that with forms, the image could be displayed anywhere on the screen. In the newer versions of ST-80, the display surfaces are limited to host windows.

Figure 7.5 shows a way to display large forms, called canvases (see the Appendix concerning the class that implements a canvas). A canvas permits having a constrained size viewport that can seemingly be moved around the entire form by grabbing the scroll bars (i.e., positioning by using the cursor) to move the canvas. The methodology used is simple, because a form is composed of bits. Whenever a scroll bar is moved by use of the cursor, bits are copied in the direction that the scroll bar is moved, thus creating the illusion of panning the viewport over the form. The code for implementing this class is available as

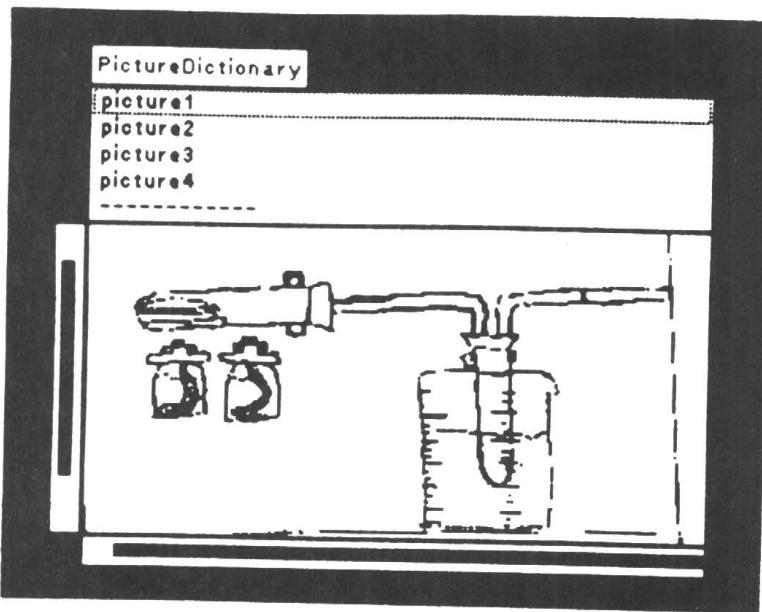


Figure 7.5 Displaying a Canvas.

indicated in the Appendix (both Version 2.5 code and Release 4 code is available).

Pen

A concept found in most early versions of Smalltalk is the pen. This idea has disappeared from Release 4. The intellectual basis for using a “pen” for drawing came originally from the Logo language (Papert, 1980). A line drawing method that used a turtle was invented for teaching children about programming ideas. Children could relate drawing to a physical turtle holding a pen that could be moved up and down. In fact, robot “turtles” with pens, that is, mobile robots with drawing instruments, were created as an output facility for teaching programming in Logo, a language very much like Lisp. The turtles were given such commands as “Pen up,” “Pen down,” north, and so on, that is, easy to understand commands sent to move the turtle and the pen. In Smalltalk, Version 2.5 and earlier, the pen was operated by creating a new pen instance, for example, `aPen := Pen new.` The pen instance can then be moved up or down (off the paper or touching the paper), turn, go a distance, go to a location, and so on. This methodology makes it extremely simple to produce line drawings. The pen nib can be any shape, including different forms.

A demonstration of the use of the pen is shown in Figure 7.6, which shows how a pen can be used to make a freehand drawing on the display. This example shows both the code and an example of having a pen follow the cursor as it is moved on the screen. In the figure, “bic” is defined as an instance variable that is assigned to the new pen instance. First, the pen is raised and moved to the location where drawing should begin and the pen lowered. Then, as long as no mouse button is pressed, the pen follows the location of the cursor. `Sensor mousePoint` returns the current location of the cursor. Compare this methodology for drawing to the examples presented using `displayLineFrom:to:` given in the previous section using the drawing method available in Release 4. One should note that there is little difference between the two techniques.

Paths

The class **Path** in pre-Release 4 ST-80 versions permitted specification of paths on the screen which could be used to display forms. For example, lines, arcs, circles, and most other kinds of figures could be displayed by creating a path. An instance of the class **Path** contained a collection of points. For example, consider Figure 7.7 which shows a very simple example of creating a

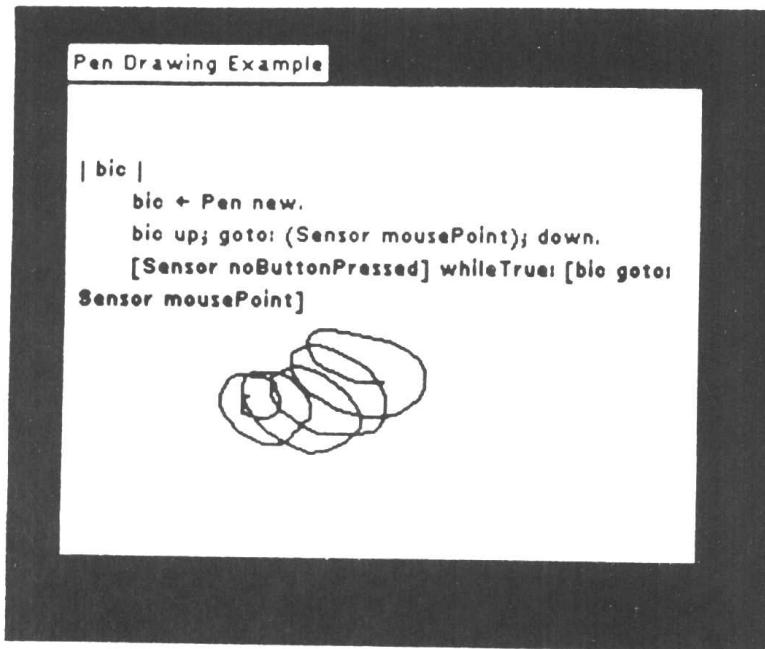


Figure 7.6 Drawing with an Instance of Class **Pen** (Version 2.5).

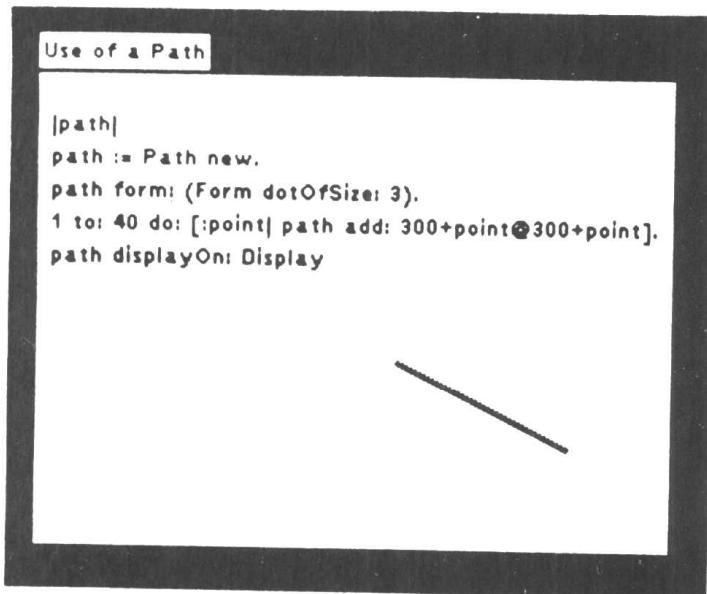


Figure 7.7 An Example of Creating a Line with Class **Path** (Version 2.5).

straight line. First, the local variable “path” is assigned to an instance of the class **Path**. The form used for drawing is assigned and the points to be employed are generated using the “do:” and finally the path is displayed on the display. In the iteration, the loop generated 40 points, each of which added a specific screen coordinate to the collection stored in the instance of **Path**. Each point has the form $x@y$; for the case shown, the points were generated from 301@301 to 340@340. Because the screen on which this example was generated had its top left corner as the origin (i.e., 0@0), the line was drawn from top left to bottom right as shown.

Generation of a path is a useful technique for animation in simulation applications. Imagine that a form is drawn on the display upon which an object should move to simulate some situation, for example, traffic moving on a grid of streets. The background grid could be generated as a form and shown on the display. Small forms to display automobiles could be created as forms or opaqueForms which could be displayed on the background form and moved along a preset path to give the appearance of animation. The same technique could be employed to simulate customers moving in a bank, ships moving from point to point on a map, and so on.

Paths have disappeared from Version 2.5 in favor of using displayPolyLines with a collection of points. Moving small images to create simulations is easily accomplished in Release 4 by having images follow a collection of points.

Animation

Animation required movement of forms or pens in Version 2.5. Pens could be used to dynamically draw pictures or sets of pens could be driven together to create the appearance of drawing simultaneity. By capturing where a pen draws and the size of the nib, a line drawn by a pen could be erased by drawing a white form over the path of the black line. This technique works fine for drawing and erasing a pen on a white background. However, if the background is not white, the background needs to be saved. Forms when moved over another form should save and restore the background (see method ‘background:at:’ in class **DisplayObject**) unless the background is white. This idea also holds for animation using images in Release 4.

7.5 Windows and Views (Release 4)

Concepts

A window forms the rectangle around either single or multiple views, including the border. Views are contained inside a window. When multiple

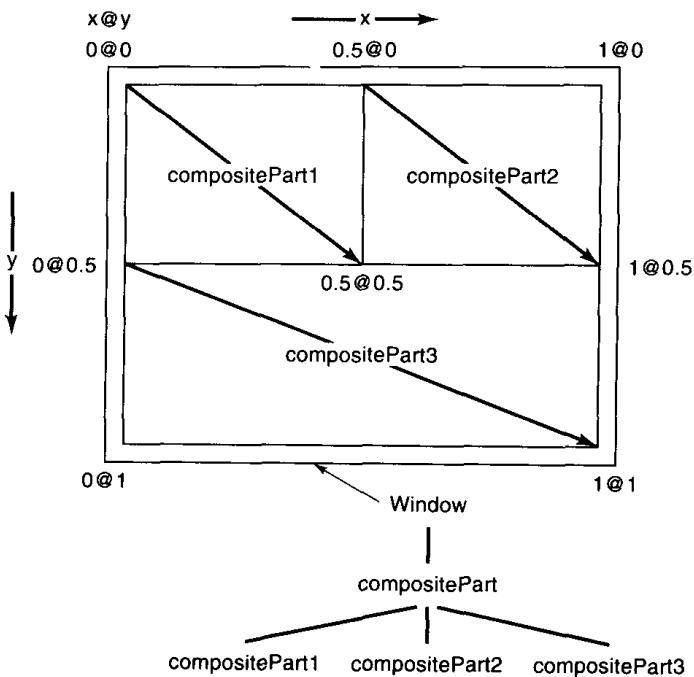


Figure 7.8 Methodology for Specification of **CompositeParts** within a Window.

views are shown within one window, wrapped views are packaged as a collection of parts, contained within an instance of class **CompositePart**. Each view interior to a window can be considered to be a composite part of the window. In some systems, multiple views existing within a window are known as “panes” (e.g., Smalltalk/V) by analogy to windowpanes. In Smalltalk-80, the class **View** is used as an abstract for other view-related classes such as **ListView**, which implements the functionality for selecting items within a list. Functionality for moving, collapsing, and resizing windows is relegated to the host windowing system. Normally, several views are created in an application window. Resizing a window can automatically resize the attendant views implemented as composite parts of the window. Figure 7.8 presents an example of one method for specifying views as fractional parts of a window. The entire window is defined to have an extent of 1, that is, 0@0 to 1@1. Each view is defined as a fraction of the window to which it belongs so that when the window is resized, all the composite parts of the window can be proportionately resized as well.

View Operations

Table 7.6 shows examples of some of the subclasses of **View**. **BooleanWidgetView** provides basic functionality for things that turn on and off. **Display-**

Table 7.6 Basic View Types in Objectworks \ Smalltalk-80 (Release 4)*

View
BooleanWidgetView
LabeledBooleanView
DisplayTextView
FractionalWidgetView
LauncherView
ScrollView
ScrollingLinesView
StringHolderView
Textview
ListView

* Abstracted from ST-80 class information, ParcPlace Systems, 1990.

Textview is a class for displaying text. The **LauncherView** provides a way to display an initial menu that is present on the screen when Smalltalk is launched that can be used for running applications and system tools. **ScrollView** and its subclasses for **Text** as well as **ListViews** provide basic functionality for text and lists. All the different view types have different pop-up menu functionality associated with the controllers that are used for operating the view. Most views, however, have a common type of operation for sizing, moving, and restoring windows. For example, Figure 7.9 shows two versions of the pop-up menu associated with the operate button on the mouse for resizing, restoring, and moving a window. Note that the appearance of the menus is somewhat different for different Smalltalk versions but has essentially the same functionality. **ScheduledWindow** is a class that is used to secure a new window from the host operating system. This class provides window functionality including sizing, coloring, and opening new windows. Many methods are available that can be examined using the system browser. A simple example will illustrate how straightforward it is to create a new window on the screen:

```
|aWindow|
aWindow := ScheduledWindow new.
aWindow label: 'Example Window';
minimumSize: 300@200;
insideColor: ColorValue blue;
open.
```

By highlighting this code in a workspace and selecting “doit” from the operation menu, a new window will appear on the screen with the label shown and a blue interior color. The window that is opened can also be closed by simply sending

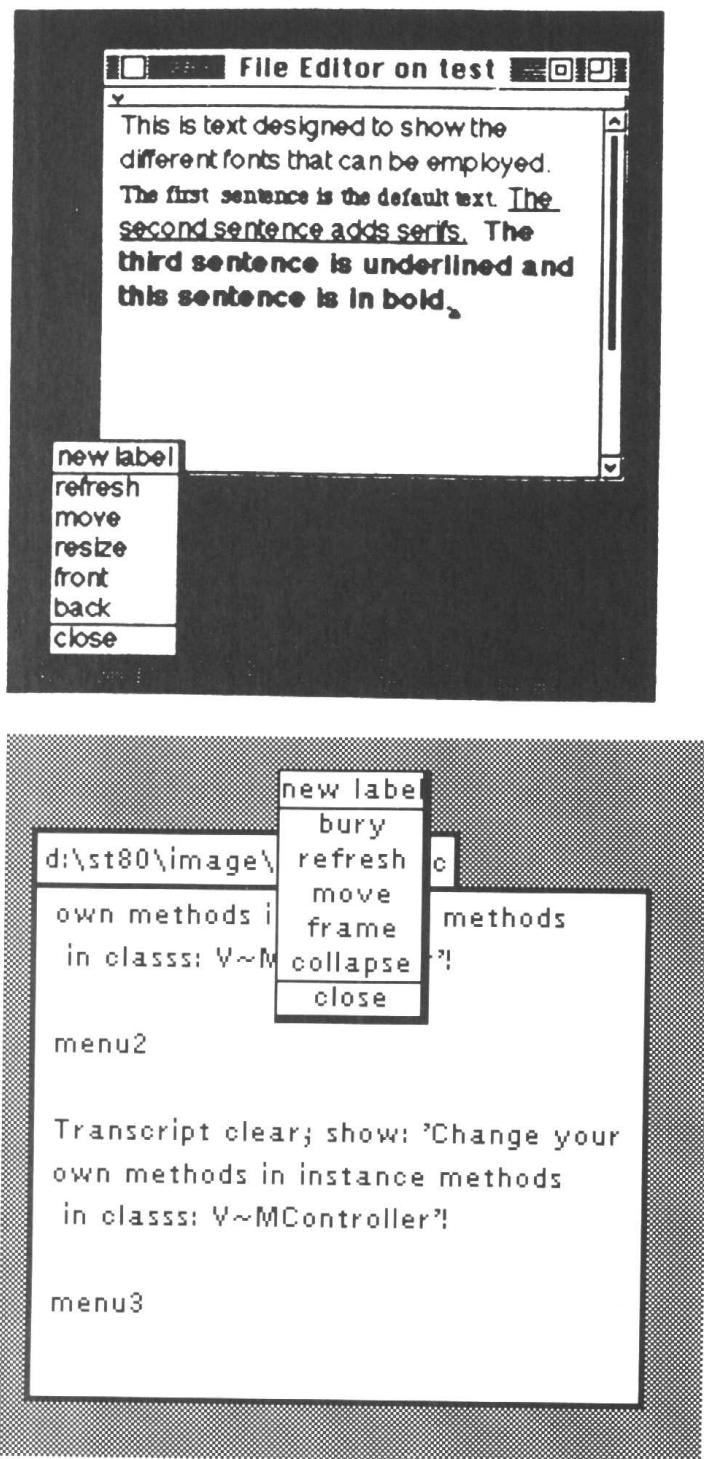


Figure 7.9 Operate Pop-up Menu for Two Different Platforms.

close to the window. For example, one can create a new window in a workspace and open it:

```
Awindow := ScheduledWindow new.  
Awindow open.
```

and then close the window by

```
Awindow close.
```

In this example, the window will have a minimum size. The global variable “Awindow” was used simply to make it easy to access the variable from a workspace. In actual programs, the variable bound to the new window would probably be contained in a class instance variable.

Displaying in Views

Normally to display views of text, lists, images, and so forth, views of a particular type (e.g., **SelectionInListView**, **DisplayTextView**, **BooleanWidgetView**, etc.) are created and added at a particular location in the container of views that fill a window. Each particular type of view will have its own capabilities, including different types of scrolling. For example, a list may scroll only up and down, but a canvas¹ will have scroll bars to scroll in two directions. Images can be created and displayed in any view.

A particularly interesting issue is the use of nonstandard views to add meaning to the display. For example, an electronic circuit might be represented as a collection of many views, each of which could be represented as an icon that has a visual identity corresponding to its physical reality (e.g., symbols for transistors and resistors). Each view might be collapsed and when expanded could show more information about the internals of the component or subcircuit that is expanded. This idea is particularly useful for representing whole-part hierarchical relationships. The alternative to this representation scheme is to place images on a background which represent the subparts of the thing being displayed. For example, for this electronic circuit, images of resistors, transistors, and so on would be positioned at various locations on an image and information about the size and extent of the positioned images would be maintained in a dictionary. The trade-off between these two alternatives is capability and speed. The first methodology gives a large capability for ease of

¹See the Appendix.

manipulating hierarchical representations, such as designs, but pays a penalty in speed for having to maintain many view relationships in a hierarchy. In contrast, the image-based methodology is much faster, simply having to display images at various locations on the screen.

7.6 Basic Direct-Manipulation Interface Design Elements

General Issues

This section examines the specific kinds of views that can be built into an application. Information about styles and the look and feel will be given here; Chapter 8 will examine specific code-related issues. The elements used in designing a user interface should, first, be open and observable. The user should be able to either intuitively understand how to use the interface or with some training understand how the system designer intended the system to be used. This latter point is relevant to the ST-80 system. It is not immediately intuitive to the novice user how to operate ST-80, because learning to operate the mouse and understanding how and when menus pop up are prerequisites to operation. A completely intuitive interface would not require such information to be learned prior to operation. For example, displaying buttons that say “Press me” would be intelligible to users who understand basic mouse operations. However, even systems with this style would fail with a novice who presses the screen with his or her finger! Hence, the first point in building an interface is that it is essential to evaluate what skills prospective users will have and evaluate how to provide users who have inadequate skills with the necessary instruction to be able to operate the system.

To build an application, the screen layout should be considered, particularly how much information can be shown on the screen at once. Having the capability of providing both tiled and overlapping windows, an assessment should be made of the complexity of tiled windows for parts of the application along with an evaluation of how often a user would need to switch between overlapping windows. Of course, overlapping windows is not a problem when the physical screen is large. It is an issue, however, when small screen sizes are employed, such as on certain Macintosh or PC screens. Views should be uncluttered; however, there is a considerable temptation to design cluttered interfaces that provide the user large amounts of information. One solution is to design a continuously visible control panel that directs the operation of the application, moving from one tiled window to another as the user proceeds through the application. Use of menus is another design issue; pop-up and pull-down menus provide the most clutter-free design but also do not yield

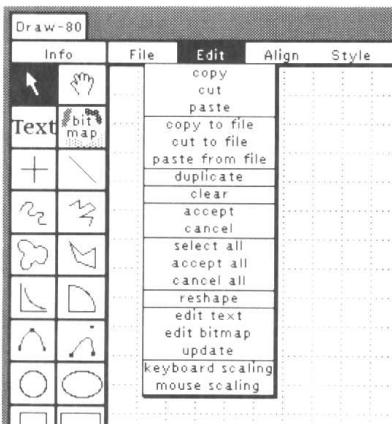
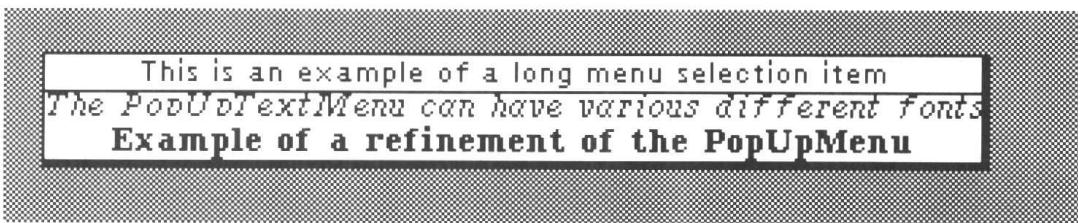
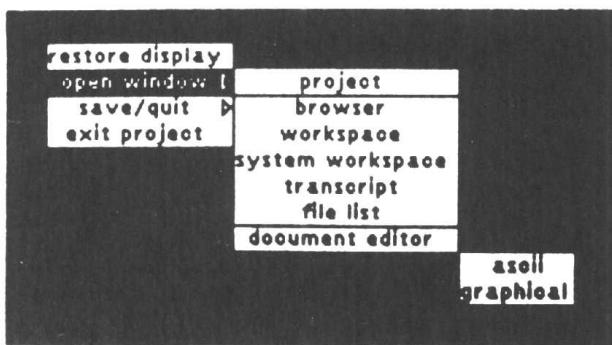


Figure 7.10 Three Types of Menus: Hierarchical, List, and Pull-Down. [Reprinted with permission of Knowledge Systems Corporation.]

much observability of the available operations in an application. One technique to solve this problem is to mix control by providing both switches and menus; in some cases, it may be useful to pop up a menu when a button is pressed!

Menus and Lists

Listing of items from which selections can be made is a typical component of user interfaces. Figure 7.10 shows several styles of menus: a simple list, a list that uses different font styles, a pull-down menu, and a hierarchical menu. Lists can be either fixed in a view and scrollable, pop-up, or pull-down. Both of the latter can be scrollable as well. The hierarchical menu probably provides the highest degree of information for the least space used. The hierarchical menu in Figure 7.10 shows how the system menu can be reimplemented using hierarchies rather than using the LauncherView (compare with the Launcher in Figure 6.1). Figure 7.11 illustrates the ease with which a hierarchical menu can be built using the ST-80 hierarchical menu building facilities. This figure shows both the code and the result of running the code. Note also in this example that the menu selection is bound to a variable called

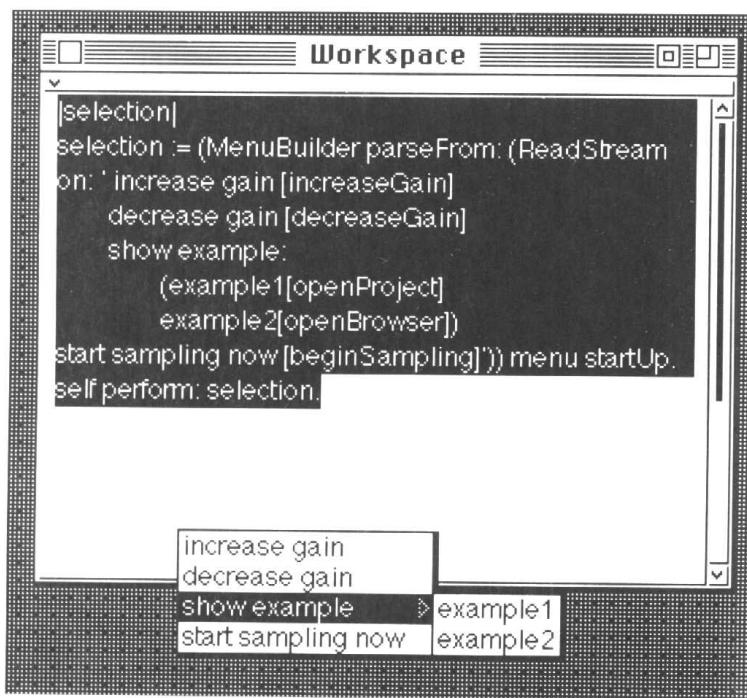


Figure 7.11 Implementing a Hierarchical Menu.

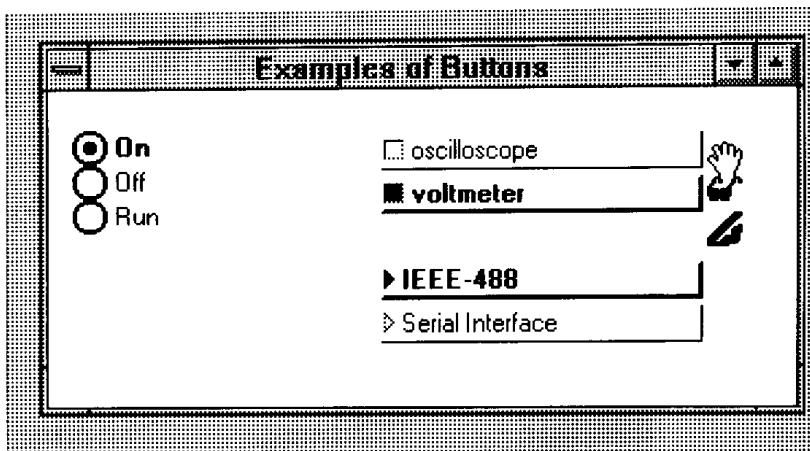


Figure 7.12 Examples of Four Different Styles of Buttons.

“selection.” When a selection is made, the selection is bound to the variable of the same name, in this case. The last line of the code in the figure illustrates the use of this information to perform some task associated with making the selection. Sending self “perform: selection” will cause a method called selection to run within “self,” where “self” might be the current class or class instance.

Buttons

Figure 7.12 shows four examples of different kinds of buttons. On the left of the figure are radio buttons. Pressing one button turns the others off. An example of switches are shown on the top right. Switches require successive mouse clicks to turn a switch on and off. The box and the label reverse color as the switch alternates between on and off. The toggle switch toggles between alternatives. Also shown are two icons, a hand and a symbol for writing, that are part of the ST-80 system. Icons can also be used as pictures on buttons.

Icons

Figure 7.13 shows the concept of the use of icons. An icon is simply a miniaturized symbol for an operation. There might be, for example, a set of icons that represent the collapsed views of things such as browsers, text editors, and other common elements in the GDMI environment. Or, icons could be used to graphically represent the actions of various buttons on a view. Any-

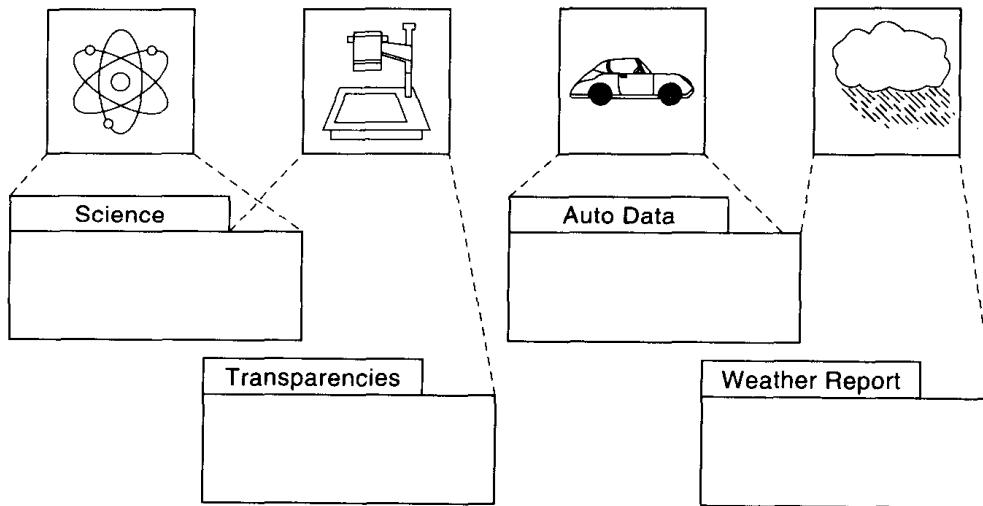


Figure 7.13 Using Icons to Represent Windows.

where that an image can be used in association with an item that is directly manipulatable, there is the opportunity to use an icon. The default behavior for icons is to write text on a small rectangle representing a collapsed view. Whether an image that has the appearance of a door, for example, or a rectangle that has a label on it is more or less understandable to the user is open to question. Proponents of icon utilization argue that these types of graphic images promote system understanding; opponents likewise argue that there is little value added. The answer is perhaps that icons should be used if there is a clear need in the application to mimic something in the real world that would make common sense and thus make the operation transfer task more clear to the user. Otherwise, text will work fine.

Text and Code Presentation

Text is easily entered and displayed using the `TextEditor` class. Figure 7.14 shows an example of using different font styles for different parts of typed text. Text can be collected and automatically formatted to the size of the workspace. Font and style selection is via the operate-button pop-up menu. Text presentation is very useful for communicating with the user. A typical use is the system transcript which provides messages from the system such as error messages. This same type of transcript can be employed by the user for special text communications as shown in Figure 7.15. This figure shows the creation of a `TextCollector` view called "Announcer" which can be sent a message from

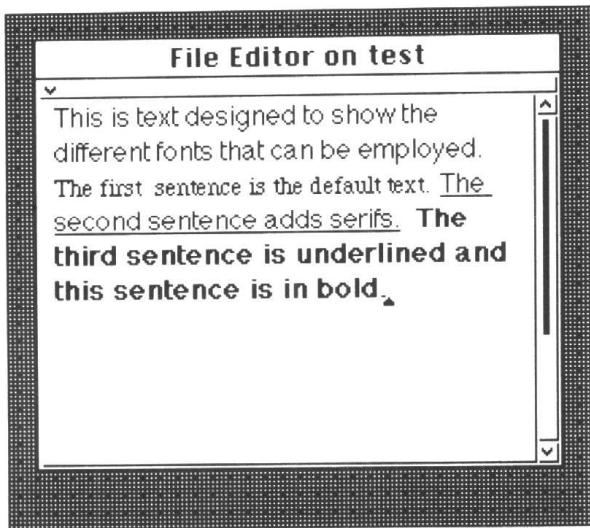


Figure 7.14 The Use of Different Fonts in Text.

anywhere in the system. The result of sending a string is shown in the demonstration window in this figure.

Typing Smalltalk code into a CodeView provides the capabilities of organizing text, for example, for building methods. In the system browser various text-handling capabilities are built in including the “format” command which provides a “pretty printing” capability for ST-80 code; that is, when “format” is selected, the code in the view will be parsed and reformatted

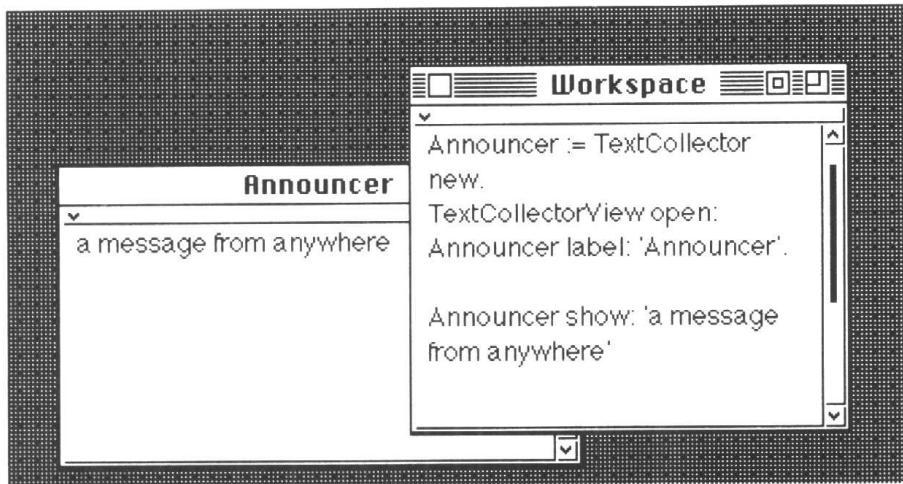


Figure 7.15 Creating a Global Text Collector.

according to formatting rules. Often, this provides more readable code. Note, however, the user must still “accept” the code, that is, compile the code and register it in the browser.

Using the Keyboard

In some cases, it may be easier to use the keyboard than the mouse for accomplishing various operations. In various word processors, it is common practice to permit a user to employ the mouse or to capture various operations using function keys or combinations of keystrokes on the keyboard. In ST-80, several examples exist that map mouse operations onto the keyboard. One specific example of the use of the keyboard is for adding text characteristics to text in a text window. Key combinations for underlining, adding bold, changing serif, and so on, are available via a keystroke combination. For example, the combination `<esc>-b` will add boldness to the highlighted text selection. Whether or not such operations are better accomplished by the use of the keyboard or by the mouse appears to be completely driven by user preference.

Mousing

Novice users of a mouse can learn mouse operation quite quickly—pointing and clicking comes naturally to users. However, becoming facile with the mouse requires more extensive use. A common experience is that users who consistently use the mouse become faster and faster with operations over time, as expected. Text editing using the cut, paste, and copy operations between multiple windows and the use of intermediate workspaces for editing text, making notes, and editing images can eventually be coordinated by rapid mouse operations. Perhaps the greatest advantages for using a mouse are that mouse-style operations: (1) implement the point-and-click metaphor, thus reducing ambiguity about operations, (2) open computer systems to persons with little typing capability, and (3) assist in focusing a user’s attention, for example, by activating a window when the mouse is moved inside the window.

7.7 Graphically Tailoring the User View

The next few paragraphs discuss options that the programmer has for creating a user interface. Both the capabilities of the standard interface and ways to extend the standard interface are discussed.

Capabilities of the Standard ST-80 User Interface

As discussed previously, there are a variety of tools that are available in the standard ST-80 user interface that permit creating user interfaces with standard features such as lists, menus, buttons, text windows, and images. These features, when combined into a window and coupled in a model-view-controller relationship, allow creation of various relatively complex user interfaces that are suitable for many applications. Even these relatively simple interface concepts can be easily tailored to provide more distinctive interfaces. For example, icons and buttons can employ user-created images to provide a more visual reminder of functionality. Views can be rearranged to be different from the standard “paned” or “tiled” views discussed. For example, there is no reason that views cannot be scattered around a window such that the views take on meaning by how they are represented. In a standard “paned” or “tiled” view, there are few hints about the intended functionality of the view, other than recognizing what is implemented in the view (e.g., a list, a button). As discussed previously, a view of a flowchart or a view of electronic components could permit the user to understand the use of the views within a larger context. For example, imagine that each block in a flowchart or each component in a schematic is a separate view. Because each view can have its own functionality apart from its organizing container in the window, these example views could permit viewing of their contents or present other relevant information. The point of this discussion is that there are many ways to tailor the user interface without making major changes to the standard components that are available.

Custom Interfaces

It is a relatively short step from using the standard ST-80 components to more complex user interfaces. Next, a number of ideas about enhanced user interfaces are discussed.

The Alternate Realities Kit (ARK; Smith, 1987) extends the ST-80 environment in several ways. The objective of Smith's work was to create an environment in which the influence of physical laws could be observed (e.g., on a planet moving around the sun) and to be able to change laws and see the effects of the change on the alternate reality created. Smith addresses the issue of literalism and “magic” in a user interface. Literalism simply means that the actions that occur are literally what the user would expect; for example, pressing a button marked exit causes that action to occur. The term “magic” comes from adding capabilities to an interface that are not so literally understood. For example, Smith added the capability of dragging buttons to an object which could be used to operate the object in different ways. If the object did not

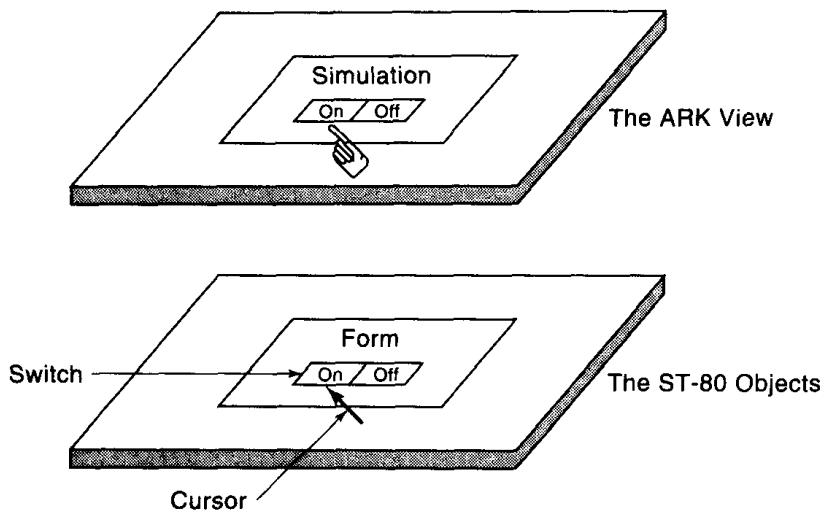


Figure 7.16 The Alternate Reality Kit Concept (explaining concepts in Smith, 1987).

understand the messages from the button to the object, then the button could not be attached to the object and would drop through it—"magic"! Figure 7.16 shows how the ARK environment is extended from ST-80. An ST-80-style interface is shown on the bottom layer of the figure with an image and a switch on the image. The mouse cursor is shown pointing to the switch. The top layer shows how the ARK user interface modifies the capability of the ST-80 environment to provide a slightly different display. Instead of a simple cursor, a moveable image showing a hand is displayed and buttons can be moved on top of an object. If the button is accepted on the object, then the hand can press the button to achieve a desired action, for example, a simulation. This type of interface is perhaps somewhat closer to reality and has some aspects that make it easier to use than the standard ST-80 environment.

An interesting type of interface is one that assists in organizing knowledge according to commonsense spatial organizations that are easily remembered. For example, Figure 7.17 shows a drawing of an interface that is configured to look like a room in a library. All the objects in the room can interact with the mouse, and actions are accomplished by clicking the mouse button while pointing the cursor at objects in the room. For example, the novice user will probably inspect the magazine on the table in the middle of the room marked "read me" when first using the system. Likewise, there are books on the bookshelves with labels and pictures on the walls that will give more information when accessed. A door to the right opens to other rooms with "more" information. Admittedly, this representation is attractive to the naive user—but is it useful to the accomplished user? The answer is probably not! The

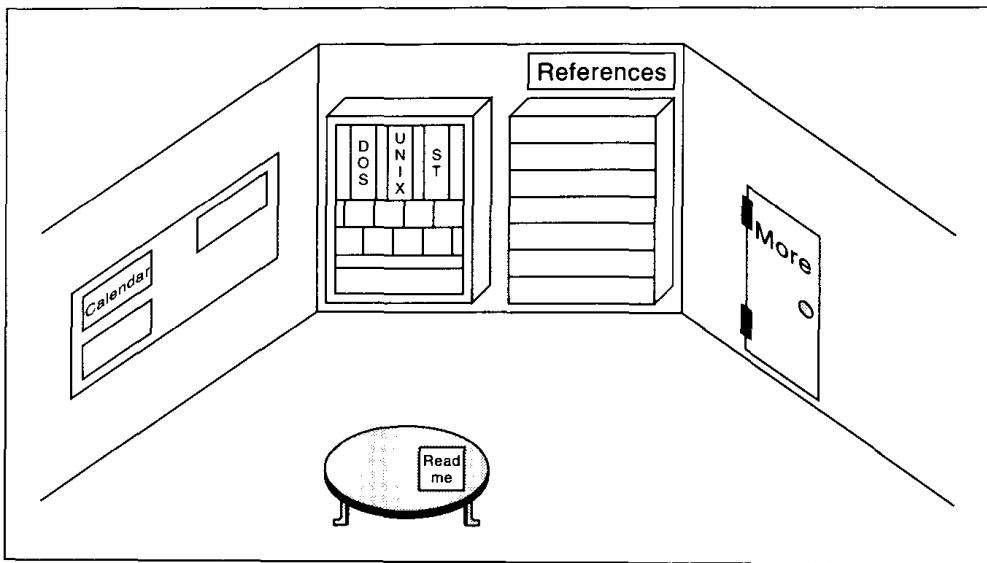


Figure 7.17 The Room Metaphor as a User Interface.

accomplished user learns to organize information and how to operate a system and thus does not need the visual cues afforded in this graphics user interface. For the novice user, the cue “read me” on the table is a useful one; yet, the same cue could be presented in a box on the screen that would serve perhaps equally well in a less elaborate interface setting.

Another type of interface is shown in Figure 7.18 patterned after the user interface style of RMG, a simulation system from Hewlett-Packard, (RMG, 1989). In this figure, commands are arrayed around the periphery of the view and moving of the view is controlled from the sliders on the right of the figure. In this example, a view of a molecule, the molecule is moving and zooming in and out is permitted. The idea of animation controlled from the buttons around the view is an interesting presentation interface.

Simulation environments can be readily created to enhance the standard features of Smalltalk. Consider the manufacturing floor plan in Figure 7.19 which shows how an electronic circuit board is created by passing the board from one machine to another along a flexible manufacturing line. This example utilizes only standard Smalltalk graphics features. The line appearing as a U shape is represented as an image with small images located at strategic locations on the image. For example, the blocks marked “k” are baskets for placing a completed unit; as a unit is manufactured and passed from one machine to another, the boxes “k” fill and empty as the board passes along. The only movement is the circuit board which can be demonstrated graphically to the

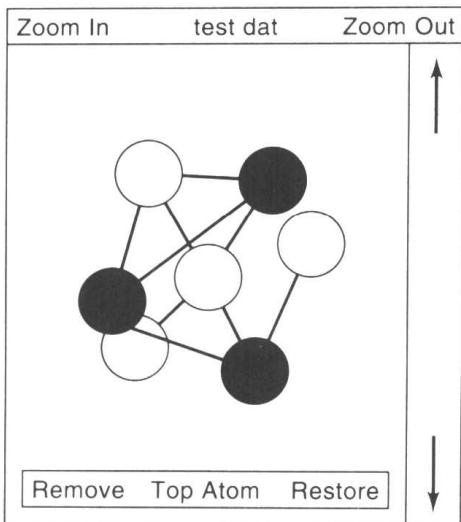


Figure 7.18 The RMG User Interface. [After RMG (1989); reprinted with permission.]

user by sequentially turning the “k” boxes black and then white. This gives the illusion of movement. The remainder of most of the features on this diagram are simply standard Smalltalk features, arrayed in different ways. One additional feature, is the use of Pluggable Gauges (Adams, 1987)² which are variable gauges that can show information about several items. In this example, gauges are used to show the percentage of team utilization, the simulation speed, the duration of the simulation, and the percentage of utilization of each machine. Chapter 14 will expand this example.

Finally, a useful addition to the types of user interfaces discussed previously is the addition of video to the presentation capabilities of the user interface. Figure 7.20 shows a schematic diagram of how to mix video and an application view. The support for this composite is the addition of a video controller which permits merging of video and a standard screen presentation. Video, supplied from various sources, such as a video disc player or camera, can be controlled by the computer (e.g., by using a serial connection to start or stop the disc player). Using a system of this type, the mixing of video and standard presentation interfaces permits presentation of animated clips of information as well as presentation of many still pictures.

²Some gauges for use in Release 4 are available to readers of this book, as indicated in the Appendix.

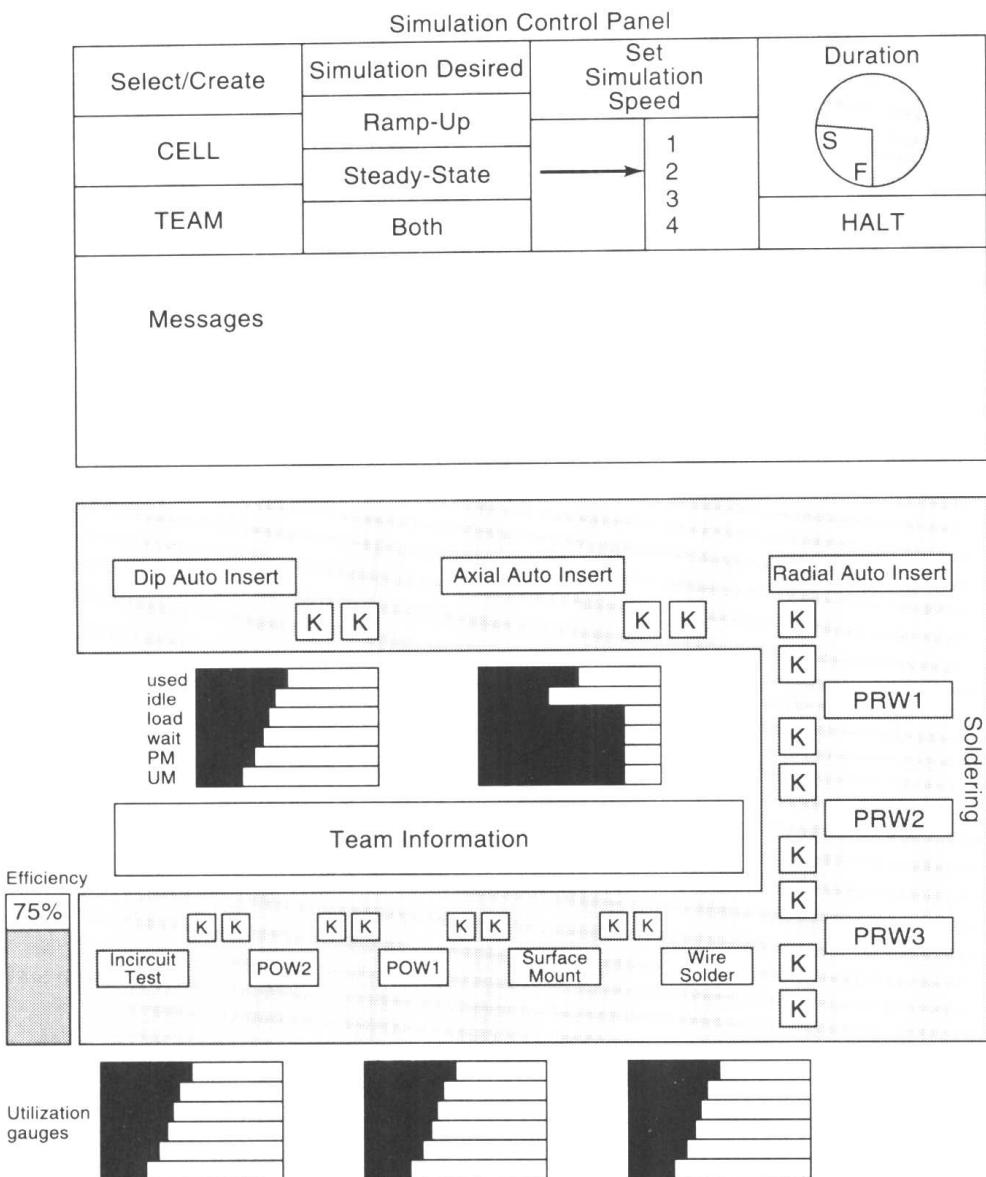


Figure 7.19 An Example of a User Interface for Manufacturing. [Dehner (1990); reprinted with permission.]

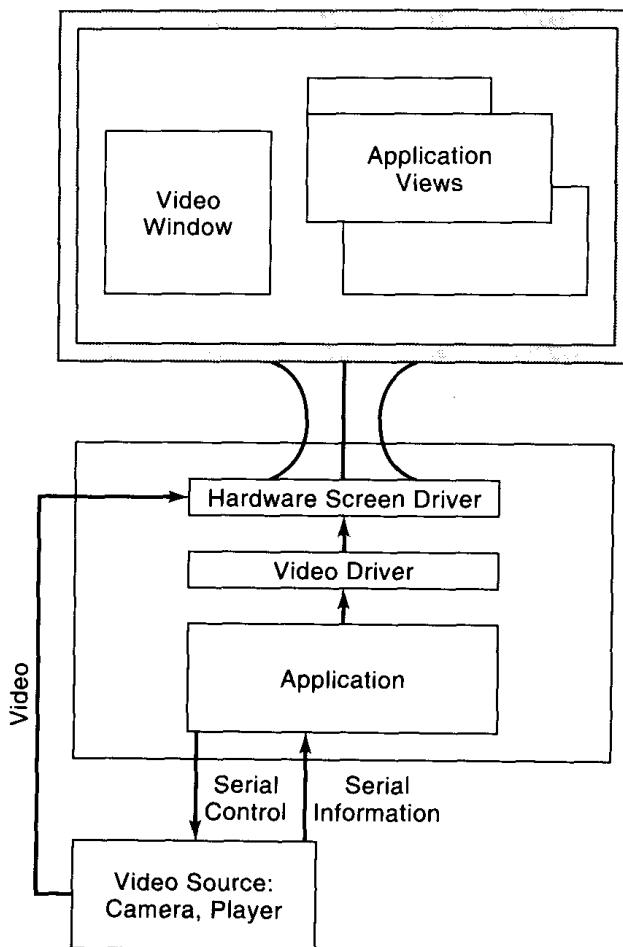


Figure 7.20 Adding a Video Interface to a GDMI.

References

- Adams, S. (1987). *PluggableGauges*. Knowledge Systems Corporation, Cary, NC.
- Dehner (1990). Class Project, EE396, Vanderbilt University, Spring Semester, 1990.
- Goldberg, Adele (1983). *Smalltalk-80, The Interactive Programming Environment*. Addison-Wesley, Reading, MA.
- Holzgang, D. A. (1988). *Understanding PostScript*. Sybex, Inc., Alameda, CA.
- Hutchins, Hollan, J., and D. A. Norman (1986). Direct Manipulation Interfaces. In *User Centered System Design, New Perspectives on Human-Computer Interaction*, D. A. Norman and S. W. Draper (Eds.). Lawrence Erlbaum Associates, Hillsdale, NJ.

- LaLonde, W., and J. Pugh (1989). Film Loops and Animation. *Journal of Object-Oriented Programming*, Vol. 1, No. 5, January/February, 64–72.
- Papert, Seymour (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York.
- Pinson, Lewis J., and Richard S. Wiener (1988). *An Introduction to Object-Oriented Programming and Smalltalk*. Addison-Wesley, Reading, MA.
- RMG, *A Tool Kit for Development of Visualization Courseware* (1989). Hewlett-Packard, Cupertino, CA.
- Schiefler, R. W., J. Gettys, and R. Newman (1989). *X Window Systems: C Library and Protocol Reference*. Digital Press, Maynard, MA.
- Shneiderman, B. (1982). The Future of Interactive Systems and the Emergence of Direct Manipulation. *Behavior and Information Technology*, Vol. 1, 237–256.
- Shneiderman, B. (1983). Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer*, Vol. 16, No. 8, 57–69.
- Smith, Randall B. (1987). Experiences with the Alternate Reality Kit: An Example of the Tension between Literalism and Magic. *IEEE Computer Graphics and Applications*, September, 42–50.

Exercises

- 7.1 Make a detailed list of command line interface commands that you frequently use. Now, create on paper a user interface using direct-manipulation principles that provide the same functionality. Are there any problems in providing the same functionality with a direct-manipulation interface?
- 7.2 Describe how copying files can be accomplished in a direct-manipulation interface.
- 7.3 There has been considerable research conducted with the objective of giving machines the capability for natural-language understanding. Compare an interface that can understand plain English to a graphics interface. Which types of applications would be most useful for both types of interfaces? Consider aspects such as response time, user capabilities (e.g., typing skills), and appropriateness.
- 7.4 As mentioned in the chapter, there are many different kinds of pointing devices that have been used. Consider and compare the use

of a mouse to the use of a touch screen. Imagine that moving your finger around on the surface of the computer screen would permit you to conduct the same operations as you conduct with the mouse. Would you prefer this type of interface?

- 7.5 Investigate images further. Try to display images on the screen at various locations. Write a routine that will permit an image to follow the cursor around the screen. One can detect the location of the cursor with *WindowSensor cursorPoint*. Use the **VisualComponent** method *follow:while:on:* and display the result in the workspace you use for the test code, or in a workspace you generate yourself. (*Hint:* Proceed in the same way as the example for displaying images and use the previously mentioned animation method.)
- 7.6 Suppose you were going to write a simulation of customers moving in a bank. Draw a sketch of how the bank would look on the screen and write the code needed along which customers would move in the simulation.
- 7.7 Look up the referenced article by Smith on ARK. Read and review the article and consider what other types of user interface mechanisms might be used.



Tools for Building Applications: The Model-View-Controller Paradigm and View Components

8.1 Model-View-Controller Concepts

This chapter describes the major interface tool available to the user for building applications in Smalltalk-80—the model-view-controller (MVC). The initial design concept for creating the model-view-controller (Krasner and Pope, 1988) was the factoring of conceptual entities when building applications. Typically, it is easy to understand how to break down the components of an application into three easily recognizable parts: the *model*, that part of the application that deals with the information to be utilized; the *view*, the presentation of the application on the display screen; and the *controller*, the code that controls the interaction of the system with the user. Example code contained in this chapter was created with Release 4 of ST-80.

The Basic MVC Concept

Figure 8.1 provides a conceptual framework for organizing a description of the MVC. As discussed in Chapter 7, the user is linked to computer system input via the mouse and keyboard and output is conveyed via the computer screen. The task of the controller is to interpret input from the user, track the mouse movements, and handle interactions with views and models. The model houses information, for example, collections of data points, images, or rules, whereas the view deals with information displayed in windows on the screen. This three-way factoring is used for building most applications in ST-80. Each view may be displayed in a window on the screen either by itself or as one

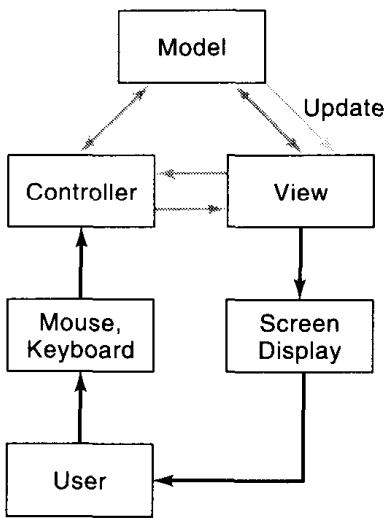


Figure 8.1 A Conceptual Framework for the Model-View-Controller. Dark hatched arrows are messages between MVC components. The broad hatched arrow is an update message from the model to the view. Nonhatched arrows indicate connectivity between the user and the MVC.

of several views arranged within a window. Views and controllers are directly linked and may directly access information about each other. Models, however, are more isolated from views, yet can indirectly inform views that the information they contain should be displayed. Models may be linked to many different view/controller pairs so that information can be displayed in different ways. The central idea is that the model maintains a collection of dependents, that is, things that are dependent on it. There can be different types of dependencies; for example, in a class for a push button, pushing any button will cause any other dependent button that is “on” to turn “off” (i.e., the buttons on a car radio). Similarly, there may be several views that are changed in some way when information in a model is changed.

As an example, consider a model that is simply a collection of data points, implemented perhaps as simply as an object with an instance variable initialized to a new orderedCollection of length 1024. Two views on the screen might be implemented, one to show the raw data and one to display the fast Fourier transform of the data, for example. If a user changes the data in the model by sampling additional points, then presumably both the plot of the original data on the screen and the FFT of the data would need to be changed. This updating of information on the screen is accomplished by creating a collection of dependents of the model—in this example, the two different views of the data. Whenever data changes, the dependent views should be automati-

cally changed, as well. This concept can be extended indefinitely to create as many dependents as needed to display different views on the screen.

Models

Model is a subclass of **Object**, adding the instance variable “dependents.” The collection “dependents” holds all the dependents of a model that the user specifies. These dependencies are created when an application is initially opened. Whenever information in the model is changed that can affect the view, the user can supply code of the form:

self changed

or

self changed: aSymbol.

or

self changed: aSymbol with: aParameter.

in any method in the model. Any of these methods results in an *update* message being sent to the view that is linked to the model. The typical method writing approach for the model is to write all the methods that implement the model and to include *changed* messages whenever what is to be shown in the view should be redisplayed. The final *changed* message listed previously is ultimately sent as a consequence of sending either of the first two messages. Sending any *changed* message results in the sending of an *update:with:from:* message to the *dependents* of the model. The argument in the “from:” part of this message can be used to indicate from which object the change comes. The utility of passing the argument “aSymbol” to a view is to permit selective updating of the visual information displayed in the view. The view should detect what changes are to be made by examining “aSymbol” and then taking an appropriate action. The dominant use of “aSymbol” is to permit differentiation among multiple views displayed. For example, imagine that an application has several different views. Whenever a change in some part of the model that is relevant to one or more

views is made, an aspect symbol can be specified in the view that, when detected, permits the view or views to be selectively updated.

Views

Each view has just one controller; yet multiple view/controller pairs can be linked to a single model. When an *update* message is received by a view from a model, the “update” method in the view should call *displayOn: aGraphicsContext* method to change the visual appearance of the view. Users should implement the *displayOn: aGraphicsContext* method in each view displayed. Because there may be only selected views or parts of a view in a window that need to be redisplayed when the model is updated, redisplaying everything in the entire window is often unnecessary. As an example, consider a simulation which requires the display of a small image at various locations on a larger image. A traffic simulation is an example in which a small image depicting an automobile is displayed on an exit ramp. If the model contains the information about where the car is to be located, then each time the model changes, the image representing the car should be moved to a new location on the background image, which does not change. Rather than redisplaying the entire image at each update, it would be useful to implement code in the *displayOn:* method of the application view that simply saves and restores the background behind the image displaying the automobile and moves the small automobile image. The time savings comes from only having to manipulate a small part of the display rather than the entire background. Another simple example would be for the display of gauges; when a value is changed, only the indicator part of a gauge needs to be redisplayed. In summary, the ‘*changed: aSymbol*’ → ‘*update: aSymbol*’ messages permit selective updating of a view. The model can indicate that only some aspect of the model (denoted by *aSymbol*) has changed. By passing this information to the view, via *update: aSymbol*, one can cause the view to selectively update itself.

Controllers

Controllers coordinate models and views with user input. There is little inherent behavior in class **Controller**, except for sensing whether the cursor is inside a view and whether the mouse buttons are pressed. **Controller-WithMenu** adds the capability of associating menus with mouse buttons. The default behavior of the controller is to initialize, enter a control loop, and finally terminate. Figure 4.6 described some of this behavior. Subclasses of controllers

can implement many different types of behaviors. For example, different types of menus can be specified, various actions can be taken when buttons are pressed or released, text editing control can be specified, and list control can be provided. Control regimes that are not common can be implemented, as well. For example, when a view appears on the screen, the cursor could be displayed as a helpful graphic symbol positioned at a predetermined point that helps the user understand what to do next. Also, subclasses of controllers could be implemented that would track the cursor and record what the user is doing. This capability would be useful in tracking the characteristics of the user so that the system could adapt its presentation format to the needs of the user.

Windows

The class **ScheduledWindow** provides the interface to the host window manager system. The capabilities of the window manager are specific to each host platform; hence, the visual appearance of the window will vary between platforms, even though the code is completely portable among platforms. Views live within the host window and are installed in specified areas of the window. Each view requires a *wrapper*, or set of code that implements different view functionality. For example, views can be wrapped to have borders (**BorderedWrapper**) and scroll and menu bars (**EdgeWidgetWrapper**). The capability of permitting applications to move from one host platform to another, adopting the interface look and feel of each new host platform, is a particularly appealing feature of ST-80.

Next, three methodologies for building MVC triads are discussed in general, followed by a description of view components. Then, concrete examples of each of the three methodologies will be presented.

The Build-It-Yourself Method

The most direct, yet most tedious, technique for building MVC triads is to write the code yourself for specifying the characteristics of the linkages between views, controllers, and models. The alternative is to utilize some type of more automated technique, two of which will be described in the following discussion. The technique utilized for building a MVC triad yourself is to specify all the relationships in the triad, as well as information about the window within which the triad lives.

The first job is to specify the window, normally in the “open” or “openOn:” class method, which can exist in the view or model. For example,

open

“ClassName open”	
window compositeParts	“temporary variables for this”
	“method”
window := ScheduledWindow new.	“the <i>new</i> message creates a new”
	“instance”
compositeParts := CompositePart new.	“create a place to hold the views”

“.... next add the views to different composite parts....”

window component:	“associate the parts with the”
compositeParts.	“window”
window open	“open the window”

Typically, to create an instance of a view and install the view as a composite part, both the model and controller are specified. For example, prior to opening the window in the preceding example, the view should be created and added to the compositePart description.

```
drawingView := DrawingView new
    model: aDrawingModel;
    controller: aDrawingController new.
```

```
compositePart add: drawingView borderedIn: (0@0 extent: 0.3@0.1).
```

In this example, drawingView is an instance of a particular class of the same name (note that the capital letter distinguishes the two). By sending *model: aDrawingModel* to the view, the model to view linkage can be established. In the same way, the view to controller link is directly specified. For each view that is to be displayed in only a portion of a window, it is necessary to tell the window where the view will be displayed, as indicated previously. In this example, the drawingView will be shown with a border in a small area in the top left of the window (as indicated by the argument to *borderedIn:.*)

The preceding example, which will be amplified in some detail in the following discussion, shows only one of the techniques in which the controller and model of a particular instance of a view are explicitly set up by sending the methods *model:* and *controller:* to an instance of a view. There are other

techniques as well; for example, in most cases a default controller can be used. Most view classes have prespecified controllers which are included if the user does not specify a new controller class.

The PluggableView Method

In the ST-80 system, there are several examples of so-called “pluggable views.” Pluggable views simplify building user interfaces by permitting the builder of an application to create instances of a variety of specific kinds of views for an application simply by passing parameters (i.e., “plugging parameters in”) to the view class. Several of these pluggable views will be explained in the following discussion.

ViewBuilder Methodology

The third technique for building views, including basic models and controllers, is to use a graphics interface which permits graphically sizing and distributing different types of views on a graphic prototype of a new window being built. The technique requires no programming knowledge. A user can create a new view by sizing a window on the screen and can add various pluggable view components just by selecting items from a list. This methodology is the simplest of the three techniques described; one drawback, however, is that because no knowledge of Smalltalk is required, the application builder may not feel the need to fully understand the MVC methodology. The use of viewBuilder is described at the end of this chapter.

View Components and Their Capabilities

The next sections of this chapter deal with specific types of prebuilt components in ST-80 which can be used for building user applications.

8.2 The View Framework

Figure 8.2 displays several different views that often might appear as composite parts in an application in a window. This view shows a gauge, a button, a switch, a selection-in-list, a text editor view, and a transcript. The use of each of these views is described in some detail in the following discussion. This figure will also serve as an example for the viewBuilder methodology

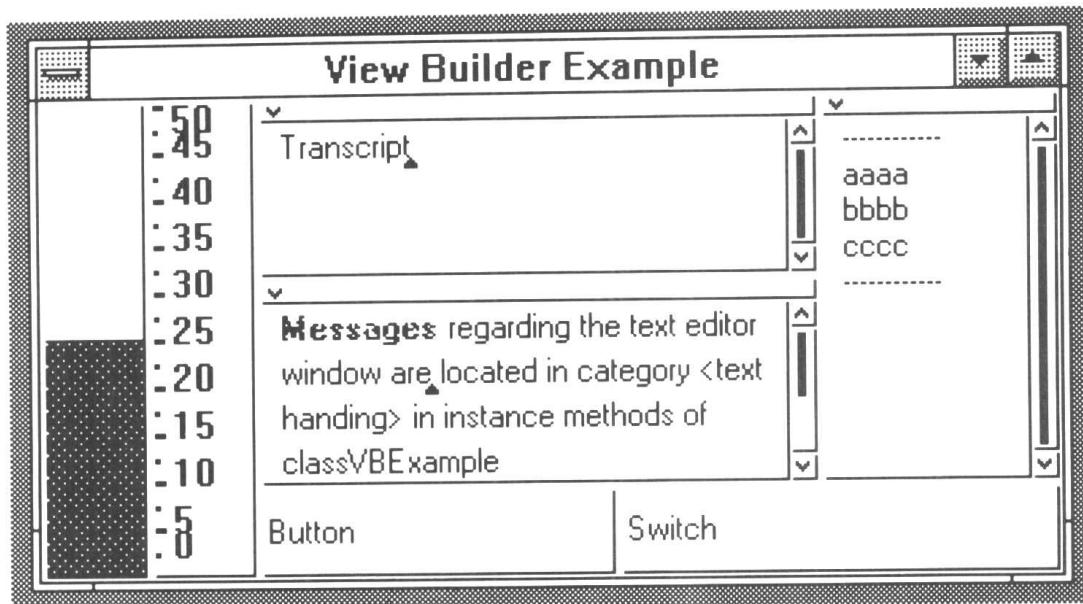


Figure 8.2 Six Example Views in a Window. Reading clockwise from left: gauge, transcript, list, switch, button, and text views.

described in the latter part of the chapter. The task of the programmer is to create applications that typically consist of either single or multiple windows filled with views, some of which may be coordinated with the models linked to view/controller pairs.

To implement a new application view, the standard methodology is to create a class method in the view or model class of an application, usually called “open” or “openOn: anObject.” As indicated previously, an instance of the class **ScheduledWindows** is created in this method to provide the interface to the host operating system and views added one by one as compositeParts of the window.

The message *openOn: anObject* can be used in preference to *open* to permit passing “anObject” to the opening of an application. This type of opening message can be used when data is passed to the application. For example, an application could be envisioned that displays the contents of any dictionary. In this type of example, the application could be opened on the contents of different dictionaries to display different contents each time the *openOn: aDictionary* message is sent to the application class. If *openOn: anObject* is sent as a message to a view class, it is useful to let the object passed as the parameter to *openOn:* be an instance of the model so that model methods and values are easily accessible to the view. In contrast, if “openOn:” is a class method in the model, the argument to *openOn:* might well be

“self new” to permit the remaining code in the opening method to access an instance of the model. For example,

open	“A method in the model”
“ExampleClass open”	“A comment that runs the example”
openOn: self new	“Code that calls the” “openOn: method” “with” “an instance of the model” “as an argument”
openOn: anInstanceOfanExampleClass	
...	
window open	

In this example, the utility of passing the argument *anInstanceOfanExampleClass* to the *openOn:* method is to have access to the instance variables of the class within the *openOn:* method. It is also worth noting that the creation and opening of windows can be accomplished either in a model class or a view class. The advantage of opening an application in a model class is that it is somewhat conceptually clearer to (1) create the view from the model and then (2) only operate within the model, sending *update* messages whenever the view should update itself.

Creating Views in Version 2.5

The creation of application views in Version 2.5 of ST-80 is quite similar to the method for creating views within the host windows of Release 4 as described previously. Next, the basic scheme for opening views in Version 2.5 is discussed for historical interest. Readers with no need to understand the relationship of Version 2.5 views and Release 4 views can skip the next paragraph.

To implement a new application view, the standard methodology is to create a class method in the view class, typically the “open” or “openOn: anObject.” The *open* method normally contains a definition of the *topView* as a **StandardSystemView** with information about the border, the label, and the model, if any. A “*topView*” is roughly equivalent to a window in Release 4 and the class **StandardSystemView** provides windowing protocol for subclasses. Subviews are then individually defined and added to the *topView*. Subviews correspond to window components in Release 4. When the programmer completes these definitions, the controller for the *topView* is opened to create the

view on the screen. The general form typically followed is

open	“A class method in a view class”
“ClassNameView open”	
topView	“local variable”
topView := StandardSystemView	
model: nil	“no model specified”
label: ‘a label string here’	“a label”
minimumSize: 50@50.	“specify the minimum” “size window”
topView borderWidth: 2.	“the width of the view border”
topView addSubView (GClockView new)	“add a clock”
in: (0.0@0.0 extent: 1.0@1.0)	“the location in the view”
borderWidth: 1.	“the border width”
topView controller open	“open the view”

In this simplified example, topView is a **StandardSystemView** with no model. Any label can be used and the minimumSize set to give the minimum size window when the view is opened. In this example, a new graphical clock¹ is created and added to the topView, covering the entire extent of the topView. Finally, topView is sent the message “controller” to get the default controller for the view and “open” is sent to the controller to open the view.

8.3 Selection-in-List

For selecting items, a pluggable view implemented in the class **SelectionInListView** is available in ST-80. A selection-in-list provides the capability of presenting to the user a scrollable list of items, from which any item can be selected using the mouse. When a selection-in-list view is opened, the user plugs into the opening method information about the model in which information resides and what to do when the user operates the list. For example, when the user makes a new selection, both the model and the methods within the model that take action when the selection occurs must be designated. The pluggable view approach provides a very useful interfacing technique which bypasses the need for the user to create explicit MVC linkages. Instead, these linkages are built by the code within the pluggable view. As an example, the creation

¹Note that **GClockView** was a Version 2.5 class that used a pen to create a display of a clock. This class does not exist in Release 4.

message used for class **SelectionInListView** is

on:aspect:change:list:menu:initialSelection:

An instance of a selection-in-list view is created by sending this message to the class **SelectionInListView**. Then, the instance view created is added as a composite part to the window. For example, consider the following list that is built to show a list of pictures to display. Each of the arguments for the *on:aspect...* message is explained later.

```
listView := SelectionInListView
    on: aModel
    aspect: #picture
    change: #picture
    list: #pictureList
    menu: #pictureMenu
    initialSelection: #pictureSelection
```

The argument to “on:” is the object to which messages are sent by the remaining arguments in the “on:aspect...” message. Note that if “aModel” were replaced by “self,” any messages would be sent to the same class or class instance in which this code appeared. “Aspect” refers to which aspect of an object is of concern. For example, an object containing several types of information could be passed to “on:.” The “aspect” might then refer to one of the object’s internal data structures or instance variables. In the case of **SelectionInListView**, the aspect selector (i.e., the symbol used to make a selection; in this case, “#picture”) can be used as the argument for updating something connected to the list. The argument to “changed:” is the selection that the user makes from the list that is passed to the method (in this example, “picture: aSelection”) specified in “aModel.” Each time a new item in the list is selected by pointing and clicking, “picture: aSelection” is sent to aModel in this example. The argument to *menu:* is the message that is sent to get the menu and the argument to *initialSelection:* is the message to be sent to select one of the items in the list, when the list is first created. If “nil” is the argument to *initialSelection:*, no initial selection will be made. *pictureList* is the method that returns a collection of items that is to be displayed in the list and *pictureMenu* returns a menu that is to be associated with the operate key of the mouse. The

“#” signs denote the named items as symbols. An example of the use of **SelectionInListView** is given later in the chapter.

8.4 Buttons and Switches

Buttons and switches can be created by sending *new* messages to the class **LabeledBooleanView**, a subclass of **BooleanWidgetView**. Different functionality for buttons and switches is provided, such as checkboxes, radio buttons, simple switches, and toggle switches. A button has the functionality of setting the model linked to it (a *valueHolder*) to be true only while the mouse *select* button is pressed. In contrast, a switch sets the model to be true or false on successive presses of the switch.

An example of creating a button is as follows:

```
aButton := LabeledBooleanView new.      "provide true and false output"
aButton beTrigger.
aButton controller beTriggerOnUp      "provide a state change when"
                                         "the user releases the button"

aButton beVisual: 'THE TEXT TO BE DISPLAYED' asComposedText.

aButton model:
((PluggableAdaptor on: aModel)           "specify the view-model link"
 getBlock:                                "the message to send to change"
   [:model | model getMethod]
 putBlock:                                "the button"
   [:model :value | model putMethod]
 updateBlock:                            "when the button is pressed"
   [:model :value :parameter | false])
```

To add this button to a container, one could write:

```
container add: aButton
in: (0.0@0.0 extent: 0.1@0.1).
```

within the opening method prior to opening the window. This code will install the button in the upper left part of the window with *x* and *y* directions extending to 10 percent of the window size in the *x* and *y* directions.

The class **PluggableAdaptor** is used to create an interface between view/controller pairs and a model, in this case, the button view/controller and the model. A model is passed through the adaptor (*aModel*) which contains the

methods mentioned in the blocks, that is, “getMethod” and “putMethod.” The code executed by sending *getMethod* to the model is used to acquire the current state of the model. This knowledge might be useful in dynamically adapting the appearance of the button, depending on information in the model. The method referenced within the *putBlock:* will be executed whenever the button is pressed. Finally, the *updateBlock* arranges for updates from the model, passing the model, an update aspect, and an update parameter. In this case, the value returned in the update block is always false, indicating that no updating is to be done.

8.5 Transcripts and Text Editors

A transcript permits a program to output information as text to a view. An example is the system transcript to which messages can be sent. The system transcript is useful for debugging purposes, in that messages such as “Transcript show: ‘message about an error’ ” can be placed in different parts of a program. The global variable “Transcript” is bound to the system transcript. To create one’s own transcript or multiple transcripts for showing different things, the code is straightforward:

```
aTranscript := TextCollector new.  
aTextCollector := TextCollectorView open: aTranscript label:  
    'FFt Information Window'.  
aTranscript show: 'Processing is starting now'.
```

By declaring the instance of **TextCollectorView** as global, the **textCollector** can be sent information from any part of an application. Furthermore, the **textCollector** view can be installed as a **compositePart** in an application window, as needed.

A text editor can be created using the **CodeView** class, as follows:

```
aTextEditorView := CodeView  
    on: aModel  
    aspect: #initText      "get initial text to show"  
    change: #acceptText   "method to run"  
    menu: #textMenu        "when text change accepted"  
    initialSelection: nil. "specification of menu"  
                           "no initial selection"
```

Opening class **CodeView** on “aModel” means that the methods referenced by the selectors specified in *aspect:change:menu:initialSelection* should be within

object “aModel.” Other objects could be specified as the argument to “on;,” for example “self.” By specifying “self,” the user would explicitly indicate that the methods specified as arguments to the **CodeView** would be within the class or class instance in which the **CodeView** creation occurred. Once specification of this **CodeView** instance is made, it can be added as a view to the window. The instance of **CodeView** created permits the standard text editing operations including: copy, cut, paste, and so on.

8.6 Building Menus

PopUpMenus

PopUpMenu is an easy-to-use class that gives the programmer the capability of creating pop-up menus with a simple class specification as follows:

(PopUpMenu labels: ‘first item \ second item \ third item’ withCRs) startUp.

Highlighting this code and executing “doit” will produce a menu. Execution (i.e., clicking on an item in the list) will return a number equivalent to the position in the list, starting with 1. In this example, carriage returns are added after each item in the string “first...item” by sending “withCRs” to the string.

Dialogs

A set of classes are provided in ST-80 that make it easy to build dialog boxes. For example, there are prefabricated classes for creating yes-no prompters, fill-in-the-blank (one and two lines) prompters, and labeled menus with scroll bars. These dialog boxes are of the pop-up variety, appearing on demand and disappearing after use. The main types are created as follows:

Yes/No Prompter:	DialogView confirm: ‘Is the voltage greater than 5V?’.
FillInTheBlank:	DialogView request: ‘What is your name?’.
Menu:	DialogView show: aMenu withLabel: ‘Pick one of the following’.
ButtonMenu:	DialogView showChoice: aMenu withLabel: ‘Pick one of the following’

In the latter two types, aMenu could be a **PopUpMenu** created as shown

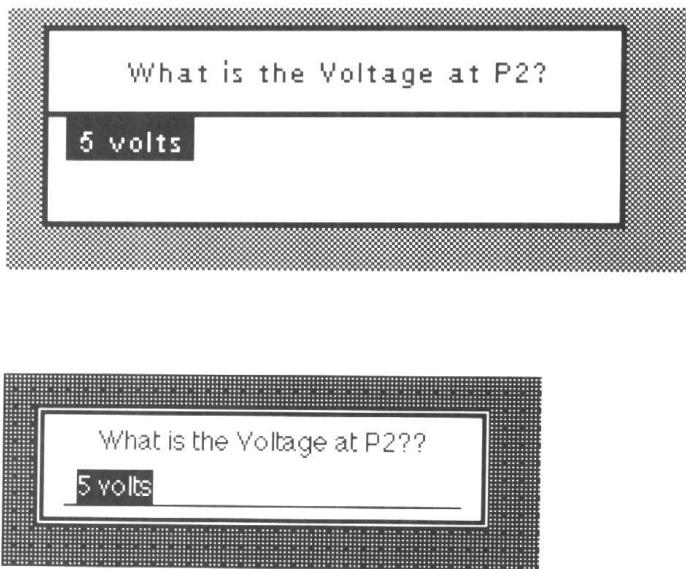


Figure 8.3 **FillInTheBlank** Pop-up View and **Dialog** View.

previously without the *startUp* message. These dialog views are similar to the **FillInTheBlank** class implemented in earlier releases of ST-80 (Version 2.5 and earlier). **FillInTheBlank** is found in backward compatibility fileIns for Release 4. Figure 8.3 shows the results of issuing the code to create an instance of this class:

```
FillInTheBlank
  request: 'What is the Voltage at P2?'
  displayAt: Sensor waitButton
  centered: true
  action: [:response | Transcript show: 'response']
  initialAnswer: '5 volts'.
```

The request in this example is a question asked the user. The pop-up window is displayed at the cursor (called a sensor) as soon as a button is pressed with the cursor centered in the view. The information that the user types is passed to the *action: aBlock* shown. For this block, the response is simply typed on the transcript. One can also specify an initial default answer as shown. Also shown in Figure 8.3 is the same style **FillInTheBlank** prompter implemented using



Figure 8.4 **LauncherView** with User Additions.

DialogView in Release 4. The implementation is somewhat simpler than the previous implementation; that is,

DialogView request: ‘What is the voltage at P2??’ initialAnswer: ‘5 volts’.

Hierarchical Menu

The methodology of creating a hierarchical menu was shown in Figure 7.11. This paragraph presents a method for adding to the hierarchical menu of the Launcher. Figure 8.4 shows a modified **LauncherView** with three additional entries that can be used to launch the applications named. The method for modifying the Launcher is to edit, accept, and reinitialize code in the LauncherView. Specifically, the method *initializeLauncherMenu* can be easily modified to add additional items to the Launcher list. Although the code in this method is implemented using PopUpMenus, a hierarchical menu could be just as easily used. Customizing the Launcher is an excellent way to organize the environment.

8.7 Using Forms

A form is a black-and-white image that contains bits which, when set, indicate where black and white dots are shown within a rectangle displayed on the screen. Forms are a display technique used with the ST-80 environment in Version 2.5 and earlier. As indicated in the previous chapter, forms have been replaced with images to provide color capabilities in later releases of the ST-80

environment. However, forms can still be used in the later releases. A form created in an earlier version can be read from a file and converted to an image and then displayed on the graphics context² of the window. For example,

```
aForm := (Form readFrom: 'and.frm') asImage.  
aForm displayOn: aView graphicsContext borderedIn:  
    (0.5@0.0 extent: 0.3@0.5).
```

8.8 Using Images

Release 4 of Smalltalk-80 was the first release of the environment to support commonly used pixel-based images. As indicated previously, prior to 1990, ST-80 used forms, consisting of rectangular displays of on-off bits, to provide graphics. Unfortunately, using images which permitted only turning a bit on or off at each pixel precluded the use of color, which requires multiple bits at each pixel. Hence, a change was made to include color in Release 4 and future releases.

The ST-80 Environment supports a hierarchy of classes to support the creation of images:

```
Object  
  VisualComponent  
    Image  
      Depth1Image      "equivalent to a Form"  
      Depth2Image  
      Depth4Image  
      Depth8Image  
      Depth16Image  
      Depth24Image
```

The depth in these classes refers to the number of bits per pixel. One bit per pixel is the same as found in a black-and-white form. Two bits permits storage of 4 different colors, four bits, 16 colors, and so on. One can create images by sending class **Image** messages, for example,

```
Image extent: anExtent depth: aNumber palette: aPaletteOfColors
```

where **extent** is the size of the image, **depth** is {1, 2, 4, 8, 16, or 24}, and **palette**

²A graphics context contains parameters that affect graphics operations.

permits specification of the colors in the image. Another alternative is to simply use the *fromUser* message to class **Image** to grab a form from the screen. Of course, to use the different **Depth*Image** classes with *fromUser*, one should know the depth of the image that can be represented on the screen of the host platform.

8.9 Building and Using Icons

Icons are small graphics representations that are used to provide the user with an abstract reminder about information underlying an icon. Icons are predominantly used in ST-80 for representing what is in a collapsed window. For example, an icon of the system browser might be a drawing that has the miniaturized appearance of a tree structure. Alternatively, an icon might be simply a textual name inserted into a box. In ST-80, when a window is collapsed, an icon appears and can be moved to different locations on the screen. Thus, one can keep on the desktop many collapsed windows that are readily accessed simply by clicking on the icon. Icons have different appearances depending on the platform on which ST-80 is running. For example, on the Macintosh, the standard icon is simply the name in the label field of the window. On other platforms, graphics representations of icons can be created that give a visual cue about the contents of the collapsed window.

One way to create an icon is to duplicate the methods outlined in the **Icon** class in methods such as *createProjectIcon* or *createDebuggerIcon*. In these icons, individual bits in hexadecimal form are specified (e.g., “16rFF” represents the number FF_{16}). Examine one of the preceding methods in the browser and compare the hex-specified bits with the icon that appears when a window is iconified. Already included in the system is a dictionary of IconConstants for all the window types of ST-80. To create a new icon, a reasonable example in the system to follow is the Launcher icon. The method *createLauncherIcon* in **Icon** retrieves a depth1 image which is created in the method *initializeGatewayImage* in the **LauncherView** class. If one examines the way that the Launcher icon is created by specifying 1's and 0's in radix 2 eight-bit representation (e.g., “2r00001111” represents 00001111_2), the icon of the Launcher on the screen can be observed to map directly to the bits shown in the *initializeGatewayImage* method. Hence, to create a new icon, all that is necessary is to edit this type of bitmap and install it into the IconConstants dictionary, following the methodology shown for the other icons. To test what an icon would look like, one can use code in a workspace such as

```
ScheduledControllers activeController view icon:  
(Icon image: (Icon createTestIcon)).
```

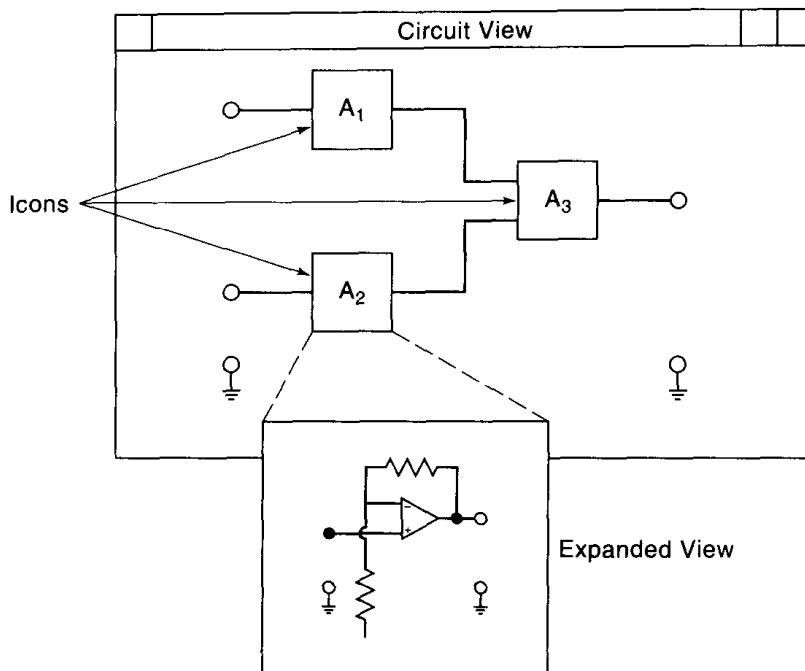


Figure 8.5 Using Icons in a Circuit Representation.

where *createTestIcon* is a method similar to *createLauncherIcon* except that a different test image should be created by the user following the image creation method outlined in the **LauncherView** class. In this example, the code “ScheduledControllers activeController view” returns the view in the workspace in which the code was entered. Once the preceding code is executed (i.e., “doit”) in the workspace, the icon can be viewed by collapsing the workspace.

Not only do icons permit graphic representation of entities, the ability to expand collapsed views that have graphical identities has the potential for creating interesting user interfaces. For example, one might create views of hierarchical designs of electronic systems by creating collapsed views of electronic gates which, when expanded, would show the internal structure of the gates. Figure 8.5 shows an example of this concept.

8.10 Other View Components

Various other types of view components are both available and conceivable. Pluggable views can be written similar to the pluggable views available in the system itself. For example, Knowledge Systems Corporation (Adams, 1987) marketed a package called Pluggable Gauges with which one could create

many types of interesting gauges.³ For example, to create a bar gauge as shown in Figure 8.2, a pluggable view can be created as follows:

```
numberHolder := NumberHolder new value: 30.
```

```
BarGaugeWithScaleView
on: numberHolder
aspect: #value
change: #value:
range: (0 to: 55 by: 5)
orientation: #vertical
needleDirection: #right.
```

NumberHolder is a simple object that contains values and uses the methods “value” and “value:” to access and set the internal value of numberHolder. Note that the gauge is opened on numberHolder. This means that the selectors in *aspect:* and *change:* are found in numberHolder. The remaining three values for range, orientation, and needleDirection are passed to set up the BarGauge view. The point worth noting here is that the concept of pluggability is fully extensible; that is, most kinds of views that can be varied using different parameters could be set up in this manner.

8.11 Putting the Components Together

In the previous sections that have examined the use of different view components, only code fragments were shown. Two complete examples will be shown next to illustrate the direct and pluggable implementation methods.

Direct Implementation of the MVC

This example is concerned with creating a view with a pop-up menu which contains a list of different colors that can be displayed on the view. In this example, two colors are used: blue and red. Figure 8.6 shows the screen appearance of the window created and Table 8.1 gives the code that implements

³See the Appendix for information about securing a Release 4 gauge package based on Adams' work.

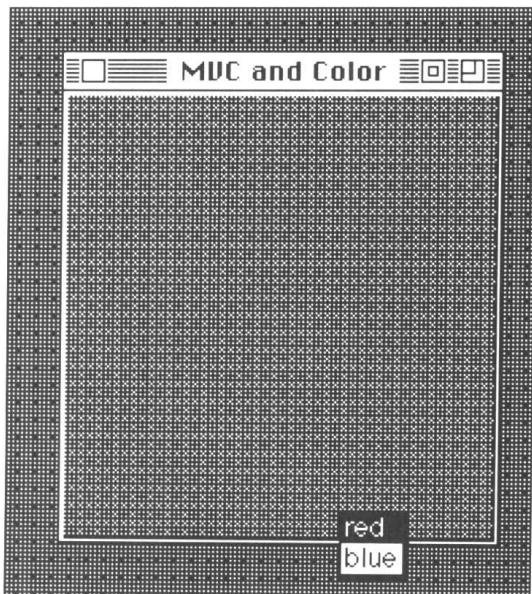


Figure 8.6 Example Use of the MVC Methodology for Showing Different Colors in a Window. When *red* is selected, the interior of the window turns red. When *blue* is selected, the interior turns blue. Note in this black-and-white printing that colors appear with different patterns (e.g., the interior and exterior of the window shown).

this window. The colors are produced in this black-and-white figure as different textures. Three classes are implemented: **DisplayDemo**, **DrawingController**, and **DrawingView**. The most direct way to understand the operation of the triad is to examine first the *open* method in **DrawingView** which follows closely the methodologies outlined in the previous examples. A model instance is created from the class **DisplayDemo** and a view instance is built which is linked to the model and the controller. A window is created as an instance of **ScheduledWindow** and opened.

In operating this example, the user selects an item from the pop-up menu associated with the controller. After the user chooses a selection, the choice is sent to the model from the controller. In the model, an image of the size of the display is created and the specified color created on a new image is stored in an instance variable called *displayImageSurface*. After drawing is complete, the view is notified that the model is changed (i.e., via the message *self changed*). The view that is linked to the model receives the information that the model has been changed via the *update:* message. In this example, *update:* simply retrieves the image to display and displays the image, using the *displayOn:* method on the *graphicsContext* of the view. Figure 8.7 graphically

Table 8.1 The DrawingView Example**The View**

```
View subclass: #DrawingView
InstanceVariableNames: 'displayImage'
classVariableNames: ''
poolDictionaries: ''
category: 'book demos'
```

This example shows a basic model-view-controller set of classes for creating a pop-up menu in a view in a window which permits selecting "blue" or "red." When selected, the interior color of the view changes to the color selected.

DrawingView methodsFor: displaying**displayOn: graphics Context**

"displayOn: displays the image in the model in the view"

```
self model displayImage displayOn: self graphicsContext
```

update: graphicsContext

"update responds to the changed message from the model"

```
self displayOn: graphicsContext
```

DrawingView class

InstanceVariableNames: ''

DrawingView class methodsFor: instance creation**open**

```
"DrawingView open"
|aModel aView aWindow aWrapper| "define local variables"
aModel := DisplayDemo new.           "create the model instance"
aModel initialize.                 "initialize the model"
aView := DrawingView new.           "create DrawingView instance"
      setModel: aModel.          "connect the model and view"
aView controller: (DrawingController new). "make a controller"
aWindow := ScheduledWindow new.     "make a new window"
aWrapper := BorderedWrapper on: aView. "with a border"
aWindow minimumSize: 200@200.       "define minimum size"
aWindow label: 'MVC and Color'.    "the label on the window"
aWindow component: aWrapper.       "add the wrapper to the window"
aWindow open                      "open the window"
```

The Model

```
Model subclass: #DisplayDemo
InstanceVariableNames: 'displayImageSurface'
classVariableNames: ''
poolDictionaries: ''
category: 'book demos'
```

Table 8.1 *Continued***DisplayDemo methodsFor: drawing****drawBlue: size**

"Defines the image and color to display"

```
displayImageSurface := Image
    extent: size extent
    depth: 1
    palette: (MappedPalette with: ColorValue Blue with: ColorValue red).
    self changed "indicate that there has been a change in the model"
```

drawRed: size

"Defines the image and color to display"

```
displayImageSurface := Image
    extent: size extent
    depth: 1
    palette: (MappedPalette with: ColorValue red with: ColorValue blue).
    self changed
```

DisplayDemo methodsFor: retrieving**displayImage**

"returns the image to display"

[^] displayImageSurface

DisplayDemo methodsFor: initialization**initialize**

"Initializes this class"

"A the display is green when the Drawing view is first shown"

```
displayImageSurface := Image
    extent: 300@300
    depth: 1
    palette: (MappedPalette with: ColorValue green with: ColorValue blue)
```

The Controller

```
ControllerWithMenu subclass: #DrawingController
instanceVariableNames: ''
classVariableNames: 'Drawing YellowButtonMenu'
poolDictionaries: ''
category: 'book demos'
```

DrawingController methodsFor: access**menu**

[^] DrawingYellowButtonMenu

Table 8.1 *Continued***DrawingController methodsFor: menu items****drawBlue**

“draw a blue view”

self model drawBlue: self view bounds

drawRed

“draw a red view”

self model drawRed: self view bounds

DrawingController class

instanceVariableNames:”

DrawingController class methodsFor: initialization**initialize**

“DrawingController initialize”

“Defines the pop-up menu and the methods to associate with menu labels”

DrawingYellowButtonMenu := PopUpMenu labelList: #(#(#red #blue))
values: #(#drawRed #drawBlue)

illustrates the linkages between the various components in the MVC triad of this example.

Using Inspectors

A method for examining model-view-controller triads is to inspect the model, view, and controller instances that are used to create an application; for example, the instances of **DrawingView**, **DrawingController**, and **DisplayDemo**. Figure 8.8 shows the collection of inspectors for each of these MVC elements. Each window displays a dictionary with scrollable keys on the left and the contents of a selected key on the right in each window. The keys of the dictionaries contain information about each of the members of the MVC triad. For example, in the **DrawingView**, the value in the key “controller” indicates that the controller is a**DrawingController**. To inspect a running application, one can place “self halt” in the code somewhere and run the application. When a halt occurs, in the debugger, select the view, model, or controller in the object lists at the bottom of the debugger and, from the operate menu, select “inspect”

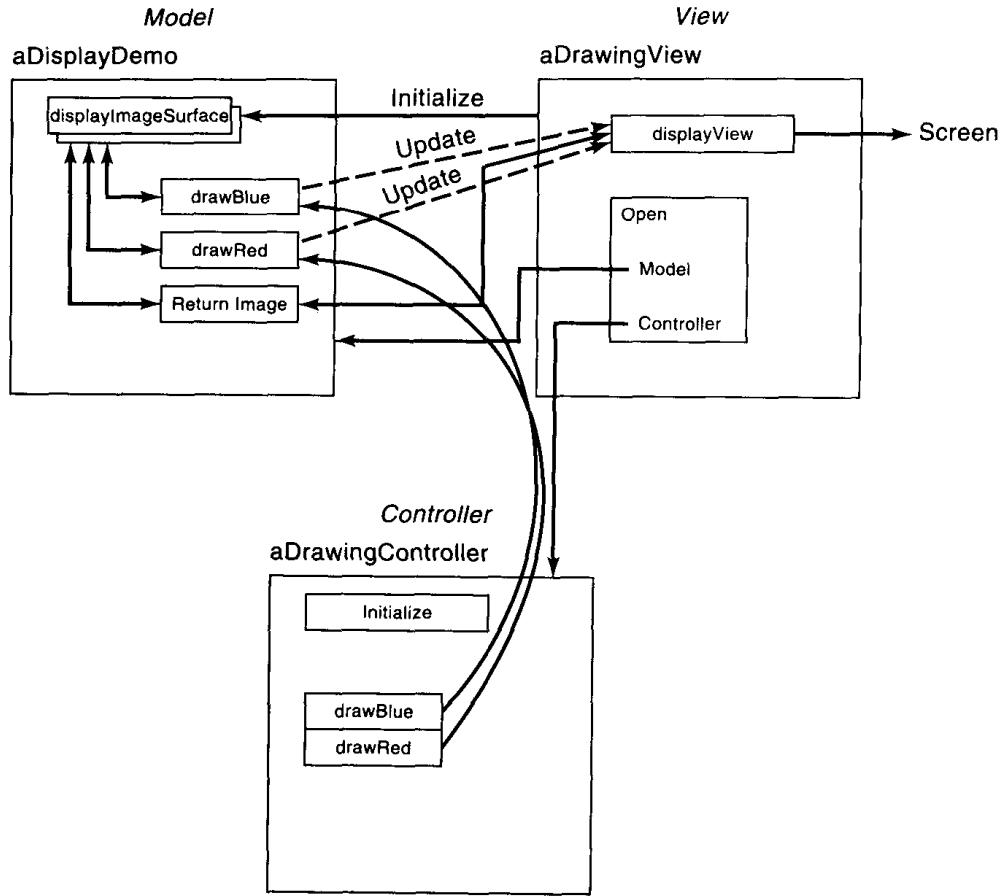


Figure 8.7 The **DisplayDemo** Example Model-View-Controller and Control Linkages. (Solid lines = messages, dotted lines = changed/update messages.)

to produce any or all of the MVC inspectors. These inspectors are useful in debugging and checking to see if all the proper linkages have been made.

Pluggable Implementations

A pluggable implementation is illustrated in Figure 8.9 which displays a window with two views: a **SelectionInListView** at the top and a **View** with a bordered wrapper at the bottom. The purpose of this example is to show how to display, in the view at the bottom of the window, a named image that is selected from items in the selection-in-list. The two images captured for this example were simply taken from the screen using *Image fromUser*. This example is

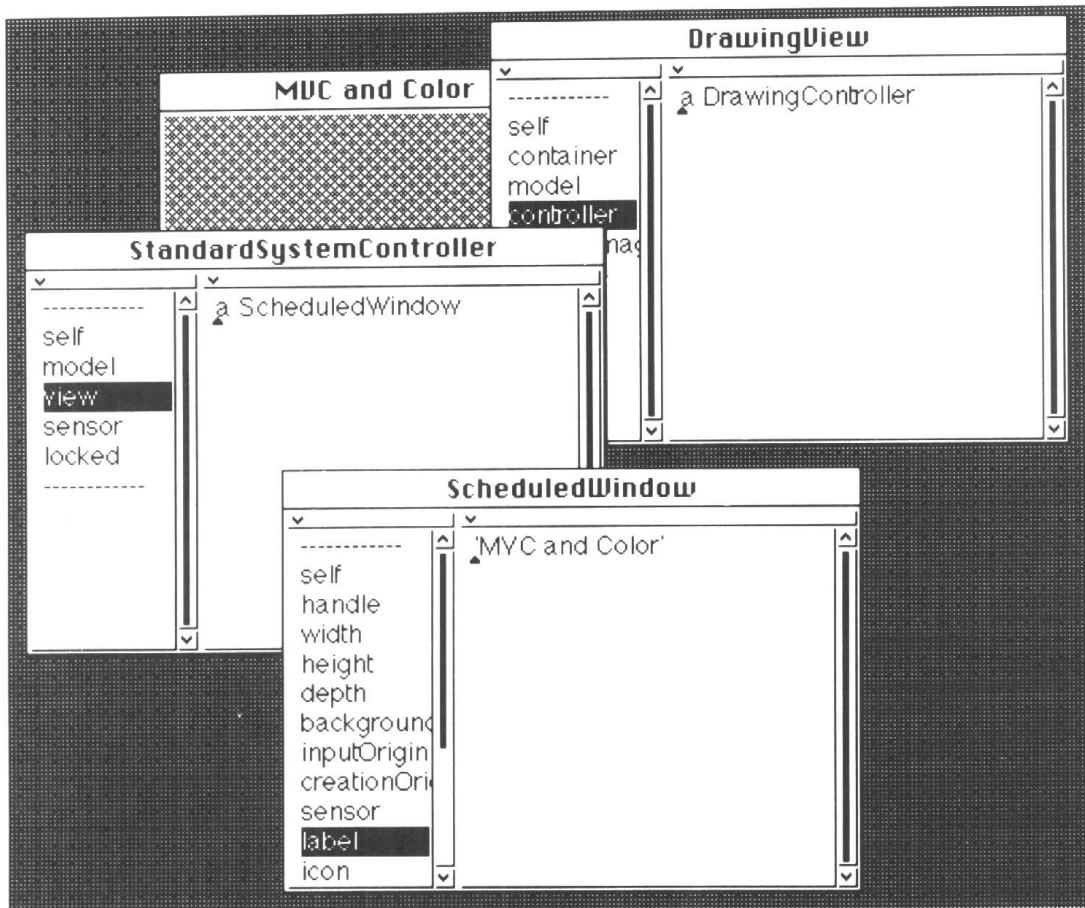


Figure 8.8 A Triad of Inspectors for Examining the Model, View, and Controller.

implemented in class **PictureView**, a subclass of **View**, and uses a global variable “PictureDictionary” to hold images. One could create a scrapbook of pictures using the technique. Table 8.2 shows the code used to implement **PictureView**. The most distinct characteristic of this code, as contrasted with the implementation of **DisplayDemo**, is the lack of explicit specification of the three parts of the MVC triad. Instead, the entire code is contained in this single class. The major work is done in the *openOn:* method. First, a window is created and the selection-in-list view built. Next, an imageView is created as an instance of **View** and both views added before opening the window. The selectors in the **SelectionInListView** refer to the remaining methods in the **PictureView** class, listed under the “menu items” protocol. An *update:* method is not used in this implementation because the selection of an item in the menu causes direct display of the image in the imageView graphicsContext each time a selection is

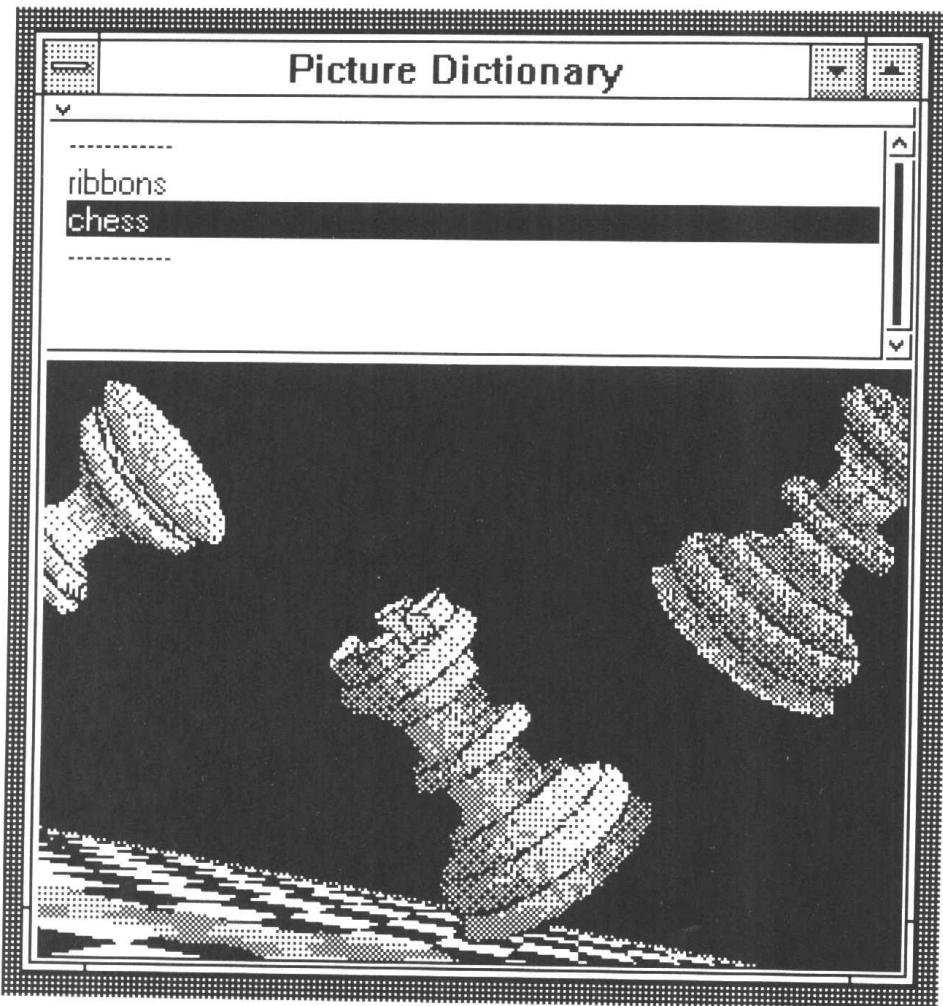


Figure 8.9 Plugging in Multiple Views: the **PictureView** Example. In this example, selecting “ribbons” or “chess” will display a picture of ribbons or chess pieces.

made. Note that without an *update* method, however, the image will disappear from the window if the window is moved on the screen. This problem can be avoided by simply adding an *update* method which contains the following code:

```
self picture displayOn: imageView graphicsContext.
```

By adding this code, when an *update* method is sent to the view due to, for example, moving the window on the screen, the current picture in the object will be redisplayed in the view.

It is, in general, simpler and more direct to use the “pluggable” methodologies rather than the direct implementation method of constructing an

Table 8.2 The PictureView Example

```

View subclass: #PictureView
  InstanceVariableNames: 'imageView currentPicture'
  classVariableNames:""
  poolDictionaries:""
  category: 'book demos'

PictureView methodsFor: menu items

picture
  ^ currentPicture

picture: aPicture
  currentPicture := PictureDictionary at: aPicture ifAbsent: [^ nil].
  currentPicture displayOn: imageView graphicsContext

pictureList
  ^ PictureDictionary keys asOrderedCollection

PictureView methodsFor: instance creation

openOn: aPictureDictionary
  "PictureView new openOn: PictureDictionary"
  |window listView aCompositePart|      "specify the local variables"
  window := ScheduledWindow new.        "make a new window"
  window label: 'Picture Dictionary'.   "with a label"
  window minimumSize: 300@300.          "define the minimum size"
  aCompositePart := CompositePart new.   "create a holder for the parts"
  imageView := View new.                "a new view"
  listView := SelectionInListView on: self "make a list"
    aspect: nil           "no aspect"
    change: #picture:    "send picture: when there is a change"
    list: #pictureList   "get the list from pictureList"
    menu: nil            "there is no menu"
    initialSelection: nil. "and no initial selection"
  aCompositePart add: (LookPreferences edgeDecorator on: listView)
  in: (0@0 extent: 1@0.3).             "make the list scroll and designate"
  aCompositePart add: imageView         "where it is to be located"
  borderedIn: (0@0.3 extent: 1@0.7).   "add the image to the composite part"
  window component: aCompositePart.    "add the view to the window"
  window open                      "open the window"

PictureView class
  InstanceVariableNames: ""

PictureView class methodsFor: initialize

setUp
  "PictureView setUp"
  PictureDictionary := Dictionary new. "make a new global dictionary"
  PictureDictionary at: #ribbons put: (Image fromUser).
  PictureDictionary at: #chess put: (Image fromUser).
  Logo := Image fromUser.

```

individual model, view, and controller. Following this line of reasoning, it is even simpler to use a graphical editor for creating the desired view. A building system for creating views has been constructed and is described next.

8.12 ViewBuilder⁴

ViewBuilder is a graphics programming system designed to permit users to create ST-80 view frameworks without having to create any code. First described by Jiang and Bourne (1991) in a Version 2.5 implementation, this system requires little knowledge about the MVC triad to create useful views that employ the view components described previously. Figure 8.10 displays the ViewBuilder interface for Release 4. Arrayed on the left side of the figure are buttons that initiate different actions including create, move, remove, resize, relabel, save, load, and compile. To create a new view, the user selects create and sizes the view to be built on the view to the right of the buttons. Next, an operate-button-connected pop-up menu permits selection of any of the types of views that can be added to the window such as **SelectionInListView**, **TextCollectorView**, or **CodeView**. The example in the figure shows building a window that contains six different types of views implemented in the system. The user simply selects the desired view type from the pop-up menu and sizes and places the view within the window. After selection of the view components is complete, the “compile” button is pressed. ViewBuilder takes the stored information built during creation of the placement of the types of views and inserts the information into sets of prototypical strings which are amalgamated to form the code of the completed view. Figure 8.2 is the result of the compilation of the skeletal form in Figure 8.10. The complete code for the view is generated and added to the ST-80 system, appearing in the system browser view as shown in Figure 8.11. The category in the browser is named according to a name given by the user when the ViewBuilder is opened. The classes built are designated with the name given by the user for the class with a prepended “V” and postpended “Controller” and “View” strings. The complete code built for this example is shown in Table 8.3. The reader is encouraged to compare the code in this table with the final view created as shown in Figure 8.2. Once this skeleton code is available, the programmer simply edits the code to create the application characteristics desired. For example, to add behavior to a button or switch, the user can simply pass arguments to the button or switch (see *self button:act:model:container:in:* in the *open:* method of **VBExampleView**) or specify the behavior in the model. Comments in the code give additional details

⁴See the Appendix for information on obtaining ViewBuilder code.

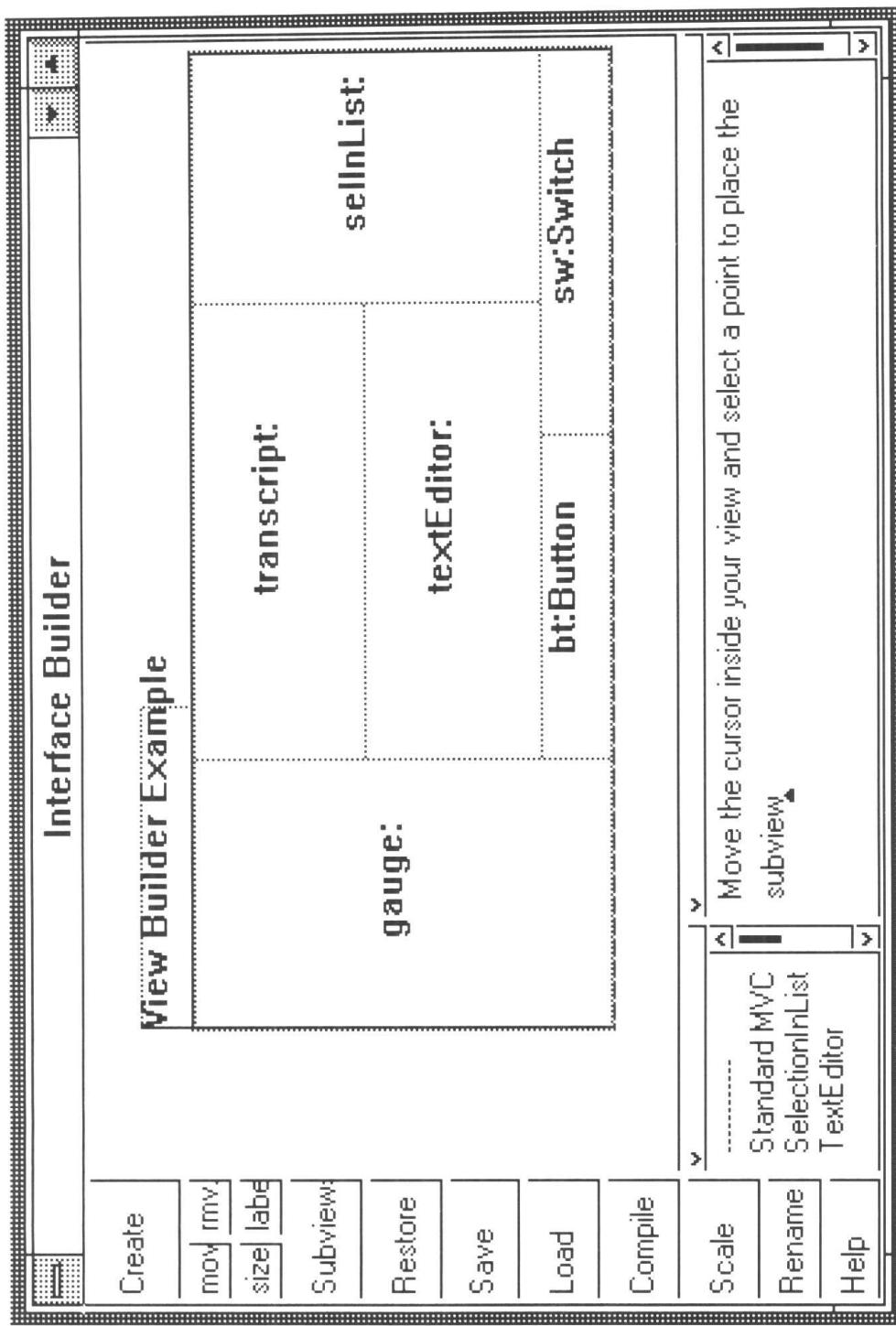


Figure 8.10 The ViewBuilder Interface.

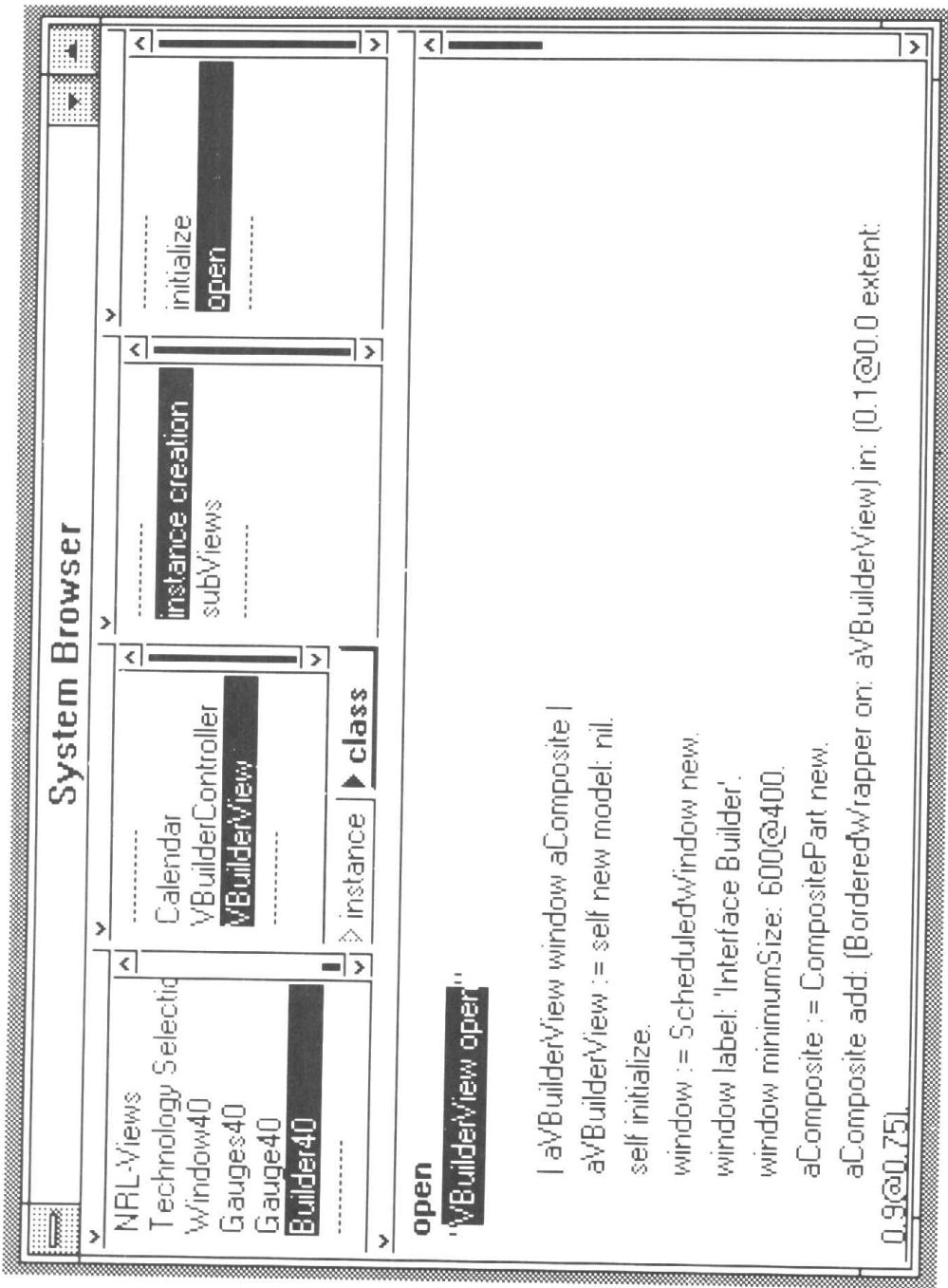


Figure 8.11 The Result of Compilation Using ViewBuilder.

Table 8.3 Code from ViewBuilder

```
View subclass: #VBExampleView
  instanceVariableNames:""
  classVariableNames:""
  poolDictionaries:""
  category: 'Builder-Example'
```

This class is the view class produced by ViewBuilder. The open method opens the application using an instance of VBExample, which is referred to by "aModel" in the open: aModel method.

VBExampleView methodsFor: private

defaultControllerClass

 ^ VBExampleController

VBExampleView methodsFor: displaying

displayOn: nothing

 "This method can be used to update views in VBExampleView. An update method with anAspect can be added in this Class to specify which aspect of the view to update, if desired"

 self displayText: ('Display object using method <displayOn: aGC> in category <displaying> in instance methods in class', self class yourself printString) as ComposedText

displayText: aText

 "This is how to display text"

 self graphicsContext clear.

 aText compositionWidth: self insetDisplayBox width.

 aText displayOn: self graphicsContext

VBExampleView class

InstanceVariableNames:"

VBExampleView class methodsFor: subViews

button: aLabel act: act model: aModel container: aContainer in: aRange

 "This message implements a button"

 "'act' is sent to 'aModel' in PluggableAdaptor"

 "'act' and 'aModel' are specified in the open: aModel method"

 "The 'act' can be specified in a method in VBExample"

 "aModel is an instance of VBExample-see open method in VBExampleView"

 |aButton|

 aButton := LabeledBooleanView new.

 aButton beTrigger.

 aButton controller beTriggerOnUp.

 aButton beVisual: aLabel asComposedText.

 aButton model: ((PluggableAdaptor on: aModel)

 getBlock: [:model| false]

 putBlock: [:model :value| model perform: act]

 updateBlock: [:model :value :parameter| false]).

Table 8.3 *Continued*

```

(aContainer
  add: aButton
  in: aRange
  borderWidth: 1)
  insideColor: ColorValue darkCyan

calendar: aContainer in: aRange
  "This method requires class Calendar"

  | aTextView|
  aTextView := CodeView
    on: Calendar new
    aspect: #calendar
    change: #calendar
    menu: #browseMenu
    initialSelection: nil.
  aContainer add: (LookPreferences edgeDecorator on: aTextView)
    borderedIn: aRange

digitGauge: aComposite in: aRange val: valRange initial: initVal

  | aDGView aDGGauge|
  aDGGauge := NumberHolder new value: initVal.
  aDGView := DigitGaugeView on: aDGGauge aspect: #value change: #value range:
  valRange.
  aComposite add: aDGView in: aRange borderWidth: 1.

gauge: aComposite o: orient t: type dir: dir in: aRange val: valRange color: aColor initial:
  initVal

  | aGauge|
  aGauge := NumberHolder new value: initVal.
  BarGaugeWithScaleView
    on: aGauge aspect: #value change: #value: range: valRange
    orientation: orient type: type needleDirection: dir composite: aComposite
    in: aRange color: aColor.

hvview: aModel container: aContainer in: aRange

  | aView|
  aView := HVScrollView new model: aModel.
  aContainer
    add: ((LookPreferences edgeDecorator on: aView) useHorizontalScrollBar)
    in: aRange.

label: aLabel container: aContainer in: aRange

  | aButton|
  aButton := LabeledBooleanView new.
  aButton beTrigger.
  aButton controller beTriggerOnUp.
  aButton beVisual: aLabel asComposedText.

```

Table 8.3 *Continued*

```

aButton model: (
    (PluggableAdaptor on: self)
        getBlock: [:model|false]
        putBlock: [:model :value|false]
        updateBlock: [:model :value :parameter|false]).
(aContainer add: aButton in: aRange borderWidth: 1)
    insideColor: ColorValue darkCyan.

list: aModel container: aContainer in: aRange

| aListView |
aListView := SelectionInListView
    on: aModel printItems: false oneItem: false
    aspect: #aspect change: #listChange: list: #list
    menu: #listMenu initialSelection: nil.
aContainer
    add: (LookPreferences edgeDecorator on: aListView)
    borderedIn: aRange.

switch: aLabel act: act off: act1 model: aModel container: aContainer in: aRange

| aButton |
aButton := LabeledBooleanView new.
aButton beTrigger.
aButton controller beTriggerOnUp.
aButton beVisual: aLabel asComposedText.
aButton model: (
    (PluggableAdaptor on: aModel)
        getBlock: [:model|false]
        putBlock: [:model :value|
            aButton insideColor = ColorValue darkCyan
                ifTrue: [act isNil ifFalse: [model perform: act].
                    aButton insideColor: ColorValue red]
                ifFalse: [act1 isNil ifFalse: [model perform: act1].
                    aButton insideColor: ColorValue darkCyan]]]
        updateBlock: [:model :value :parameter|false]).
(aContainer add: aButton in: aRange borderWidth: 1)
    insideColor: ColorValue darkCyan.

textView: aModel container: aContainer in: aRange

| aTextView |
aTextView := CodeView
    on: aModel aspect: #text change: #acceptText:
    menu: #textMenu initialSelection: nil.
aContainer
    add: (LookPreferences edgeDecorator on: aTextView)
    borderedIn: aRange.

transcript: aTranscript container: aContainer in: aRange

| aTranscriptView |
aTranscriptView := TextCollectorView new model: aTranscript.

```

Table 8.3 *Continued*

```

aContainer
  add: (LookPreferences edgeDecorator on: aTranscriptView)
  borderedIn: aRange.

view: aModel container: aContainer in: aRange

| aView |
aView := self new model: aModel.
aContainer
  add: (BorderedWrapper on: aView)
  in: aRange

VBExampleView class methodsFor: Instance creation

open

“VBExampleView open”
self open: (VBExample new)

open: aModel

| window aComposite |
window := ScheduledWindow new. “make a new window”
window label: ‘View Builder Example’.
window minimumSize: 367@174. “size selected by user created here”
aComposite := CompositePart new.

“next create a gauge”
self gauge: aComposite o: #vertical t: #bar dir: #right in: (0.0@0.0 extent: 0.212534@1.0)
val: (0 to: 50 by: 5) color: #blue initial: 25.
“create a button”
self button: ‘Button’ act: #button model: aModel container: aComposite in:
(0.212534@0.810345 extent: 0.351499@0.189655).
“create a switch”
self switch: ‘Switch’ act: #switchOn off: #switchOff model: aModel container: aComposite in:
(0.564033@0.810345 extent: 0.435967@0.189655).
“create a list”
self list: aModel container: aComposite in:
(0.762943@0.0 extent: 0.237057@0.810345).
“a text view”
self textView: aModel container: aComposite in: (0.212534@0.37931 extent:
0.550409@0.431034).
“and a transcript”
self transcript: aModel transcript container: aComposite in: (0.212534@0.0 extent:
0.550409@0.37931).

window component: aComposite.
window open

ControllerWithMenu subclass: #VBExampleController
InstanceVariableNames:”
classVariableNames:”
poolDictionaries:”
category: ‘Builder-Example’

```

Table 8.3 *Continued*

This class contains the skeleton for specifying menu associated with the example.

VBExampleController methodsFor: control defaults

menu

```
^ PopUpMenu
  labels: 'Item \ Item2 \ Item11 \ Item12' withCRs
  lines: #(2)
  values: #(#item1 #item2 #item 11 #item12).
```

redButtonActivity

view display Text: ('Mouse button message is located in category <control defaults> in instance methods in class called', self class yourself printString) asComposedText

VBExampleController methodsFor: menu messages

item1

view display Text: ('Menu messages are located in category <menu messages> in instance methods in class called', (self class yourself printString)) asComposedText

```
item11
self item1
```

item12

```
self item1
```

item2

```
self item1
```

Model subclass: #VBExample

```
InstanceVariableNames: 'transcript text'
classVariableNames: ''
poolDictionaries: ''
category: 'Builder-Example'
```

VBExample is the model created when an example MVC application is built. A protocol in the model that is associated with the views built provide "dummy" methods in which the user can insert specific code.

VBExample methodsFor: text handling

acceptText: aText

```
aText isNil ifTrue: [^ nil].
text := aText.
^ true
```

text

```
text isNil ifTrue: [text := 'Messages regarding text editor window is located in category <text handling>']
```

Table 8.3 *Continued*

```

in instance methods of class', self class yourself printString].
text := text asText.
text size > 10 ifTrue: [text
    emphasizeFrom: 1
    to: 9
    with: (Array with: #bold with: #color → ColorValue blue)].
^ text

textMenu

^ PopUpMenu
labelList: #((again undo) (copy cut paste)(accept cancel))
values: #(again undo copySelection cut paste accept cancel)

VBExample methodsFor: transcript

showOnTranscript: aString

transcript cr; show: aString

transcript

transcript isNil ifTrue: [transcript := TextCollector new].
^ transcript

VBExample methodsFor: selectionInList

aspect

^ self list

list

^ OrderedCollection with: 'aaaa' with: 'bbbb' with: 'cccc'

listChange: aSelection

aSelection isNil ifTrue: [^ nil].
self ins: 'You have selected an item in the selectionInListView' cat: '<(selectionInList)>' in:
instance'.

listMenu

^ PopUpMenu
labelList: #((menu1 menu2)(menu11 menu12))
values: #(menu1 menu2 menu11 menu12)

“Or the following method
^ PopUpMenu
labels: ‘menu1 \ menu2 \ menu11 \ menu12’ withCRs
lines: #(2)
values: #(#menu1 #menu2 #menu11 #menu12).”

```

Table 8.3 *Continued***menu1**

```
self ins: 'A menu item in the selectionInListView is selected' cat: '<selectionInList>' in:
'instance'.
```

menu11

```
self menu1
```

menu12

```
self menu1
```

menu2

```
self menu1
```

VBExample methodsFor: switch**switchOff**

"When the switch is turned off, code in this method is activated"

```
self
```

```
ins: 'A switch is turned off,'  
cat: '<switch>'  
in: 'instance'
```

switchOn

```
self
```

```
ins: 'A switch is turned on,'  
cat: '<switch>'  
in: 'instance'
```

VBExample methodsFor: button**button**

"This is the method specified in the VBExampleView openOn: aMethod, that is one of the arguments to 'self button...'. Hence, the user can insert a desired action here that is to be associated with the button press"

```
self
```

```
ins: 'A button is pressed'  
cat: '<button>'  
in: 'instance'
```

VBExample methodsFor: private**Ins: aString**

```
Transcript cr; show: aString
```

Table 8.3 *Continued***Ins: aString cat: c1 in: i1**

Transcript cr; show: aString, 'its method is located in category', c1, 'in', i1, 'methods of class', self class printString

Ins: aString cat: c1 in: i1 cl: k1

Transcript cr; show: aString, 'its method is located in category', c1, 'in', i1, 'methods of class', k1

about how to modify this example code. As constructed, the skeleton code framework contains messages to the transcript such that as the prototypical framework is operated, messages appear in the transcript that direct the user on how to change the code for an application. The code for ViewBuilder is part of the code package that is described in the Appendix.

8.13 Comparison to ST/V Interface

This text employs examples written in ST-80 for illustrating problems in engineering analysis and design. It is worth noting, however, that ST/V (Digitalk, 1986) is suitable for undertaking the same activities described in ST-80. For the most part, Smalltalk code for the two Smalltalk versions is quite similar. The model-view-controller is called the model-pane-dispatcher in ST/V, but operates in a very similar fashion. The transition between using these two versions of Smalltalk requires little effort for experienced programmers.

References

- Adams, S. (1987). *Pluggable Gauges User's Manual*. Knowledge Systems Corporation, Cary, NC.
- Jiang, Zhihe, and John R. Bourne (1991). A Visual Programming Environment for Building Smalltalk-80 Views. *Journal of Object-Oriented Programming*, May.
- Krasner, Glenn E., and Stephen T. Pope (1988). A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, Vol. 1, No. 3, August/September, 26–49.
- Smalltalk/V, Tutorial and Programming Handbook* (1986). Digitalk, Inc.

Exercises

- 8.1 See if you can build the **DisplayDemo** example yourself. Study the implementation carefully and determine how the model, controller, and view communicate.
- 8.2 Make a list of the different display entities described in this chapter. Now write a sentence for each type of view explaining when it would be useful. Can you think of any display entities that have been left out of ST-80 that would be useful? For the things left out, describe how to implement them.
- 8.3 Smalltalk supports design of user interfaces consisting of several graphical components organized in a window. Release 4 organizes all components placed on a **ScheduledWindow** into a whole-part hierarchy. Explain how components of the hierarchy interact with each other. For example, explain how resizing the **ScheduledWindow** affects its interior components.
- 8.4 In Exercise 3.2, ACOM cards were created for a specific application. Building on the cards created in that exercise, create drawings of the way that the applications proposed using the ACOM cards would look, if implemented.
- a. Consider using the following types of elements in the views:
1. Lists: Use to select items from.
 2. Buttons: Press to get an action.
 3. Moveable forms: Small icons that you can move around and place on the screen.
 4. Switches: For example, “on-off,” or “radio-button” switches.
 5. Text windows, to give information.
 6. Graphics representations of things.
 7. Animation, for example, sequences of images moving along a path at some rate.
 8. Mouse cursors that change the way they look depending on where they are on the screen.
 9. Pop-up menus, pop-up dialog boxes.
 10. Dictionaries for help and information.

- b. Consider what other types of graphics user interface constructs would be useful and invent them. You can get help by looking ahead to the examples in Chapter 9; however, before looking up this information, it would be useful to

consider what you think should be done. Only after that, look up and see how other applications have been created.

- c. Build a series of sketches (“storyboards”) that explain how your application would work, if it were implemented.

- 8.5 Explain how the class **SelectionInListView** operates.
- 8.6 For the **PictureView** example presented in Section 8.11, add the suggested updating method to the code. Explain why update works for this example.
- 8.7 If you have access to ViewBuilder, recreate the **PictureView** example using ViewBuilder.

Exercises for Users with Version 2.5 and Release 4

- 8.8 In class **Pen**, examine the method *new* which permits drawing directly on the display. Add the method *new: aForm* that draws on a new form specified by the user. (*Hint:* Look at the method *new* in class **Pen** and modify this method.) This problem can only be worked using Version 2.5 of ST-80 because there is no class **Pen** in Release 4.
- 8.9 Because **Pen** has been removed from Release 4, how can you draw on a view?
- 8.10 An interesting demonstration of drawing with a pen is an example of drawing a dragon curve. The code for this curve (from ST-80 Version 2.5) is

Pen Class
dragon: order

```
order = 0
    ifTrue: [self go: 10]
    ifFalse: [order > 0
        ifTrue: [self dragon: order -1; turn 90; dragon: 1-order]
        ifFalse: [self dragon: -1-order; turn -90; dragon: 1 + order]]
```

Reimplement this code in Release 4 using the *displayLineOn:* method for drawing on a *graphicsContext*.

Part Three

Applications in

Engineering

The third and final section of this text deals with applications in engineering. The specific intention is to show that object-oriented methodologies are useful across a broad range of engineering applications. To set the stage, Chapter 9 examines the larger subfields of engineering from an abstract point of view to determine if there are commonalities that can be alleged to exist between the different branches in engineering. Then, practical examples of problems solved using object-oriented methodologies are given and related directly to the tools described in Part Two of the text. Constraint-related methods are examined in Chapter 10, the methodologies for capturing and representing knowledge are explained in Chapter 11, integration of object-oriented systems with the external world are discussed in Chapter 12, rapid prototyping methodologies are presented in Chapter 13, and object-oriented simulation is examined in Chapter 14. Finally, in Chapter 15, some conclusions about the lessons learned from these examinations and evaluations are presented.



Relating Engineering Problems to Object-Oriented Methodologies

9.1 Objective

The hypothesis discussed in this chapter is that many engineering problems can be cast into, or at least facilitated by, an object-oriented problem-solving framework. To argue for the validity of this hypothesis, the nature of engineering will be explicated by example through brief looks at typical engineering tasks in traditional engineering domains. An attempt will be made to abstract these tasks to show that they are directly mappable to object-oriented problem-solving methodologies.

9.2 The Nature of Engineering

Introduction

Engineering is the application of the scientific methodology and thought processes to the solution of real-world problems. Work in engineering spans the continuum from design to basic engineering science. Design is the process of creating new artifacts for our world. Basic engineering science seeks to uncover new truths about our world that will eventually contribute to the solution of real-world problems. Many of the various engineering subfields tend toward one of these two extremes, whereas other subfields have strong components across the continuum.

Few would disagree that the major component of engineering is *problem solving*, both algorithmic and heuristic. Many engineering problems are

solved using mathematical methodologies captured in algorithms, whereas other problems use inexact “rules of thumb,” or heuristic reasoning.

The unifying thread of engineering is problem solving. Major conceptual methodologies support problem solving, such as knowledge organization and reasoning methodologies. Knowledge organization is an essential ingredient of engineering practice, required for concept derivation, background information composition, and analytical reasoning. Library information and common-sense knowledge have long served the engineering profession; however, as the limits of knowledge about things and how they work grow, traditional knowledge organization methods may be insufficient for future generations of engineers. Similarly, reasoning has traditionally taken two forms: *algorithmic* and *heuristic*. The former deals with clearly procedurally specifiable methodologies, for example, the calculation of a Fourier transform, stresses on mechanical parts, friction and wear coefficients, and so on. Normally derived from basic scientific principles, algorithmic methods have been a fundamental ingredient in the engineer’s tool kit for many decades. Heuristic reasoning is seemingly quite different; heuristics, or rules of thumb have also been a traditional tool in the engineer’s tool kit. Much of the “feel” of engineering lies in the application of heuristics to problems. An expert engineer can often solve problems quickly through the use of heuristic approximations. This type of reasoning may well represent a mental distillation of algorithmic methods into “intuition,” or a sense of what the outcome would be if an algorithmic method were applied to a problem. Indeed, heuristic and algorithmic methods are simply different ways of approaching problem solving in less and more structured ways.

Engineering Science and Design

Engineering science and the pure scientific method differ very little, perhaps only with respect to ultimate goals. That is, science seeks new knowledge without specific purpose other than understanding the world better. Engineering science has the same thrust but with the goal of understanding in order to facilitate application. Acquiring new knowledge requires observation and experimentation, evaluation, and modeling. Analytical methodologies attempt to decompose a problem into subparts, whereas synthesis seeks to recompose parts into a whole.

At the opposite end of the engineering spectrum lies design, the set of activities that leads to the creation of new objects. Design consists of conceptualization, specification, the use of analogy and analysis, prototyping, refinement, and simulation. Following design lies manufacturing, testing, and maintenance which are also part of engineering.

The tasks of engineering, discussed previously, are generalizable across disciplinary areas of engineering and, indeed, to other disciplines as well. There are clear parallels between the engineering method and the object-oriented methods which will be explicated later in this chapter. Concepts discussed in previous chapters concerning abstraction, encapsulation, reusability, and object-oriented methods for analysis and design are not often discussed in engineering. It appears plausible to suggest that the techniques that have been studied should be applicable to a wide range of engineering problems. After a deeper examination of problem solving in engineering, an attempt will be made to show how object-oriented problem-solving techniques can facilitate engineering.

The Engineering Fields

The next paragraphs examine the characteristics of several engineering fields in an attempt to discover commonalities and abstractions of the kinds of activities that are conducted in engineering. Table 9.1 displays a listing of some of the most prominent engineering fields, along with a rough assessment of the amounts of design and engineering science components found in each area. This assessment is shown as a continuum across the *design ↔ science* spectrum.

Table 9.1 Loose Approximation of the Design and Engineering Science Components of Subdisciplines in Engineering

Subfield	Design ← → Engineering Science
Civil engineering	↔
Chemical engineering	↔
Electrical engineering	↔
Computer engineering	↔
Mechanical engineering	↔
Biomedical engineering	↔
Environmental engineering	↔
Agricultural engineering	↔
Aerospace engineering	↔
Materials engineering	↔
Manufacturing industrial engineering	↔

Civil Engineering. Generally speaking, civil engineering is concerned with the analysis and design of structures, including buildings and transportation systems. Design in civil engineering often requires extensive analytical tools; yet other aspects utilize heuristic elements of design. Knowledge organization, reasoning, and algorithmic solutions are all needed in civil engineering.

Chemical Engineering. Chemical engineering applies basic knowledge about chemistry to the design and operation of plants and processes for the production of materials for use in our society. For example, chemical engineers study methods for designing plants and processes to manufacture such products as paper, pharmaceuticals, fibers, paints, and so forth. Simulation of plants, understanding of constraints in processes, and design analogy are components of the knowledge needs of chemical engineering.

Electrical Engineering. Electrical engineering is concerned with electrical phenomena, including electronic circuits, computers, communications, control, and the study of systems. Electrical engineering is both knowledge and design intensive. Characterized as the broadest engineering field, electrical engineering requires all the basic knowledge intensive tools, as well as both algorithmic and heuristic reasoning techniques.

Computer Engineering. Computer engineering is concerned with the design of computers and the software required to operate computer systems. Computer engineering deals with all aspects of computer systems, including design and fabrication of computers, as well as the creation of application software, for example, software for computer-aided design and electronic simulation. In contrast to computer science, which seeks to discover new knowledge about computation, computer engineering is a practical field emphasizing design.

Mechanical Engineering. Mechanical engineering deals with machines and mechanical processes. Mechanical engineering requires considerable three-dimensional visualization capabilities, relies heavily on analogy for design, and shares many common algorithmic methodologies for control with electrical engineering.

Biomedical Engineering. Biomedical engineering is concerned with the application of engineering principles to the field of medicine. Biomedical engineering is one of the more science-oriented engineering disciplines, tending toward the use of analytical methods more than design.

Environmental Engineering. Environmental engineering seeks an understanding of how to improve our environment through the use of engineering methodolo-

gies. Modeling and simulation are heavily used and there are many cross linkages with chemical engineering.

Agricultural Engineering. Agricultural engineering applies engineering principles to food and food production. Combinations of analytical methods apply here, as well as heuristic reasoning.

Aerospace Engineering. Aerospace engineering deals with the design and fabrication of airplanes, rockets, and the other elements of aeronautical endeavors. One of the more highly mathematically and computationally bound engineering disciplines, heuristics tend to play a minor role.

Material Engineering and Science. This field studies the basic properties of materials. The basic scientific method is found in this field in much more prominence than design. Presentation and understanding of information are critical.

Manufacturing /Industrial Engineering. Turning designs into reality is the responsibility of manufacturing and industrial engineers. Methodologies for automating manufacturing facilities are studied, as well as applying engineering methodologies to industrial processes.

Education. Education in engineering is normally broken into subfields, including those discussed previously. An analysis of the attributes of the activities in each of the engineering subfields argues that there should perhaps be more common educational activities because the engineering disciplines appear to share many common needs and methods.

9.3 Evaluation of Common Requirements in Engineering

Examination of the fields of engineering permits consideration of the commonalities in the different engineering fields and identification of the abstract activities that provide bridges across the fields.

Knowledge Representation

Knowledge representation appears to be a central issue in all the engineering fields. Indeed, the need to represent knowledge extends into every corner of human inquiry, because representation is almost a prerequisite to understanding and problem solving. What is knowledge representation? In

examining the fields of engineering, knowledge in engineering is the representation of the area of interest in terms of a *model*, where a model may take on very different representations depending on the area. For example, a model might be a complex set of equations that describe a physical phenomenon being studied or a set of constraints in a spreadsheet. The term *model* is so broad that it can be used in most cases to simply indicate that knowledge is being organized in a way to make concepts more understandable. In this chapter, with particular reference to the *model* of the model-view-controller paradigm, a *model* is any representation technique that describes the problem being examined without reference to the *view* or *controller*.

Laws, Algorithms, and Methodologies

Basic knowledge about the physical laws, the algorithms that implement procedures, and the methodologies that describe ways of accomplishing activities are the most usual knowledge that needs representation in engineering problems. Laws, as manifested in algorithmically specified equations, are the backbone of engineering. Basic “commonsense” engineering relates to the laws derived from basic science. Algorithms, of course, are the manifestation of mathematical expressions in a computer-consumable form. Methodologies can be described as plans, or sequences of steps that when applied to a problem result in a solution. Most engineering activities include considerable use of mathematics and statistics for problem solving.

Engineering Reasoning

Reasoning methodologies are not as clearly understood as the use of laws, algorithms, and methodologies for problem solving. Reasoning can often take the form of applying well-understood algorithms to a problem. However, problems are not always amenable to the use of solutions that fit a problem exactly. Hence, there is a need for representation of heuristic reasoning methods that capture rules of thumb.

Domain-Specific Knowledge

Each domain in engineering has detailed specific knowledge that must be representable in retrievable and understandable fashions. For all the engineering fields discussed previously, it is clear that domain-specific knowledge is essential. Each field works with different knowledge; yet it is plausible to make a conjecture that all could utilize common organization and understanding methodologies.

Presentation and Visualization

Common to all the engineering disciplines is the clear need for presentation and visualization facilities. Presentation mechanisms have become well developed in engineering, resulting in many graphics and computer-aided-design packages that permit engineers to view the results of analysis and design activities. However, mechanisms for visualization are not so well developed. Decomposition of problems and the graphical visualization of problem parts and how parts interact is an immature field. For example, the visualization of causal relationships in design and trade-offs in the design process are currently difficult to visualize in engineering design. For assisting in understanding knowledge, automated visualization tool kits are also underdeveloped. There are few methods available that provide simple pluggable graphics interfaces to engineering systems.

Modeling and Simulation

In many ways, review of the engineering fields leads toward the observation that modeling and simulation are central to the engineering effort. Modeling, of course, is directly related to knowledge representation in that the model of a system is indeed the representation of information about the system. Simulation, in contrast, is the behavioral manifestation of the model and its results are best presented in an easily observable (i.e., graphical) fashion.

Many simulation systems used in engineering require users to interpret the results of a quantitative simulation but provide few facilities for understanding how the simulation relates to the problem. "Tedious" and "difficult to use" are terms that could be applied to many traditional simulation systems. Why? Probably because there is a fundamental cognitive mismatch between the representation and the problem. Historically, in most cases, simulation systems were developed using procedural languages long before the concepts of mapping programming languages to the world were invented. Hence, traditional simulation systems reflect the mismatch between the programming paradigms used and the needs of the engineering simulation.

Control

Control is an aspect of engineering common to electrical, chemical, and mechanical engineering. Automatic control, which is control untouched by humans, is usually algorithmic and mappable to procedures. However, the ability to experiment with control algorithms has been severely limited by the

algorithmic nature of the coding methodologies. Perturbations to, and experimentation with, control methods are hard to visualize in simulated systems.

A second method of control is the direct linkage of direct-manipulation interfaces to systems that require control. Humans controlling complex systems need long experience and understanding to provide reasonable control. For example, control of a nuclear power plant requires definite expertise or disastrous results could occur. Direct-manipulation interfaces with built-in intelligence could assist in providing more intelligent control for complex devices that would assist a human user.

Next, a direct comparison of engineering problem-solving requirements and object-oriented methods will be undertaken.

9.4 Engineering and Object-Oriented Problem-Solving Methodologies

The characteristics of object-oriented problem solving for engineering will be described in general in this section. Section 9.5 will present several examples that tie together the general concepts presented in this section. Figure 9.1 compares typical engineering and object-oriented problem-solving methods. The left column displays the typical steps involved in engineering design and the right column displays the equivalent steps using object-oriented techniques.

Concept Formation

The first step in understanding how to solve a problem is to organize the concepts about a problem. In analytical problems, the end-point goals for the analysis are specified, and for design problems, the specification for the object being designed is created. The engineering method has no clear formalism at this point; indeed, ad hoc methodologies for concept formation seem to be the norm. For object-oriented methods, the same is essentially true, except that the framework into which the problem will ultimately be cast gives some structure to concept formation. For example, one can consider several alternative structures for a model, ask what an ideal view should be, and question how the system created should perform. Although such questions fit cleanly to problem understanding in the context of the MVC-paradigm organizations presented in the previous chapter, it appears that the same questions might also lead to engineering design concept formalisms.

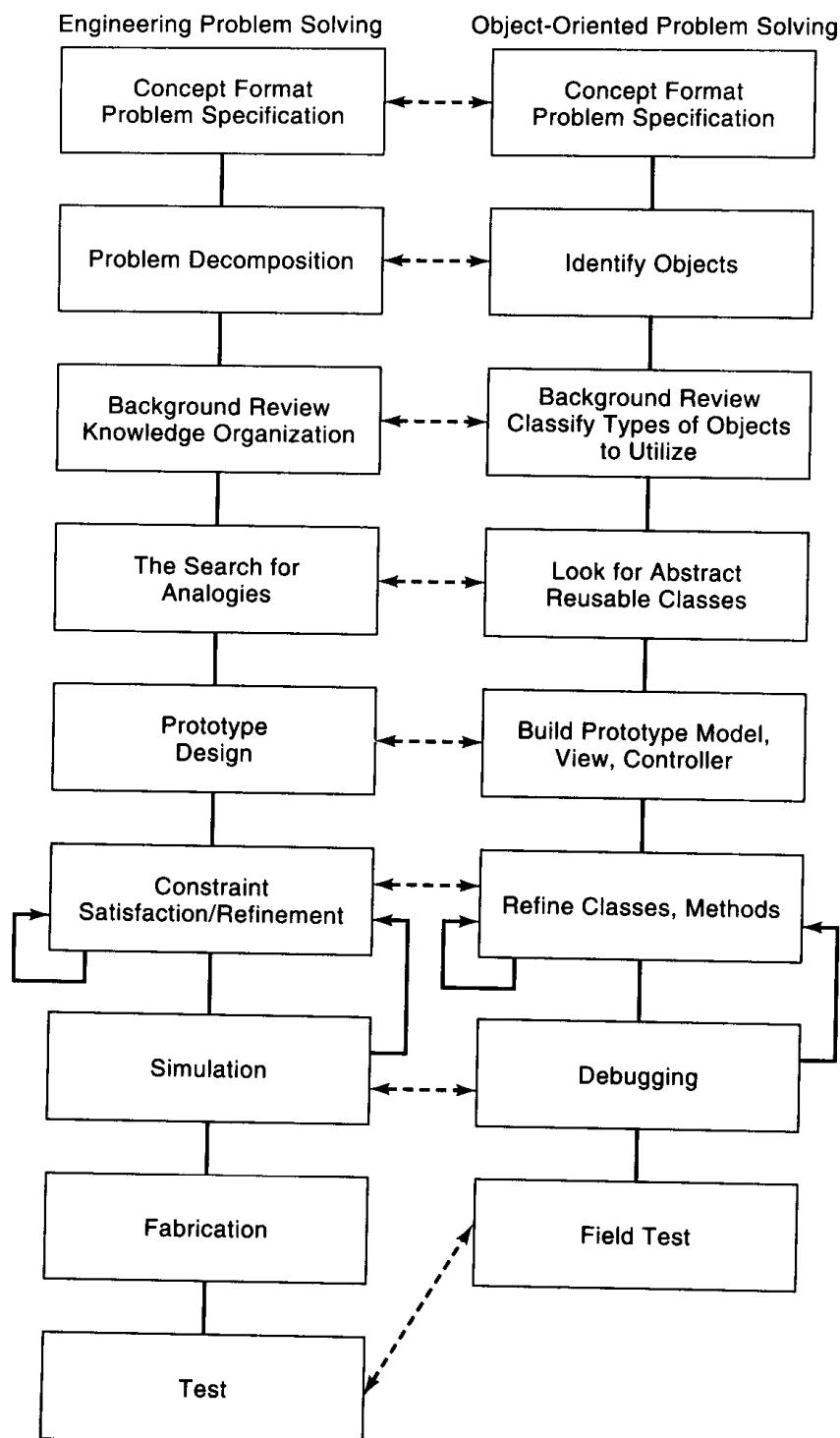


Figure 9.1 Comparison of Engineering Problem Solving and Object-Oriented Problem Solving.

Problem Decomposition

Once concepts are formed, a problem should be decomposed into understandable parts. Decomposition is a typical engineering methodology that works for object-oriented systems as well. For the latter, however, one can decompose according to whole-part hierarchies *and* inheritance hierarchies (see Chapter 2), whereas in engineering design and analysis, decomposition is traditionally only whole-part.

Background Review/Organization

Problem solving requires background knowledge. Typical engineering efforts review background material and attempt to secure as much information as possible in the domain of interest. For software systems, this would include searching for code that can be used in an application. For object-oriented methods, a review of objects in available class libraries is needed so that a minimum amount of coding will be required.

Organization of knowledge in an application is critical for the design of any system. In engineering systems, attributes of the designed artifacts will need to be specified according to accepted norms and referenced to constraints that may change attribute values. Similarly, for object-oriented design, the influence of objects on each other as manifested in the methods in protocols needs to be clarified. Once knowledge grows to a level that exceeds raw human comprehension, automated tools for organizing knowledge must be sought, for example, hypertext organizations of information that permit handling large amounts of information (see Chapter 11).

Prototypical Design

Once preliminary design decisions are made for both engineering designs and object-oriented designs, specifications can be made and a prototype created. In engineering, it is typical to fabricate a rather complete specification for the end product and to work to build that product. In contrast, object-oriented design prefers the so-called “scruffy” approach in which a prototype is built from a very loose design and refined continually until the product is complete (some say that object-based systems are “debugged” into existence). The fact that these design preferences even exist gives some hint about the nature of design using traditional techniques and design using objects. In the traditional top-down approach, a complete specification drives the creation of a system, yielding little to experimentation or modification along the route. Thus, initial conceptualization and specification must be correct. In contrast, object-

oriented methods permit simple refinement as implementation of a prototype proceeds. In part, this capability is due to the modularity afforded by object-oriented systems and, in part, to the capability for refinement by inheritance.

Analogy is used in engineering design; that is, old techniques and knowledge that have worked before are examined and reused to the greatest extent possible. Although analogy is heavily used in engineering, there is no explicit formalism for supporting reuse as there is in object-oriented design. Standard engineering components such as circuits, electrical components, and mechanical subassemblies are commonly reused in the process of engineering design. This type of reuse is roughly analogous to object-oriented design, although no specific formalism is widely used to assist engineering designers in reusing existing components. Reusability of objects in the object-oriented methodology is a key factor in object-oriented design. Designing from reusable components permits the creation of new objects much more quickly than having to begin from scratch. Auer's (1989) observation perhaps captures the essence of this concept: "The fastest way to produce code which is well designed, written, and tested is to already have it."

Constraint Satisfaction

Constraint satisfaction is the process of considering trade-offs. Financial constraints can be handled in engineering projects in a direct fashion using spreadsheets, for example. However, beyond simple understandable numeric relationships, the matter of understanding trade-offs becomes difficult. For example, consider the creation of a new product in which there are various competing constraints: time to market, customer satisfaction, quality, development costs, and product costs. These different quantities and qualities impose constraints on the design and ultimate manufacturability of a product. In engineering systems, working with constraints of this type are poorly handled. Constraint satisfaction in object-oriented methodologies is also a problem, perhaps less so because of the built-in abstraction provided by inheritance using abstract classes and the constraining paradigm afforded by the model-view-controller. Indeed, the trade-off problem discussed previously may be no easier to understand using object-oriented methodologies, but the code would likely be easier to code and maintain than traditional code.

Simulation

There are few, if any, clearly systematic methods for simulation in traditional engineering. Each domain contains hundreds if not thousands of special-purpose simulation systems which may or may not meet the needs of any

particular user. Most packages have been built without regard to interfacing with other packages; hence, there are virtually no standards! This situation is, unfortunately, true also in the object-oriented programming world. However, there is a conceptual base for standards in simulation, both analog and digital. Various ideas for interfacing to simulation packages and creating simulation frameworks will be discussed in upcoming chapters.

The analogy between simulation in an engineering design and an object-oriented design is the rapid prototyping/debugging cycle in the latter. Simulation in engineering yields knowledge about whether the projected product will work. In creating object-centered systems, applications are often debugged into existence yielding the same type of knowledge, except that the knowledge now permits a prototype of the product to continually mature during rapid refinement cycles.

Refinement

Refinement is required in any analytical or design setting. Certainly, the coupling of refinement steps and simulation is suitable, because it is not cost effective to manufacture a product until a simulation of the product has been satisfactorily refined. Refinement in object-oriented design is not as critical; in fact, refinement needs to be taken more seriously in object-oriented programming. There is a tendency, once a program works, to simply stop coding. Refining steps should be to abstract the common features of objects and to reorganize the class inheritance hierarchies, so that the objects created can be reused by others.

Fabrication

Fabrication is the actual creation of an entity from a design. In engineering, this step is taken by actually manufacturing the product. In programming, the step is taking by extensive testing and polishing of the user interface.

System Creation Considerations

Presentation Methods. There are, of course, many alternatives for presentation of information and different techniques for implementing user interaction with engineering and computing systems. Problems in presentation of information are essentially the same for both engineering and computing systems. Engineering systems, although often designed for special purposes, have reached the

point where presentation and user interface considerations are essentially the same as for systems that are software intensive. In general, one can hardly make a clear distinction between a hardware-intensive system and a software-intensive system. In both, there is a significant need for interaction with the user and presentation of information. In fact, some hardware systems are now actually software systems that mimic hardware (e.g., the modern oscilloscope has a highly developed user interface that appears similar to an interface to a traditional hardware oscilloscope, but is, in fact, created by software!).

For computer-based systems that organize information, interpret findings, provide control, give advice, and assist in the analysis and design of systems, the direct-manipulation methods discussed in the previous chapters are among the most appropriate type of interface. Decisions about the modalities of presentation, such as whether to use video, voice commands, speech output, command line interfaces as well as GDMIs, remain a difficult problem for the designer. In building most engineering systems, because integral computer facilities are normally part of most complex engineering systems, these issues must be addressed by engineering designers. These same considerations are also true for software designers using object-oriented techniques. The object paradigm, per se, does little to assist in understanding how best to construct the user interface for a system. The major advantage to the object-oriented techniques discussed in this text are the organization and simplification capabilities afforded by the MVC factorization methodology.

Inference. A second issue deals with the level of inference or “intelligence” that systems should contain. It is easy to understand the plausibility of adding intelligence to most engineering systems. For example, an intelligent measurement instrument would have the capability of making inferences about what is measured or an intelligent traffic control system could adapt its control strategies to the information it secures on-line about the current state of traffic. In computer systems, the same notions are plausible and can be extended to the concept of monitoring the user of a computer system and adapting the way that the system presents information based on the needs of the user. Large analysis and design systems could contain mechanisms for making inferences during analysis or design sessions to permit dynamic restructuring of analysis or design procedures according to the inferences made.

Design Frameworks. The concept of frameworks relates to problem solving in engineering. A framework is a skeletal tool that helps a user construct or analyze situations. Smalltalk-80 can, in fact, be considered a framework, because the language plus the environment facilitates the design of software systems. In the electronics world, frameworks capture specific knowledge about components and methodologies that are useful for design. Other domains have similar frameworks that capture knowledge in their domains. The evolution of

design, analysis, and synthesis frameworks speaks to the continually evolving and expanding knowledge that exists in the world. There are clear needs to capture this knowledge and present it to users in a way that the knowledge is readily usable. Object-oriented analysis and design methodologies precisely fit this need!

Knowledge Organization and Modeling. In the design of any engineering system, ways of organizing knowledge and capturing and presenting models of the world in which the designed systems work are critical. Knowledge can be organized as frames, in rules, captured in objects or in many types of data structures. Again, as discussed previously for the framework problem, organizing schema are needed to permit designers/analysts to create standard and sharable organizations of knowledge and models. Until this step is taken, design frameworks will remain monolithic entities with little capability for sharing information. Object-oriented techniques, trading particularly on the encapsulation concept, have the potential to fill this need for methods for standardization or, at least, provide conversion protocols between different applications that use different kinds of representation mechanisms.

Refinement. In today's engineering systems, there is often little capability for changing the characteristics of a system once it is built. For most software packages, this rigidity is also found, but to a lesser extent. It is plausible to suspect that future systems will permit augmentation of system capabilities by the user to fit specific needs. Although few current commercial engineering packages permit such flexibility, one can see the outlines of these possibilities in object-oriented packages that permit refinement because of the specialization of classes.

Summary of Abstract Concepts

In the preceding sections, a number of abstract concepts associated with engineering design, analysis and synthesis, knowledge representation and modeling, simulation, visualization, understanding, and analogy were discussed. In several cases, it was suggested that object-oriented methodologies permit implementation and facilitation of these abstract concepts in real systems. In the early 1990s, systems that are completely inspectable, modular, and that allow simple and understandable extensions are not yet commonplace and no frameworks have been developed to assist users in organizing knowledge according to these abstract concepts. It is postulated that the object-oriented methodologies described in this text will assist in reaching the goals of producing flexible, understandable, and adaptable engineering systems in the future.

Early examples of the capabilities of such systems exist in simulation and design frameworks, knowledge browsing systems, and inference systems. The next section of this chapter demonstrates some early steps taken in creating object-oriented systems for use in engineering domains. Examples are given in several primary areas in engineering.

9.5 Characterizing Problems Using the Model-View-Controller Paradigm

The model-view-controller paradigm of Smalltalk-80 provides a unique and useful organizing methodology for creating real working systems for engineering analysis and design problems. Some concrete examples of systems that have been built or designed based on the MVC are described next. Note that the utility of the concepts presented are not limited to ST-80; however, because ST-80 has one of the best developed environments for supporting building examples using the paradigm, only ST-80 examples are given. In each example, the components examined in detail in Chapter 8 will be discussed. The reader is encouraged to consider each example and attempt to relate the programming structures that support each application to the fundamental models, views, and controllers discussed previously.

Civil Engineering

Figure 9.2 revisits the traffic flow example presented in Chapter 1. In this example, the primary purpose is to permit visualization of traffic approaching an intersection. The user is able to change stochastic (i.e., random) variations in traffic density in order to study the influence of how control of the lights at the intersections will affect waiting queues. The main view of the traffic flow simulator shows a diagram of a street plan feeding a highway. The concept is to provide the user control of the traffic lights and specification of the random variation of the arrival of automobiles at the intersections. The display permits viewing of the traffic density on the highway, as well as the waiting queues at the intersection of the feeder streets.

A canvas view (see the Appendix) is used for displaying the street pattern, which can be drawn using a **FormEditor** (Version 2.5) or host platform drawing package (Release 4) and used in a **CanvasView**. The small rectangles shown represent the automobiles queuing at the lights. Ordered collections are used to represent each queue for each light. The arrows in the view represent the direction of traffic flow. The only animation components in this image are adding and removing small iconic images, which simplify creating the appear-

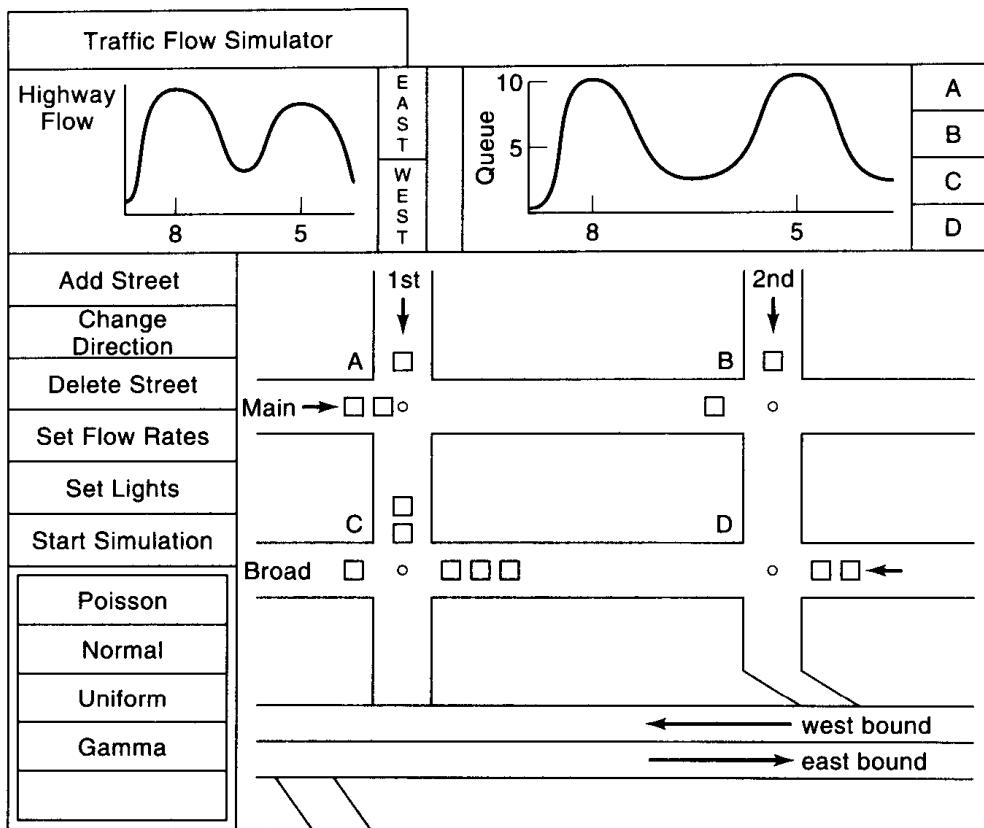


Figure 9.2 A Traffic Flow Simulator.

ance of animation. On the left of the window is (1) a list of buttons for controlling the simulation and (2) a scrollable **SelectionInListView** that contains the different options for probability distributions. Each button, when pressed, permits more options to be explored in a separate view. For example, “Add Street” will allow editing of the image used in the canvas. A more sophisticated system would allow automated addition of or modification to a street. The “Change Direction” button produces a crossHair cursor which the user can position over the arrows in the canvas; clicking once changes the direction, twice terminates. Setting light and flow rates brings up a table which allows selection of the lights and specification of their stochastic attributes and specification of the flow rates on each of the streets.

The top of the window shows the output results: Two diagrams are displayed in individual views that indicate traffic flow rates on the highway and on the streets. Note the selection buttons at the left of each of the data displays which allow choosing which intersection to view and whether to examine traffic

flowing east or west on the highway. The flow graphs are created by drawing on the image. The underlying model of this simulation is a discrete-event simulation system which will be discussed in Chapter 14.

Chemical Engineering

Figure 9.3 shows an example of an assistant system created for simulating chemical separation problems in chemical engineering (Debelak, 1990). The main view contains a network of representations of chemical separator systems. The concept is that a compound containing various chemical components is to be separated into various products and subproducts by arranging a series of separators, each of which will separate components in different ways. The user can select building a specialized system by using the buttons marked "add separator" and "connect separator." The input to the system can be selected using the compound dictionary shown at the top left of the view and pressing the button marked "select feed," where the "feed" is the input to the series of separators. There are various rules attached to the simulation, such as "select the cheapest compound to remove first," which are embodied in rules. These rules can be edited by pushing the "edit rules" button.

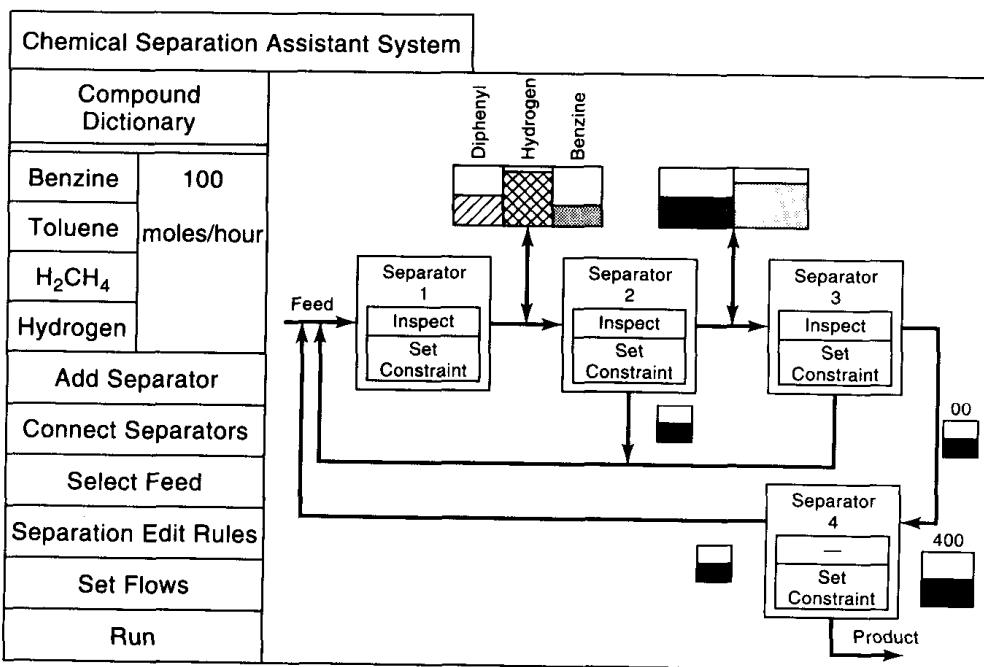


Figure 9.3 A Chemical Separation Assistant System.

Flow rates can be set, as indicated, For each of the separators, various constraint equations can be used to propagate information forward about the materials separated in the simulated system. Each separator contains its own constraints which can be modified by pressing the button marked "set constraint" in the separator representation. Also, information about each separator can be inspected by pressing "inspect." Gauges are used at the output of each separator to show the amount of material in the compound at each stage of the process. Finally, "run" permits the simulation to run with the values set.

Electrical Engineering

Figure 9.4 displays a view from a diagnostic system created to analyze failures in electronic circuit boards. Based on the Dempster-Shafer theory (Shafer and Logan, 1987), the system displays a hierarchical diagram of causes and effects (shown in the large view in the background of the figure and in the "greeked" view at the bottom left of the window) and a view of the components on a view of the actual circuit board. The organization of the view is simple: A series of control buttons are at the top of the window. In different parts of the diagnostic process, buttons appear with labels; at some points the labels are blank, thus providing no action. A transcript is provided at the left of the view and a **FillInTheBlank** pop-up view is used to secure information from the user, as shown. By answering questions about the status of a physical board, recommendations on how to repair a circuit board are given. The probe shown on the figure is interesting. It is simply an opaqueForm (i.e., an OpaqueImage) that can be moved to different positions on the board to permit sensing of what the proper voltage is at that point. Points are sensed by comparison of the probe point location with a dictionary of points at which voltage levels are available.

Figure 9.5 shows a second electrical engineering example dealing with circuits. In this example, an electronic lab bench is simulated that permits the placing and monitoring of electronic components. The array of buttons to the left allows placing of various types of electronic components on the canvas as opaqueForms (opaqueImages in Release 4) and controlling of the operation of the system created. This particular example was taken from a prototype of a tutorial system for electronics built in Smalltalk-80.

Mechanical / Manufacturing Engineering

Figures 9.6 and 9.7 display two views of parts of a system built for the evaluation of statistical process control (Bourne et al., 1989). This system was designed (1) to acquire knowledge about the causes of out-of-control conditions

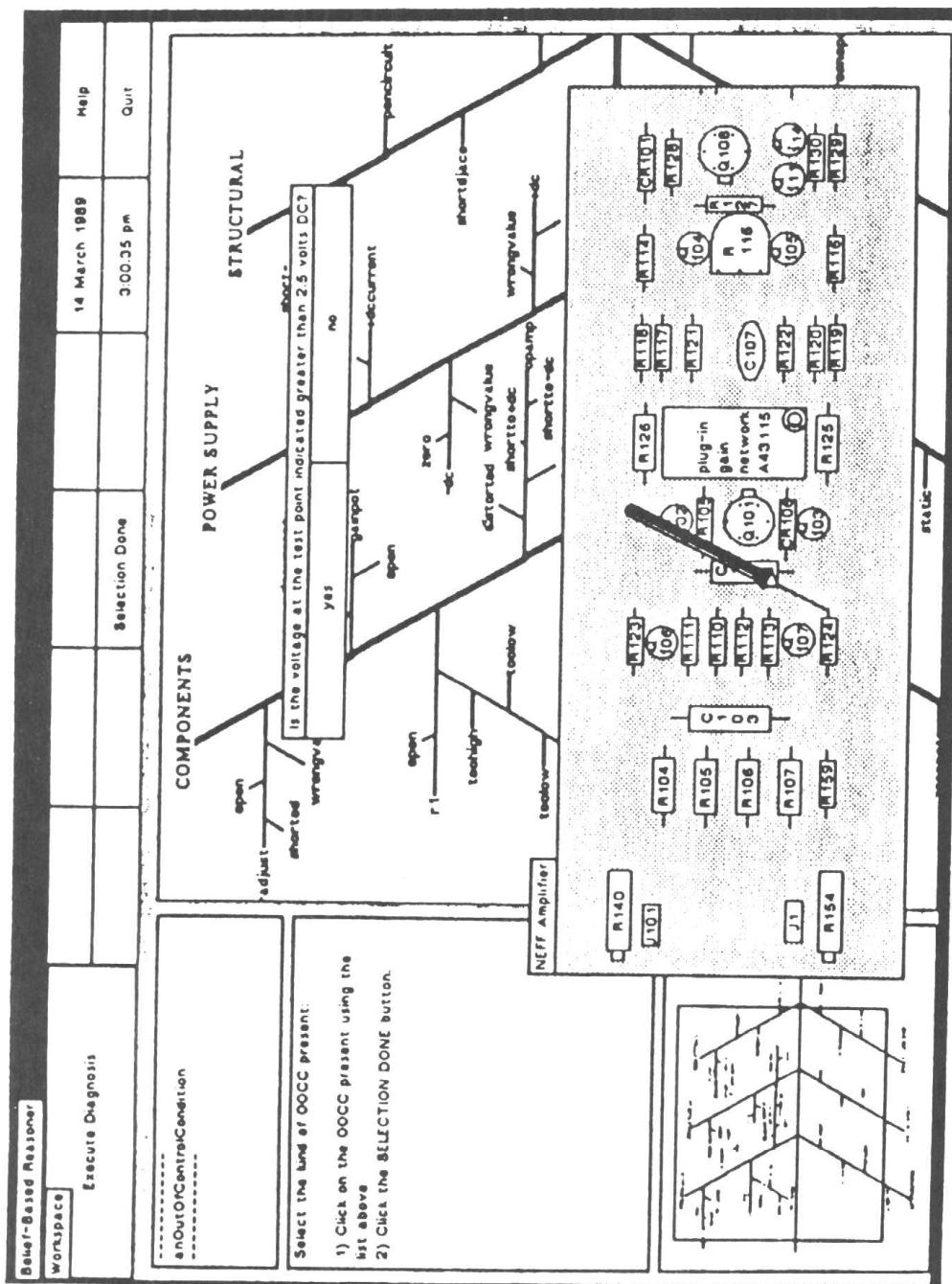


Figure 9.4 Electronic Diagnostic System. (© 1989 Association for Computing Machinery, Inc. Reprinted by permission.)

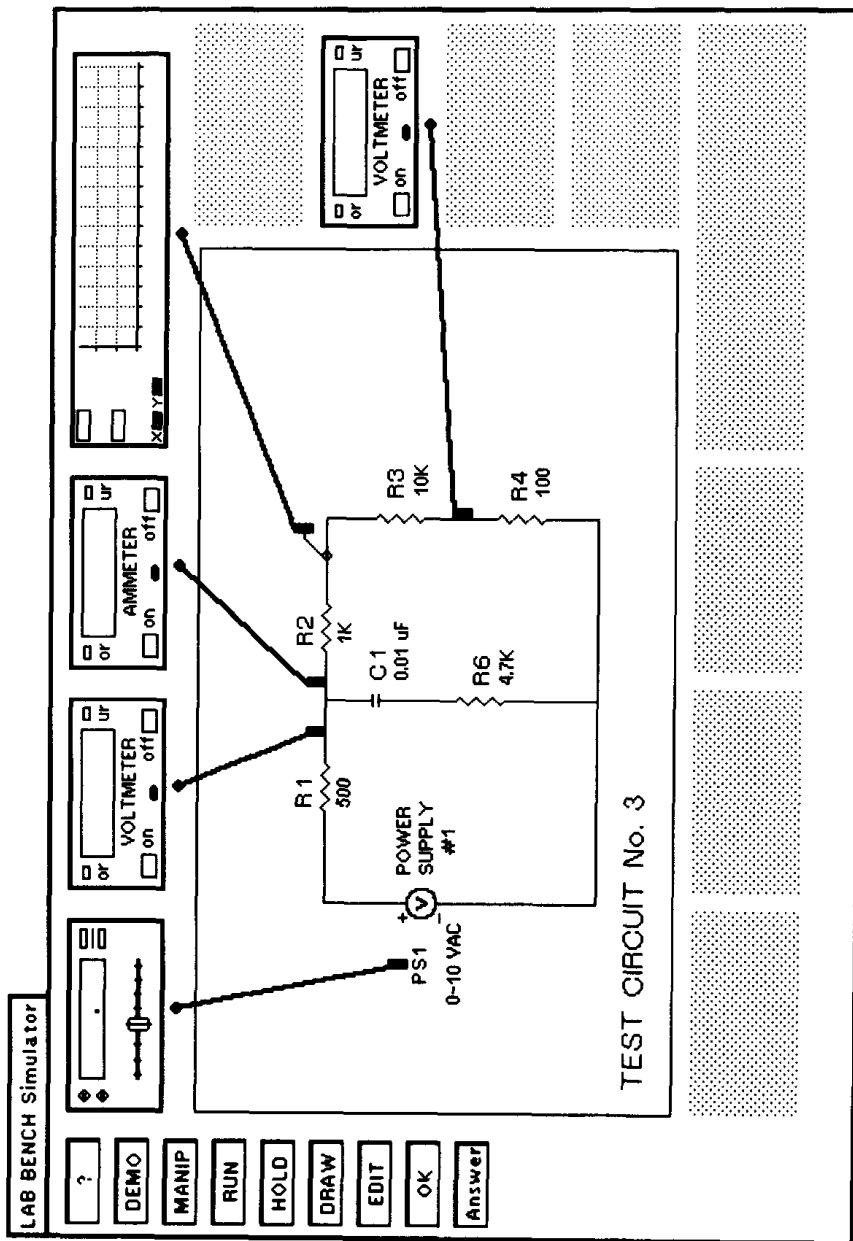


Figure 9.5 A Simulated Electronics Laboratory.

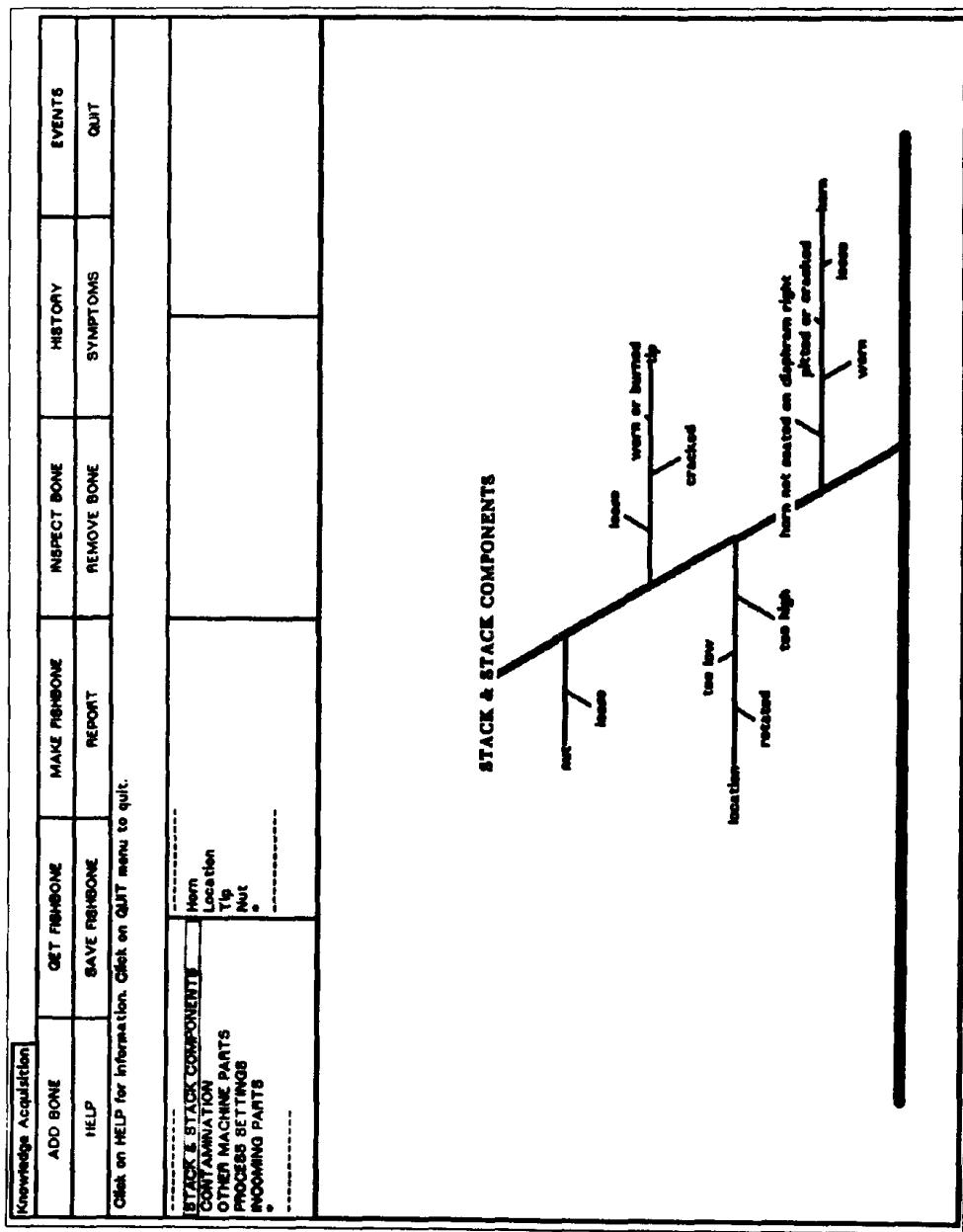


Figure 9.6 A Knowledge Acquisition View for Statistical Process Control. (With permission; Kirk Karwan and James Sweigart.)

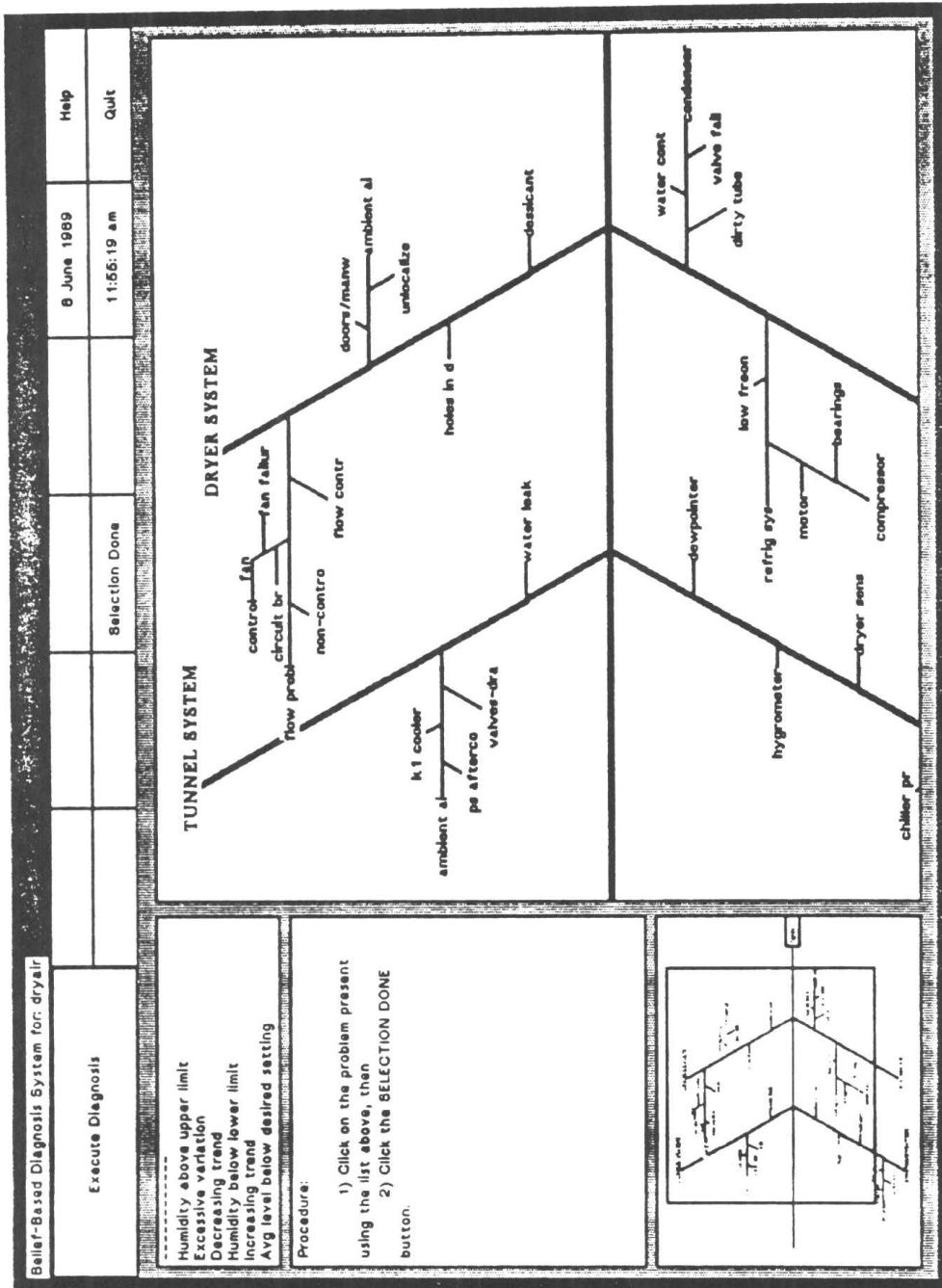


Figure 9.7 The Diagnostic View for a Statistical Process Control Advisory System.

in manufacturing processes and (2) to diagnose problems in malfunctioning manufacturing processes. Because the knowledge acquisition part of the system precedes the use of the system for diagnosis, two separate views were created. The first view, as shown in Figure 9.6, was used to acquire knowledge in the form of a hierarchical, or fishbone, diagram which related causes of problems to the problem. The window layout consisted of control buttons at the top of the window, a transcript view, four SelectionInListViews that corresponded to the successive subelements of parts of the tree, and a graphical view of parts of the tree. Once this information was collected, including probabilistic information, links between events and problems, and historical information, the system could be used for diagnosis as shown in Figure 9.7. This figure shows the diagnostic view, in which the fishbone diagram is exhibited and diagnostic results given in the left view of the window. This figure is basically the same as the diagnostic system described previously for electronic systems.

Environmental Engineering

An example of using the MVC paradigm in ST-80 for building a groundwater tutoring system in environmental engineering is shown in Figures 9.8 and 9.9. This system has multiple views which are activated from a control panel that allows the opening of several different windows. Figure 9.8 shows an "information center," which basically systematizes information about groundwater into a set of hypertext graphs. As shown, there are a series of views, each containing information. The graph in the second view holds a set of nodes and links. Each node contains additional information which, when pointed to by the cursor and when the mouse button is clicked, appears in successive views and the textView at the bottom of the window. Figure 9.9 shows how data can be easily displayed for the groundwater problem. The problem is described graphically at the top right with attendant data just below. A family of curves is plotted to the left of the view based on the data acquired. As usual, a series of buttons are arrayed along the bottom of the window to provide control. Complete details of this system are contained in Bourne et al. (1989).

Biomedical Engineering

Figure 9.10 shows an example of the use of the MVC paradigm in solving a biomedical engineering problem (Haden, 1989). This example shows a large view on the right of the window which is used to display different sections of the brain and to enlarge and reduce images. These images are CAT scans

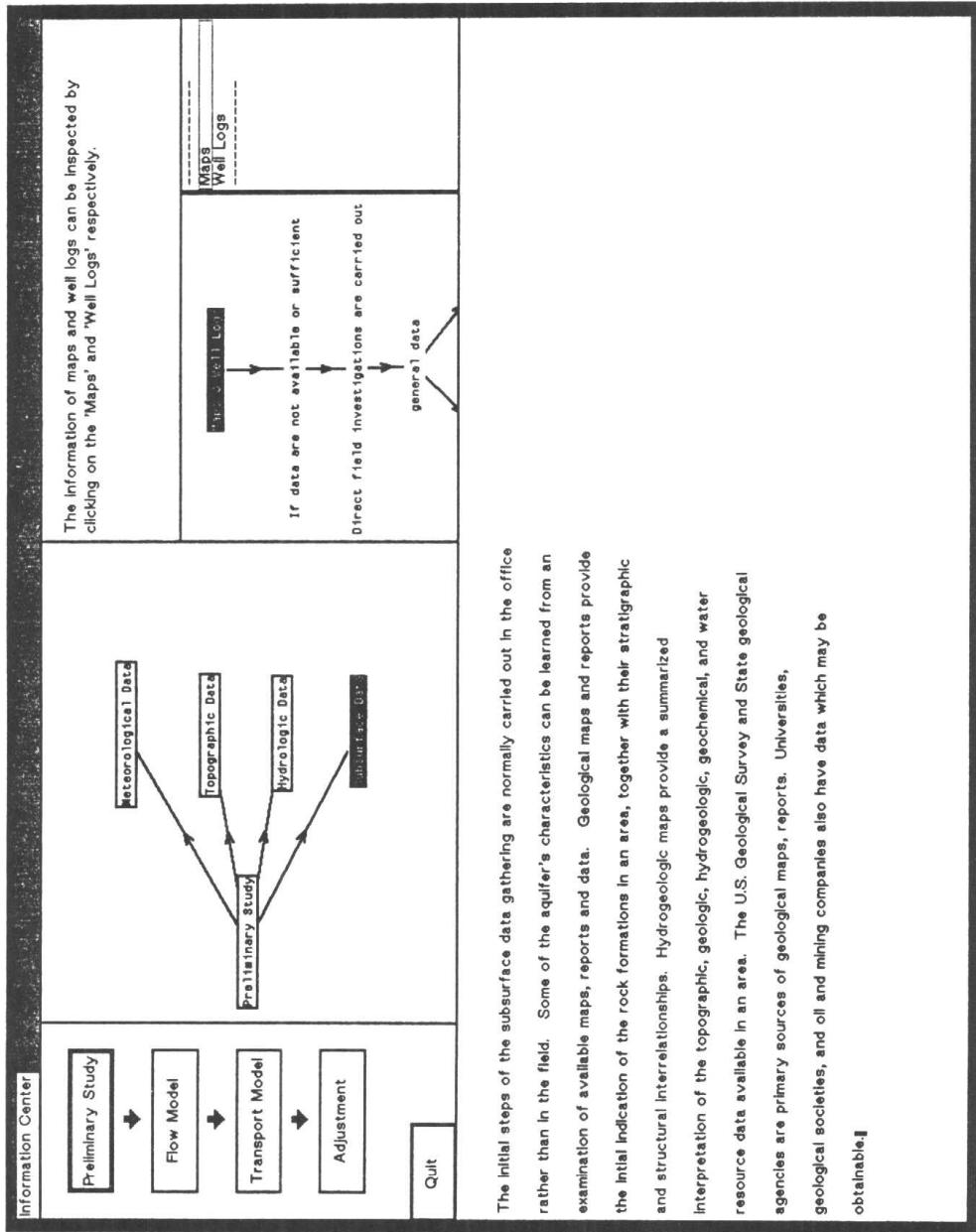


Figure 9.8 The Information Center for the Groundwater Advisory System. (With permission; Academic Computing.)

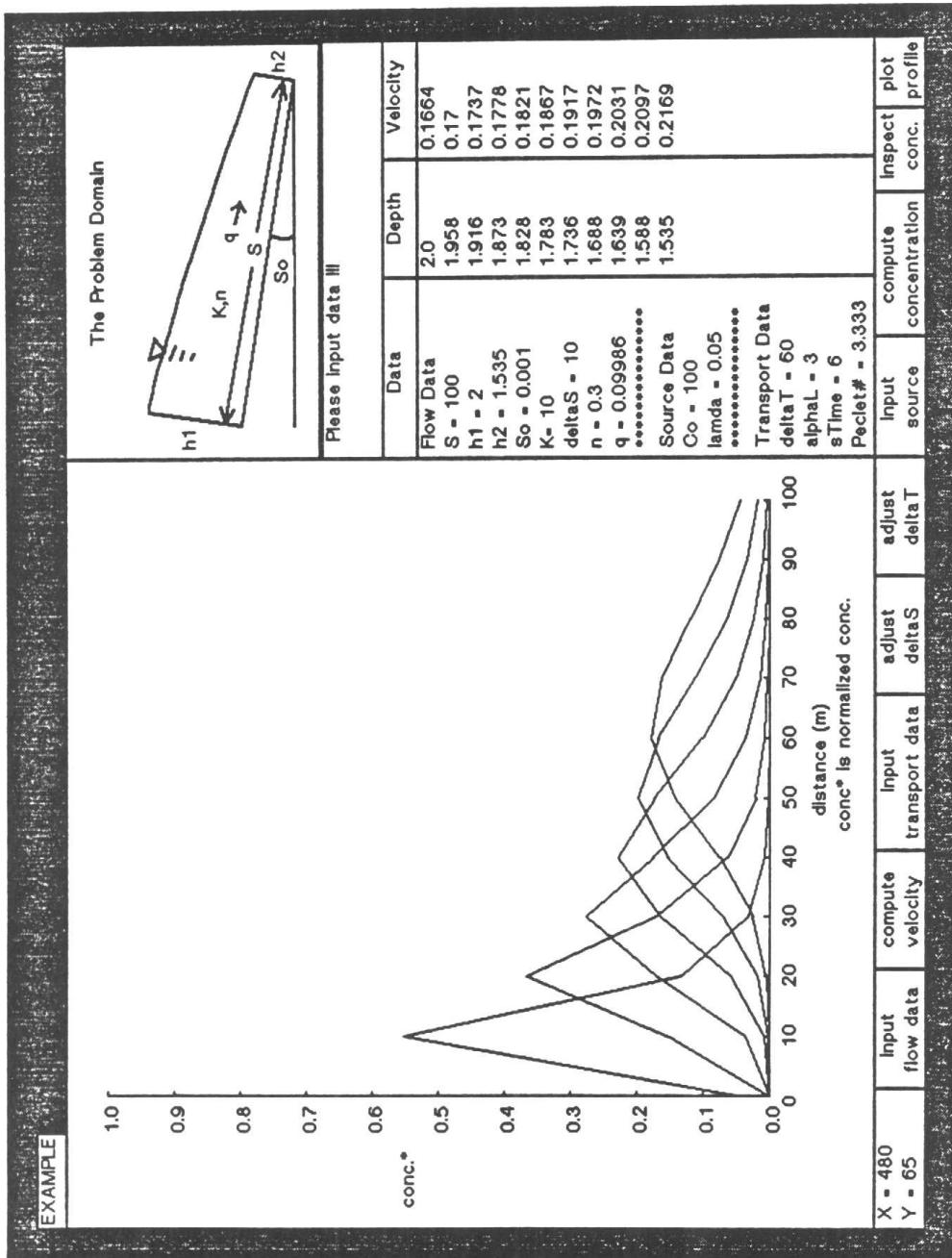


Figure 9.9 Interaction of Flow and Solute Transport in the Groundwater Advisory System. (With permission; Academic Computing.)

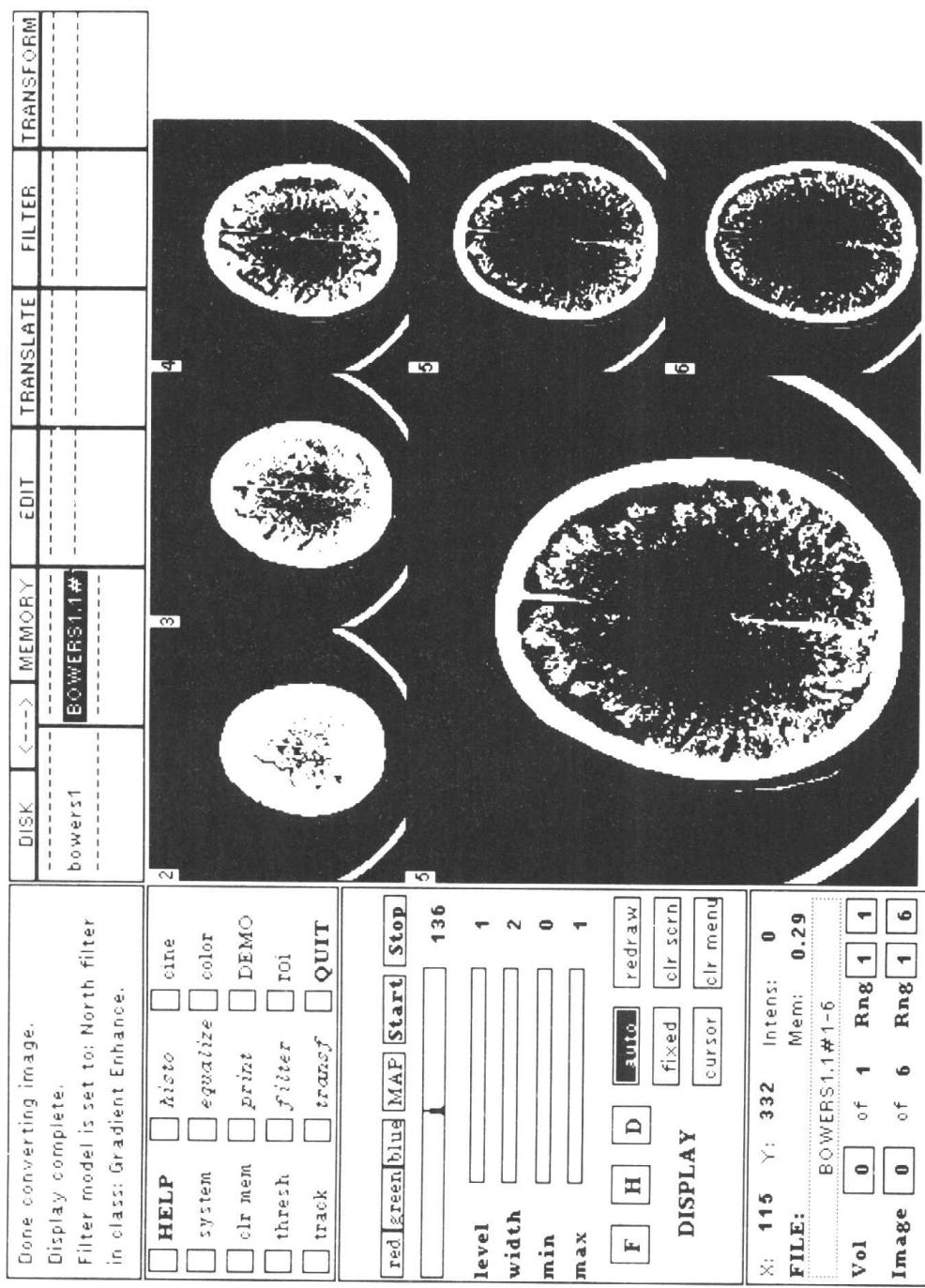


Figure 9.10 A User Interface in Biomedical Engineering: Display of CAT Scans. (With permission; G. Haden.)

that provide multiple-color images; the black-and-white images shown here provide a striking display when viewed in color. The design of this system was similar to the systems described previously; all control information is on the left of the window. There are some different features, however. For example, there is a transcript window at the top of the view that provides continuous information about the display. Next, buttons are shown as checkboxes with the names beside them. This type of display can be easily created by instantiating small buttons and writing the names of their function adjacent to the button rectangle (see the class `DialogView`). Gauges are used for color control in this example and `SelectionInListViews` provide other image information as shown at the bottom of the figure. Perhaps the most interesting part of this example is the finding that one is not constrained by the original “look and feel” of the ST-80 system: It is simple and straightforward to create alternative view appearances.

So, what are the lessons learned from examination of these different examples in several subfields of engineering? One lesson is that it is surprisingly easy to use the same representation paradigms to represent and visualize information in different fields. Although these examples do not, of course, pretend to represent all of the things that are possible to do in engineering, they do at least provide some evidence that information factoring into the MVC triad is useful in a variety of engineering situations. It would not be surprising to find that the methods suggested can be broadly applied. Another lesson is that it is easy to transform the tools supplied into unique and interesting displays of information. The last lesson is that it is straightforward to integrate these visualization and display tools with algorithmic methods as in the environmental engineering example.

9.6 Capitalizing on Abstraction and Reusability

To build object-oriented systems that assist in solving engineering problems most efficiently, one needs to capitalize on abstraction and reusability. As discussed in some detail previously, there are currently no high-level abstract objects that can be specialized that deal with topics such as design, analysis, analogy, and so on. However, there are various lower-level abstractions in the ST-80 system that can be used for system building. Furthermore, there is considerable code that is reusable and does not have to be recreated. The next few paragraphs examine some of the classes that can be used for system building. The reader should consider if it is possible to build abstract and reusable classes that are more specific to engineering that would augment the classes discussed next.

Table 9.2 Common Abstract Classes

Class Name	Description
Object	Superclass of all classes in ST-80
Stream	Superclass of all stream operations, PositionableStream, ReadWriteStream, TextStreams, etc.
Collection	Superclass of many classes including: Bag, OrderedCollection, Set, Array, String, Dictionary
Magnitude	Superclass of arithmetic operations including: Number, Fraction, Integer, Float, Point
Model	Adds dependencies to the class Object
View	Superclass of the views, e.g., ListView, TextView
Controller	Controller for the MVC triad: including StandardSystemController and ControllerWithMenu

Abstraction

Table 9.2 displays a number of common abstract classes in ST-80 that are subclassed in the ST-80 system. These abstract classes, when subclassed, provide many of the classes commonly needed by Smalltalk programmers. Most classes have been discussed extensively in previous chapters. This listing is provided to link with consideration of building of abstract classes for supporting applications. When building applications, it is useful to keep the abstraction paradigm in mind and to try hard not to reinvent new code. For example, in building representations for electronic circuits in Chapter 3, the **TwoInputGate** was discussed as an abstract class for the **AndGate** and **OrGate**. The **TwoInputGate** is never instantiated but provides a common protocol for its subclasses. Similarly, if one were creating a system for use in chemical engineering dealing with flow through pipes, one might create a class called **Pipe**, which could be subclassed into, for example, **PipeWithFriction** and **PipeWithLeak**.

Reusability

The issue of reusability is very keenly argued in many quarters. Engineers and programmers who create new software normally do not consider the possibility of code reuse, except for using libraries of functions or using standard modules that solve specific problems (e.g., an FFT module). Most often, software is written “from the ground up” and the concept of spending

Table 9.3 Common Reusable Classes in Smalltalk-80**The classes used for models**

Arrays
Ordered and SortedCollections
Sets and Bags
Dictionaries
Strings

The classes used for views

Buttons
DialogView
Text and CodeViews
ListView and SelectionInListViews
Various Pluggable GaugeViews

The classes used for controllers

StandardSystemController
ControllerWithMenu

time to understand how to reuse code is a foreign one. Moreover, the reuse of code is discouraged because of code infringement and the rights of writers of software. In making a new commercial product, one wishes the code to be completely new. Hence, there is a problem! Reusability permits rapid building of new systems; yet code must be in the public domain or part of purchased packages before it can be used. ST-80 code is completely reusable—but only to those who purchase the license for ST-80. Fortunately, licenses are quite inexpensive, particularly for university users. The point to be made is that, if progress toward reusability of code in engineering is to be made, libraries of useful abstractions in code will need to be made available or widely sold.

Table 9.3 displays some of the more prominent classes that are reused frequently by Smalltalk-80 programmers. The classes shown are by no means all that one should consider; however, these classes were used extensively to create the examples given previously. Consider that Table 9.3 only shows one “foreign” package of tools, Pluggable Gauges (Adams, 1987)—used for creating gauges in one of the systems described previously; the remainder of the standard tools are in the ST-80 package. If many pluggable components became available for use, it is anticipated that programming productivity would soar.

9.7 Building Object-Oriented Solutions to Problems

The major advantages of creating object-oriented solutions to engineering problems in ST-80 are (1) the availability of excellent graphics and user

interface facilities, (2) reusability of classes, and (3) integrated environmental tools that facilitate creation of applications. In considering building an application in the model-view-controller paradigm, it is useful to think of the MVC triad as representation (= model), the user interface (= view), and control (= controller).

The application builder should first consider the representation desired to determine if the information that needs to be represented can be cast into one of the available classes in ST-80. Usually, this is possible, although it may be useful to create other representations for special purposes. For example, representing a semantic net would require additional representation capabilities (see Chapter 11).

The most thought in building an application should be given to the user interface as manifested in the views. This line of thinking is somewhat different than normal application building techniques. Designing starting with the interface has the advantage that one can consider a series of scenarios or “storyboards” that represent what the user should see. The idea of a storyboard is that one creates a story of the interaction with the system. Because system components are directly controlled in the direct-manipulation interface, the application builder can readily allocate interface connections (buttons, etc.) to the conceptual components of the application.

In initially building an application, it is normally not necessary for the programmer to consider control aspects. The default facilities provided by the StandardSystemController and ControllerWithMenu are usually sufficient for an initial prototype.

There is, in some applications, the need to interface to other application programs or framework packages outside Smalltalk. Next, how to integrate such external programs to MVC-built systems will be examined.

9.8 Coupling to Traditional Solutions: Building an Integrated Problem-Solving Environment

Even though this text promotes the use of object-oriented solutions to engineering problems, there remain various problems that can be solved directly using “traditional” solution paths. Next, we consider the nature of traditional methodologies and tools and discuss how such systems can be integrated with the object-oriented approach.

Traditional Solutions

Typical computer-based programs that provide solutions to complex problems have almost always been domain-knowledge intensive. For example,

many complex and detailed design and simulation packages and algorithmic analysis packages have been created with great effort over a period of many years. Recoding such packages in an object-oriented style would be unreasonable, because detailed knowledge has been gathered and much effort expended to create special-purpose application packages. If such packages can be used to assist in problem solving, it is worthwhile to consider “coupled” systems in which object-oriented methods are coupled to standalone application packages. Efficient software realizations of algorithms fall in the domain of “traditional” solutions. For example, it would make little sense to recode a fast Fourier transform algorithm in an object-oriented paradigm if it already exists. Instead, it makes more sense to simply integrate the code with the object-oriented code.

Coupled Systems

Figure 9.11 graphically indicates the nature of coupled systems for object-oriented systems. The central core of the diagram shown contains the ST-80 language and environment. A C-language interface is shown around the core that provides an interface to various types of traditional code including, but not limited to, simulation packages, frameworks, drawing and word processing packages, algorithms, and inference tools. ST-80 provides a C-language interface that permits message sending in ST-80 that results in execution of a C procedure. Variables can be passed and returned between the languages. The basic methodology used is to link C object files to the Smalltalk virtual machine and to provide primitive calls that are built into methods that call C code. For example, a Smalltalk method for running a primitive in a class called **InferenceEngine** might be

```
run
  "InferenceEngine run"
  <primitive: 10001>
```

The “<primitive: 10001>” statement provides a link to specific user-defined compiled code that will be executed when the message *run* is sent to the class **InferenceEngine**. Details about how to interface primitives to Smalltalk-80 are given in Chapter 12.

Inference

Inference engines can be built in Smalltalk with some effort [see, e.g., Piersol (1987)], yet are relatively slow in comparison with optimized inference

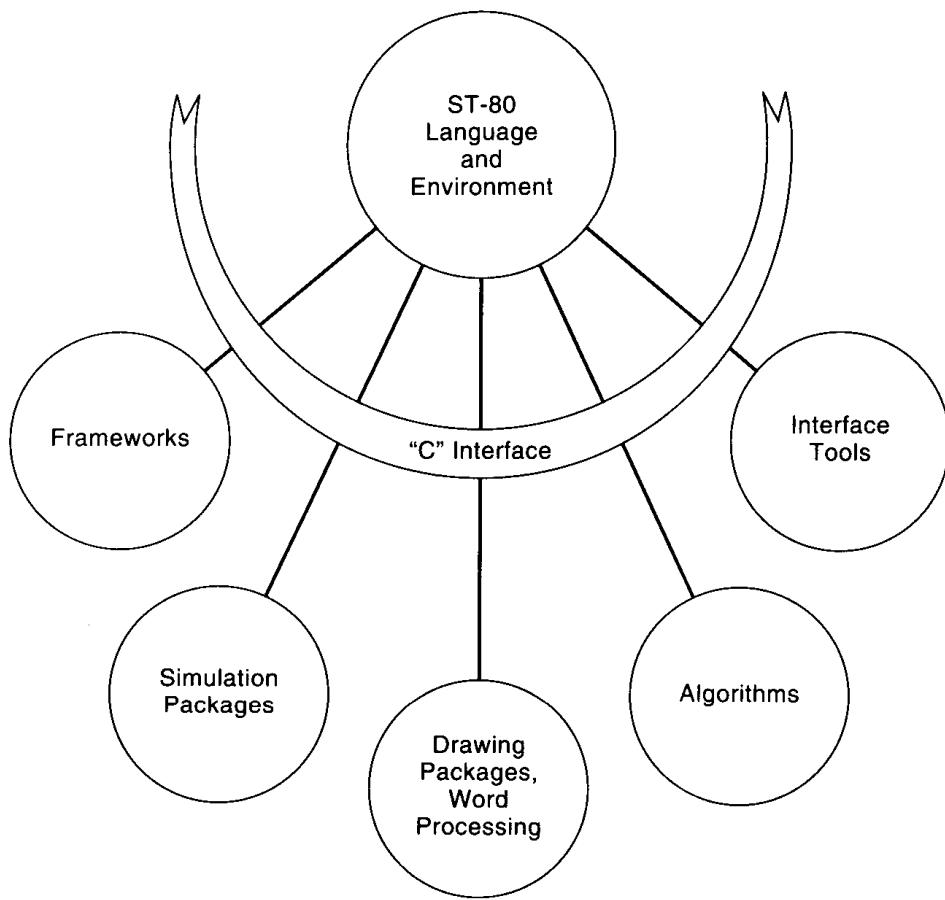


Figure 9.11 Coupling Smalltalk-80 to External Software.

engines written in completely compiled languages. An inference engine is computer code that supports the running of rules of the general form:

IF (various conditions) THEN (various actions).

A typical use of inference engines has been to build expert systems. In building applications of various types using object-oriented techniques, inference capabilities are of use in such activities as (1) inferring what the user knows and does not know (e.g., for forming a user model), (2) providing advice based on information acquired, and (3) analyzing heuristic information. Two advantages of some inference systems built using C or other compiled languages are high speed and the optimization of rules into networks [see, e.g., OPS5; Brownston et al. (1986)]. Coupling engines of this type to Smalltalk makes sense in order to improve system reasoning performance.

Algorithms

Algorithms coded in compiled procedural languages can be coupled to Smalltalk in order to secure a speed advantage. For applications that do not require intensive computation, coupling to algorithmic implementations outside Smalltalk does not make sense. In general, one can hope for only, *at best*, an order-of-magnitude speedup. Moreover, the compilation facilities of Smalltalk have improved with each successive release; hence, differences in speed between Smalltalk and procedural languages may well disappear in time.

Simulation Packages and Frameworks

Complex simulation packages and existing design and analysis frameworks are good candidates for coupling to object-oriented problem solutions. Using the same argument as for algorithms, speed improvements can be achieved. However, the greatest gains are likely because simulation code that has been developed over long periods of time can be used and does not have to be recoded.

Word Processing, Equations

Although early releases of the Smalltalk environment (Versions 2.5 and earlier) contained the capabilities for word processing and building of equations using the form editor, these capabilities were not as well developed as some commercial packages designed specifically for these tasks. Text editing capabilities remain about the same in Release 4. However, integration of Release 4 with the host environment permitted using the word processing and equation editing facilities of the host software. For example, using Microsoft Windows, any of the standard word processing programs are probably more useful than the ST-80 editing capabilities. It is worth noting, however, that Xerox Special Information Systems markets a superb text editor written in ST-80 that is part of the Analyst package, an integrated office environment (XSIS, 1989).

The Architecture of Coupled Systems

Figure 9.12 displays an architectural diagram which shows how coupled systems can be implemented in the ST-80 environment. At the top of the figure is the standard user interface. In the second layer, various application

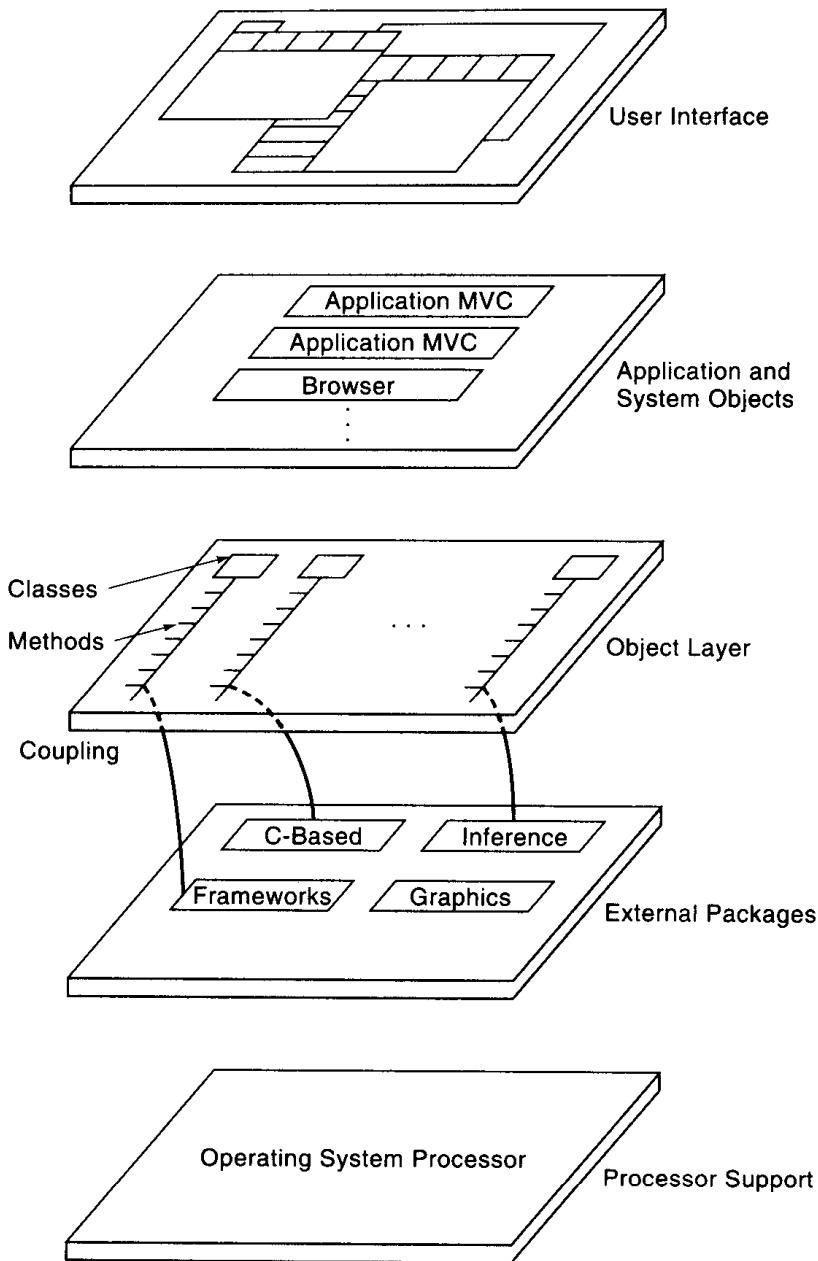


Figure 9.12 The Architecture of Coupled Systems.

MVC triads are shown which are supported by various classes in the third layer of the diagram. Within various classes, there should exist methods that contain primitives that have links to packages external to the ST-80 environment. Inside ST-80, sending a method to a particular class that has the primitives implemented gives no hint that external code is being called; the programmer can program entirely in Smalltalk but retain the speed and complexity reduction capabilities afforded by using coupled systems. On the negative side, as soon as special-purpose links to external programs are added, Smalltalk programs become much less portable.

9.9 Summary

Object-oriented methodologies fit engineering problems. In this chapter, an attempt has been made to show that common problems in engineering can be either solved or facilitated by using object-oriented methodologies. Ideas surrounding the needs of engineering science and design were projected into specific examples of building systems for use in several different engineering domains. Various common needs were identified, including representation of knowledge and simulation. The upcoming chapters will take a closer look at these and allied areas that are useful for engineering problem solving.

References

- Adams, S. (1987). *Pluggable Gauges User's Manual*. Knowledge Systems Corporation, Cary, NC.
- Auer, Ken (1989). Which Object-Oriented Language Should We Choose? *Hotline on Object-Oriented Technology*, Vol. 1, No. 1, November, 1–6.
- Bourne, J., H. Liu, C. Orog, S. Uckun, K. Hensley, and C. Buenzli (1989). Intelligent Systems for Quality Control. *Proceedings of the Third International Conference on Expert Systems and the Leading Edge in Production and Operations Management*. Management Science Department, University of South Carolina, May, 321–332.
- Brownston, L., R. Farrell, E. Kant, and N. Martin (1986). *Programming Expert Systems in OPS5*. Addison-Wesley, Reading, MA.
- Debelak, K. (1990). Personal Communication.
- Haden, Gerald (1989). Medical Image Processing: Using Object-Oriented Programming Techniques. MS Thesis, Vanderbilt University, Nashville.
- Piersol, K. W. (1987). *Humble V2.0 Reference Manual*. Xerox Special Information Systems, Xerox Corporation, Pasadena, CA.

Shafer, G., and R. Logan (1987). Implementing Dempster's Rule for Hierarchical Evidence. *Artificial Intelligence*, Vol. 33, No. 3, 271–298. XSIS (Xerox Special Information Systems), Analyst (1989). Xerox Corporation, Pasadena, CA.

Exercises

- 9.1 Consider each of the domain examples given in Section 9.5. Give a critical appraisal of the examples and develop a similar example of your own in each field. This task will require consultation with others in the various disciplines discussed.
- 9.2 Take one of the examples in Exercise 9.1 that you are very familiar with and describe the specific characteristics of the model-view-controller that you would create for the application.
- 9.3 As a more complex exercise, develop a detailed architecture for the system you described in Exercise 9.2, patterned after the architectural example given in Section 9.8.
- 9.4 Imagine that you are asked to describe how a system would be developed for the application that you have chosen to work with in the preceding exercise. Create a series of scenarios that you can present to a group. One technique is to create a set of “storyboards” that describe how a user would interact with the application. In this case, a “Storyboard” set would consist of multiple sketches of the user interface at different times and would also show how the interface relates to a model or models and to control of the application.
- 9.5 At the beginning of this chapter, the commonalities between engineering fields were mentioned. See if you can find out what some commonalities are. Write a short paper (e.g., not more than 5 pages) about interfield analogies in engineering. For example, consider how design is conducted in different engineering fields, such as electrical engineering and chemical engineering. Interview knowledgeable persons in these fields and determine how they do design. Then compare and contrast your findings.

Constraint Methods

10.1 Basic Constraint Concepts

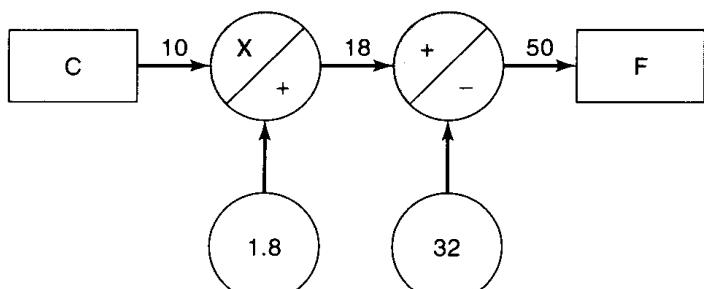
What is the meaning of the term *constraint*? The dictionary defines a constraint as: “something that restricts, limits or regulates” (Morris, 1981). In mathematical terms, “A constraint consists of a set of variables and a relation on these variables” (Güsgen, 1989) and in terms of objects, “A constraint expresses a desired relationship between one or more objects” (Leifer, 1988). A favorite example used to explain constraints is the representation of the Fahrenheit to centigrade conversion equation (Abelson and Sussman, 1985; Leifer, 1988):

$$C = \frac{5}{9} (F - 32)$$

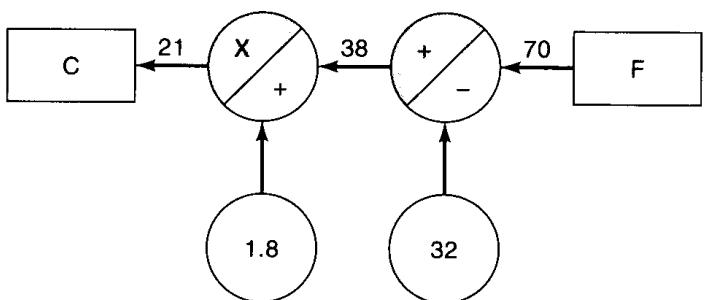
or

$$F = 1.8 * C + 32$$

This simple relationship between the two variables C (centigrade) and F (Fahrenheit) can be expressed graphically as shown in Figure 10.1. This figure illustrates propagation of information in two directions in the graph. From centigrade to Fahrenheit the constraint is shown in the gray part of each circle; in the reverse direction, the constraint is shown in white. For each case, a numerical example is given: Converting 10°C to 50°F is shown in the first case and converting 70°F to 21°C is shown in the second case. The operations specified are inverses of each other when the data flows in opposite directions. In this simple case, the meaning of constraint is simply a mathematical relationship between the variables entering a node. In more complex situations, more complicated combination functions or even symbolic computations could be used in a node.



Centigrade to Fahrenheit (gray constraint)



Fahrenheit to Centigrade (white constraint)

Figure 10.1 Example Constraint Graphs for Temperature Conversion. Circles are operators; gray indicates forward propagation (to the right); white indicates reverse propagation (to the left). Squares indicate variables.

Note in the temperature conversion example, the direct analogy between the constraint graph and the object-oriented methods discussed in this text. Each node can be considered to contain a mathematical relationship, with the kind of relationship determined by which inputs are present. That is, for example, when an input to the leftmost constraint is present, the multiplication constraint is used, and in the reverse direction, division is used. Objects can be used to represent each constraint, for example:

```

Object subclass #Constraint
  instanceVariableNames:
    'constraintHolderForward
     constraintHolderReverse
     input
     output'
  classVariableNames:""
  poolDictionaries ""
  category: 'constraint-examples'
```

where the input and output values are stored in the designated instanceVariables and the constraintHolders contain blocks of code that couple the input to the output and the output to the input, respectively.

Propagation of information through a simple network of this type is called local propagation (Steele, 1979). When there are circularities in the graph, one cannot solve the graph directly; relaxation methods (Leler, 1988) can be used to find a solution.

In the context of previous chapters when ST-80 implementation of dependencies was discussed, it is worth noting that there is a direct analogy to the dependencies studied in Chapter 8 and the constraint graphs. In the MVC paradigm, objects can contain a list of dependent objects to which a message is sent whenever information in the object (model) is changed. It is easy to see how the dependency methodology is essentially the same as a constraint graph. Each object is linked to another (equivalent to adding to the dependents list) and an action is taken when data arrives at a node, which is equivalent to *update*: in the MVC.

Most interest in constraint programming has been in the area of equation solving and graphical solutions. Various examples abound; some of the more prominent are reviewed in the following discussion. The major emphasis in the topic from the viewpoint of engineering analysis and design is the direct mapping of the constraint methodology to the simulation of physical systems. In the examples given in the previous chapters about engineering systems, several systems were discussed that utilize the constraint technique. For example, the chemical engineering example involved propagation of information among various separators and the digital circuit simulator passed information between chips in a circuit. A detailed look at the latter example will be given later in this chapter.

10.2 Using Constraints

The history of the use of constraint techniques begins with the dissertation of Ivan Sutherland in 1963 on SKETCHPAD (Sutherland, 1963) which investigated the use of graphics as a communications medium for user-machine interaction. In SKETCHPAD, the user interacted with the display using a light pen to manipulate a drawing. Concepts such as "rubber banding" and other graphics manipulation ideas, which are now commonplace, were introduced by Sutherland in this landmark work that was well before its time. The "rubber band" concept serves to illustrate the graphics constraint idea. In this metaphor, a line appears to follow the cursor as it is moved on the screen and does not become permanent until the user, for example, clicks the mouse button. If one wishes to create a graphical image, a square for example, the

lines drawn can be made to snap to vertical and horizontal constraints. That is, the lines can be constrained to be only vertical or horizontal. This idea is incorporated in most modern computer drawing packages.

TK!Solver (Frank et al., 1987) is a constraint solution system for numerical problems that is rule based in the sense that one is permitted to enter constraint equations in any order according to a specified format. Once entered, TK!Solver solves the equations, using relaxation techniques (Borning, 1981). Relaxation changes the numerical values in sets of equations to try to minimize the error expressions of the constraints. The basic idea is to represent approximations to the constraints as a set of linear equations and find a least-mean-square fit to the equations. The solutions created to sets of equations by TK!Solver are provided for any of the variables entered in the system; hence, an output can be determined in terms of an input, and vice versa. Equations and data can be stored in simple declarative statements. Also supported in TK!Solver Plus is the ability to combine logical operations as well as mathematical calculations. TK!Solver has enjoyed some commercial success during the last decade.

ThingLab (Borning, 1979, 1981) was probably the first attempt to create a constraint-based simulation laboratory. Based on roots in Smalltalk, ThingLab extended the capabilities of Smalltalk by providing the user the ability to graphically program constraint relationships. The ThingLab view, constructed in Smalltalk, was quite similar to the ClassBrowser of Smalltalk with four selection-in-list views across the window and a picture view (i.e., the display of the constraint diagram) across the bottom of the window. The four selection-in-list views contained: (1) the names of classes of objects that could be displayed, (2) the aspect of the object selected (e.g., the prototype of the object), (3) messages, and (4) arguments to the messages. To use the browser, one could make selections across the four views; for example: “triangle → prototype → inset → line,” where “triangle” was selected in the leftmost list, “prototype” in the next list, and so forth. This style of interface provided the capability for the user to create objects in the picture view without understanding the syntax of any language. Borning presented using ThingLab to create a temperature conversion system that included gauges on the input and output of the constraint network. Figure 10.2 displays a drawing that shows how ThingLab could represent the temperature conversion problem. The gauges show the relationship between centigrade and Fahrenheit; changes in one gauge propagate through the constraints and register on the other gauge.

Steamer (Hollan, Hutchins, and Weitzman, 1984) was constructed to evaluate the use of artificial intelligence for computer-based training systems. The domain chosen was for training naval personnel about steam engines in ships. The major advance made in this project was the representation of a steam propulsion plant on a computer screen, complete with gauges and dials that

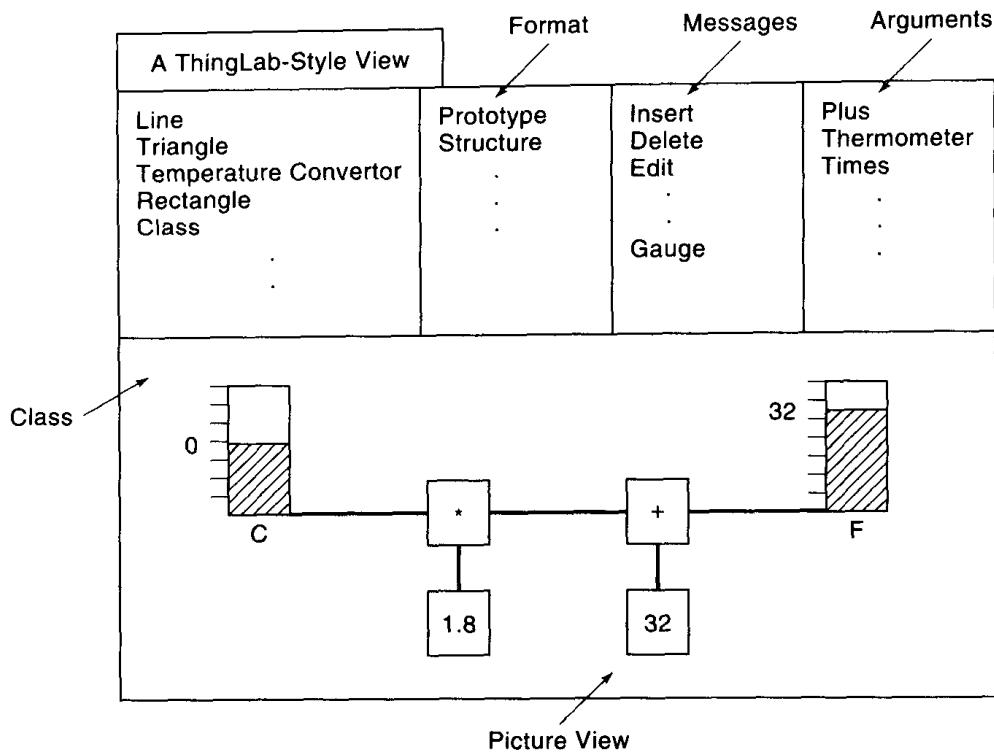


Figure 10.2 The ThingLab Style Representation of a Temperature Conversion Constraint System.
[After Borning (1981).]

monitored the operation of the steam engine. Steamer was implemented using object-oriented methods in ZetaLisp using Flavors. Figure 10.3 shows a graphical rendition of a steam engine that is presented to the user of the system. The conceptual fidelity in this system was good, in that the user could recognize the schematic of the engine and relate it to the physical system. Furthermore, operating the system produced the close-to-reality results needed for teaching how the system operated. Steamer is described here because it can be considered to be a constraint-based system in that objects representing the various elements in a steam propulsion system were implemented as objects that acted as constraints in the overall description of the engine.

Constraints other than mathematical equation solvers or graphics constraint systems are also well known such as constraints in spreadsheets or symbolic equation-solving systems. In spreadsheets, equations at cells in a matrix constrain values at other cells in the matrix. Perhaps the spreadsheet is the most commonly used constraint-based system. Other systems also exist for symbolic manipulation, for example, MACSYMA (MathLab, 1983), used for solving symbolic mathematical equations.

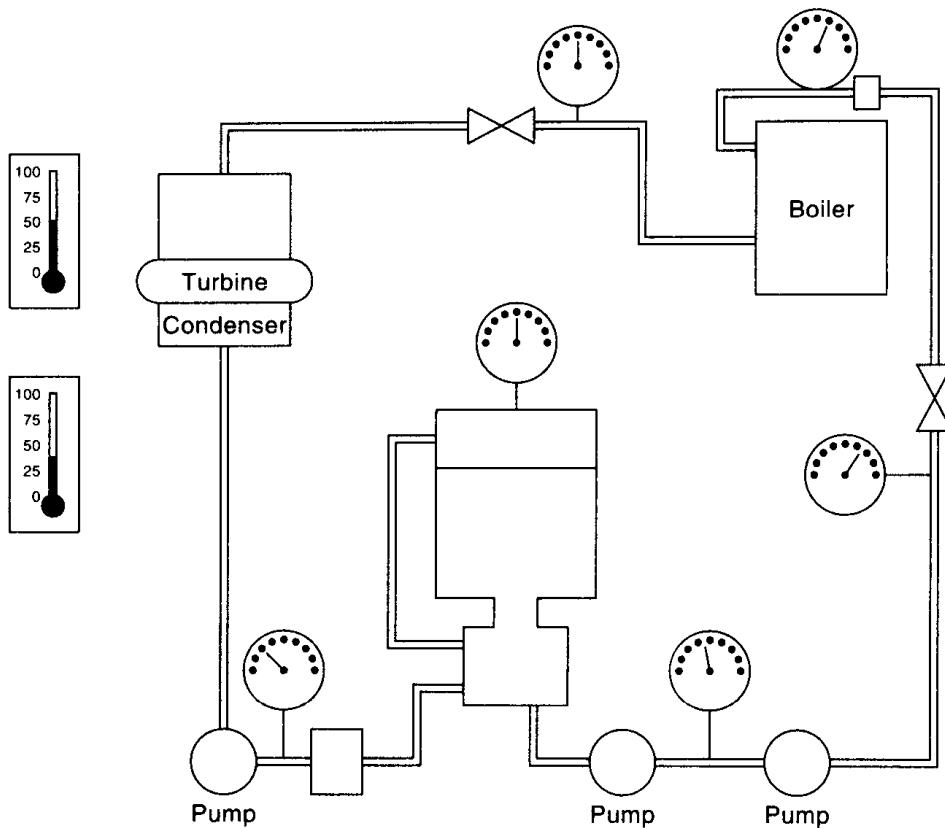


Figure 10.3 A Model of a Steam Propulsion Plant. [After Stevens (1982); with permission. © Bolt Beranek and Newman, Inc.]

10.3 Implementing Constraints: An Example for Digital Circuit Simulation (DCS)

This section of the chapter shows how to implement a simple constraint system for the digital circuit simulation system that appears in different aspects in several sections of this textbook. The implementation shown is in ST-80 and demonstrates only single directional flow of information. In digital circuits, actual information flows in just one direction through electronic components; hence, it would make no sense to implement bidirectional flow for this example.

DCS Concepts

In Chapter 3, a discussion of object-oriented analysis culminated in the demonstration of the use of ACOM (application/class organization method)

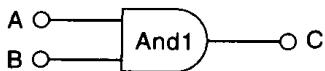


Figure 10.4 An And Gate.

notecards for analyzing the component elements of an application. The application addressed in that chapter was a digital circuit simulator. The descriptions of the components of the digital circuit simulator specified on ACOM notecards in Chapter 3 will be turned into ST-80 code and tested in this chapter.

First, as background, the utility of digital circuits is to combine logical operations to create various electronic systems such as adders, counters, flipflops, and even, at a much higher level of complexity, digital computers. The purpose of a digital circuit simulator is to create computer models of actual physical electronic devices which mimic the behavior of the physical devices themselves. For very simple circuit elements, this is not difficult to understand. Consider the two very simple circuits shown in Figures 10.4 and 10.5 which display an AND gate and a combination of an AND gate and an OR gate, respectively. The AND gate's behavior is simply that a 1 appears on the output (C) whenever a 1 is placed on both inputs (A and B) simultaneously. Likewise, output of the OR gate is a 1 whenever either input is high (a 1). The digital circuit combination of an AND gate and an OR gate shown in Figure 10.5 will obey these combination rules, propagating information from the input to the output (i.e., A and B to E) according to the wiring of the circuit and the definition of the *constraints* contained in each of the circuit elements.

Complexity in the design of digital circuits can be lowered by using hierarchical design; that is, each element in a circuit can represent combinations of other elements in a hierarchical fashion. Consider Figure 10.6 which shows an example of how circuit elements can be combined hierarchically. At the top level of this figure is shown a D flipflop, a gate that has two states Q (1) and notQ (0) that change back and forth depending on the state of an input clock signal. The actual implementation of the D flipflop is a collection of AND gates connected as shown in the lower plane in the figure. At the top level of the representation, only the functionality of the flipflop is observable and the

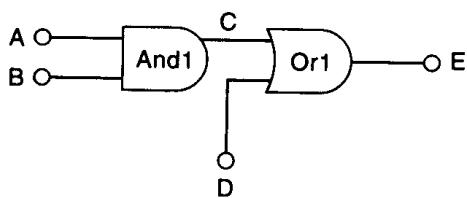


Figure 10.5 A Very Simple Digital Circuit.

underlying implementation, in terms of the interconnected AND gates, is hidden. This style decomposition can be carried on to various levels in order to create systems of arbitrary complexity. To simulate the behavior of the flipflop, the combined behaviors of the AND gates will yield the behavior of the flipflop. The problem, of course, in circuit simulations of this type is that implementation of the behavioral characteristics of the top-level circuit in terms of their constituent elements will result in very slow simulations when the number of circuit elements grows large. Nevertheless, the concept of hierarchical design is a powerful one. Next, specific methods for implementing the behavior of primitive AND, OR, and NOT gates is examined. Using the information given in the following discussion, it would be straightforward to implement the flipflop example given in Figure 10.6.

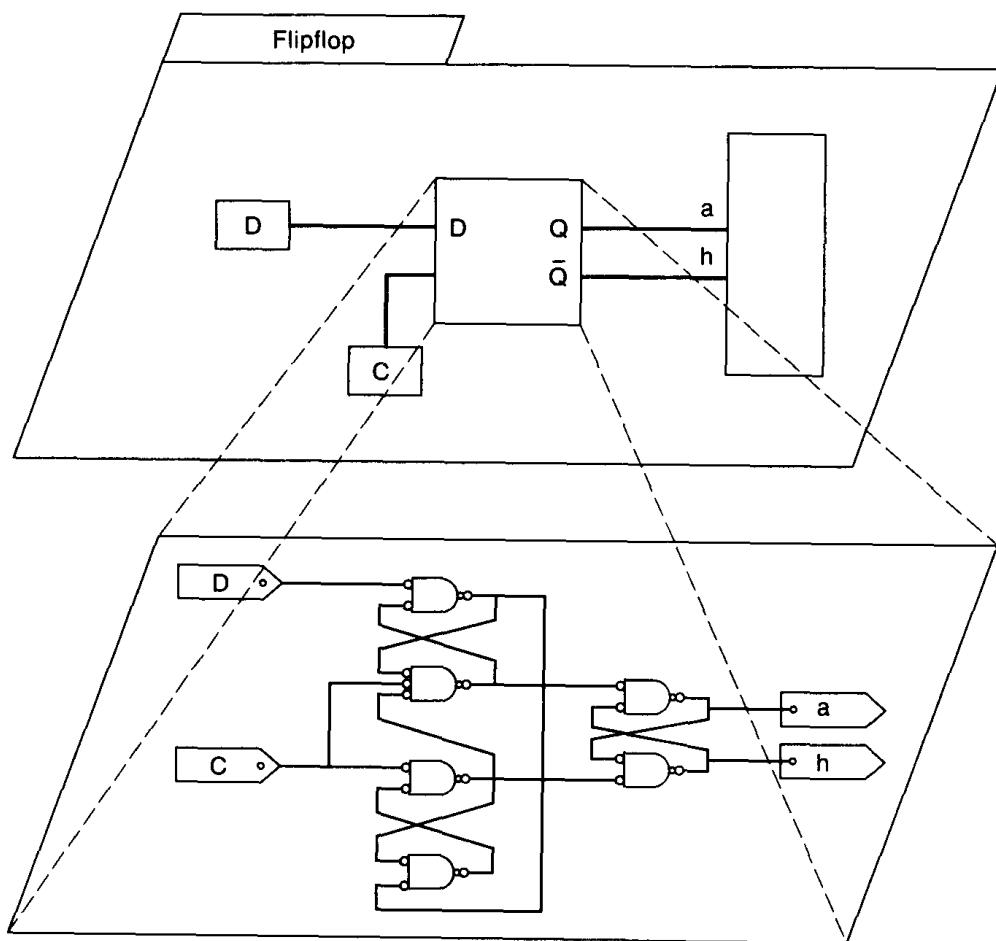


Figure 10.6 Hierarchical Digital Design Concept.

Table 10.1 The Block Evaluation Protocol

Message	Explanation
value	Evaluate the block represented by the receiver
value: anArgument	Evaluate the block represented by the receiver using one argument
value: value:	Use two arguments
value: value: value:	Use three arguments
valueWithArguments: arguments	Use anArray of arguments
Example	
[:x:y (x*x) + (y*y)] value: 3 value: 4 returns: 25	

Implementing Constraints

Implementing a constraint or relationship between several variables is quite simple in ST-80. The concept of a block may be utilized in which code may be stored for later execution. A block has the form:

[:variable1 :variable2 | |localVariable1 localVariable2| codeBlock ...]

where the variable names preceded by colons are variables that are passed to the code block. Above, where “codeBlock...” is shown, Smalltalk code would be inserted. A block with no arguments can be evaluated simply by sending the message *value* to the block and values can be passed to a block using several different messages, for example,

```
constraint value: aValue
constraint value: aValue1 value: aValue2
```

Table 10.1 displays a partial set of messages for block evaluation and gives a simple example. This style of forming constraint expressions is quite convenient for specifying the behavioral characteristics of gates in the digital circuit simulation. The specific implementation for several elementary gates will be discussed in the next section.

Models

In this chapter, the description of building a digital circuit simulator will focus on the model; Chapter 13 will examine the interface to the user.

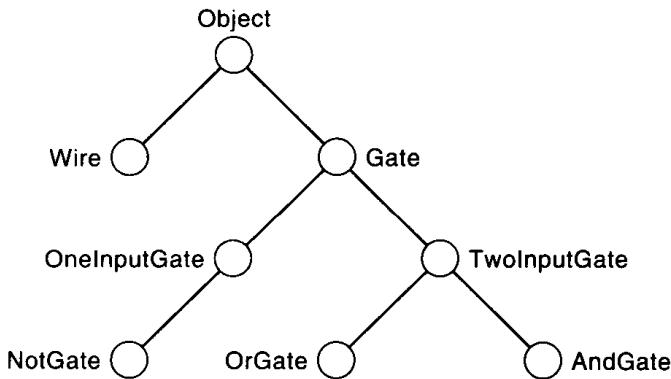


Figure 10.7 The Digital Circuit Simulator Example Class Hierarchy.

Figure 10.7 displays a simple hierarchy created for the DCS. **Gate** is a subclass of **Object**; **OneInputGate** and **TwoInputGate** are subclasses of **Gate** and both **AndGate** and **OrGate** are subclasses of **TwoInputGate**. **Gate** is used in Chapter 13 to keep track of the location of any gate on the screen; hence, it is a superclass of all other gates that need to inherit this capability.

The digital circuit simulation system hierarchy could be quite complex. For example, many electronic components such as nand, nor, plus, times, invertors, memory elements, clocks, analyzers, meters, power supplies, signal sources, probes, and so forth could be implemented. The example system presented contains only the minimum components needed to explain a few basic concepts. The reader is encouraged to add additional circuit elements as an exercise. Table 10.2 lists the components defined for the example system.

Table 10.2 Components in a Digital Circuit Simulation

Component Name	Description
AndGate	Two inputs, one output; when two inputs are high, the output is high; inherits from TwoInputGate
OrGate	Two inputs, one output; when either input is high, the output is high; inherits from TwoInputGate
NotGate	One input, one output; inverts the input; inherits from OneInputGate
Wire	Connects components
TwoInputGate	Abstract component; two inputs, one output
OneInputGate	Abstract component; one input, one output
Gate	Abstract component; monitors display location

Table 10.3 Constraints for DCS Components

Component	ConstraintBlock
AndGate	[:input1 :input2 (input1 = 1) & (input2 = 1) ifTrue: [1] ifFalse: [0]]
OrGate	[:input1 :input2 (input1 = 1) (input2 = 1) if True: [1] ifFalse: [0]]
NotGate	[:input (input = 1) ifTrue: [0] ifFalse: [1]]

Gate Models

Table 10.3 presents the constraint code for each of the gates used in the DCS. As shown, each constraint contains the number of input variables of the gate. The output value is returned as the value of the block evaluated.

Wire Model

Table 10.4 displays the definition and protocols for the class **Wire**. **Wire** is a subclass of **Object**; instances of **Wire** are used to connect components in the digital circuit simulator. Basically, a wire remembers its value and where each end of the wire is connected. If a value is sent to a wire object using *set: aValue*, the wire remembers the value sent and propagates the value to wherever the wire is connected.

Implementing Delays

In the example given, delays are not implemented. However, it would be quite straightforward to insert delays in the constraintBlocks for each of the components. ST-80 provides a class named **Delay** which permits real-time delays, useful for animation. For example,

(Delay forMilliseconds: 1000) wait.

or

(Delay forSeconds: 1) wait.

will delay the process in which the delay is embedded for 1 second. One can

Table 10.4 The Wire Protocol and Code

```
Object subclass: #Wire
  instanceVariables:
    'inputConnection
     outputConnection
     value'
```

Protocol	Method	Comment
Initialize	init	Sets value = 0, makes output an OrderedCollection
Access	inputConnection: aGate	Connects input
	outputConnection: aGate	Connects output
	set: aValue	Sets the value
	value	Retrieves the value

```
Object subclass: #Wire
  instanceVariableNames: 'inputConnection outputConnections value name
pointsDescribingWire'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'digital-demo'
```

Wires are used to connect digital components.

Wire methodsFor: access

```
getWirePoints
  ^ pointsDescribingWire
```

```
inputConnection: aGate
  inputConnection := aGate.
```

```
name
  ^ name
```

```
name: aName
  name := aName
```

```
outputConnection: aGate
  outputConnections add: aGate.
```

```
set: aValue
  value := aValue.
  outputConnections isEmpty
    ifTrue:
      [Transcript show: self name printString.
       Transcript show: self value printString.
       Transcript cr.]
    ifFalse: [outputConnections do: [:eachGate| eachGate propagate]]
```

```
storeWire: aWire
  pointsDescribingWire := aWire.
```

```
value
  ^ value
```

Wire methodsFor: initialize

```
init
  value := 0.
  outputConnections := OrderedCollection new.
```

Table 10.5 Code for the Digital Circuit Simulator

```
Object subclass: #OneInputGate
    instanceVariableNames: 'inputWire outputWires constraint delay'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Digital Demo'

OneInputGate methodsFor: initialize

init
    delay := 0.
    outputWires := OrderedCollection new.

input: aWire1 output: aWire2
    inputWire := aWire1.
    outputWires add: aWire2.
    inputWire outputConnection: self.
    aWire2 inputConnection: self.

OneInputGate methodsFor: action

propagate
    |constraintOutputValue|
    constraintOutputValue :=
        constraint value: inputWire value.
    outputWires do: [:aWire|aWire set: constraintOutputValue]

OneInputGate subclass: #NotGate
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Digital Demo'

NotGate methodsFor: initialize

init
    super init.
    constraint :=
        [:input| (input = 1)
            ifTrue: [0]
            ifFalse: [1]].

Object subclass: #TwoInputGate
    instanceVariableNames: 'inputWire1 inputWire2 outputWires constraint delay'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Digital Demo'

TwoInputGate methodsFor: access

constraint
    ^ constraint

inputWire1
    ^ inputWire1
```

Table 10.5 *Continued*

InputWire2

 ^ inputWire2

TwoInputGate methodsFor: action

propagate

 | constraintOutputValue |

 constraintOutputValue := constraint value: inputWire1 value value: inputWire2 value.

 outputWires do: [:aWire | aWire set: constraintOutputValue]

TwoInputGate methodsFor: initialize

init

 delay := 0.

 outputWires := OrderedCollection new.

input1: aWire1 input2: aWire2 output: aWire3

 inputWire1 := aWire1.

 inputWire2 := aWire2.

 outputWires add: aWire3.

 inputWire1 outputConnection: self.

 inputWire2 outputConnection: self.

 aWire3 inputConnection: self.

TwoInputGate subclass: #OrGate

InstanceVariableNames:"

classVariableNames:"

poolDictionaries:"

category: 'Digital Demo'

OrGate methodsFor: initialize

Init

 super init.

 constraint := [:input1 :input2 |

 (input1 = 1) | (input2 = 1)

 ifTrue: [1]

 ifFalse: [0]].

TwoInputGate subclass: #AndGate

InstanceVariableNames:"

classVariableNames:"

poolDictionaries:"

category: 'Digital Demo'

AndGate methodsFor: initialize

Init

 super init.

 constraint := [:input1 :input2 |

 (input1 = 1) & (input2 = 1)

 ifTrue: [1]

 ifFalse: [0]].

AndGate class

InstanceVariableNames:"

make an assignment such as

```
minuteDelay := Delay forSeconds: 60
```

and then send “minuteDelay wait” at some later point. For example, for the digital circuit simulation system, different delays might be inserted into each of the constraints to simulate various delay times present in different gate types. Thus, one might have a constraintBlock that contains

```
[:input1 :input2|andGateDelay wait.  
  (input1 = 1) & (input2 = 1)  
   ifTrue: [1]  
   ifFalse: [0]]
```

where andGateDelay has been set to be a particular time delay.

Code for the Digital Circuit Simulator Example

Table 10.5 displays the code for the digital circuit simulation models presented previously. Code is organized in the form provided by printing out classes in the ST-80 system. The initial information presented for each class shows its superclass, instance variable names, class variable names, poolDictionaries, and the category in which it is contained. Then, in alphabetic ordering by protocol, the methods are shown for both instance and class methods. The printouts capture all information contained in the system browser, including comments, shown here in italics after the initial definition of the class.

10.4 Wiring and Testing Example Digital Circuit Simulations

Several examples of how to wire together objects and test the digital circuit simulations are given next. To facilitate the testing and observation of variables in these examples, variables have been declared as global to make it easy to inspect changes in the examples given. In normal programming practice, the use of global variables, as shown in these examples, would not be employed.

Testing a Component. The easiest example to begin with is to create one digital component and test it. To create the AND gate shown in Figure 10.4, one can follow the steps given in Table 10.6 which shows first the creation of three wires: A, B, and C. A new AND gate, called And1 is created and the wires just

Table 10.6 Example of Creating an AND Gate**AndGate class methodsFor: examples****example1**

"AndGate example1"

A := Wire new init. "Create a wire with the global name A"
 B := Wire new init. "Create a wire with the global name B"
 C := Wire new init.

And1 := AndGate new init. "Make a new AndGate"

And1

 input1: A "Attach the wires on the Gate"

 input2: B

 output: C.

B set: 1. "test by setting the inputs"

A set: 0. "and inspecting the output wire"

C inspect.

A set: 1.

C inspect.

Table 10.7 Example of Wiring Together Two Gates**example2**

"AndGate example2"

A := Wire new init. "make wires"

B := Wire new init.

C := Wire new init.

D := Wire new init.

E := Wire new init.

And1 := AndGate new init. "make the and gate"

And1

 input1: A

 input2: B

 output: C.

Or1 := OrGate new init. "make an or gate"

Or1

 input1: C

 input2: D

 output: E.

B set: 1.

A set: 1.

D set: 0.

E inspect.

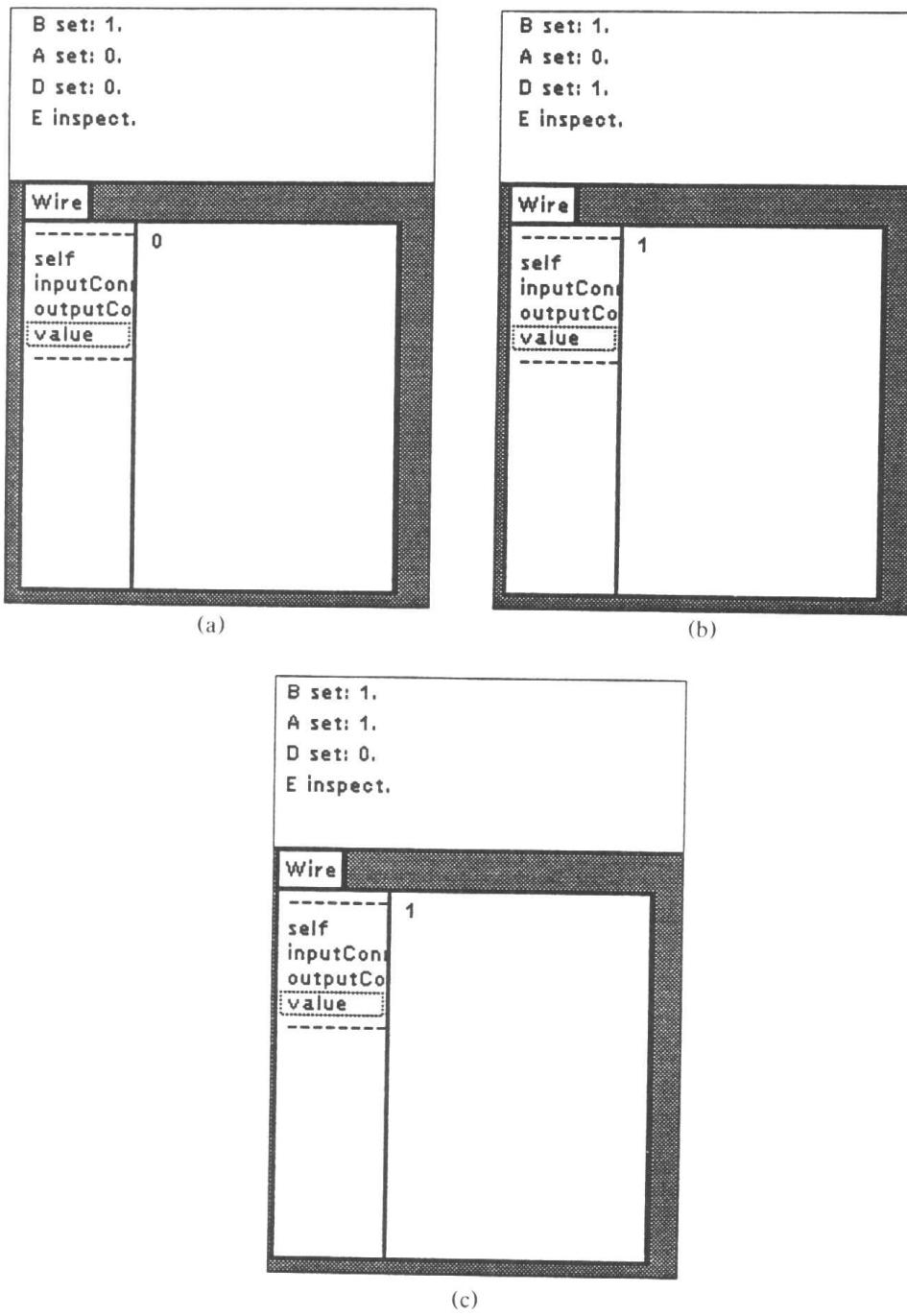


Figure 10.8 Testing the Circuit in Figure 10.5.

Table 10.8 Example of Creating the Functionality of a NAND Gate**example3**

"AndGate example3"

A := Wire new init. "Create a wire with the global name A"
 B := Wire new init. "Create a wire with the global name B"
 C := Wire new init.

And1 := AndGate new init. "Make a new AndGate"

And1
 input1: A "Attach the wires on the Gate"
 input2: B
 output: C.

Not1 := NotGate new init. "Make a new NotGate"

D := Wire new init. "and an output wire"

Not1 input: C output: D.

A set: 1. B set: 1. D inspect. "test the combination NAND gate"
 A set: 0. B set: 0. D inspect.

instantiated attached using the *input1:input2:output* message. Table 10.6 displays the code used for each of these steps. At the end of the table, the behavior of the gate created can be tested by sending the input wires created 1's and 0's and inspecting the output by sending the output wire the message *inspect*. This latter message will create an inspector window in which one can observe the value contained in the wire object as new values are placed on the input wires.

Wiring Gates Together. Table 10.7 demonstrates how to wire together two gates as shown in Figure 10.5. First, all the wires in the simulation are identified (A, B, C, D, and E) and new wires created using "Wire new init." Next, the two logic components are created and the wires for their inputs and outputs specified. Finally, the circuit created is tested. Figures 10.8a, b, and c demonstrate the testing of this circuit. In each of the three cases, A, B, and D are set to different values and E is inspected. As shown, in each case the wire E is

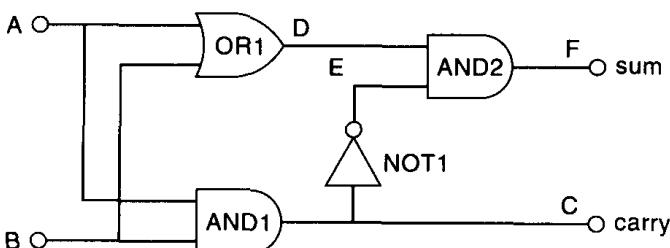
**Figure 10.9** The Half-Adder.

Table 10.9 Example of Creating a Half-Adder**example4**

```

"AndGate example4"
"A Half Adder example"

A := Wire new init.
B := Wire new init.
C := Wire new init.
D := Wire new init.
E := Wire new init.
F := Wire new init.
And1 := BasicAndGate new init.
And1 input1: B input2: A output: C.
And2 := BasicAndGate new init.
And2 input1: D input2: E output: F.
Or1 := BasicOrGate new init.
Or1 input1: A input2: B output: D.
Not1 := BasicNotGate new init.
Not1 input: C output: E.
A set: 1. B set: 0. F inspect. C inspect.

```

inspected and the value that the wire contains shown in the right window of the inspector. The reader should verify that the values shown are correct.

A Nand Gate. Table 10.8 displays an example of creating the functionality of a NAND gate, that is, a combination of an AND gate followed by a NOT operation. As in the previous example, wires are defined and the output of the AND gate is connected to the output of the NOT gate.

A Half-Adder. The final circuit example is shown in Figure 10.9, along with the testing code in Table 10.9. This circuit is the same half-adder circuit described in an earlier chapter. The functionality of the half-adder is to sum the two inputs, providing a summation and a carry signal at the output. Hence, a 1 and 0 on the two inputs create a sum equal to 1 and a carry equal to 0. Setting both inputs to 1 creates a 0 sum and a 1 for the carry. The annotations provided on the figure match the designations for the wires given in Table 10.9.

10.5 Enhancing the Digital Circuit Simulation System: An Example in Tutoring Systems

Figures 10.10 and 10.11 provide a concrete example of the use of the constraint propagation methodologies for implementing a tutorial system for digital circuits (Balasubramanyam, 1990). Figure 10.10 shows a window created

ANALYZER	QUIT	HELP	LAWS	GET PROBLEM	EXPRESSION TO CIRCUIT	CIRCUIT TO EXPRESSION	KNOWLEDGE LVL
	PROBLEM STATEMENT WINDOW						TUTOR WINDOW
	<p>The expression is: $\neg(x_1 + x_2)$</p> <p>For the given expression, construct the simplified logic circuit using the gates supplied on the leftmost column of your screen. Use the options - label input and label output in the yellow button menu for creating the input and output ports respectively</p> <p>You may also analyze the created logic circuit by using the analyzer option in the yellow button menu.</p>						
		<input type="button" value="CAN'T DO"/>	<input type="button" value="DONE!"/>		<input type="button" value="CLEAR ANSWER"/>		
							DIGITAL SYSTEMS
							<p>A digital system is a combination of devices designed to manipulate quantities that can be represented in digital form. Unlike an analog system, where the quantities can vary over a continuous range of values, a representable quantity in a digital system can take on only discrete values.</p> <p>Although the real world is mainly analog, representing a system in digital form offers advantages of greater accuracy and precision, easier information storage, easier design and operation and higher robustness. The area of digital systems spans the system design, the logic design and the circuit design. Through TSDI, the student will learn about the basic concepts involved in the logic design of a digital system. Logic design involves determining how to interconnect basic logic building blocks efficiently to perform a specific task.</p>
							<p>One of the most important applications of digital systems has been in the field of electronics and that is switching systems. A switching system consists consists of one or more inputs and one or more outputs which take on digital values. TSDI will be covering the combinational type of switching system in which the output values depend on the present values of inputs and not on past values. This is in contrast with sequential type where the past values of inputs influence the the present values of outputs. The basic building blocks to construct combinational switching system (or circuit) are logic gates. A logic designer must determine how to interconnect these logic gates in order to convert the input values into desired output values (or signals).</p>

Figure 10.10 A Digital Circuit Tutorial System Implemented Using Constraints.

ANALYZER	QUIT	HELP	Expression to Circuit	Circuit to Expression	STUDENT	Map Factor: 1
	EXPRESSIONS/STUDENT DATA					
	1 2 3 4 5 6 7 8 9 10	<p>For the given expression, enter the simplified expression.</p>				
LEVELS/STUDENTS						

The following section describes the procedure to input an circuit-to-expression problem into the system at a selected index. Remember, you are required to input a circuit here. Upon selecting the kind of problem (use button) the system will list a sequence of numbers on the left view. These numbers represent the indices at which the problem will be stored. Then, select the index at which you want the problem to reside. If there is a problem already existing at the selected index then all the information associated with it will automatically appear at the appropriate places.

On the right-view, key in the text that will appear as the problem statement for the student. In addition to a formal problem statement, the text may include hints, warnings and other related information. After the completion of this task, press the yellow-button-menu and select 'accept' option to store the text. The system responds by presenting you with a fill-in-the-blank view into which you must key in the category to which the problem belongs. Also, construct the circuit which you want to present to the student as as problem by making use of the button-gates listed on the leftmost column. Once the circuit has been made, select the 'store' option from the YBM to register the problem into the system.

Figure 10.11 The Instructor Interface to the Digital Circuit Tutorial System.

that assists a student in learning about digital circuits. The individual circuits that can be used are graphically depicted as icons on the left side of the figure and a graphics view in which a circuit can be graphically composed is shown immediately adjacent to the right of the icons. The remainder of the views in the system communicate with the student giving the problem statement, providing tutoring, and furnishing text presentations about aspects of the circuit created. Control is via the buttons shown at the top of the figure and include help and choosing whether the student wants to learn about how digital expressions can be expressed graphically, or vice versa. Figure 10.11 displays the instructor interface in which the instructor can structure the knowledge that is to be presented to the student. The same capabilities are used; that is, the icon builder mechanisms remain but capabilities are added for providing text editing and structurization of lessons.

The design of the electronics tutor evolved from the simple constraint mechanisms described previously. The ability to create icons related to objects that constrain descriptions of the behavior of electronic components coupled with the capability of coupling components together with wire objects provides a powerful mechanism for assisting students in visualizing how signals propagate through an electronic circuit.

10.6 A Petri Net Example

The final example of a constraint propagation system is the implementation of a Petri net simulator. The example given here will be expanded and discussed in Chapter 14, including a comparison with discrete-event simulation methods. Petri nets (Peterson, 1981) are useful mechanisms for simulating activities in the world. Petri nets have four parts: (1) a set of places that represent conditions, (2) a set of transitions that represent events, (3) an input function that represents a precondition to a transition, and (4) an output function that represents a postcondition of a transition. Places can be thought of as nodes in a semantic net that hold tokens representing the fulfillment of conditions. Arcs are drawn between places and transitions to indicate flow between entities in a Petri net. Transitions provide control by firing when each of their input places contain at least as many tokens in them as arcs from the places to the transitions. Imagine a series of repositories for tokens (i.e., the places) that are separated by transitions and a set of tokens (things) that can migrate through the net according to conditions placed at the transitions. When a transition fires, tokens on the input are placed at the output and propagation proceeds until no further propagation is possible. Consider the example shown in Figure 10.12 which demonstrates the use of a Petri net for modeling the building of a house. The transitions in this net are shown by the narrow

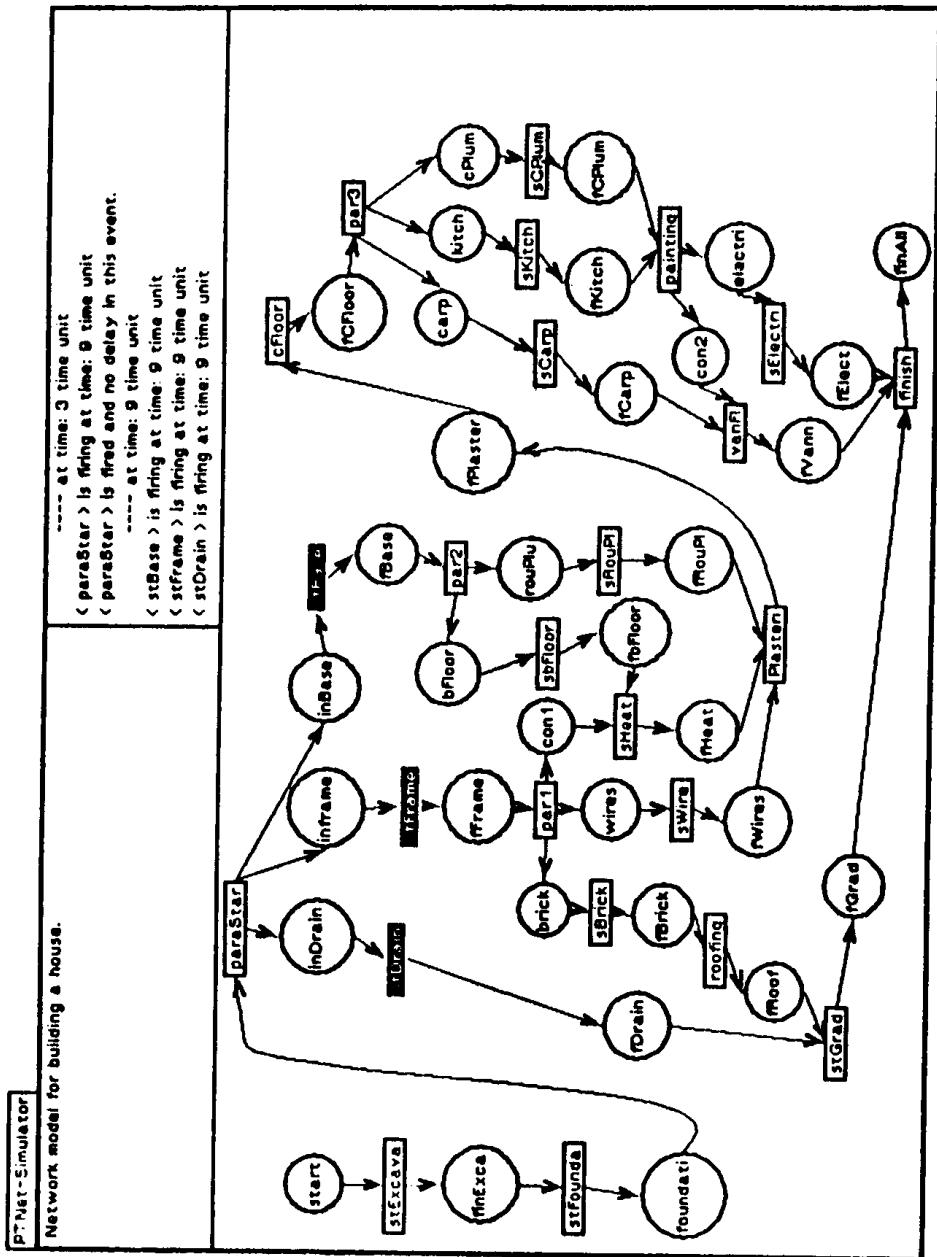


Figure 10.12 Building a House: An Example of the Use of Petri Nets.

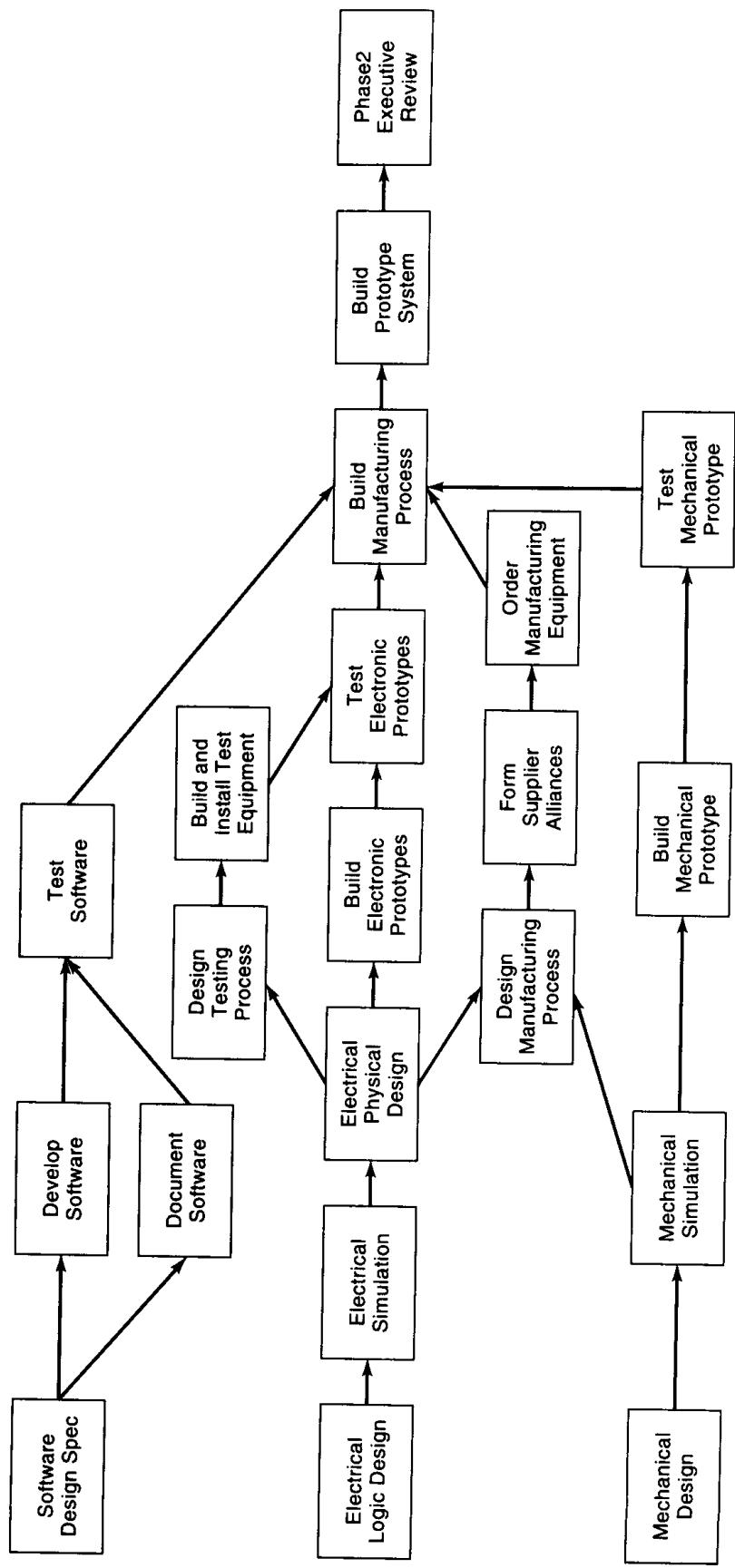


Figure 10.13 Petri Nets for the New Product Introduction Cycle. [With permission; Dehner (1990).]

rectangles and the places are denoted by circles. Beginning at the input place "start," a transition "stExcava" (start Excavation) is made to the new place (i.e., condition) "finExcava" (finish Excavation). Activity in the net continues throughout the net, sometimes dividing into parallel activities until the house is built. This example is an implementation of an example given by Peterson (1981), page 72.

The example of the house building system is constructed as a timed Petri net; that is, at a transition, a delay can be introduced to simulate the amount of time an activity will take. Class **Delay**, as discussed previously, is used for this implementation. The remainder of the implementation is quite similar to the digital circuit simulator except that objects in the simulation obey the Petri net rules, as indicated previously, and use a time delay to introduce more realism into the simulation. The top right of the window of the example shown contains a transcript that shows which node is firing as the simulation runs.

Figure 10.13 displays another implementation of a Petri net simulation for a different domain—that of introducing a new product (Dehner, 1990). The figure shown displays part of the new product introduction process which deals with mechanical, electrical, and software design. The transition nodes are shown as rectangles in this diagram and places are implicitly embedded in the arcs. As design of a product moves from transition to transition in this simulation, the rectangular forms containing text are reversed from white to black while the transition occurs. In this way one can observe the simulated progress of a design along the parallel paths in the simulation.

Petri nets pass discrete events (tokens) between transitions. The transitions contain constraints that are in the form of decisions about when and where to pass a token. Because the actions taken by the transitions are discrete, Petri nets are useful for modeling and simulating physical systems in which noncontinuous models are used. The two preceding examples have presented a first look at simulation using discrete events; Chapter 14 will give a more detailed presentation and specifically examine how to create discrete-event simulators.

10.7 General Use of Constraint-Based Techniques

This chapter has attempted to develop the case for using constraint-based techniques for simulation of engineering activities. Constraint techniques are useful for many different types of simulations, including solution of mathematical equations and representation of physical objects that can be separated into discrete components, each with internalized behaviors (e.g., digital circuits, pipes, etc.). Finally, constraints implemented in objects can realize even relatively advanced concepts such as Petri nets that provide a logical framework for

simulation of many types of models. Adding time to simulations has been shown to be easily added in the ST-80 framework using the **Delay** class.

Finally, the major lesson to be learned from this chapter is that object representations can be used naturally to model and simulate the real world. Objects have built-in capabilities for maintaining state, implementing behavior, and communicating that makes the object-oriented paradigm ideal for constraint-based systems. There are, however, other simulation methodologies such as discrete-event-based simulation that can be used to enhance simulation capabilities for timed simulations. These methods will be examined in Chapter 14.

References

- Abelson, H., and G. J. Sussman (1985). *Structure and Interpretation of Computer Programs*. McGraw-Hill, New York.
- Balasubramanyam, Ravi (1990). A Tutorial System for Digital Circuits. MS Thesis, Vanderbilt University, Nashville.
- Borning, A. (1979). ThingLab—A Constraint-Oriented Simulation Laboratory. PhD Dissertation, Stanford University.
- Borning, A. (1981). The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 4, October, 353–387.
- Dehner, J. (1990). Personal Communication.
- Frank, W., L. Kloepping, M. Nordahl, K. Olafsson, and D. Shook (1987). *TK!Solver Plus Manual*. Universal Technical Systems.
- Güsgen, H. W. (1989). *CONSAT: A System for Constraint Satisfaction*. Morgan Kaufmann, Los Altos, CA.
- Hollan, J. D., E. L. Hutchins, and L. Weitzman (1984). STEAMER: An Interactive Inspectable Simulation-Based Training System. *AI Magazine*, Vol. 5, No. 2, Summer. In *Readings from the AI Magazine* (1988) 581–593.
- Jiang, Zhihe (1990). An Intelligent Petri-Net-Based Simulation System. MS Thesis, Vanderbilt University, Nashville.
- Lefer, William (1988). *Constraint Programming Languages*. Addison-Wesley, Reading, MA.
- MathLab, MACSYMA Reference Manual (1983). The MathLab Group, Laboratory for Computer Science, MIT, Cambridge, MA.
- Morris, William (Ed.) (1981). *The American Heritage Dictionary of the English Language*. Houghton Mifflin, Boston, MA.
- Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ.
- Steele, G. L. (1979). The Definition and Implementation of a Com-

puter Programming Language Based on Constraints. MIT Artificial Intelligence Laboratory Technical Report 595, MIT, Cambridge, MA.
Sutherland, I. E. (1963). SKETCHPAD: A man-machine graphical communications system. Technical Report 296, MIT Lincoln Laboratory, Cambridge, MA.

Exercises

- 10.1 Extend the half-adder example to implement a full-adder. Refer to the diagram for a full-adder found in Figure 3.12.
- 10.2 Add additional components to the example system, for example, a NAND gate, a logic level sensor, and so on. Verify that the components that you create work correctly.
- 10.3 Add delays to the constraints of the DCS example and test.
- 10.4 Verify and explain the code given in Tables 10.6 through 10.9.
- 10.5 Review the STEAMER project by reading the paper by Hollan, Hutchins, and Weitzman listed in the references. Write a short paper about how to create constraints similar to those in STEAMER.

Representing and Using Engineering Knowledge

A key problem in creating computer-based systems for applications in engineering is understanding how to represent knowledge. Knowledge can be represented in so many different ways that it may be difficult to match available representational structures to problem requirements. There has been little guidance available to designers of applications on how to choose representations and how to understand the rationale behind the choices that are available. Most frequently, system designs involving knowledge-intensive tasks are driven by which software packages are available and/or prior designer familiarity with various representation paradigms. This chapter will try to systematically uncover the rationale behind using several different representation methodologies and to show how the different techniques relate to each other. Many of the representation types discussed are derived from the field of artificial intelligence (AI). This field has been seriously engaged for some decades in attempting to unravel the representation issue (Brachman and Levesque, 1985). First, an attempt will be made to summarize the general characteristics of engineering knowledge and the general representation categories that are appropriate for different tasks. Next, basic concepts in knowledge representation will be presented and finally, examples for each of the major representation categories will be given. An explicit effort will be made to reveal when object-oriented methodologies are useful and when these methodologies can be beneficially combined with other approaches.

11.1 Engineering Knowledge

What is engineering knowledge? Most engineers give little thought to this question. Rather, consideration of knowledge structurization is almost

implicit in problem solving, unseen yet fundamental. Software application projects created by engineering students normally follow no formal software engineering process, such as the waterfall model (Royce, 1970). This model of the software engineering process steps designers through the following type of problem-solving script: requirements analysis, specification fabrication, software design, implementation, integration, and finally operation. A test and verify cycle is included at each stage of the design. The lack of software systemization in engineering has translated into the preeminence of ad hoc software methodologies for engineering problem solving. Even relatively rigidized methodologies for software specification do not yet appear to have greatly influenced knowledge-intensive engineering problem solving. For example, the waterfall method is not widely used, except implicitly, in many engineering applications. Indeed, there appears to be considerable difficulty in linking traditional software engineering methods (Schach, 1990) to knowledge-intensive engineering problem solving. In part, this situation derives from the complexity of understanding the characteristics of engineering knowledge and the difficulty of specifying ways to represent that knowledge. Specification of techniques for organizing the building of software (i.e., the field of software engineering) has not become widespread for building engineering applications. An allegation made in this text is that the introduction of object-oriented methodologies in engineering will eventually alleviate this situation.

Next, common knowledge types used in engineering are examined. For several different fundamental engineering tasks, such as design and analysis, the knowledge types required are discussed.

Models

Engineers use models extensively to describe the way the world works and to organize knowledge for designing new entities. The major need for a model is for capturing and organizing knowledge in frameworks in a way that permits easy viewing, understanding, and retrieval. In addition to the need for capturing declarative knowledge, engineering uses two other dominant modeling activities: (1) creating and using mathematical models and (2) employing heuristic models or “rules of thumb.” Knowledge represented by these three modeling types serves as the core of the engineering discipline.

Declarative Models. Models that capture declarative knowledge are essential. Declarative knowledge (i.e., static knowledge) can be thought of as encyclopedic knowledge in a particular case. Reference information, libraries of designs, tabular listings of algorithms and their actions, histories, anecdotal information, glossaries, and interpretations are among the kinds of declarative information

that need to be captured. Some attempts have been made to capture knowledge on a grand scale (e.g., the CYC project; Lenat, Prakas, and Shephard, 1986); unfortunately, human knowledge is too broad to permit encyclopedia capture. A more reasonable solution is to capture extensive knowledge in restricted domains.

Techniques for organizing, understanding, and communicating knowledge are essential. Without domain-specific knowledge, algorithmic and reasoning methodologies become virtually useless. Hence, a critical requirement for knowledge representation in engineering (or any field, for that matter) is the creation of acquisition, access, inquiry, and manipulation of static knowledge methodologies that can be used in applications as well as analysis and design systems.

Mathematical Models and Algorithms. The working of the world in terms of physical-law-based mathematical equations has traditionally been a central part of knowledge representation methods used by engineers. Many facets of the dynamic world that engineering seeks to characterize can be described in terms of differential equations. Both static and dynamic behaviors of systems as well as stochastic (random) behaviors can be modeled mathematically. Mathematical techniques, descriptions, and algorithms have been built, taught to students, and used in engineering for many years.

Algorithms are defined as “Any mechanical or recursive computational procedure” (Morris, 1981). Algorithmic descriptions of procedures for doing things tend to become almost “black-box” or object-like in nature over some period of time. The term *black box* refers to an encapsulated procedure which takes an input and produces an output, without the user needing to understand the internal working of the procedure. In contrast, an observable system is termed a *glass box*. In time, new knowledge about a procedure moves from the glass-box stage to the black-box stage; that is, the knowledge becomes encapsulated. Encapsulated procedures are routinely used throughout engineering without a deep understanding of the internal working of procedural contents. An example is the use of the fast Fourier transform (FFT) in various fields. The FFT converts time-domain signals to frequency-domain signals, providing the user with an understanding of the frequency content in a signal. Such analyses would be useful, for example, in measuring the vibration in a helicopter blade, examining vibrations in a tall building, understanding sonar signals, or computing the frequency components of brain waves. Users of algorithms of this sort are content to understand the requirements for input into the black box and understand the implications of the data produced. A direct analogy can be made with the basic nature of object-oriented encapsulation in which the user is shielded from understanding the internal functioning of an object and must only send a message to the object to elicit a desired behavior.

Mental Models. The term *mental model* is used to indicate a model formed in the mind (Gentner and Stevens, 1983). The term is not often employed in engineering but is useful in this description to indicate the distinction between mathematical models and heuristic or rule-of-thumb models of how things work. Engineers, as problem solvers, use mental models of the world in terms of rules and, in all likelihood, qualitative models. Rules capture heuristics in terms of causality, that is,

IF (a condition or conditions is true) THEN (a conclusion)

and qualitative models capture the working of physical mechanisms in terms that indicate qualitative relationships. Each of these model types are explained next.

Rules: Reasoning Models. The use of rules has been studied extensively in the area of “expert systems” (Hayes-Roth, Waterman, and Lenat, 1983) in which attempts have been made to mimic the reasoning of experts. By capturing many causal relationships in a domain, numerous software systems have been built during the last decade that mimic the reasoning of experts. The use of these so-called “shallow” reasoning systems has been commonplace in engineering. A shallow model is one that captures only surface level causalities but does not deal with “deep” or algorithmic law-related knowledge. There remain many debates about the utility of “shallow” versus “deep” reasoning. For the purposes of this text, both are considered important—the first captured in rules and qualitative models and the second in physical-law-related algorithms.

Qualitative Models. Qualitative models (Bobrow, 1985) can be thought of as mental models of physical systems. Quantitative information is absent for qualitative models and changes are represented in terms of qualitative change. Whereas quantitative representations employ equations that relate directly to physical laws, qualitative models describe how things work in a much less precise way, usually simply in terms of variable increases, decreases, or no change. For example, if a variable (e.g., the pressure in a model of a steam plant) in a model increases, that increase may cause another variable to decrease. In turn, that decrease may cause other changes, and so forth. This type of description is excellent for visualization and understanding how causality-linked components operate in a physical system.

The use of qualitative mental models in engineering is probably quite widespread, even though there has been little attention given to this concept in many engineering circles. Consider, for example, the use of a model of a steam engine. Although the way a steam engine works is describable by mathematical

equations, it is also describable by qualitative relationships such as “if the pressure at point A is increasing rapidly, then the plant is about to blow up.” Experts who understand physical systems may well capture complex models in these terms, permitting almost instant visualization of relationships between entities in a system. Such models aid in understanding how a physical system works without having to understand equations that describe a system.

Knowledge Types Needed for Engineering Tasks

Analysis. To build applications for engineering analysis, declarative knowledge models, algorithms, and rules appear to be the dominant representation techniques required. Analysis includes situation observation and application of evaluative techniques. Here, declarative knowledge representations that provide information on how to conduct analyses are primary requirements followed by the availability of algorithmic tools that permit transformation of data into understandable terms. Once evaluative data is secured, rule-of-thumb methods for interpretation are employed. These steps are typically carried out when humans analyze problems. Machine-based analysis and/or machine-facilitated analytical systems can capture these same steps using the representational mechanisms described.

Design. Design, the creation of new artifacts, begins with a specification of requirements and ends with the product, as discussed in Chapter 9. Representation must include ways of specifying what has been done in the past because much of design is conducted by making analogies to prior designs. In design, heuristic reasoning is required perhaps more than in any other engineering activity. Design is thought by some to be more “art” than algorithms, meaning that designers create through innovation and experience. To capture this type of knowledge for building automated design systems, knowledge in the form of heuristic rules should be captured. Also, in design, the capability of evaluating design trade-offs is required. Such knowledge can be captured in rules as well, or perhaps in the form of networks of constraints.

Simulation. For simulation, a *runnable* model of an engineering system under study is required. The goal of simulation is to verify the behavioral operation of a system prior to actually fabricating a design. The implementation of a simulation might be a constraint model, a discrete-event model, or a set of interconnected algorithms which permit interpretation of the behavior of the system. Chapter 14 will contain information about the implementation of simulation systems using object-oriented methods.

Knowledge Organization and Object-Oriented Methods

The previous paragraphs explain some of the representation needs that are required in engineering. There are doubtless other needs as well. However, the systematic linking of representation mechanisms with engineering task needs is a continuously evolving study and no consensus knowledge about the proper means for understanding these linkages is yet available. However, an interesting observation is that different engineering tasks have similar representational needs: knowledge in the form of declarative knowledge, algorithms, rules, constraints, and qualitative models. In the upcoming sections of this chapter the ways that the object-oriented representation model can be used to facilitate knowledge representation tasks will be examined.

11.2 Representing and Acquiring Knowledge

To be useful in building computer-based systems, knowledge must be acquired and cast in some machine-consumable fashion. Books, video media, algorithms and system implementations in computer code, verbal anecdotes, and other materials currently comprise the vast corpus of engineering knowledge. To capture this knowledge in a machine-understandable way will require a significant effort in building knowledge-based organization methodologies. This section will examine some general techniques that can be used for this purpose. Subsequent sections will look at specific implementation strategies.

Representation

Two dominant forms of knowledge representation have been categorized: declarative and procedural knowledge. Declarative knowledge is “what is” or static knowledge. Procedural knowledge is “how to” knowledge or knowledge that is represented as a series of steps. During the early history of work in knowledge representation, there were many arguments about which representation style was better (Barr and Feigenbaum, 1981). Although there was considerable debate on this issue, the issue resolved itself as people came to recognize that both types of knowledge were important. From the discussion in earlier parts of this chapter, it is indeed evident that both types of knowledge are needed for representing knowledge in engineering.

Over the last two decades, there has been voluminous experimentation with different styles of knowledge representation [see, e.g., Brachman and Levesque (1985)]. The major categories that have been examined by legions of

investigators include logic-based methods, procedural representations, semantic networks, rule systems, frames, scripts, and, more recently, objects.

Logic. Early logic-based systems described the world in terms of mathematically representable sentences such as

“All transistors have three leads”

Logic systems capture many rules of this sort and through a logical inference process are able to make deductions about other facts. Prolog (Clocksin and Mellish, 1984) is an example of a logic-based language that became popular for programming in logic (see Chapter 5).

Procedural Systems. The original concept of procedural systems was to capture knowledge in small bits of code about how to do things. Now, in light of detailed evaluations of representational needs, the use of procedures is well established as one of the techniques needed to represent knowledge. In object-oriented methodologies, methods are implemented by procedures!

Semantic Nets. The word *semantic* is defined as “pertaining to meaning” (Morris, 1981). Semantic nets are pieces of knowledge linked together in a network which, when taken as a whole, have meaning. Originally developed by Quillian (1968), semantic nets were invented to serve as a model for human associative memory. As shall be seen in this and later chapters, the representation technique is useful for many different kinds of knowledge. A semantic net consists of nodes that represent objects and links that represent the interrelationships between nodes.

Rule/Production Systems. Early research on rule systems [see, e.g., Newell and Simon (1972)], then called production systems, also focused on explicating human cognition. In the 1980s, research and development of rule systems that captured human reasoning for practical purposes gained high visibility and became used throughout industry to capture heuristic knowledge in the form of if-then rules. Systems with these characteristics have become widely popular and are often referred to as *expert systems*.

Frames. Frames were developed for capturing static knowledge about the world and during the 1980s became the dominant technique for representing declarative knowledge. Frames permit representing prototypical forms which have slots (names) and values with which one can capture information that describe things. Chapter 2 presented an example of capturing knowledge in frames and also described methods of enhancing frames to couple procedures to frames. As will

be described in the upcoming paragraphs, frames and object descriptions have virtually identical characteristics for describing declarative knowledge! This point will be made clear via an example later in this chapter. Frames also share representational mechanisms with semantic nets, requiring only small changes to permit addition of different kinds of linkages to pull together nodes of information with different meanings.

Scripts. The term *script* comes from a theatrical metaphor in which actions are scripted to give players directions of what to do. Scripts, as popularized by Schank and Abelson (1977), permit descriptions of sequences of actions that should be carried out to accomplish some goal. For example, a script could be created to fabricate a printed circuit board. Scripts share a common ground with procedural methods and, indeed, can be thought of as simply packaged procedures.

Objects. Finally, the most recently investigated representation paradigm is that of objects, the central topic of this text. Objects combine most of the good features of representation techniques investigated in the past and, in addition, include other features as have already been discussed in some detail in previous chapters.

Of the major historically useful representation paradigms listed previously, frames, semantic nets, rules, and objects have played dominant roles. The novice user who simply wishes to represent knowledge in an application has a difficult problem in choosing which of the popular representations should be used because there are massive overlaps between the capabilities of the different techniques. Furthermore, once one or more representations are chosen, one must acquire knowledge prior to use. Next, the acquisition process will be examined to be followed by implementation examples.

The next sections examine the different representation types using examples and relate the characteristics of the representations to the object-oriented style.

11.3 Frames, Semantic Nets, and Object Representations

Frames

In Chapter 2, the general concepts of frames were discussed, including how to attach procedures to frames. For the purposes of comparison of frames and object representations, the example of a frame shown in Table 11.1 will be used. This frame gives the basic outline of a frame that describes the characteristics of a computer. As shown in this partially filled out example of a frame,

Table 11.1 Example Characteristics of a Frame

```
(defFrame ComputerCharacteristics
  (physical
    (memory
      (options
        (type {simm sip chip} typeConstraint)
        (maxMem {2 4 8 16 32} memoryConstraint)
        (mounting {vertical horizontal})
        (cpuSpeed {16 20 25 33})
        (userAccess {mouse Keyboard}))
      )
    )
    (functional
      (software
        (editors
          (word (characteristics (memory 2)
            (speed > 12)
            (suitability suitPointer)
            (cost 140)
            )
          )
        )
        (wordPerfect (characteristics
          ...)))
      )
      (compilers...)
      (spreadsheets...)
      (databases...)
    )
    (cost
      (vendor
        (HP
          (models
            {QS16 RS20 RS25 QS16S} typeinformer)
        )
        (IBM
          ...)
        (Apple
          ...)
      )
    )
  )
)
```

information might be included, such as the physical, functional, and cost characteristics of a computer. In each of the subcategories, additional specifications would be included. For example, for physical characteristics, information about memory, the type of mountings that are available, cpu speed, and user access methods could be included. Of course, many additional attributes might be added as well. Note the way that subframes are nested: memory, mounting, cpuSpeed, and userAccess are all subframes that contain additional information. Furthermore, any subframe can be extended in the same way. Note the

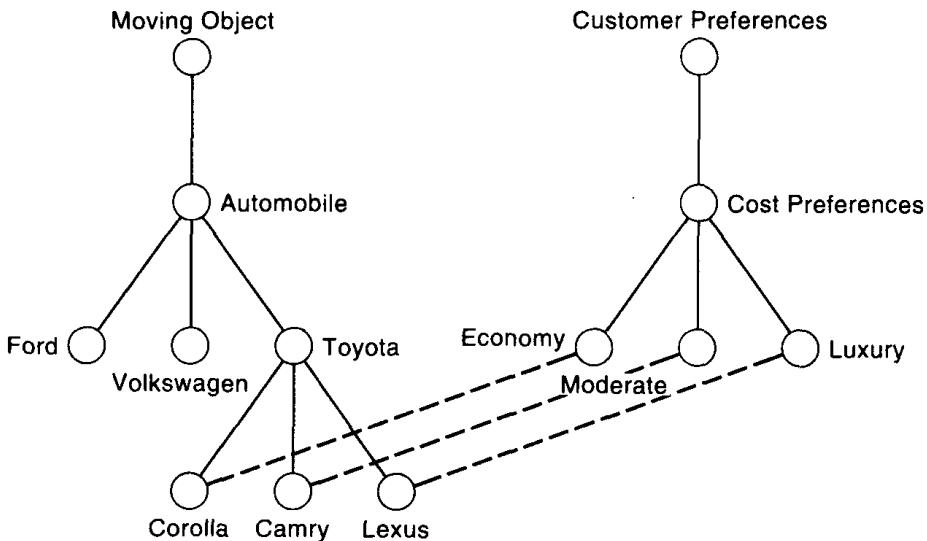


Figure 11.1 Strict Hierarchical and Tangled Hierarchies.

syntax of showing alternatives with curly braces “{ . . . }.” A purely declarative frame would simply contain information with no additional procedural information. Thus, one might retrieve the information that (computerCharacteristics physical memory options type) would contain {simm sipp dram}.

The example frame displayed in Table 11.1 is strictly hierarchical; that is, each node in the tree has only one ancestor. If multiple ancestors are permitted, then the tree is said to be “tangled.” If the tree is organized such that nodes further removed from the root can inherit information from multiple nodes closer to the root, then the organization is said to permit “multiple inheritance.” Consider the example in Figure 11.1 in which single inheritance is shown with solid lines and multiple inheritance with dotted lines. The nodes “corolla,” “camry,” and “lexus” inherit information from both hierarchical structures shown. The total structure is said to be tangled and, because information is inherited from more than one place, inheritance is multiple.

Some implementations of frames extend the paradigm to include active values, or procedures that run when a slot or slots are accessed in the frame. In the example, the names of procedures to run when information in a frame is accessed are shown following the alternatives in curly braces; for example, typeConstraint and memoryConstraint. Various implementation strategies exist such as running one procedure when a value is put into a slot or running another procedure when values are accessed. In this example, running the procedure “typeConstraint” might put information into other slots in the

same frame (or another frame, for that matter). Suppose that the subframe "cost" contains information about vendors and that some of the slots in that subframe have places to fill in the memory type. Furthermore, suppose that the programmer has created a program to examine trade-offs in selecting computer characteristics. If the user selects a particular memory type (e.g., SIMM), the procedure "typeConstraint" could access another slot and put in that slot the information that SIMM had been selected. Then, if a particular style of computer is selected, the SIMM constraint might limit the available manufacturer selections to the ones that used SIMMS as a memory mounting methodology. Thus, based on this concept of interlinking sets of constraints attached to values of slots in a frame, one could construct a system that would permit trade-offs between interrelated sets of information to be made.

A cautionary note is worthwhile here. From Table 11.1 and the preceding discussion, it is easy to see that the complexity of representing information in this fashion increases very rapidly as more detail is put into the frame. Procedures can be attached to slots or rules can even be run that manipulate the same or other frames. After some time of adding information in this way, conceptual spaghetti results. Furthermore, implementing interacting constrained relationships in this way results in the near impossibility of modification. Representing frames in graph form with links and their effects shown graphically would assist the user in understanding how a frame-implemented constraint system works, however.

Representing Frames in Objects. Frames are essentially identical to objects as far as the representation of tree-like hierarchical knowledge is concerned. It is quite simple to mimic static frame representation by using dictionaries in the instance variables of objects. Consider Figure 11.2 which shows the same implementation that was presented in Table 11.1 implemented in an object/dictionary style. An object, perhaps in a class called **DeclarativeInformation**, contains instance variables, one of which might be "computerDesignKnowledge." In this instance variable, one could store the dictionary "computerCharacteristics," and in the slots of this dictionary other dictionaries containing more information could be stored. One could even add procedures in dictionary slots to completely mimic the frame-based active value methodology. However, the same problems encountered in frames would appear. Because objects are utilized, improved understandability is achieved by making procedural methods in objects and attempting to divorce control from representation.

The end point of this discussion of frames versus objects is that frame and object representations can be used in the same conceptual way. Frames organize knowledge in hierarchical or tangled organizations. Objects also have the capability for representing just the same structures.

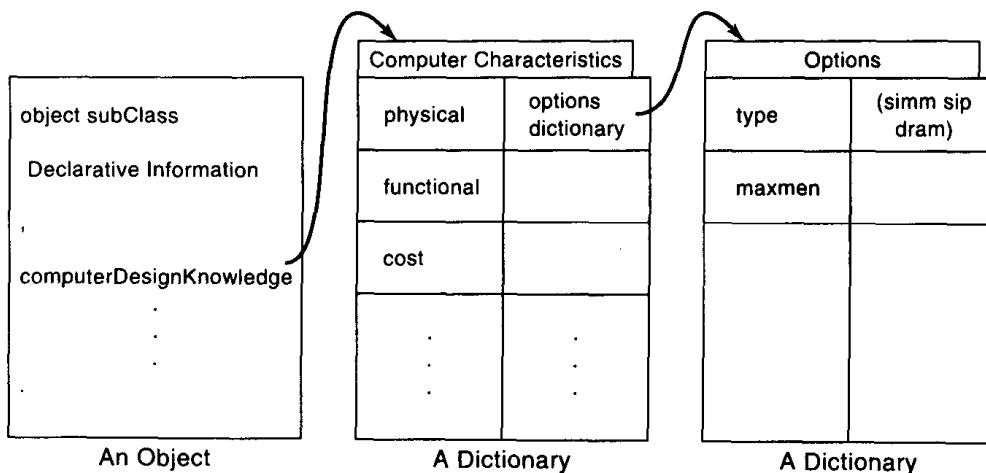


Figure 11.2 Frame Representation in Objects: Recasting the Frame of Table 11.1.

Scripts

The concept of a script is simple. One of the definitions of the term is that a script is the text of a play. Generalizing this concept extends the definition to mean that a script captures knowledge about sequences of events. Hence, scripts can be a useful representation mechanism for capturing lists of actions, such as a series of steps necessary to solve a problem.

There are various ways that one could implement a script. Because the intention is to record a set of procedures, a list of actions could be captured as procedures in a frame slot or as different terms in several slots in a frame. Actions could also be captured in if-then rules or in a list of functions. Also, scripts can adapt to a context by having alternative actions to take based on input data.

Table 11.2 displays a simple way to implement a script in ST-80. For demonstration purposes this example can be executed in a workspace, if desired. A local variable “aScript” is defined and bound to a new ordered collection. Three blocks are added to “aScript” that can be executed at a later

Table 11.2 Implementing a Script

```
[aScript]
aScript := OrderedCollection new.
aScript add: [Transcript show: 'action I'];
           add: [Transcript show: 'action II'];
           add: [Transcript show: 'action III'].
aScript do: [:aBlock| aBlock value].
```

time. Each block can contain any executable code; in this case a simple message is written on the transcript. Once “aScript” contains an ordered collection of executable blocks, the blocks can be passed one by one to another block using the *do:* message. Then, sending *value* to each executable block will run any code specified inside the block. The reader is encouraged to type in other specific actions in the blocks and observe the running of this script.

Semantic Nets

Figure 11.3 displays a semantic net representing the meaning about transistors. The primary relationships in this diagram are the “isa” links which represent the inheritance linkages in the net. The term “isa” is commonly used to mean “is a” which refers to the inheritance property of the link. That is, a transistor *is an* electronic device and a Mosfet *is a* transistor, and so forth.

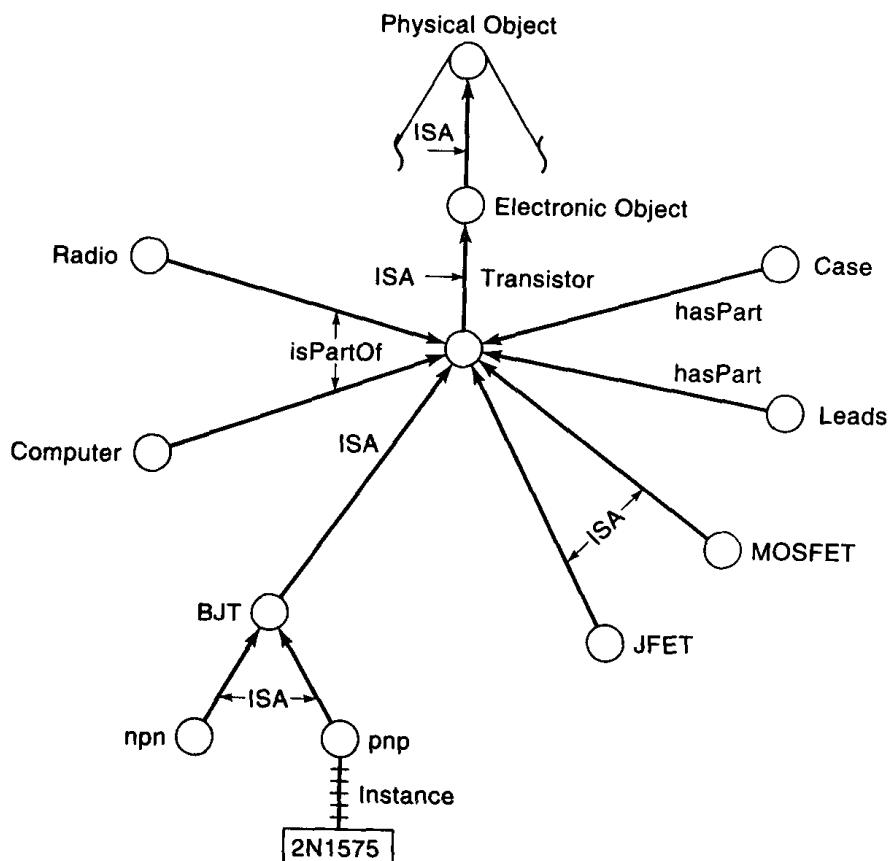


Figure 11.3 Example of a Semantic Net. [After Bourne et al. (1989).]

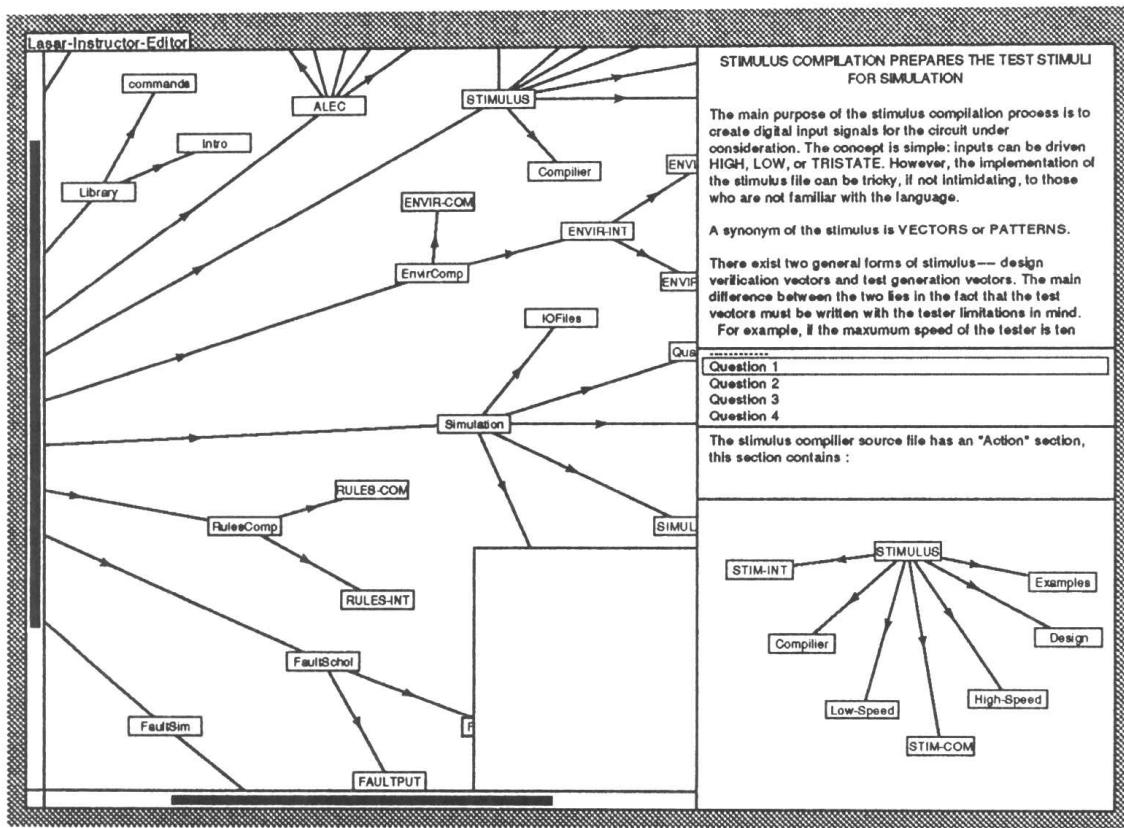


Figure 11.4 An Application Built Using a Semantic Net. (With permission; Academic Computing.)

Other kinds of links can be used, such as `isPartOf` and `hasPart`, to represent other types of semantic relationships. In principle, many kinds of links could be defined; however, the more link types that are created, the more complex the organization of the net will become.

Creating a network representation system in Smalltalk-80 is an interesting exercise. Figure 11.4 displays the window of an ST-80 application called NRL (Network Representation Language) that was created to permit construction of networks (Bourne et al., 1989). Table 11.3 shows the objects and their instance variables that were created for this application and Figure 11.5 recasts the same information as a class network.¹ The design intent of NRL was to permit users to create semantic nets by moving forms onto a canvas to create network nodes and to provide the ability to connect nodes with different types of links. This general network representation system has served as the basis for

¹NRL was originally created in ST-80, Version 2.5. The hierarchy shown is changed for Release 4, by the substitution of `ControllerWithMenu` for `MouseMenuController`. The Release 4 version is available as indicated in the Appendix.

Table 11.3 Class Hierarchy for NRL

Object ()
View ('model' 'controller' 'superView' 'subViews' 'transformation' 'viewpoint' 'window' 'displayTransformation' 'insetDisplayBox' 'borderWidth' 'borderColor' 'insideColor' 'boundingBox')
NetView('activeNode' 'canvas' 'greekedView' 'externModel' 'partMsg' 'acceptMsg' 'menuMsg' 'displayLabel')
ContextEditorView ('parentNet')
Object ()
Controller ('model' 'view' 'sensor')
MouseMenuController ('redButtonMenu' 'redButtonMessages' 'yellowButtonMenu' 'yellowButtonMessages' 'blueButtonMenu' 'blueButtonMessages')
CanvasController ('scrollBar' 'marker' 'horScrollBar' 'horMarker' 'oldPoint' 'corner')
NetController ()
BrowserController ()
ContextDisplayController ()
ContextEditorController ()
InstructorController ()
NetViewPluggableController ('lastLoc')
Object ()
SemanticNet ('nodes' 'links' 'label' 'student' 'studentRecord' 'levels')
NRLContext ('contextnodes' 'contextlinks' 'name' 'associatedNode' 'locations' 'textLocations')
Object ()
Links ('input' 'output' 'name' 'sensor')
AttributeLinks ()
HierarchyLinks ()
RelationalLinks ()
Object ()
NetNodes ('nodeContext')
ConceptNode ()
LessonNode ('questions' 'currentQuestion' 'currentChoice' 'tmpVal')
AttributeName ()
InstanceNode ()

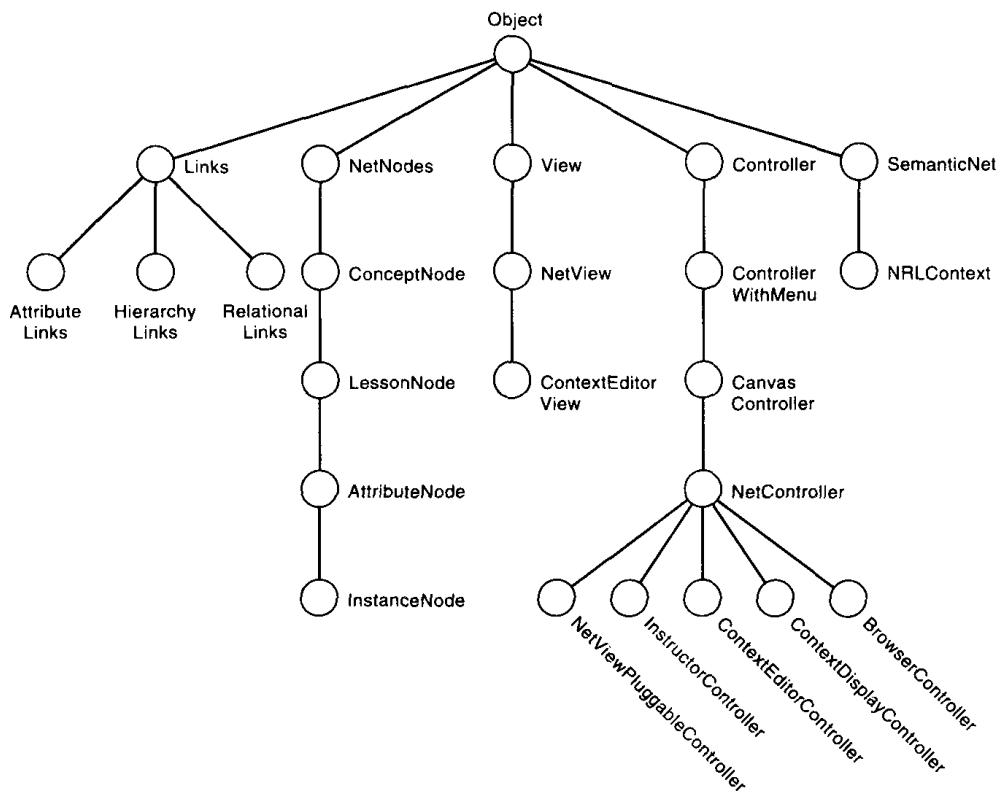


Figure 11.5 The Class Hierarchy for NRL.

various types of representation schemes, several of which will be subsequently described. NRL is available in the public domain, as described in the Appendix.

Figure 11.4 contains several application design features of interest. The left view contains the network represented on a **CanvasView** which is scrollable in two directions (see Chapter 8 and the Appendix). In the bottom right of this view is a small “greeked” view that is the same view “shrunkBy” the amount needed to fit in the window. In the remainder of the view, information about the network is shown that is related to each node in the network. In this example, the application is using the semantic net as a tool for teaching. More information about this application is given in several places (Bourne et al., 1989; Antao et al., 1990) and is discussed in more detail in the next section.

Hypertext/Hypermedia

Closely allied to semantic net representations are concepts in hypertext and hypermedia. These two relatively new terms are defined as follows.

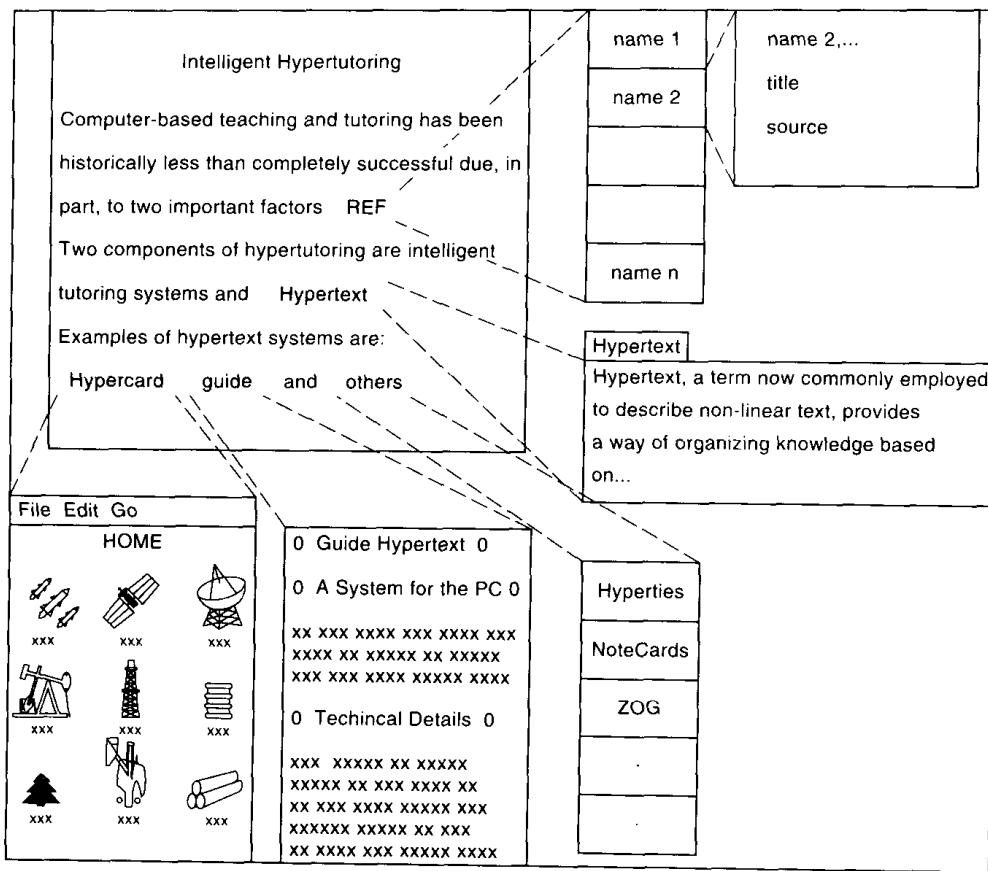


Figure 11.6 Traditional Hypertext. (With permission; Academic Computing.)

Hypertext is nonlinear text and hypermedia is nonlinear media. These concepts can be explained by examining Figure 11.6 which displays a set of text in a window in the upper left of the figure. Highlighting words or phrases in the text, as shown, can pop up other windows that contain more information about the highlighted items. When the items that appear are text, the knowledge organization is hypertext, and when presentations other than text are shown (e.g., such as video, animations, sound, etc.), then the presentation combination is called hypermedia, because different types of media are used. Information can be organized in a hierarchical graph as well, as shown in Figure 11.7. Arraying information in this fashion is called graph-based hypermedia (Bourne et al., 1989). The methodologies used for creating semantic nets described in the previous section work well for creating graph-based hypertext/hypermedia systems.

The importance of including hypertext/hypermedia as a representation in building applications is that this knowledge organization can be (1) used

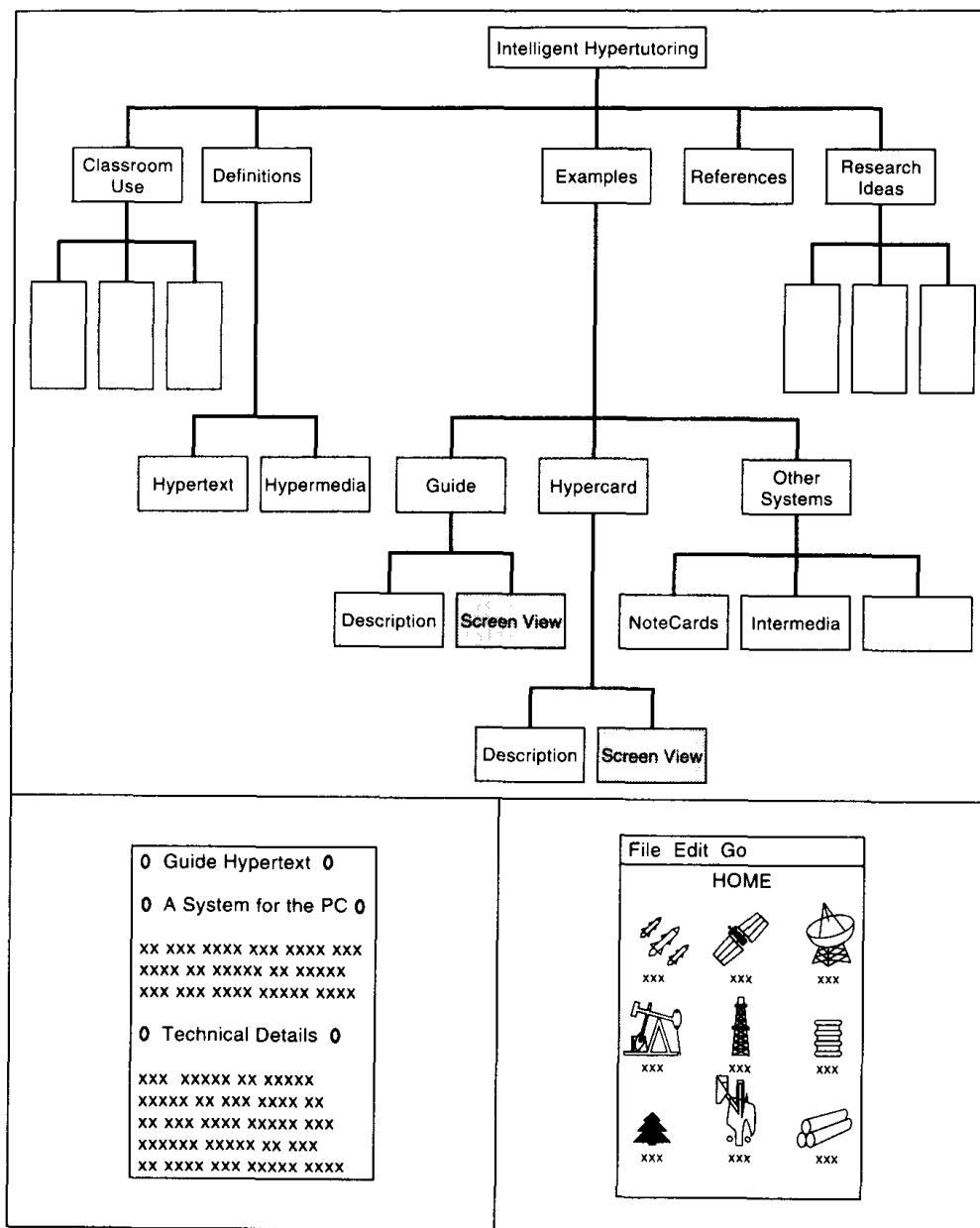


Figure 11.7 Graph-Based Hypertext. (With permission; Academic Computing.)

by the program and (2) accessed and displayed to the end user. Knowledge captured and organized in this graphic fashion permits browsable representations of knowledge which facilitate direct modification and observation of the contents of nodes. Hence, in applications that are built for engineering analysis and design, as well as other systems, hypertext graphs can be included for presenting and organizing knowledge about the application. The utilization of the knowledge captured in this fashion could facilitate various tasks, such as (1) teaching how to use an application, (2) defining terms, or (3) exploring knowledge in a domain.

Knowledge Chunking and Browsing

Another concept associated with knowledge organization is concerned with capturing, representing, and browsing knowledge. To capture knowledge, one can handcraft domain-specific graphs of knowledge using semantic nets or hypertext, as indicated previously. A related, yet more general approach, is to build small “chunks” of knowledge that are accessible via a browser, much like the class browser of Smalltalk. Consider, for example, the knowledge browser shown in Figure 11.8 which displays a browser tailored for engineering knowledge (van der Molen, 1989). Arrayed across the top of the view are views listing

Area	Subject	Module	Items
Bio-Medical Engineering	DC Circuit Analysis	Electric & Magnetic fields	Part D
Chemical Engineering	Electromagnetic Theory	Historical introduction	
Civil Engineering	Electronic Engineering	Introduction	
Computer Science	Intro Digital Circuits	Maxwell's equations	
Electrical Engineering	Microprocessors	Wave propagation	
Mathematics	Network Theory I		
Mechanical Engineering	Power Electronics		

Laws Prereqs
 Texts EOI
 Examples Lessons
 Analogies Assocs
 Graphs Exams
 Images
 Simulation
 Exercises

Figure 11.8 A Knowledge Browser.

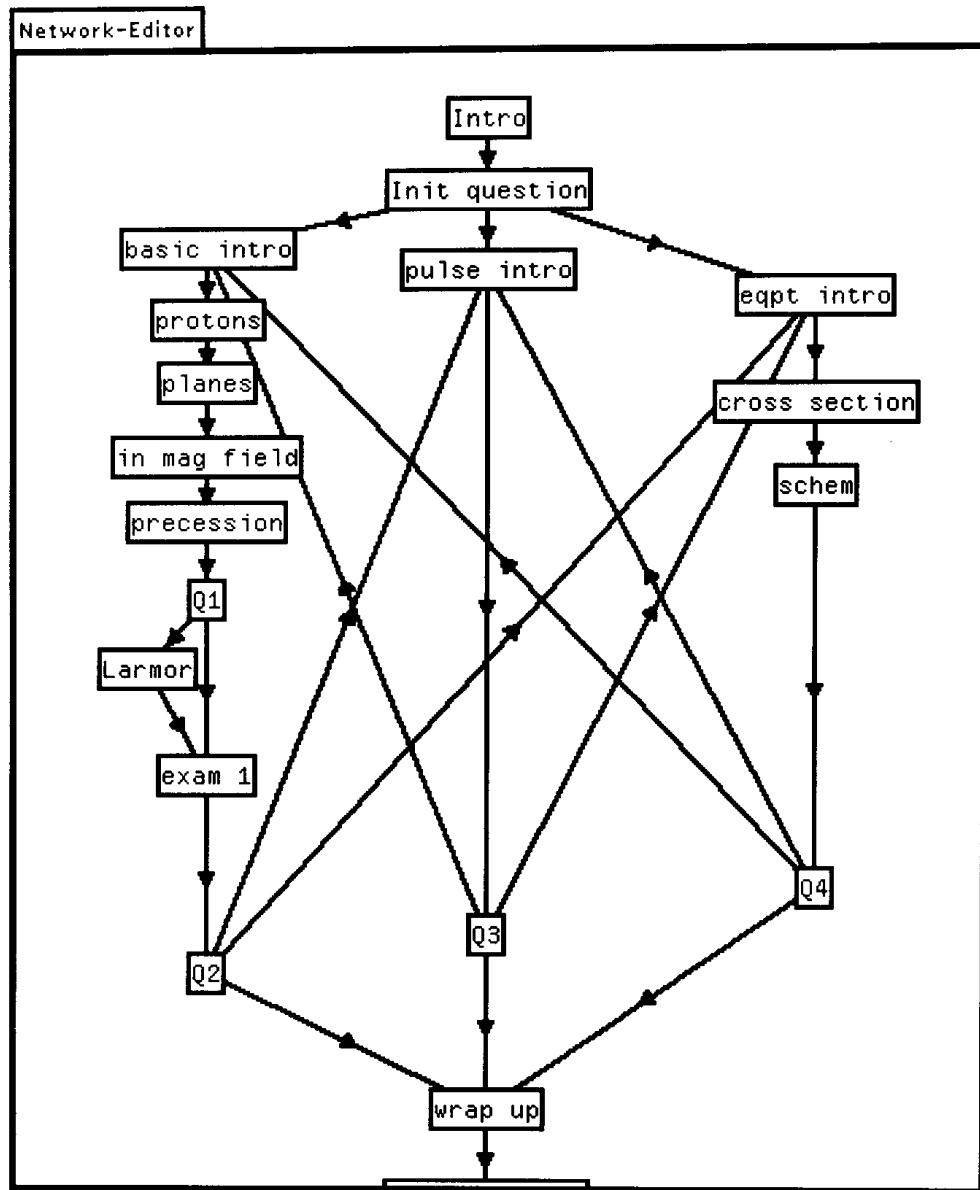


Figure 11.9 A Network of Knowledge Chunks for Presenting Tutorial Information.

(1) the general area, (2) the subject area within the general area, (3) modules in the subject area, and (4) items for the module. Different perspectives for the items are selected by pressing the buttons shown at the bottom left of the figure. The perspectives or views of the knowledge listed are, for the most part, familiar. For example, laws, texts, examples, graphics, and so forth are listed.² The utility of this type of knowledge browsing system is for constructing lessons in computer-based training systems. This browser is directly modeled on the class browser of ST-80.

Figure 11.9 shows another part of a tutoring/training system consisting of a network of linked knowledge chunks that display the presentation sequence of lessons to be given to a student. This representation is a semantic net that was built using the NRL methodology discussed earlier in this chapter. In this example, the presentation sequence proceeds from the top to the bottom of the graph, presenting lessons in the form of knowledge chunks with branch points along the way to change the flow of control depending on the way that the student answers questions posed by the system. For example, introductory information is first presented (“intro”), and after an introductory question, either basic, pulse, or equipment introductory knowledge chunks are presented based on the selection of the student.

11.4 Rules, Procedures, and Algorithms

Rules, procedures, and algorithms all represent sequences of “how to” knowledge, that is, sequences of things that are to be done. An algorithm captures a methodology for solving a problem in a finite number of steps. The implementation of an algorithm is made by concatenating sequences of executable computer statements known as procedures. Rules are also procedural because sequences of rules capture a line of reasoning. Reasoning using rules proceeds either from data toward a goal or attempts to prove goal statements by reasoning from the goal, by proving subgoals and ultimately reaching the data.

Forward-chaining rule systems reason from data toward a goal and backward-chaining rule systems reason from the goal to data. The common architecture for rule systems is to have some type of “working memory” or temporary storage area in which data is stored. Forward-chained rule systems match the items in their if parts (known as antecedents) against working memory and if an item is there, they fire the rule which may produce a message or deposit a new item in working memory. The inference engine (i.e., the program machinery that matches the elements in working memory with the

²EOI = Events of Instruction; Assoc = Associations.

rules) continues to test each rule until no more rules can fire and leaves the results in working memory. In contrast, backward chaining starts with a number of goals and runs all the rules that can directly or indirectly prove the hypothesis. A hypothesis is a tentative goal solution, of which there may be several that would prove a goal. In backward chaining, there may well be a long chain of rules representing many subgoals that must be found before a hypothesis is proven.

Rule systems have been deeply examined and explained in a wide variety of papers and textbooks (Giarratano and Riley, 1989; Brownston et al., 1985). Both forward- and backward-chaining inference methods are widely used; which paradigm to use depends on the nature of the problem to be solved. Forward chaining is particularly useful for making inferences from data and backward chaining is normally employed for securing conclusions starting with several different hypotheses that should be proved. Depending on the characteristics of the problem, either or both of the methods might be used. Commercial tools for inference are normally based on one technique or the other, yet some incorporate both techniques (e.g., ART, the Automated Reasoning Tool). The interested reader is referred to the two texts listed previously for detailed information on rule systems.

The importance of this discussion on rule systems is that rules permit making inferences, a task that is difficult to do in pure object-oriented systems. Hence, if inference is needed in an application built using object-based methods, it may be necessary to build a simple rule system or to interface the object-oriented system to an existing rule system. Both methods will be illustrated in the remainder of this section.

To illustrate the simplicity of a forward-chained rule system, Table 11.4 demonstrates how to construct this type of inference system in ST-80. All

Table 11.4 A Forward-Chaining Rule System for ST-80

Object subclass: #RBS
InstanceVariableNames: 'rules facts'
classVariableNames:"
poolDictionaries:"
category: 'Trivial RBS'

This is a simple example that implements a forward chaining rule system. All methods in RBS inference are accessed by sending messages to 'self'

RBS methodsFor: Inference

forwardchain

```
rules do: [:rule| (self tryrule: rule)
           ifTrue: [↑ true]].
          ↑ false
```

Table 11.4 *Continued***Infer**

```
self forwardchain ifFalse: [↑ nil]. "continue to
forwardchain until no rules fire"
self infer
```

store: fact

```
(facts includes: fact)
ifTrue: [↑ false]
ifFalse: [facts add: fact. ↑ true]
```

testIfPartOfRule: rule

```
lifs progress|
its := rule first. "get the antecedents"
its do: [:if|facts includes: if] "do the facts include the antecedents?"
ifTrue: [progress := true]
ifFalse: [↑ false].
↑ progress
```

tryrule: rule

```
(self testIfPartOfRule: rule)
ifTrue: [(self use ThenPartOfRule: rule)
ifTrue: [↑ true]
ifFalse: [↑ false]]
ifFalse: [↑ false]
```

useThenPartOfRule: rule

```
[thens|
thens := rule last.
thens do: [:then|(self store: then)
ifTrue:
[Transcript cr; show: (rules keyAtValue: rule)
asString, 'deduces', then; cr. ↑ true]
ifFalse: [↑ false]]]
```

RBS methodsFor: initialize**example**

```
facts add: 'a'; add: 'b'.
rules at: #rule1 put: #('a' 'b') ('c').
rules at: #rule2 put: #('c') ('d').
rules at: #rule3 put: #('d') ('e').
rules at: #rule4 put: #('e') ('f').
rules at: #rule5 put: #('f') ('g').
rules at: #rule6 put: #('g') ('h').
```

```
"Temp := RBS new.
Temp init; example; infer.
Temp inspect"
```

Init

```
rules ← Dictionary new.
facts ← OrderedCollection new.
```

```
(defrule gain
  (voltageIn ?inputV)
  (voltageOut ?outputV)
  =>
  (assert(gain = (/?outputV ?inputV)))))

  (assert (voltageIn 1))
  (assert (voltageOut 5))
  (run)
result: →(gain 5)
```

Figure 11.10 Example of a Rule from CLIPS and the Result of Running the Inference Engine.

methods for inference are contained in the protocol of the class **RBS** named “inference” and consist of six simple methods that are easily understood. In the class **RBS** “initialize” protocol, an example is given that illustrates the use of this class. The reader is encouraged to use this code and create a rule system in ST-80. The example first creates an instance of **RBS**, initializes the system, and makes six simple rules and two facts. Finally, the message *infer* is sent to the system. The method *infer* runs itself until no more rules fire. The *useThenPartOfRule:* contains a message to the transcript which prints out a message for every rule firing that yields a way for the user to observe the functioning of this simple system.

There are many types of rule systems available for both forward- and backward-chaining inference procedures. Among the popular forward-chaining rule systems are CLIPS (Giarratano and Riley, 1989) and OPS5 (Brownston et al., 1985). Figure 11.10 displays the syntax and use of a typical forward-chaining rule system. This example is from CLIPS, and OPS5 differs in only minor syntactic details (e.g., the symbol “ \Rightarrow ” in CLIPS is “ \rightarrow ” in OPS5). Shown at the top of the figure is the definition of a rule named “gain.” The syntax of the rule is that rule antecedents (the if part) precede the “ \Rightarrow ” and consequents (the then part) follow. In the antecedent, two slots named “voltageIn” and “voltageOut” are created with variables in the two slots that are bound to any data with variables that have the same slot names when the rule runs. The consequent contains what is to be concluded; in this case, the value of the gain computed between the input and output is added to working memory. Following the definition of the rule, the way that the rule is run is shown. First, two working memory elements are asserted and the inference engine is instructed to run. The result is that the elements in the if part of the rule match the elements in working memory and the rule is fired (i.e., the consequent is executed) and the result is posted in working memory. Note the mechanism of binding variables “inputV” and “outputV” to the values “1” and “5” respectively and the use of these values in the consequent to compute the gain

(GAIN = output voltage/input voltage). The prepended "?" to "inputV" and "outputV" indicates that "inputV" and "outputV" are variables.

Although there are several variations on the rule running theme, the basic data-driven rule execution method is to match all rules in an inference system against working memory (the data) until no rule fires as all the if parts of the rules are compared to working memory. Quite complex inference schemes can be constructed in this manner; however, once the number of rules become large, it is quite difficult to understand the functioning of the body of rules and, in consequence, it is difficult to maintain and modify.

Various large commercial inference systems include both forward and backward chaining such as KEE (Kehler and Clemenson, 1984) and ART. Such systems contain various rule acquisition and building tools, as well as visualization aids. Furthermore, there are at least two inference systems written in Smalltalk, for example, HUMBLE (Piersol, 1987), a backward-chaining system manufactured by Xerox Special Information Systems, and SMART, created by Knowledge Systems Corporation.

How do rules relate to graphs? The answer is: very closely. Indeed, some rule systems such as CLIPS and OPS5 convert the rules created into a graph to permit more rapid execution. Consider, for example, Figure 11.11 which gives an idea of how a set of rules might be converted to a simple graph. Two rules are shown in this example and three facts. In a forward-chaining rule system, adding facts to working memory and then running the rules would result in "c" being concluded first and then "e" being concluded. Note that "c" had to be concluded before the second rule would run. Adding data to the input nodes

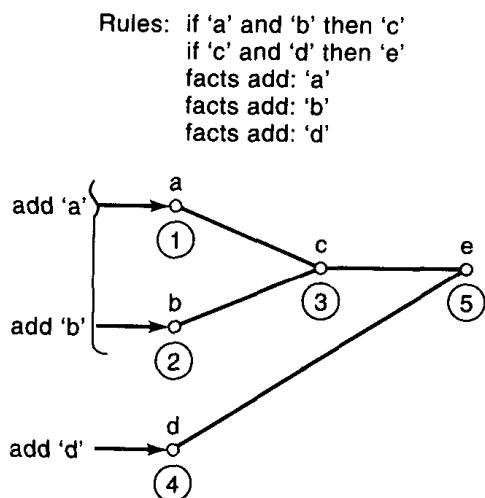


Figure 11.11 Mapping Rules to a Graph.

of the graph produces the same result; as shown, asserting “a” and “b” results in concluding “c” and asserting “d” results in “e” being concluded. Rather than constructing code to interpret rules, it is more efficient to create a graph that will produce the same results. An order of magnitude or better improvement in speed is often observed by converting rules to graphs.

The format of rules and the representation capabilities of object-oriented systems are not well matched. Rules are procedural and require an imperative style of programming (i.e., a series of code statements) to implement forward chaining or backward chaining—not the style that has been discussed throughout this text for object-oriented programming. Nevertheless, representation of knowledge in rules is frequently desirable in building applications in which heuristic decisions need to be made. The alternatives available to the system builder are: (1) write the needed code procedurally in the object-oriented language, (2) write an inference engine, or (3) couple the object-oriented program to an inference engine. The first option should be taken if the number of rules is small and if the rules are changed infrequently. The second option can be selected if there are only a few rules but changes are often required. The third option should be chosen if there are many rules. These options are particularly relevant to Smalltalk-80. Because ST-80 runs more slowly than common procedural languages such as C, coupling to an efficient graph-based inference engine written in that language will result in a significant speed up. The next chapter will discuss how to build such an interface and will demonstrate the resulting class created for coupling CLIPS to ST-80.

11.5 Acquiring Knowledge

Once representation methodologies have been selected for building an application, knowledge must be acquired in the specific domain of interest. There are various options available for acquiring knowledge that include: (1) entering the knowledge by hand, (2) using a knowledge acquisition tool to organize knowledge, and (3) permitting the system to discover knowledge as it works. By far the most common method is the first. The second option is utilized in a number of experimental knowledge acquisition systems, and the third method remains as basic research. In the first and most frequently used case, the user is required, for example, to create rules, frames, or objects. The methodology for object creation has been discussed in Chapter 3 in terms of analyzing a problem to capture object specifications. If rules are the representation used, then rules are written which can be data driven or goal driven. In the first case, that is, forward chaining, conclusions are made by writing rules that examine the data first and work toward a conclusion. For goal-driven rule systems or backward chaining, the possible conclusions that can be made

(hypotheses) are listed and an attempt is made to collect rules that will prove the hypotheses. Either one or the other of these two modes of thinking about solving a problem using rules are normally used in the knowledge acquisition process. If frames are used in a representation, slots in the frames must be filled with information much like that in the computer frame example shown earlier.

For handcrafting knowledge entry into specific representations, the dominant method has been to debrief an expert and then directly enter the secured knowledge into the representation used. Individuals performing the latter task have come to be known as "knowledge engineers" (Hayes-Roth, Waterman, and Lenat, 1983). More advanced methods employ tools for assisting the user. For example, the KEE system (Kehler and Clemenson, 1984) uses graphical tools for frame building. Beyond methods that simply provide visualization and organization support are tools that actually assist in classification of the knowledge [see, e.g., Kitto and Boose (1988)] according to a taxometric scheme. Some knowledge acquisition systems can be tuned to assist users in acquiring and categorizing knowledge into areas such as anticipatory, circumstantial, refining, and qualifying knowledge (Eshelman et al., 1988) that satisfy the requirements of a particular domain.

Learning or discovery systems are the least developed and possibly the most interesting category for acquiring knowledge. During recent years, systems that learn by examining cases have become popular (Hammond, 1989) as have neural nets (Rummelhart and McClelland, 1986) which learn from collecting many observations. These techniques are most useful for data that can be separated into distinct categories.

11.6 Representational Relationships

Although a number of different representation methodologies were mentioned in the outset of this chapter, three primary representation techniques emerged that are suitable for building knowledge-intensive applications; *rules*, *frames*, and *objects*. Rules are best for representing knowledge about inference and frames are superior for capturing static knowledge. Rules can be represented in frames and frames can be used to conduct inference by using procedures attached to slots. Furthermore, knowledge represented in frames can be represented in an identical manner in objects. Rules can be represented either by coupling to an external rule system or by recoding the knowledge into procedures. Moreover, rule systems can be written using objects. The point is that one can "mix and match" the various styles of representation—not quite freely, but any goal that one has can probably be achieved with any of the representations. Object-oriented representations tend to be the easiest to maintain and understand due to the modularity of declarative knowledge. Rules,

however, are best understood in a rule format, that is, a listing of near-English language rules that capture a reasoning process. Objects, however, can represent reasoning by making rules into the form of a constraint graph which also permits easy understanding by observation of graph connections.

The preceding explanation of the differences between representations leads to the following conclusions. Most representation mechanisms can be used for problem solving. Some are better than others for some tasks. The viewpoint taken in this text is that the optimal system architecture will permit the use of different representations but will not force the use of a methodology. In line with the general theme of the text, the conclusion is made that object-oriented methodologies will satisfy most application needs, with the exception of inference. For cases in which fast and robust inference engines are needed, it is perhaps best to couple object to rule paradigms.

11.7 Identifying the Knowledge Organization Needed for Problem Solving

Problem solving consists of several facets, including problem analysis, creation of plans to pursue, and understanding ways of dealing with dead ends (i.e., unreachable solutions) and incomplete knowledge and data. To produce applications that have the components of problem solving built into the software, it is essential that a representation scheme be provided that can capture the knowledge needed to solve problems. As discussed previously, there are a number of prominent ways of representing knowledge for this purpose. Yet, there are few clear guidelines for making proper paradigm selections. Some alternatives are clear. For example, it is clear that rules should be used to capture reasoning. However, it is not as clear what kind of rule system should be used and why. Furthermore, there are alternatives to rules, such as coding the information in procedures or creating graphs that have the same effect as rules. Another clear alternative is representing declarative knowledge, such as the information in text books or manuals. Hypertext/hypermedia methods, presented in some form of a semantic net or frame, are quite useful for this style of representation. Knowledge captured in nets or frames can serve as useful background knowledge for problem solving. Both frames and nets are directly represented with objects. Object-oriented methods hold the edge for creating simulations, especially in situations in which a simulated system can be directly mapped into objects with behaviors that mimic the behavior of the objects in the real world.

In summary, there are many useful knowledge representation methodologies available, including the primary methodologies of rules, frames, and objects. Research on knowledge representation is continually evolving. The

methods that we have discussed will work for producing many applications; yet, it can be anticipated that new advances will, at some point, replace or augment the techniques discussed in this chapter. As shown, object-oriented methodologies can be used for implementing most, if not all, representations required to solve problems.

References

- Antao, B. A. A., A. J. Brodersen, J. R. Bourne, and J. R. Cantwell (1991). Building Intelligent Tutorial Systems for Teaching Simulation in Engineering Education. *IEEE Transactions on Engineering Education*.
- ART, The Automated Reasoning Tool. Inference Corporation. Los Angeles.
- Barr, A., and E. Feigenbaum (1981). *The Handbook of Artificial Intelligence*, Vol. 1, William Kaufmann, Los Altos, CA.
- Bobrow, D. G. (Ed.) (1985). *Qualitative Reasoning about Physical Systems*. MIT Press, Cambridge, MA.
- Bourne, John R., Jeff Cantwell, Arthur J. Brodersen, Brian Antao, Antonis Koussis, and Yen-Chun Huang (1989). Intelligent Hypertutoring in Engineering. *Academic Computing*, September, 18–48.
- Brachman, R. J., and H. J. Levesque (Eds.) (1985). *Readings in Knowledge Representation*. Morgan Kaufmann, Los Altos, CA.
- Brownston, L., R. Farrell, E. Kant, and N. Martin (1985). *Programming Expert Systems in OPS5*. Addison-Wesley, Reading, MA.
- Clocksin, W. F., and C. S. Mellish (1984). *Programming in Prolog*. Springer-Verlag, New York.
- Eshelman, L., D. Ehret, J. McDermott, and M. Tan (1988). MOLE: A Tenacious Knowledge Acquisition Tool. In *Knowledge Acquisition Tools for Expert Systems*, J. Boose and B. Gaines (Eds.), pages 95–108. Academic, New York.
- Genter, D., and A. Stevens (1983). *Mental Models*. Lawrence Erlbaum Associates, Hillsdale, NJ.
- Giarratano, J., and G. Riley (1989). *Expert Systems: Principles and Programming*. PWS-KENT, Boston, MA.
- Hammond, K. (Ed.) (1989). *Proceedings: Case-Based Reasoning Workshop*. Morgan Kaufmann, San Mateo, CA.
- Hayes-Roth, Frederick, Donald A. Waterman, and Douglas B. Lenat (1983). *Building Expert Systems*. Addison-Wesley, Reading, MA.
- Kehler, T. P., and G. D. Clemenson (1984). An Application Development System for Expert Systems. *System Software*, Vol. 3, No. 1, January, 212–224.

- Kitto, C. M., and J. H. Boose (1988). Heuristics for Expertise Transfer: An Implementation of a Dialog Manager for Knowledge Acquisition. In *Knowledge Acquisition Tools for Expert Systems*, J. Boose and B. Gaines (Eds.), pages 175–194. Academic, San Diego, CA.
- Lenat, D., M. Prakas, and M. Shephard, (1986). CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks. *AI Magazine*, Vol. 6, No. 4, Winter, 65–85.
- Michalski, R. S., Jaime G. Carbonell, and Tom M. Mitchell (1983). *Machine Learning*. Tioga Publishing, Palo Alto, CA.
- Morris, William (Ed.) (1981). *The American Heritage Dictionary of the English Language*. Houghton Mifflin. Boston, MA.
- Newell, A. and H. A. Simon, (1972). *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ.
- Piersol, Kurt W. (1987). *HUMBLE V2.0 Reference Manual*. Xerox Special Information Systems, Xerox Corporation, Pasadena, CA.
- Quillian, M. R. (1968). Semantic Memory. In *Semantic Information Processing*, M. Minsky (Ed.), pages 227–270. MIT Press, Cambridge, MA.
- Royce, W. W. (1970). Managing the Development of Large Software Systems: Concepts and Techniques. *Proceedings of WestCon*, August.
- Rummelhart, D. E., and J. L. McClelland (1986). *Parallel Distributed Processing*. MIT Press, Cambridge, MA.
- Schach, S. (1990). *Software Engineering*. Aksen Associates. Homewood, IL.
- Schank, R. C. and R. P. Abelson, (1977). *Scripts, Plans, Goals, and Understanding*, Lawrence Erlbaum, Hilldale, NJ.
- van der Molen, H. (1989). Knowledge Acquisition in Intelligent Tutoring Systems. Class Project, EE 355, Vanderbilt University, Nashville.

Exercises

- 11.1 Explain the difference between declarative and procedural knowledge. Give an example of each.
- 11.2 What are three ways to represent knowledge? Briefly summarize the advantages and disadvantages of each technique.
- 11.3 Frames, semantic nets, and objects are very similar in concept. Describe the similarities and differences among these different representation methodologies.
- 11.4 Explain the differences between the terms *syntactic* and *semantic*.

- 11.5 Is hypertext more like a system for programming or more like a system for indexing information?
- 11.6 As engineering knowledge grows, the corpus of the body of this knowledge cannot be understood by a single individual. How would you design systems that could assist individuals and teams working together to understand and deal with the complexity of engineering knowledge in the 21st century?
- 11.7 Rule systems are useful for making inferences. When would it be useful to include rules in an application? Consider, for example, the use of rules for capturing scarce expertise or for monitoring the user and modifying what the user sees on the screen.
- 11.8 Write a simple script in the form shown in Table 11.2. Show how you can use a script to control the flow of actions in an application.
- 11.9 Do engineers really use rules of thumb in problem solving? Or are models, simulations, and algorithms always used? What do you think?

Integrating Object-Based Environments with the Real World

12.1 Concepts

Prior to this chapter, computer-based application systems have been treated as essentially divorced from the physical world. In previous chapters, the only connection to the outside world discussed has been the graphics and direct-manipulation interfaces to users that were examined intensively in Chapter 7. Yet, for creating applications in engineering other types of interfaces are frequently required. For example, many engineering tasks need the ability to acquire data, communicate electronically with others, and control a host of different types of instruments. These needs can be met in an object-oriented system such as Smalltalk-80 by enhancement of the basic environment. This chapter will explore the capabilities already present in the ST-80 environment for interacting with the world and discuss how to add additional features. Several examples will be presented to indicate how an enhanced environment could be employed to solve various engineering tasks. It is worth noting that the concepts discussed are general in nature and not completely specific to the ST-80 environment. Nevertheless, ST-80 examples are given, in part because these examples can be implemented using the techniques described throughout the text but also because of the clarity in code presentation that Smalltalk and the object-oriented paradigm affords.

Figure 12.1 displays a block diagram of a computer system that contains many types of computer I/O (input/output) cards attached to their related peripherals. The peripheral devices shown cover the common types of devices manufactured to connect computer systems to the outside world. There are other devices that can be added, such as image scanners. It is unlikely that any computer would ever be configured in the way shown; however, it is useful

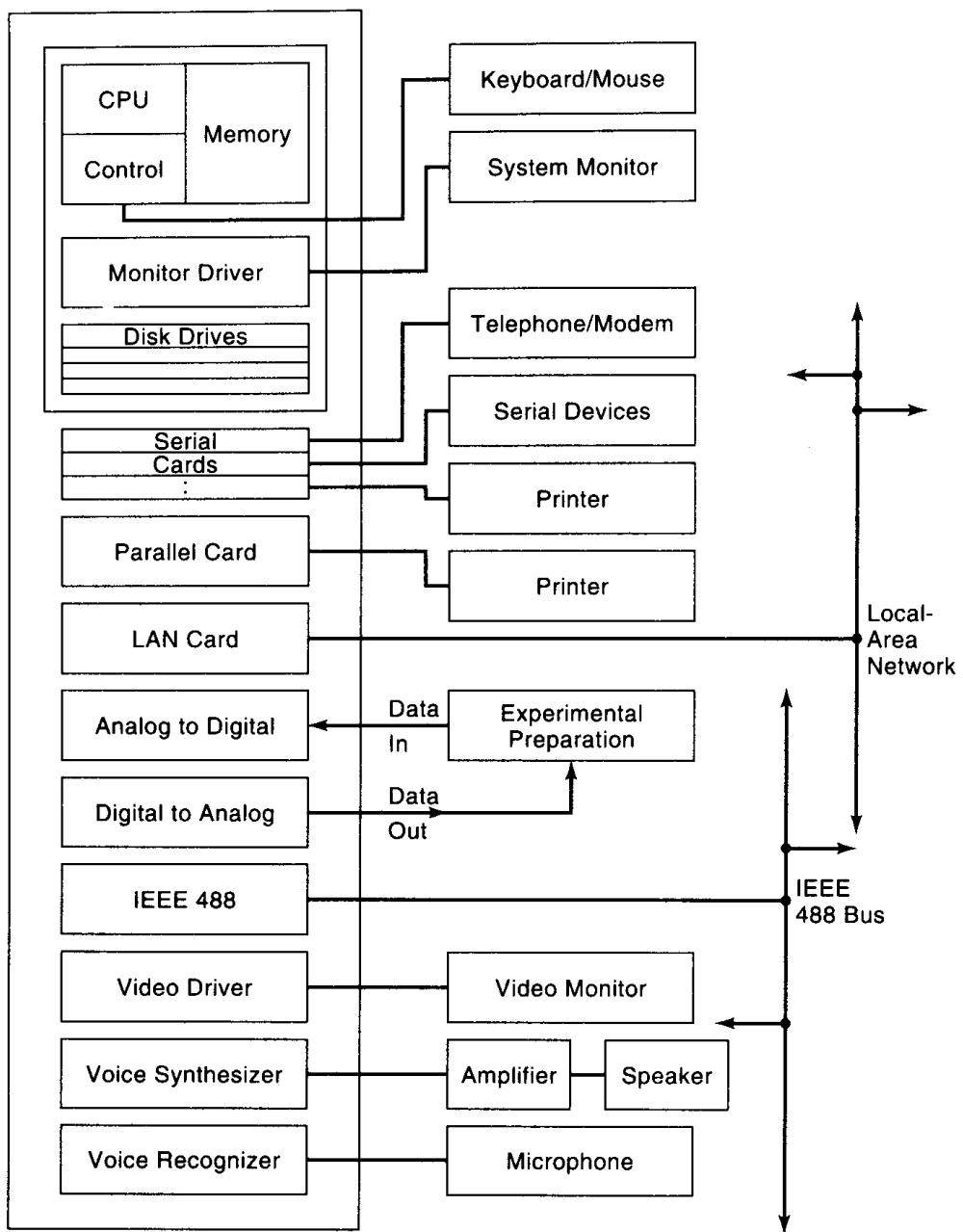


Figure 12.1 Computer Peripherals and Their Connection to the External Environment.

to indicate that a wide range of functionality can be achieved by adding different types of cards. Serial cards permit communication with external devices, such as modems for indirect access to the telephone network. Serial cards can also be used to drive printers or any other serial devices, such as serial printers or a serial mouse. Parallel cards are useful for driving multiple input and output lines for controlling printers and other parallel devices. Note that typical printers are either serial or parallel and sometimes even both types of connections are available. LAN (local-area network) cards allow access to networks that use coaxial cable or fiber-optic cable for rapid transmission of information. Analog to digital (A to D, or A/D) and digital to analog (D/A) cards convert externally acquired analog signals to digital signals, and vice versa. IEEE-488 cards implement a control and communications protocol that is useful for accessing and manipulating instrumentation. Video drivers permit control of video sources, including digitizing video information. Finally, voice input and output capabilities permit implementation of a verbal interface with a computing system.

The most popular types of interfaces to the external world will be examined next, followed by a description of how to interface external devices and non-Smalltalk procedures to ST-80. Examples of the use of various interface types will be presented in the remainder of the chapter.

12.2 The External Environment

Table 12.1 lists four major areas that can enhance computer I/O capabilities by connecting computer peripherals to the external environment. *Communications* is the best known category, because most computers interface to other computer systems either via a modem, hardwired cable, or network connection. Intercomputer communications, either via direct serial connections or local-and wide-area networks, is a widely used medium for transmitting information and facilitating human communication through electronic mail. An additional communications medium, the IEEE-488 bus, basically a multiwire cable, is a standard for communication between computers and instrumentation. *Control* is another prominent category in which computer peripherals are directly interfaced to various types of instrumentation or other devices such as printers. The IEEE-488 bus is also widely used for control of instrumentation. *Analog signals* acquired from the world outside a computer system must be converted to digital representation prior to use by a digital computer; the converse is true as well. Analog to digital converters (A/D) and digital to analog converters (D/A) are widely used I/O devices that provide this transformation. Finally, the communications capability of computers can be enhanced by the addition of new modalities, such as video and voice. Video permits the

Table 12.1 External Environment Connections**Communications**

- Serial
- Local-area net
- IEEE 488

Control

- Parallel
- IEEE 488

Acquisition of signals and creation of signals

- Analog to digital
- Digital to analog

Modality enhancement

- Voice input
- Voice output
- Video presentation

viewing of scenes from the world, including animation. Voice communication (both input and output) interfaces add additional capabilities to the typical screen, keyboard, and mouse interfaces.

Serial Interfaces

Figure 12.2 illustrates the way serial connections are made between a serial card in a computer and a serial device. A typical cable between the two devices contains 3 to 10 connections. Below the diagram is shown how wires in a serial cable are allocated to different tasks. The most used connections are pins 2, 3, and 7 of a common 25-wire cable which provide links for transmitting data, receiving data, and ground, respectively. The remaining connections are used for applications that require control of the data transmission (such as stopping the transmission of data due to printer speed) and control of a modem, for example. Signals are sent in series on the receive and transmit wires in the cable. Figure 12.3 illustrates the way serial information is transmitted. High and low bits are sent at a fixed rate (e.g., 9600 baud). After an initial low bit to indicate “start,” eight high and low data bits, which represent a byte of information, are sent. Codes that represent bytes of information belong to a set of codes defined by the American Standard Code for Information Interchange, and are thus referred to as ASCII codes. Individual numbers, characters, and symbols are assigned byte codes. For example, an “A” has an ASCII value of 65 and a “1” has a value of 49.

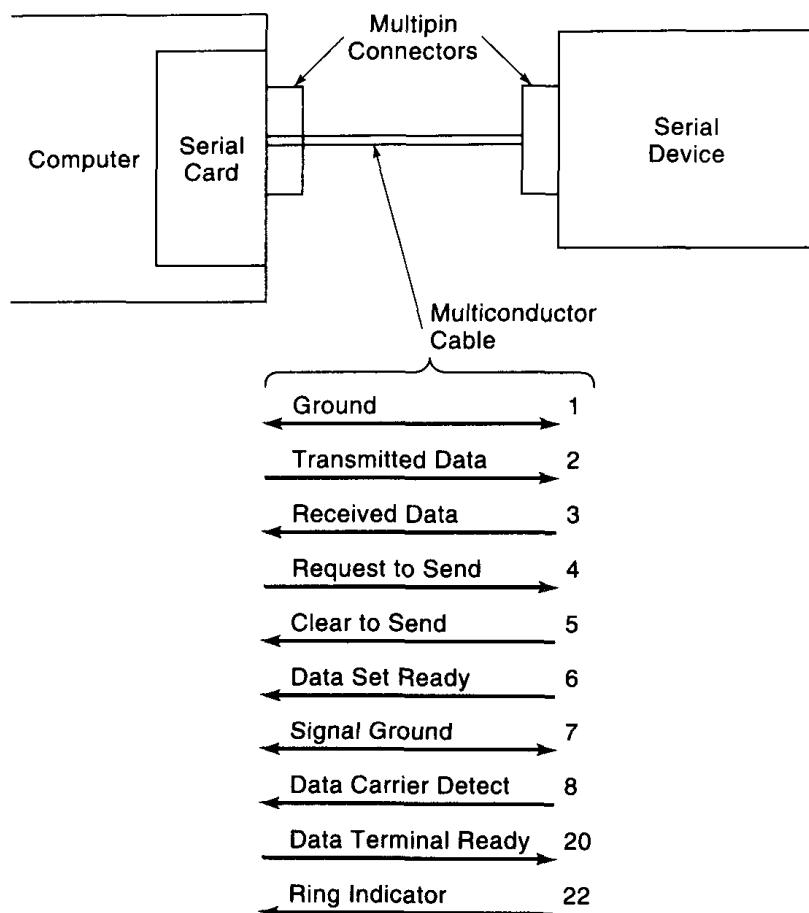


Figure 12.2 Serial Communications. Numbers shown are pins commonly used.

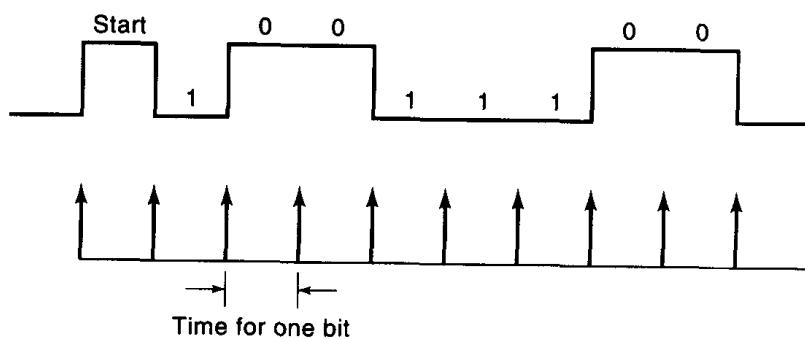


Figure 12.3 Serial Information Transmission.

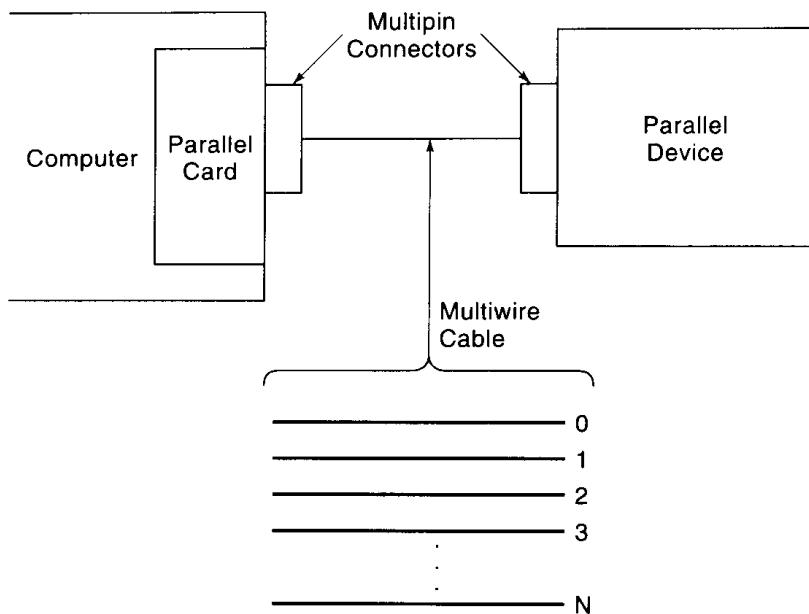


Figure 12.4 Parallel Interface.

Parallel Interfaces

Figure 12.4 shows a multiwire connection between a parallel interface in a computer and a parallel device. The major distinction between this configuration and the serial interface discussed previously is that data is sent *in parallel* across all the wires simultaneously. Each lead is dedicated to a different bit, in contrast to each bit being sent serially across a pair of wires in the serial interface. As a consequence of the parallel nature of this interface, parallel interfaces can normally transmit data at much faster rates than serial interfaces.

LANs and WANs

LAN stands for local-area network and WAN for wide-area network. Figure 12.5 shows a connection of a LAN card from a computer to a network cable. Cables, in order to achieve high speed, are typically coaxial or optical fiber. Local-area nets are commonly used to wire together local clusters of computers, for example, those not separated by more than a mile. Connection to wide-area networks, which permit national and international communications, are typically via bridges (i.e., hardware protocol conversion interfaces) to network cables that utilize leased telephone lines or use fiber-optics cable utilized by various communications companies.

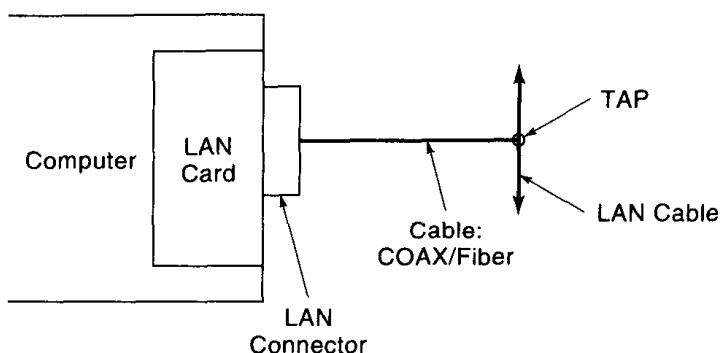


Figure 12.5 ThinLan Connections.

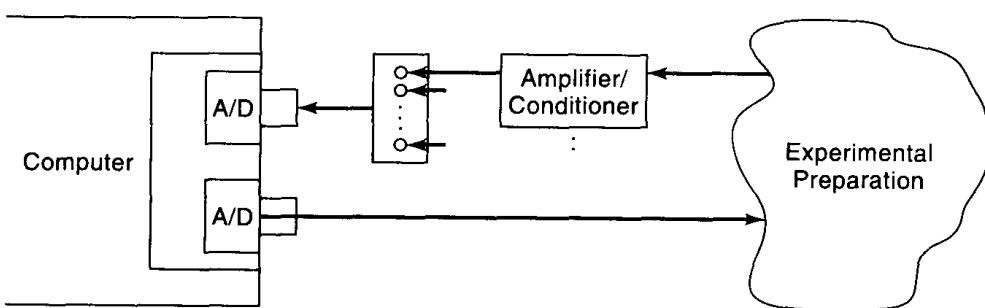


Figure 12.6 Analog to Digital and Digital to Analog Connections.

Analog to Digital and Digital to Analog Conversion

Figure 12.6 depicts a combination card with both A/D and D/A capabilities. Normal connection to the world outside the computer is via a set of leads from the card that connects to a box or connector panel from which connections are made to an external process, for example, a laboratory experiment or process that is to be controlled. Signal amplitudes, if not similar to the input range of the A/D card, normally need to be amplified or attenuated prior to being fed to the A/D converter. Examples of this latter requirement might be the amplification of the voltage measured from a strain gauge or the signals secured from electrodes placed in a muscle.

IEEE-488 Protocol

The IEEE-488 bus is a very popular communications and control protocol standard (Gates, 1989) that has been promoted by the Institute of Electrical and Electronic Engineers. It is used as a standard by hundreds of

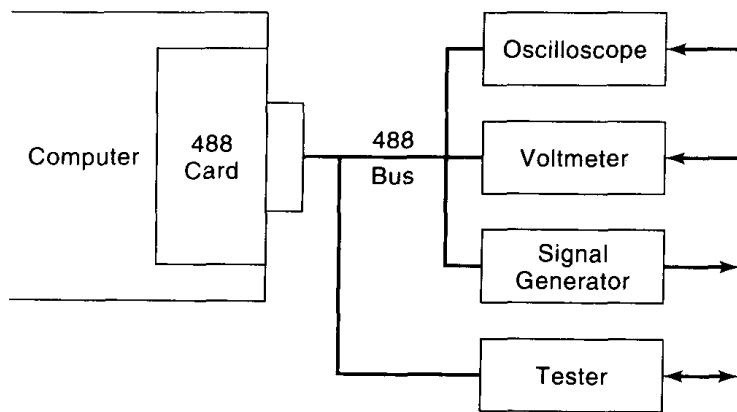


Figure 12.7 IEEE-488 Connection to Instrumentation.

instrument manufacturers which fabricate literally thousands of different instruments. A bus, in this case, is nothing more than a cable consisting of 24 lines, 8 for data, 5 for management, and 3 for handshaking (i.e., requesting and acknowledging) and ground and auxiliary lines. The bus can be connected as a “daisy chain,” star, or in various other topological combinations. One computer serves as a controller for all the instruments. As shown in Figure 12.7, various instruments can be connected directly to an IEEE-488 interface card. Another option is shown in Figure 12.8, in which the serial port of a computer is connected to a hardware serial to 488 bus converter which connects to the various instruments.

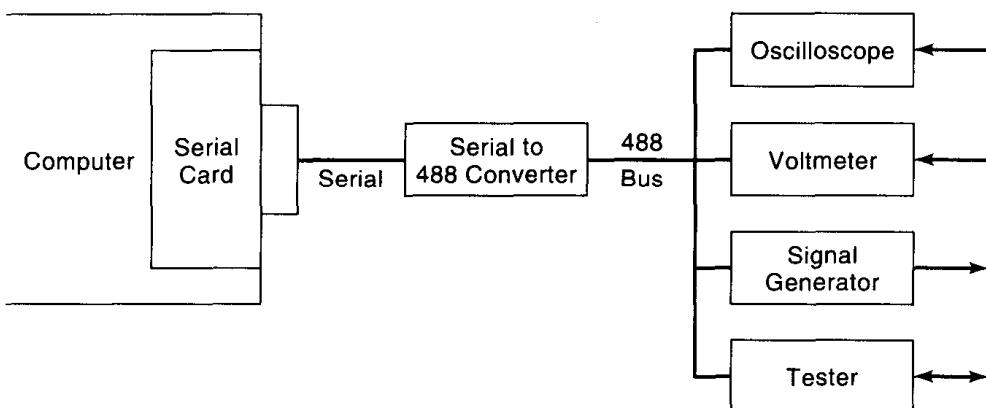


Figure 12.8 Connecting to an IEEE-488 Bus Using a Serial to 488 Converter.

12.3 Interfacing Procedures and External Devices to the ST-80 Environment

Concepts in Interfacing

In previous chapters, the graphics interface and rapid prototyping capabilities of the Smalltalk-80 environment have been discussed in some detail. This robust environment, however, has few “hooks” to the real world, with the exception of the serial interfaces. Furthermore, the ST-80 language has some perceived deficiencies in speed and seamless integration with software written in other languages. To ameliorate these two problems, ST-80 provides a mechanism for integrating C-language routines into the system, such that the routines can be easily used as methods in Smalltalk classes. This capability solves the two problems just mentioned by (1) permitting the writing of drivers for peripheral cards in C and (2) allowing simple and direct access to compiled C programs. Examples of both these capabilities will be given later in this chapter. The C language is one of the most widely used procedural languages and, in addition, produces very efficient code. Hence, combining these capabilities with the strong object-oriented graphical environment of Smalltalk results in a complete system with a wider range of strong capabilities.

External I/O Access Using ST-80

ST-80 provides several classes for interfacing to UNIX, DOS, and Macintosh operating system primitives that permit I/O access. Figure 12.9 displays the class hierarchy for accessing the facilities of supported operating systems. As shown, there are separate classes for DOS, Macintosh, and UNIX errorHolders (i.e., repositories for error handling). The major superclass is **IOAccessor** which serves as a superclass for classes that implement specific I/O for three operating systems: Mac OS, DOS, and UNIX. ST-80 implements subclasses of **IOAccessor** that contain interfaces to Disk and “TTY” (i.e., serial communications¹). As shown in the figure, other subclasses could be implemented that would support other types of I/O devices (e.g., IEEE-488 instruments). Also, subclassing **IOAccessor** permits adding interfaces for other operating systems. Note how the inheritance property of ST-80 simplifies the capturing of I/O interfaces for different operating systems within the same environment.

¹In Objectworks \Smalltalk, Release 4, TTY communications classes are located in \utils\serial.st.

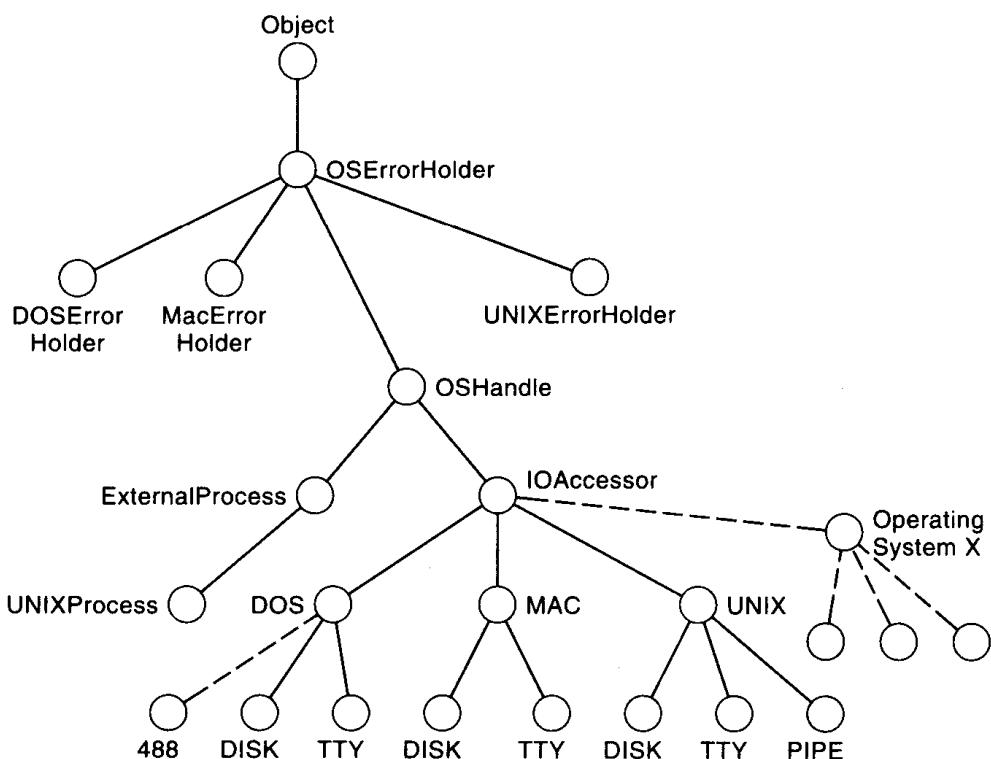


Figure 12.9 Class Hierarchy for Various Operating Systems. (DOS, MAC, and UNIX are abbreviations for **DosIOAccessor**, **MacIOAccessor**, and **UnixIOAccessor**, respectively.)

This feature, in part, is why ST-80 can be moved relatively easily between various hardware platforms.

An Interface to Procedures

The second area of interfacing is the ST-80 interface to procedures written in the C language. C represents a good choice for melding with the ST-80 environment due to its popularity and wide availability. Furthermore, C code can be written that will interface to other languages such as Fortran or Pascal, so that code written in these languages can be interfaced to the ST-80 environment as well.

Table 12.2 presents an example of an ST-80 class named **CLIPS** that interfaces methods in this class to the C functions of the CLIPS inference engine (discussed in Chapter 11). Functions in CLIPS are accessed through primitives in the **CLIPS** class methods. A blackboard of information is kept in a data structure implemented in C. Functions for accessing the blackboard and

Table 12.2 Interfacing to the CLIPS Inference Engine

```
Object subclass: #CLIPS
  InstanceVariableNames: ""
  classVariableNames: ""
  poolDictionaries: ""
  category: 'ITS-CLIPS'

CLIPS class
  InstanceVariableNames: ""

CLIPS class methodsFor: primitives

assert: aString
  <primitive: 10118>

bload: aFileName
  <primitive: 10106>

bsave: aFileName
  <primitive: 10107>

clearClips
  <primitive: 10102>

exciseRule: aString
  <primitive: 10119>

loadRules: aFileName
  <primitive: 10104>

readBB
  <primitive: 10120>

resetClips
  <primitive: 10101>

run: aRunLimit
  <primitive: 10103>

CLIPS class methodsFor: examples

example1

  "CLIPS example1"

  CLIPS clearClips.
  CLIPS loadRules: 'gain.rul'.
  CLIPS assert: 'voltageIn 1'.
  CLIPS assert: 'voltageOut 5'.
  CLIPS run: -1.
  Transcript cr; show: CLIPS readBB.
  Transcript cr; show: CLIPS readBB.

  "example rules in clips"
  "(defrule gain
    (voltagein ?inputV)
    (voltageOut ?OutputV)
    -->
    (assert (gain = (/ ?outputV ?inputV)))
```

Table 12.2 *Continued*

```
(defrule adjustGain
  (gain ?value&: (< ?value 10))
  →
  (bbadd adjustGain))"
```

example2

```
"CLIPS example2"
CLIPS clearClips.
CLIPS bload: 'c:\clips\mab.bin'.
CLIPS resetClips.
CLIPS run: -1.
Transcript cr; show: CLIPS readBB.
Transcript cr; show: CLIPS readBB
```

running the inference engine are accessible in the class **CLIPS** using the syntax `<primitive: xxxx>`, where `xxxx` is a unique numeric value identifying a primitive written in C. Each method calls a different primitive, passing a predetermined number of variables. The mechanism for passing this information and constructing the interface will be described in the following discussion. Readers not interested in these details should skip to Section 12.4.

Examples of the use of the **CLIPS** class interface to the CLIPS inference engine are shown at the end of Table 12.2. In the first example, the working memory of CLIPS is cleared (“clearclips”), rules in an external file are loaded, two strings are loaded into working memory, and the inference engine is run. The “-1” argument to **CLIPS** indicates that the inference engine should run until no more rules fire. The example rules shown in the comment are the contents of the rule file loaded. The values of `voltageIn` and `voltageOut` are bound to the two variables `?inputV` and `?outputV`, respectively, which are subsequently used to compute the value of the gain. The second rule accesses the value posted in working memory by the first rule and, if the gain is too low, posts information on the blackboard accessible to Smalltalk to “adjustGain.” This last string is the information that can be read by Smalltalk by sending the message “readBB” to the class **CLIPS**. The second part of the example shows these steps and prints the information on the system transcript. When the “readBB” message is sent a second time and the results printed, a message that the blackboard is empty is displayed in the transcript. To accomplish these interactions between Smalltalk and C, several specific interfacing procedures are required which are explained next.

The C Interface to ST-80

ST-80 can be interfaced to software written in the C language. Although it is not the intention of this text to show "advanced" concepts or explicit details of code not related to ST-80, the next paragraphs are included to give the reader an idea of how the interface between ST-80 and C can be constructed. The information to be imparted from the example includes (1) how to set up a C routine for interfacing, (2) the characteristics of what can be put in the routine, and (3) how to name and link to ST-80. The code and libraries used will differ for different hardware platforms. Different C-language compilers are required for each different platform. The High-C Compiler (MetaWare, 1989) is used for a DOS platform. The Macintosh (MPW C) and UNIX machines require different C compilers.

The basic steps for creating your own primitive is to create a C routine called UPinstall(). This routine should add your C program by invoking a routine called UPaddPrimitive. For example, a file might contain C code of the following form:

```
#include "userprim.h"                                /* initial definitions */
char *UPinstall()
{
    UPaddPrimitive (10010, cfunc, 2);      /* the number, name of the
                                                C function */
    return "Test primitive";                /* and number of arguments */
}
void cfunc(receiver, arg1, arg2)
upHandle receiver, arg1, arg2;
{
    /* insert any C code here */
}
```

In the preceding code framework, a user primitive number is specified in the UPaddPrimitive function to identify the user primitive code. These numbers are to range between $10,000_{10}$ and 19,999. There are many functions available for the user to employ for passing arguments, converting data between C and Smalltalk, and so on. The reader is referred to the *Objectworks \ Smalltalk User's Guide* (1991) for more information on the user-defined primitive interface. Inside any Smalltalk method, the C code can be directly accessed by referring to the primitive number. For example, one might create a test class such as **PrimitiveTestClass** containing a class method *test:with:* to try out the interface.

```
test: arg1 with: arg2
    "PrimitiveTestClass test: 3 with: 4"
    <primitive: 10010>      "Note the same primitive number as above"
```

Now, whenever the preceding code in the quotes is executed, the method *test: with:* and its arguments will be passed to the C code inside the primitive numbered 10010_{10} .

To complete the installation of the code, the original Smalltalk object file is linked with the object file obtained by compiling the C-coded primitive function. Then, rather than using the original executable file for running Smalltalk (i.e., ST80.exe), a new executable file that includes the primitive code created is utilized (e.g., ST80User.exe).

The preceding description extends somewhat beyond the intended scope of this text and is included so that readers who wish to use these methods can have a better understanding of some of the potentially difficult details. However, for the reader who only wishes an understanding of the methodology, the main point is that C programs can be compiled and linked to Smalltalk. Many C routines can be linked to create a very versatile set of routines in the environment. A cautionary note is necessary, however. Although the technique just described appears quite appealing, the portability of the Smalltalk image is lost by adding various C functions. Hence, it is prudent to determine if any desired functionality can be added without having to resort to using the C-language linkages.

Interfacing with Multiple Applications in Microsoft Windows

Release 4 of Objectworks \ Smalltalk includes classes that can be filed in which provide a simple and direct interface with the Microsoft Windows environment. The fileIn can be found in the standard release in \utils\extfunc.st. Filing in this code will create a category name OS-External-Interface which contains several classes including **MSWindowsInterface**. Contained in this class are a number of examples that can help guide the user in understanding how to interface Smalltalk with other applications in Windows. A simple example illustrates the use of the **MSWindowsInterface** class:

In a workspace, one can type:

```
MSWindowsAPI
define: 'WinExec'
parameters: #( #string #word )
returns: #word.
MSWindowsAPI at: #WinExec
callWith: 'calendar' with 1.
```

highlight the code, and execute. The calendar from Windows will appear on the screen. In the preceding code, MSWindowsAPI is an external name space defined in the interface code, WinExec is a function defined in the Microsoft Windows Software Development Kit (SDK; Microsoft, 1990), parameters define the types of the input needed for “WinExec,” and returns define the type of the output required for this function. Actually activating the calendar is accomplished by passing a string naming the executable file (i.e., calendar.exe) and an argument denoting how the new window is to be shown on the screen (e.g., normal, iconified, etc.). One can use this interface to start any Windows application; furthermore, links between running applications and Smalltalk can be created. The reader is encouraged to experiment with the examples given in the system. One problem is that the SDK documentation will be required in order to be able to determine the available functions and their parameters.

12.4 Applications Using the External Environment

Introduction

The preceding descriptions have indicated that it is possible to tailor the ST-80 environment for communication with the environment outside the computing system. This section examines a number of examples that describe when and how one might want to create applications that access the environment.

One of the simpler concepts that emerges from understanding the capabilities of the ST-80 environment is the idea of building control panels for controlling devices connected, either directly or indirectly, to a computer system. For example, the capabilities described in detail in Chapter 8 which permit building of a wide variety of interface functions such as gauges, lists, buttons, and so forth are ideal for the creation of control panels that can replace hardware control panels. In addition to replacement, the creation of new concepts for control and monitoring is simple to consider because all the tools needed to build and experiment with new styles of panels are at hand. For example, one might consider the creation of control panels that permit controlling and organizing the activity of multiple cooperative instruments. By using such a control panel, a single human operator might be able to operate a much larger number of machines than by using traditional hardware control methods. Furthermore, it is certainly conceivable that dynamically configurable control panels could be implemented that would adapt to the environment. An example of building a control panel is given in an upcoming section.

Next, the kinds of interface activities that might be appropriate for the hardware discussed in the previous section of this chapter will be examined.

Interface Applications

The Use of Serial and Parallel Connections. Serial and parallel I/O ports are the most widely used means of communicating with a variety of devices. Often, serial and parallel communication is interchangeable. For example, printers frequently can be connected to either type of port, and within the operating system, output can ordinarily be redirected from one port to another. Various styles of communicating with printers exist. For example, some printers, such as the Apple LaserWriter, communicate in a language called PostScript (Holzgang, 1988) which is interpreted internally by the printer prior to producing an image on the printer. PostScript is a page description language that employs ASCII character representation to describe (albeit rather cryptically) how to print a page. In contrast, some other laser printers (e.g., Hewlett-Packard printers) use a raster scan technique for sending bitmaps to the printer, in which each dot is described. The raster scan technique is directly analogous to the form or Depth1Image display mechanism of ST-80, in which each bit of an image is assigned a 1 or 0 to indicate whether a white or black pixel should be displayed. Another text (Pinson and Wiener, 1988) gives a good example of how to interface a raster scan printer to Smalltalk.

Several communication and instrumentation examples are presented next.

Serial Control. Consider the scenario shown in Figure 12.10 in which a robot is controlled by a program written in ST-80. The application is the automated observation of parts on a manufacturing line. A camera mounted on the robot arm acquires video images and returns the images to the computer. Control of the robot is needed to adjust the height of the camera to be able to inspect different types of parts that might be run on the line. In the figure, only two cables are shown connected from the serial and video cards in a computer to the robot, although there might well be additional cables for controlling the flow of parts on the line shown. Internal to the robot is a small interpreter that can digest serial commands sent over the cable. Commands might be of the type "UP," "DOWN," "LEFT," "RESET," and so on, which can be readily created and stored in dictionaries in the application program. Both the robot and the belt could be controlled serially, based on video information returned to the computer, for detecting, for example, errors in the physical fabrication of a board. Shown on the right side of the diagram is a conceptualization of how the window containing a control panel might look for an application of this type. A selectionInListView could be created which would permit the user to choose various options. The video picture that the camera sees, perhaps including the highlighting of problem areas, might be shown in a view. Finally, a dialog box

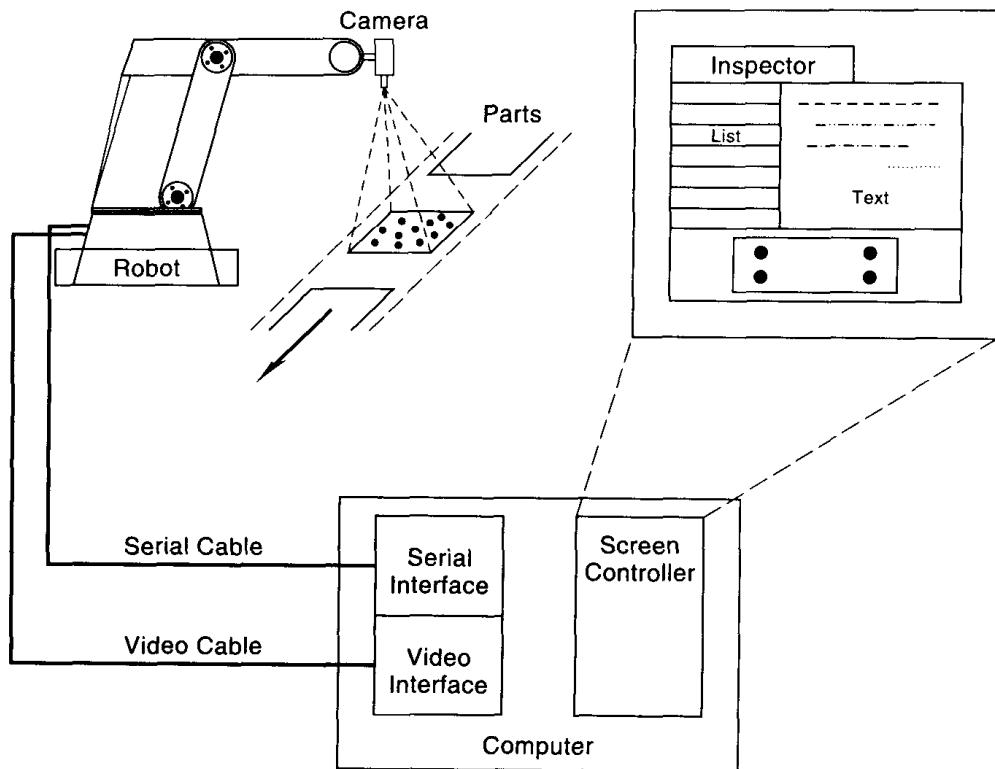


Figure 12.10 Controlling a Robotic Parts Inspector.

(textCollector) could be displayed in which advice could be presented to the operator of the system.

Serial Communications. Smalltalk-80 provides the capability for connecting to serial ports on any of the supported hardware. Table 12.3 displays how a serial I/O port can be opened for three different hardware platforms. Once an identification for the opened port is obtained from the operating system (termed “handle,” i.e., a name to call the port by), various options for the serial port, such as speed and number of bits, can be set. More information and examples about these options can be found using the Smalltalk browser.

In Release 4 of ST-80, the ObjectKit \ Smalltalk Advanced Programming software supplied by ParcPlace systems provides classes **SunTerminal** and **VT100Terminal** that permit displaying text collected from the serial I/O port. An example of a VT100 terminal window is shown in Figure 12.11. The functionality of this terminal window is virtually identical to the functionality provided by standard serial communications packages. Hence, it is questionable

Table 12.3 Examples of Serial Port Usage in ST-80

(Note: These classes need to be filed in for Objectworks\Smalltalk Release 4.)

Opening a Port

```
handle := UnixRealTtyAccessor open: '/dev/ttya'.
handle := DosTtyAccessor open: 'COM1:'.
handle := MacTtyAccessor open: 'modem'.
```

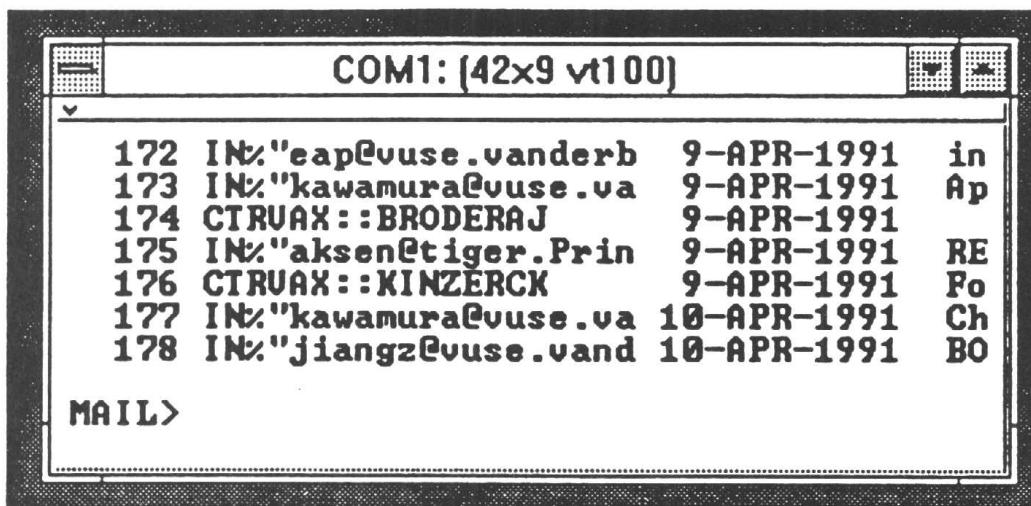
Setting Serial Options

```
options := DosTtyOptions new.
options bits: 8;
speed: 9600;
stopBits: 1;
parity: nil.
```

```
handle setOptions: options.
```

See OS classes in the ST-80 browser for examples.*

whether this facility will serve a useful purpose when other communications packages are available. However, if an application required embedding a terminal window as part of multiple views in a window, the classes provided would indeed be useful. One might enhance the communications facility of ST-80 to use multiple textCollector views that could sort electronic mail as it is received, for example, into different categories, such as "urgent," "read later," or "junk mail." This capability would be straightforward to implement by reading incoming text and routing the mail messages to dictionaries associated with different textCollector views. Another variation on this same concept

**Figure 12.11** A Terminal Window in ST-80.

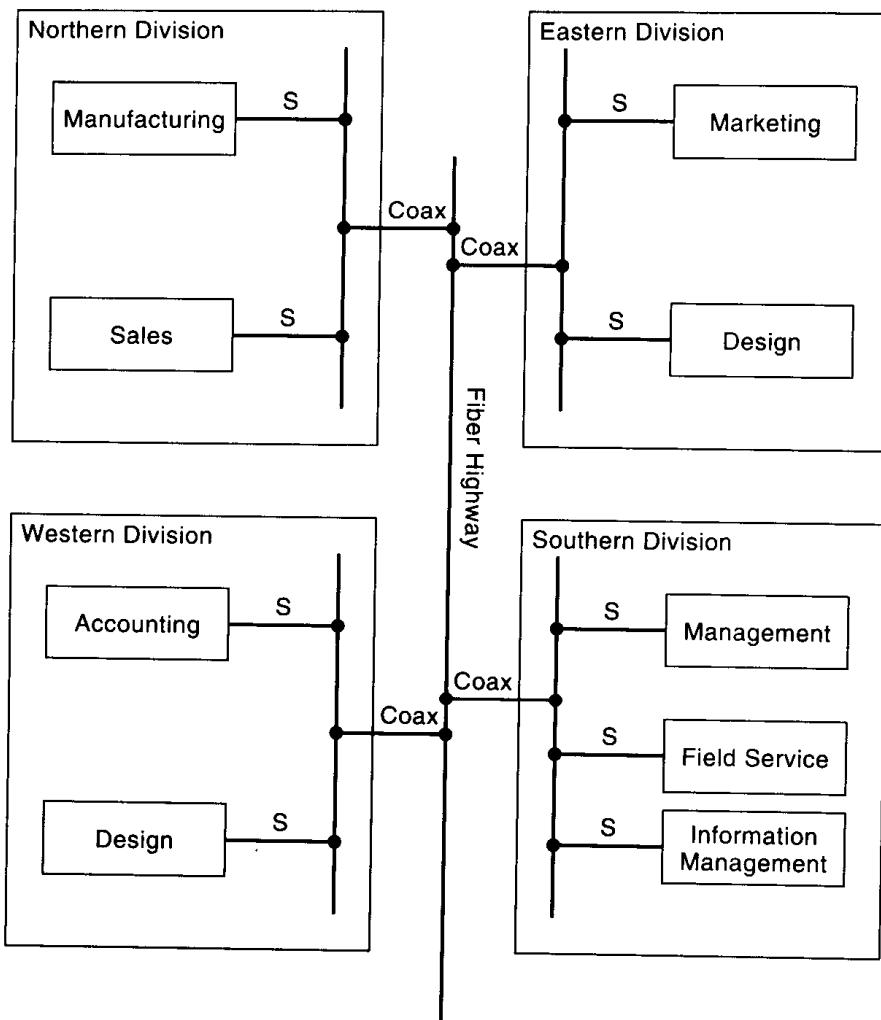


Figure 12.12 Communications Concepts for Group Work Example. (S = serial).

would be to facilitate groups of individuals working together. For example, multiple views in a terminal window could be created such that there might be different views of messages sorted from different members of a team collaborating on a project. A framework for an example of this kind of idea is outlined in Figure 12.12 in which individuals representing different kinds of functionality in a manufacturing organization are all connected via a network. Individual workstations (i.e., PCs, Mac, etc.) could be connected using serial connections to a wide-area net, perhaps implemented using fiber-optics communications media. Creating a multiperson communications facility for a group such as the one shown should be a fairly straightforward exercise of extending the MVC triad

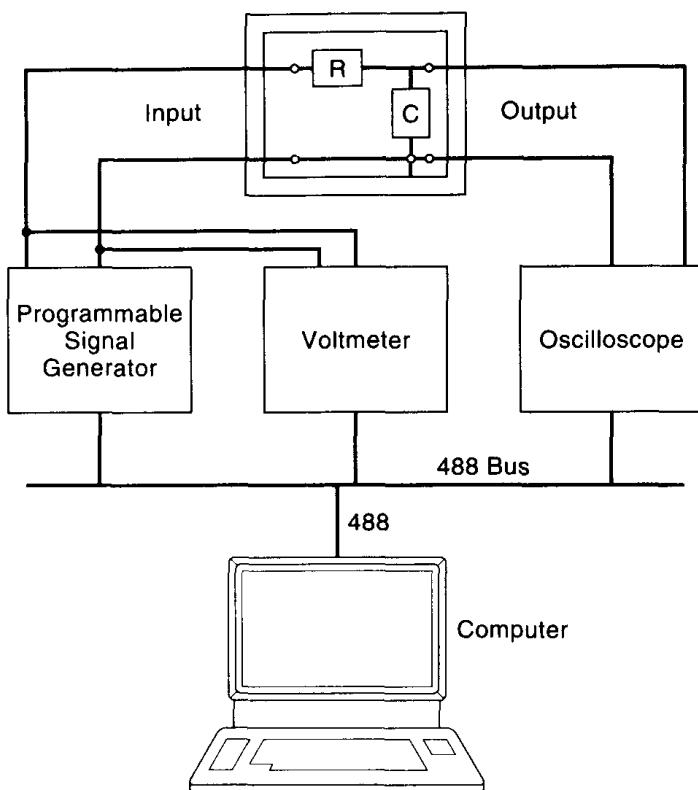


Figure 12.13 Laboratory Control of Instruments.

that implements a terminal window. Each workstation might display multiple views of text commentary by individual team members, or the sorting concept mentioned previously might be used to organize the work of the team into a browser that lists the messages by category.

Direct Control of Instruments. Another interface application is the connection of measurement instrumentation to the ST-80 environment, for the purpose of controlling and communicating with instruments. Figure 12.13 presents the conceptualization of how an instrumentation control system might be configured for conducting an experiment on testing the characteristics of a simple electronic circuit. A laboratory bench preparation could contain a breadboard of electronic components with input and output signal connections. The input could be driven by a programmable signal generator, with the signal level monitored by a voltmeter and the output of the system monitored by an oscilloscope, which could relay information about the contents of the signal measured back to the computer. All instruments could be synchronized via the

Table 12.4 Controlling a Digital Voltmeter with IEEE-488 Commands

Command Category	Command List	Function	ASCII Code
Function step		Set meter to measure dc	F1
		Set meter to measure ac	F2
		dc amps	F3
		ac amps	F4
Range selection		AutoRanging	R1
		100 mV, 0.1 A	R2
		1 V, 1 A	R3
		10 V, 10 A	R4
		100 V	R5
Triggering		Internal trigger	T1
		External trigger	T2
Autozero		Zeros the instrument on	Z0
		Zeros the instrument off	Z1
Example Command String			
"F1R2T1Z0"			

posting of control messages on the IEEE-488 bus (see the following examples). As in the previous example, a multiview control panel could be created which would permit control of laboratory experiments and summarization and plotting of results in a view.

Table 12.4 presents information about how to control an example instrument—a digital voltmeter. Control is via the transmission of a series of ASCII codes over the IEEE 488 bus. At the top of the table is a set of sample commands similar to those found in physical digital voltmeters that use the IEEE-488 bus. The commands are mnemonic; that is, they provide a terse rendition of a command in English by using the first letter of that command followed by a digit. Shown at the bottom of the table is a sample string that sets the voltmeter to measure dc voltage in the 100-mV range, using the internal trigger and with autozero turned on.

A computer can access the IEEE-488 bus by (1) simply sending serial strings from a standard serial port to a hardware device that converts serially coded information into the IEEE-488 format or (2) by directly driving an IEEE-488 interface board directly plugged into the computer. Several relatively complex factors need to be understood about how to accomplish direct program-to-hardware communication with a physical board. To explicate how to construct such an interface, a simple example will be given for a D/A converter,

Table 12.5 Controlling a D/A Using a C-Language Interface

```
#define DA 0×2EC
#define COMMAND 0×2ED
/* Interface code as in Table 12.3 */

main ()
{
    outp(COMMAND, 0×f); /* setup board */
    outp(COMMAND, 0×1); /* clear errors */
    outp(COMMAND, 0×8); /* write D/A immediate */
    outp(DA, 0×0); /* specify channel 0 */
    outp(DA, 0×100); /* output a value */

}
/* Interface code */
```

one of the most straightforward I/O devices that can be explained. Interfaces to more complicated devices will proceed in the same manner, although with added complexity.

Direct I/O Port Connection Methods. Table 12.5 presents an example of interfacing a digital to analog converter to the ST-80 system. The virtue of explaining the interfacing of a D/A converter is that it is conceptually and actually easy to understand the code needed to drive the converter using C, even if one does not understand the C language. The C code shown in the table follows the same format as the general interfacing example methodology discussed previously in Section 12.3. The first line of Table 12.5 defines DA as a physical address: $2EC_{16}$; the preceding 0x simply indicates that the number should be interpreted by the C compiler as hexadecimal. In C-language compilers, there are typically mnemonics such as “inp” or “inport” and “outp” or “outport” that are compiled to assembly level code commands that address the I/O ports in DOS machines. In the example, “outp” is used to put a particular number into an I/O port. In other machines, for example, those that use Motorola cpu’s, direct memory addresses are normally accessed rather than the ports to control I/O cards. In this case, a C structure can be created which represents the memory locations to be accessed and values put into these locations to control the I/O card.

A digital to analog converter converts a numerical value into an analog voltage, which is made available on a cable to the external world. To operate a typical D/A converter, two registers are employed: the command register (here named COMMAND) and the data register (designated as DA, in this example). Operation proceeds by sending a setup command, a clear errors

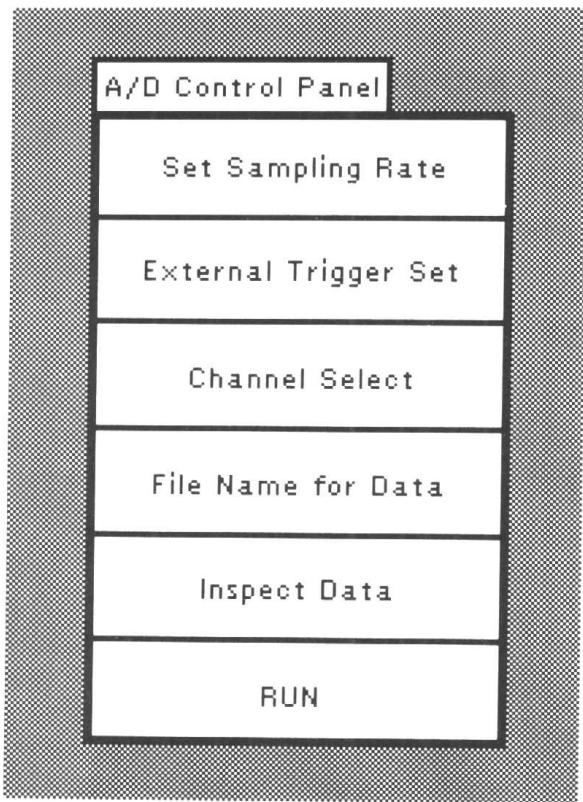


Figure 12.14 A Control Panel for an Analog to Digital Converter.

command, a command about what to do, the channel number, and finally the data.

In the example, the first argument to `outp` is an address, as indicated in the earlier definitions in this code fragment, and the second argument is code sent to the board for control purposes or data. The port commands on a PC can be used for accessing any of the machine ports. Because peripheral devices are usually connected to port or directly to a memory location, it is straightforward to address ports or addresses in C programs interfaced to Smalltalk.

Code to interface I/O devices to Smalltalk can be written in the manner described in the last few paragraphs. However, it is essential to determine if the manufacturer supplies the needed information about the characteristics of the card.

Creating a Control Panel Example. As an example of creating a control panel in ST-80, Figure 12.14 displays a control panel for an analog to digital (A/D) converter. The example was created using the ViewBuilder program described

in Chapter 8. The design begins with defining the external functions of the A/D that need to be accessed in order to use the A/D board. Six main functions in this example were identified: setting the sampling rate, controlling the external trigger, selecting a channel, designating a file in which to store data, initiating inspection of acquired data, and running the system. The six items are actuated by pressing any of the six buttons shown in the figure. Recalling from the description in Chapter 8 that a button press executes code in a code block, one can add calls to primitives written in C which actually carry the functionality written on the buttons in the control panel.

12.5 The Wrapper Concept²

The concept of a “wrapper” comes from the commonsense usage of the term in the context of software. Many software packages are difficult to understand and use; hence, it would be useful to “wrap” these packages with a layer that would assist the user in using such systems and would assist in interpreting the output of a package. The ST-80 environment, particularly the graphics interface mechanisms of the MVC, permits the creation of interfaces that are more user friendly than the typical interface constructed for engineering software packages. Indeed, during the last few decades, many large and complex software packages for use in engineering have been created in a variety of computer languages with little regard for portability or reusability. For example, there are many quite expensive software packages for engineering modeling in a variety of the subfields of engineering that could benefit by having a “friendly” wrapper. Although such wrappings are becoming more commonplace with the increased use of window environments for PC and workstation environments, it is frequently difficult to understand how one might create custom wrapping for existing programs. Some scenarios for creation of simple wrappers are described next in the context of the MVC tools studied in this text.

Figure 12.15 diagrammatically illustrates the wrapper idea. Imagine that one has a large simulation package, as indicated at the top of the figure. Surrounding the package are interface functions that interpret both the input and output of the package through conversion routines that interface to a standard ST-80 window. The interface in the window might contain buttons for control, lists for selecting options, and a view in which results can be displayed.

²The wrapper concepts presented in this section differ from the wrapper concepts presented in Chapter 8. The class **Wrapper** in ST-80 provides different view functionality; here wrapper refers to surrounding or enclosing an application.

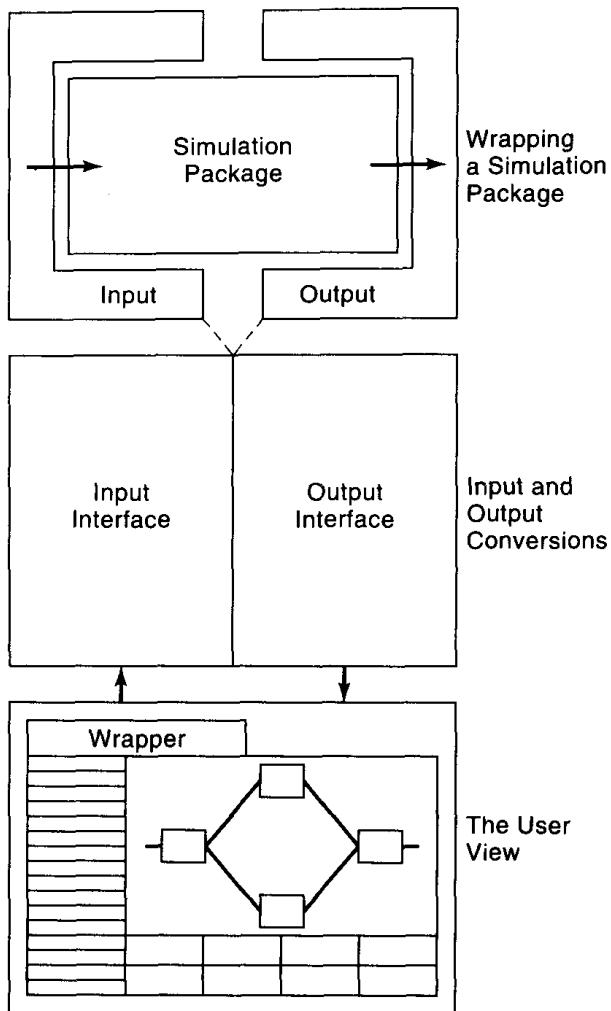


Figure 12.15 The Wrapper Concept.

Figure 12.16 shows an actual implementation of a wrapper (Bourne et al., 1989) that uses two views. The view in front is an ST-80 image created from information returned from a simulator and the view in the background is the actual wrapper view. Although part of this view is occluded, it is easily describable. First, there are four `selectionInListViews` across the top of the window in which different alternatives are available for operating the simulation system that is wrapped. At the bottom of the view are two `textViews`. On the left, prepackaged listings of commands that are available for driving a simulator can be viewed. On the right, occluded in this example, is a blank `textView` to which operational commands can be cut and pasted. Once the user creates a

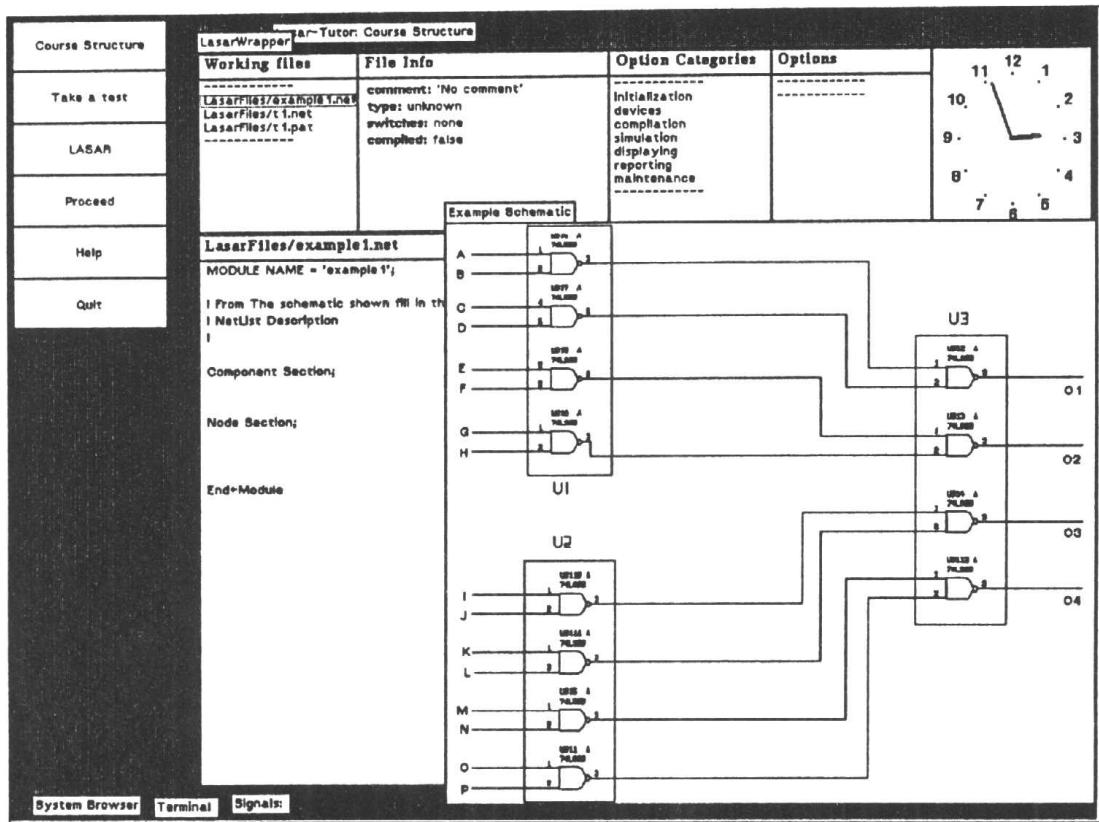


Figure 12.16 A View of a Wrapper Implementation.

command file, by analogy to the command files shown in the left view, the command listing can then be executed. The virtue of this scheme is that the user can find by browsing a set of command lines analogous to his or her current needs and tailor the listing selected simply by copying and pasting to a new view that is simultaneously observable with the original command lines.

The particular implementation shown in Figure 12.16 is reprinted from an application in which a large digital circuit simulator (LASAR; Teradyne, 1988) was wrapped. In this case, ST-80 was used to create the interface and the actual simulator code ran on a physically separated machine (Bourne et al., 1989). UNIX versions of ST-80 provide the capabilities for linking across multiple machines, hence, it was simple to create an interface on one machine and have the simulation run on another. ST-80 provides access to the UNIX facilities for “socketing,” that is, creating pluggable communications channels across multiple machines. Unfortunately, the DOS and Macintosh versions of ST-80 do not currently have this capability.

12.6 Object-Oriented Database Management

Some ideas about supporting groups of individuals working together using computer-supported communications media have been mentioned previously. This type of distributed work environment introduces the problem that there must be some central repository for information that can be accessed by multiple individuals. This issue is well known for management of large information systems that require central data storage for access by a large number of users. Examples include stock market information access, airline reservation systems, and library information retrieval. Some common methods of storing information in databases include both relational databases (i.e., in tabular record-oriented representations) and hierarchical databases. Problems with traditional database management systems include limitations in available representational structures and stylized query mechanisms for retrieval of information. Also, there are typically "impedance" mismatches (i.e., the nonmatching of one program to another) between a database management system and an application program. The idea behind object-oriented database management systems (OODBMS) is to provide the capability for storing objects, no matter how large or how complex, in order to reduce application development time and increase modeling power. An example of an OODBMS is the Gemstone system. Gemstone (Maier and Stein, 1987) is an object-oriented database management system that capitalizes on the Smalltalk model, providing excellent facilities for sharing data among multiple users. Gemstone, written in C, provides Smalltalk and other interfaces access to shared data manipulation facilities. For a comprehensive look at this relatively new field of object-oriented databases, the reader is referred to Kim and Lochovsky (1989).

The addition of an object-oriented database management system to Smalltalk application programs can significantly enhance the capabilities of Smalltalk because Smalltalk was originally designed to be a personal computational environment on a single workstation. Objects created in Smalltalk are not persistent (i.e., they are not always there, e.g., the objects do not remain accessible when one exits the image) and are difficult to share among users except by the rudimentary means of sharing ASCII files of objects. The versions of Smalltalk-80 from Version 2.5 on have a binary object storage system (BOSS) for storing objects on files in a more parsimonious fashion than writing text files. This mechanism implements a form of persistent objects, but is not as sophisticated as the capabilities of Gemstone which provides multiple-user access to a database management system, including transaction and concurrency control (i.e., provides the capability for multiple users to share and update commonly shared objects).

An OODBMS can provide the capability of sharing information among various users without the need to capture a complete copy of objects in

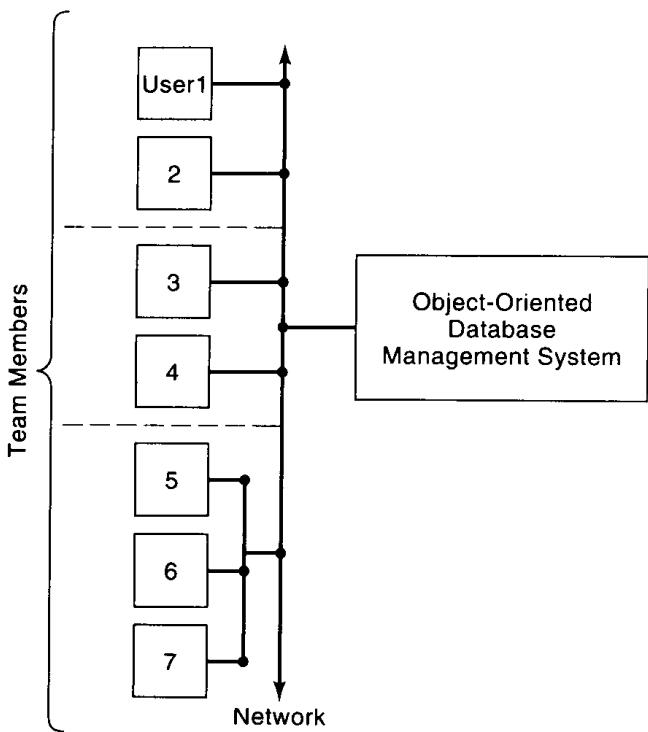


Figure 12.17 Group Work with an Object-Oriented Database Management System.

each user's image or disk. Figure 12.17 conceptualizes how a group of individuals might work together on a network, sharing information resources on an OODBMS. In this scenario, each team member could be provided with the inter-team-member communication facilities described earlier in this chapter, coupled with the ability to view and manipulate persistent objects in a shared OODBMS, as shown on the right side of Figure 12.17.

12.7 Summary

The primary conclusion to be drawn from this chapter is that the Smalltalk-80 environment can be readily interfaced to the world outside the computer. This finding extends well beyond ST-80 in the sense that the interfacing concepts discussed are general in terms of language interfacing and application creation mechanisms. In previous chapters, the excellent graphics and user tools, including debuggers, of ST-80 have been extolled as providing mechanisms for the user to permit creating rapid prototypes of various types in engineering. In this chapter, areas that are difficult to approach using ST-80 alone have been examined—areas in which preexisting and well-defined classes

do not always exist. Because engineering often needs interfaces to the real world, methods for providing such interfaces have been surveyed and found to be relatively easy to create. The combination of an integrated programming environment with interfaces to the external world and to other computer languages offers a strong problem-solving system that should find excellent utility in many areas in engineering.

References

- Bourne, John R., J. Cantwell, A. J. Brodersen, B. Antao, A. Koussis and Y.-C. Huang, (1989). Intelligent Hypertutoring in Engineering. *Academic Computing*, September, 18–47.
- Dittrich, K. R. (Ed.) (1988). *Advances in Object-Oriented Database Systems. Lecture Notes in Computer Science*. Springer-Verlag, New York.
- Gates, S. C. (1989). *Laboratory Automation Using the IBM PC*. Prentice-Hall, Englewood Cliffs, NJ.
- High-C Compiler Manual* (1989). Meta Ware Incorporated, Santa Cruz, CA.
- Holzgang, D. A. (1988). *Understanding PostScript*. Sybex, Inc., Alameda, CA.
- Kim, W., and F. H. Lochovsky (Eds.) (1989). *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley, Reading, MA.
- Maier, D., and J. Stein (1987). Development and Implementation of an Object-Oriented DBMS. In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner (Eds.), pages 355–392. MIT Press, Cambridge, MA.
- Microsoft Windows Software Development Kit (1990). Reference Manuals, Microsoft Corporation, Redmond, WA.
- Objectworks \ Smalltalk User's Guide*, Release 4 (1991). ParcPlace Systems, Mountain View, CA.
- Pinson, Lewis J., and Richard S. Wiener (1988). *An Introduction to Object-Oriented Programming and Smalltalk*. Addison-Wesley, Reading, MA.
- Teradyne Corporation (1988). *LASAR Version 6 Software Users Guide*. Boston, MA.

Exercises

12.1

For the computers that you have access to, determine which peripherals are available and study how these peripherals operate with the facilities of the operating system that you are using.

- 12.2 Browse the ST-80 system. Study carefully the serial interface. Connect a serial cable from your computer to a “dumb” terminal or another computer using a communications program and see if you can send characters from one system to the other using the examples given in the serial interface class.
- 12.3 Project: Implement an A/D and D/A interface to ST-80.
- 12.4 Project: Implement an IEEE 488 interface to ST-80.
- 12.5 Study the manual provided in ST-80 on interfacing C to Smalltalk and write a brief explanatory report.
- 12.6 Undertake a survey of object-oriented database management systems. Contact manufacturers and compare and contrast the capabilities of the various systems you find.

Building an Application: Concepts in Rapid Prototyping

13.1 Introduction

Many of the concepts and methodologies described throughout this text are tied together in a single prototypical application presented in this chapter. The ideas and the concrete programming tools discussed in earlier chapters are used here to create an example of a working digital circuit simulator. Because the focus of the chapter is on rapid prototyping, no attempt is made to present a completed and well-refined example; instead, the skeletal framework of the application is developed and code presented that can serve as a basis for continued development. Classes and code for the digital circuit model, developed in Chapter 3 using the application/class organization method (ACOM) modeling method, and the view building methods discussed in Chapter 8 are used to create a prototypical view of a digital circuit simulator. The purposes of the chapter are (1) to discuss general analytical and design methods, (2) to apply the concepts described to a digital circuit design problem, and (3) to show how the design can be implemented. More code is exhibited in this chapter than in prior chapters due to the need to describe the operation of the complete example. Coding methodologies are explained in detail because the techniques used are generally useful in many different kinds of rapid prototyping applications.

What is rapid prototyping? A prototype is defined as “An original type, form or instance that serves as a model on which later stages are based or judged” (Morris, 1981). As a model, a prototype can be thought of as a skeletal framework which can be expanded or modified to serve various needs. The addition of the term “rapid” to the beginning of the phrase connotes the ability to put together a model of an application rapidly. Hence, rapid prototyping has

come to mean the rapid fabrication of a working model of an application which can be used to demonstrate a concept. Rapid prototypes are useful for explaining what could be done if a prototype would be developed into a complete product or for simply experimenting with concepts to determine the utility of an idea. Rather than spend detailed design and implementation time, the rapid prototyping methodology permits trying out many different ideas without the high cost of detailed design and implementation.

13.2 The Digital Circuit Simulator Problem

Digital circuits, as described in Chapter 3, operate on digital signals, using 0 and 1 logic levels for signals. Collections of digital circuits, such as *and* gates, *or* gates, and invertors (i.e., *not* gates) can be connected to form logic circuits that implement a wide variety of devices such as adders, counters, and even digital computers. A digital circuit simulator is a program that can mimic the behavior of actual digital logic gates wired together and tested by applying a signal to the input of the combination of gates. A simple example used at several points throughout the text is the half-adder, a simple digital logic circuit composed of two *and* gates, one *or* gate, and one *not* gate. When physically wiring a circuit together on a workbench, students learning electronics can observe the behavior of the circuit combinations that they create. A viable alternative to physically wiring a circuit, applying input signals, and observing the output is to use a digital circuit simulator which permits the simulation of a physical system in software. A circuit simulator can provide close to the same educational experience as physically wiring the circuit, if the simulator has sufficient capabilities. At worst, a simulator might have only the capability of logically verifying what the output of a circuit should be given a particular input value or values. At best, the truly sophisticated simulator would contain many more components and instruments than those found on a typical physical workbench. The user would operate in a near-real-world simulated environment with the capability of rapidly connecting various combinations of circuits and observing circuit operation. With many types of simulated circuits available and the capability of abstracting the behavior of collections of components [see, e.g., van der Meulen (1989)], a circuit simulator becomes a potentially powerful design tool with which new electronic designs can be implemented and tested without the need for ever fabricating an actual circuit until the design is complete. Once a circuit is designed and debugged using such software-based simulation methodologies, actual physical devices can be created using the designs obtained from the simulation. Indeed, a circuit simulator, when devel-

oped fully, can be considered to be a productivity tool for design, as well as a learning tool for students.

As discussed in Chapter 3, a digital circuit simulator should permit connecting digital circuit components such as *and*, *or*, *not*, and other gates together with simulated wires. Using the ACOM methods, a description of various gates and wires was given in Chapter 3 and ultimately implemented and illustrated in Chapter 10. Neither in the class descriptions given in Chapter 3 nor in the discussion of constraints in Chapter 10 was any graphical means of display implemented. In the digital circuit simulation implementation in Chapter 3, the user could view the output signal values of a wired circuit on the system transcript. Later, in Chapter 10, inspector views were used to examine signals in a constraint network. The purpose of these implementations was to illustrate some basic properties of object-oriented programming, including inheritance, and to introduce the ACOM method. Although the few classes created permitted verifying the operation of simple logic circuits, the implementation technique required the user to build and name each object and wire in a workspace, including specification of the wiring of the individual components. Moreover, although one could inspect variables in this simulation, it was difficult to secure an overall global appraisal of the operation of the entire circuit.

To create a graphics-oriented circuit simulation system, the models described in Chapters 3 and 10 should be linked to a view in which graphical representations of circuit elements can be displayed and the circuit animated by displaying the signals propagating through the simulated circuit elements. Furthermore, a graphics builder interface should be added that would permit a user to construct a simulated system using only the point-and-click metaphor. The goal of this chapter is to show how to create a prototype of a digital circuit simulation system that has both circuit building and animation capabilities.

Basic conceptualization methods for creation of the prototype are examined next followed by a detailed description of the implementation code. The code builds directly on the code described in previous chapters. Although not complete, the code presented furnishes a sufficient framework to permit a serious student to develop a more complex example within a short time period. This chapter will conclude with a discussion of ideas about different directions in which the prototype could be developed.

13.3 Maxims for Rapid Prototyping

This section introduces some ideas about rapid prototyping, incorporating and reviewing concepts already covered that are germane. The maxims

given are almost commonsense knowledge and fit well with the conceptualization of rapid prototyping as an analysis phase followed by design and implementation. The maxims given in the following discussion are divided into the four areas: analysis, design, implementation, and incremental refinement.

The Analysis Phase

At the outset of the analysis phase of creating a rapid prototype, an assessment of the design intent should be made (i.e., the teleology, as discussed in Chapter 2). This assessment is a kind of requirements analysis, but not extremely detailed. Particularly important at this point is determining what the functionality of the system being created should be, in concept. An especially useful mental (and paper-and-pencil) operation is to attempt to sketch out what the behavior of the prototype should be; that is, how does a user interact with the system and what does the system do while the user is engaged with the system? At this point in creating a prototype, the chief notion is to try to link your idea about a new system with what the system should *do* when it is completed. Given the background on implementation paradigms presented earlier in the text, it is useful to think in terms that ultimately can be implemented. For example, if there are multiple functions to be carried out which might be activated by pressing buttons, buttons should be included in a prototype to activate the different designed functionalities.

A second important analysis requirement is to consider deeply how the system should appear to the user and how the user should operate the system. A useful construct here is the use of *storyboards*, that is, a series of sketches that illustrate how a system would work, if constructed. It is very useful to produce a set of storyboards that show each possible view that the user would see, including switches, menus, and lists and how they might work.

A third analytical requirement is the consideration of the kinds of knowledge and data representations required for the prototype. Knowledge and data may fit easily recognized objects with which the designer is familiar, such as dictionaries or ordered collections. At this stage, the designer may recognize other requirements, such as the need for reasoning or representation of a rich set of semantic constructs. Recognizing such representational needs would lead to the conclusion that rule systems and semantic nets would be needed in the prototype. In general, however, rather than resorting to complex representational structures for a prototype, simpler style representations are preferable simply from the debugging and understandability position. Once a system operates properly in the prototype, more complex representational schemes can be substituted as desired.

The Design Phase

Once concepts have been formed by analyzing the requirements for the system, the design phase commences with specification of the actual components of the application. First, a model that will represent the required knowledge and data should be specified. This step is identical to the ACOM method outlined in Chapter 3. The model should be tested using the simple evaluation techniques of creating code in a workspace and exercising the code to see that it performs the intended function (e.g., such as testing digital circuit operation as was done in a workspace in Chapter 10).

For all objects in the model, a specification of the instance and class variables needed should be determined. Application-wide variables should be identified early; that is, are there class variables that need to be identified that will require access by all objects in the system? In many cases, only instance variables will be needed; only occasionally are class variables needed that are accessible to all instances of the class.

For organizational purposes, opening an application is normally accomplished by using a class method or instance method, often called “open” in the application view or model. Inside the “open” method, linkages are made between the model, view, and controller to permit subsequent communication among these three major parts of an application. Typically, the application class is given a descriptive name and stored in a named category created using the class browser.

The remaining design steps are the specification of the characteristics of the view and controller. Either the **Controller** or the **ControllerWithMenu** class can be normally used in a prototype because this class provides most of the control functionality needed. The class **Controller** is provided with the creation of any **View** subclass. The design of the view requires additional thought, however. In particular, the placement and type of views in the window should be designed. This step essentially matches the requirements described in the storyboards by specifying the actual view types and placement. As outlined in Chapter 8, it is useful to employ a tool such as ViewBuilder (Jiang and Bourne, 1991) to create a view and controller that can be interfaced to the model.

The Implementation Phase

To implement a design, the use of ViewBuilder is a good place to start. Graphically prototyping the view and controller components with this tool permits rapidly creating the skeletal view framework. Without ViewBuilder, the user should write an “open” method that follows the same schema shown in Chapter 8 for creating a window and adding views in a container. In fact, the

examples shown in Chapter 8 can be followed directly. The most reasonable scheme is to create a view in a window first, then open the window and determine that the view placement is reasonable. If you do not use View-Builder, special care must be given to defining the placement and the extent of each view.

Once the view is built and the controller implemented (normally by doing nothing special and simply allowing the default controller to be used), the application model can be added to the views by sending the message *model: aModel* to the views that require a model. After linking in the model, debugging and incremental refinement of the application can commence. Debugging is accomplished by trying to run the application and activating the various buttons, list selections, and gauges used to control the application. Any errors found while running the application will result in a debugger window popping up. The debugger window (see Chapter 6) will highlight the message send that caused an error. Frequently, it is necessary to move down the message send stack (i.e., move to the previous item in the message sends listed in the list at the top of the debugger view). The usual problem is that an object is being sent a message that it does not understand. One can correct the problem in the debugger and continue or restart the application.

Incremental Refinement Phase

The incremental refinement phase of the implementation is probably the most interesting because new features rapidly appear in the prototype. Starting with a working simple prototype generated as described previously, small refinements can be made one by one and tried out. New features can be added to buttons in the view, and different styles of menus or icons can be experimented with and/or modified.

13.4 The Digital Circuit System: Teleological and Functional Assessment

Teleology of the Simulator

The specific example designed and implemented in this chapter should permit a user to select digital components from a list or set of buttons and place the icons representing the components on a viewable graphics context. As each component is placed, the wires connected to the component should be specified. Once all components and wires are connected, the user should be able to set the signals on the input and observe the signal flow through the connected

gates. To demonstrate this design intent, a half-adder circuit consisting of two *and* gates, one *or* gate, one *inverter*, and six wires will be constructed.

Functionality of the Simulator

Described next are the various components of the constituent elements of the digital circuit simulation which, when combined, are a description of the functionality of the system. The functional items are listed as in a specification.

Creating Components

To create a component, the user should press a button with the name of the component. After pressing the button, an image representing the component follows the cursor until the user clicks any mouse button, depositing the image with the component icon on the view graphics context.

Creating Wires

After completing specification of each component, a pop-up list should appear from which the user selects the wires to be connected to the component in the order: (input1 input2 out) for the **TwoInputGate** class and (input output) for one input gates such as invertors. This design prespecifies an arbitrary number of wires that can be used for connecting components in a circuit.

Visually Connecting Components

The capability for displaying the wires should be included in the design. Drawing of wires should include the capability for ensuring that the wires are exactly vertical or horizontal when the user draws wires in approximately vertical or horizontal directions.

Setting the Inputs

To demonstrate the feasibility of the prototype, buttons should be included that permit setting combinations of the input signal so that the behavior of the simulation can be verified. For the half-adder example to be

constructed, there are two inputs; hence, each input should be able to be set to either 0 or 1. For ease of construction, the wires on which signals can be set are specified to be wires A and B in the pop-up wire list.

Animation

The progression of signals through the circuit should be viewable when setting the input signals.

Encapsulation

Each component should contain information about connections to wires and behavior when activated. This behavior should include how to display itself and how to propagate information to connected wires.

13.5 DCS Architectural Organization

Figure 13.1 displays the architecture of the prototype. The three standard MVC elements are present; the initial conceptualization of the contents of each are given next. Throughout the next sections, specific reference to parts of the code presented in Table 13.4 are given; the reader can benefit from a simultaneous reading of the following information and examining the code. This operation will probably take several readings.

The parts of the design to be considered in detail are as follows.

Model Design

The model for this application is very simple, consisting of ordered collections of gates and wires. The wire collection is initialized for the wires "1" through "10," so that the wires can be referenced by their position in the collection. Pop-up menus refer to the collection by using the letters "A" through "J." Hence, wires at: 1 is the wire "A," wires at: 2 is the wire "B," and so on.

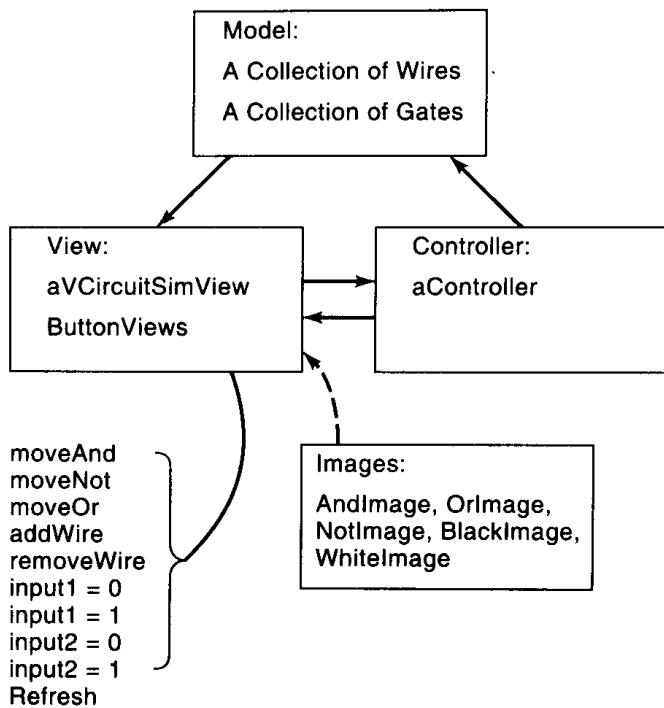


Figure 13.1 Architecture of the Digital Circuit Simulation System.

View Design

The view requires the most design. Ten button views are specified that completely control the application and one view defined in which the circuit can be graphically built.

Controller Design

The controller is specified by default (i.e., **Controller**) via the default controller in the class **View**, although a controller with a menu could be used to permit the adding of pop-up menus for additional functionality at some later time. By simply specifying the view class as a subclass of **View**, the default controller is supplied. For the example given, the **ControllerWithMenu** would work equally well. To reduce the volume of the code shown, controller classes are not displayed.

Auxiliary Images

Five auxiliary images are used in the system. Three of these images are used for icons (*AndImage*, *NotImage*, and *OrImage*) and the remaining two (*BlackImage* and *WhiteImage*) are used respectively to show a black token on an icon when a signal appears at the component and to erase the black token. All images were created using a host drawing package and imported into ST-80 using code such as *AndImage := Image fromUser*. Each small image is pointed to by a global variable in the Smalltalk dictionary. An alternative methodology is to read the images from files using the binary object storage system of ST-80. Another alternative is to use black-and-white forms created by the bit editor in Version 2.5 and earlier systems and to read the forms from a file.

13.6 Organizing the Information to Store

The Model

Components

The digital circuit objects described in Chapter 3 exhibited no behavior other than forwarding their input states to their output. The behaviors of the *and*, *or*, and *not* gates presented here have been changed to add the additional capability of displaying their behavior on the view's graphics context. A more proper way to add this behavior would be to create a subclass of each component that contains the modified methods. For example, a subclass of **And** could be **AndWithAnimation** which would contain a new constraint block. In addition to the constraint modification, a variable "location" was added to each constraint to indicate the location of any particular gate in the window. Also added was a variable to keep track of the *graphicsContext* of the view.

The behavior of each component used in this animated example has been changed to permit display of a black image in the icon for that component. The original code for an *and* gate was, as described in Chapter 3:

```
constraint := [:input1 :input2| (input1 & input2) ifTrue: [1] ifFalse: [0]].
```

Thus, any *and* gate with this constraint will return a 1 when both its inputs are 1 and 0 otherwise. To change the behavior of this constraint to provide an animated view of the operation of the constraint, one can simply add a few new

statements inside this block, as follows:

```
constraint :=
  [:input1 :input2 :aGraphicsContext :location|
   (input1 = 1) & (input2 = 1) ifTrue:
    [BlackImage displayOn: aGraphicsContext at: (location + 5@10).
     (Delay forSeconds: 1) wait.
    WhiteImage displayOn: aGraphicsContext at: (location + 5@10). 1]
   ifFalse: [0]].
```

The changes to the code within this block are to add three statements: (1) to display an image of a black dot, (2) wait a short while, and (3) display another image with a white dot. The images were cut from the host drawing package and imported into ST-80 using *Image fromUser*. The white image is sized to cover the black dot. The action of the constraint is, when sent a signal, to blink the black dot on the icon of the gate. The location at which the dot appears, for illustration, is offset from the location passed to the block, which in this example is the top left of where the icon should be displayed. The offset was arbitrarily chosen to be 5 pixels in the *x* direction and 10 pixels in the *y* direction so that the dot would appear inside the image of the *andGate*.

Figure 13.2 displays the knowledge organization of the instance variables in the model. The knowledge organization for the gates (i.e., the *and*, *or*, and *not* gates in this example) remains similar to the organization used in Chapter 3. In the model, the instance variable “gates” holds an ordered

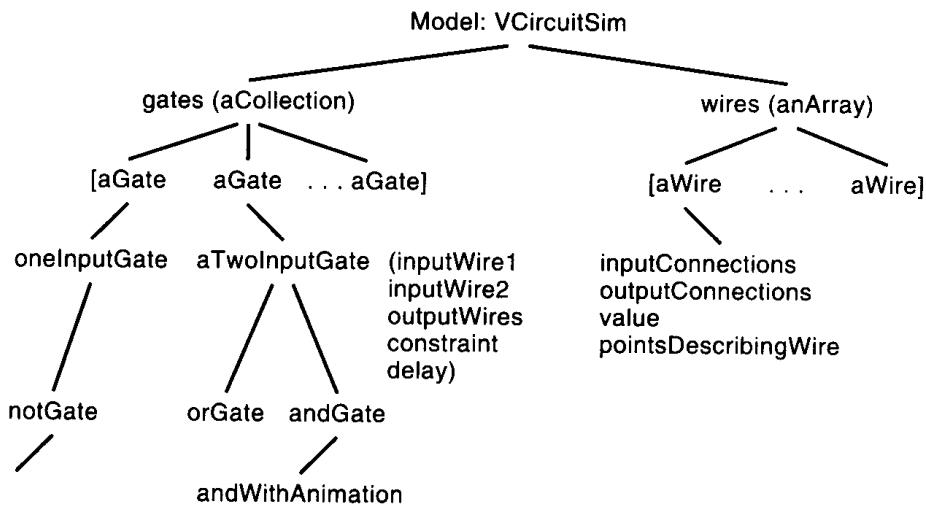


Figure 13.2 Knowledge Organization in the Digital Circuit Simulation System.

collection of gate objects. Each gate object is one of the gates defined when the user places a gate image on the screen. The instance variable “wires” in the model contains a collection of all the wires that are defined. Each wire defined contains the wire input, outputs, value, and a collection of points defining the path of the wire. To make naming wires simple for this example, all wire names are predefined so that each wire can be related to a popUpMenu with the wire names that the user can employ when wiring the system.

The View

Wire Manipulation

Figure 13.3 displays a rather nondescriptive picture of a few wires. There is a point to be made, however, that requires examination of this figure. The **GraphicsContext** method *displayLineFrom: to:* will draw from any point on a **graphicsContext** to any other point. Simply having the points follow the cursor is insufficient however, because it is quite difficult to draw straight lines using only the mouse. It is instructive to try drawing a straight line, as in the example given in Figure 7.4 to verify this problem. Lines in digital circuits are conventionally drawn either vertically or horizontally. To create this example, two possibilities for drawing algorithms were considered: (1) given the end points of a wire, create the wires automatically or (2) permit the user to create and route the wires by continued choosing of points using the mouse button. The latter method was chosen to avoid the complexities of wire routing and to demonstrate the use of a graphics primitive interacting with the mouse.

The protocols in **VCircuitSimView** dealing with wire manipulation contain a number of interesting methods. The method *addwire* is called from

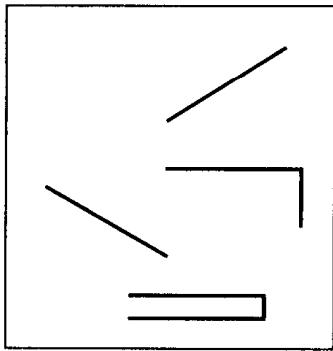


Figure 13.3 Drawing Wires.

the opening view, activated by pressing the button with the same name. The main code in this method is a loop that checks to see if the user has clicked the mouse button twice: *checkForDoubleClick*. When this is not true, the dependent block is executed continuously showing a cursor and adding a line to the drawing when the user clicks a button. The method *straightenLines* is used to ensure that each line, prior to drawing, is either vertical or horizontal, if the mouse is within 10 pixels either vertically or horizontally from the previous point. This technique for drawing permits both horizontal lines, vertical lines, angled lines, and multisegmented lines to be drawn, as displayed in Figure 13.3, by simply clicking on the first and successive points and double clicking on the last point. Hence, when complete, a wire is simply an ordered collection of points that can be stored in one of the wires in the model. To facilitate storing of a wire, upon completion of drawing each wire, the user is asked to select which wire has been placed from those available in the array of wires in the model.

All locations stored for icons and for wires in these examples are specified as offsets from the top left corner of the circuitView defined in the opening method of the view. Note that the graphicsContext of the view must be used to draw on. The reason for specifying component offset from the view corner is that all components should be drawn at the same relative spot on the view no matter where the window is opened on the display. Because icon and wire locations relative to the top left of the view are stored in the model, redisplaying the icons and wires is simple whenever the window is moved or resized.

Images

Table 13.1 displays the names of the auxiliary images used in the DCS example. Each image was created using a drawing package on the host platform and exported to ST-80 using the message *fromUser* sent to class **Image**. For Version 2.5 of ST-80, an alternative technique to create these images would be

Table 13.1 Auxiliary Images Used in the Digital Circuit Simulator Example

Image Name	Explanation
AndImage	Icon for an And gate
OrImage	Icon for an Or gate
NotImage	Icon for an Inverter
BlackImage	A black spot used for animation
WhiteImage	A white spot to erase a black spot

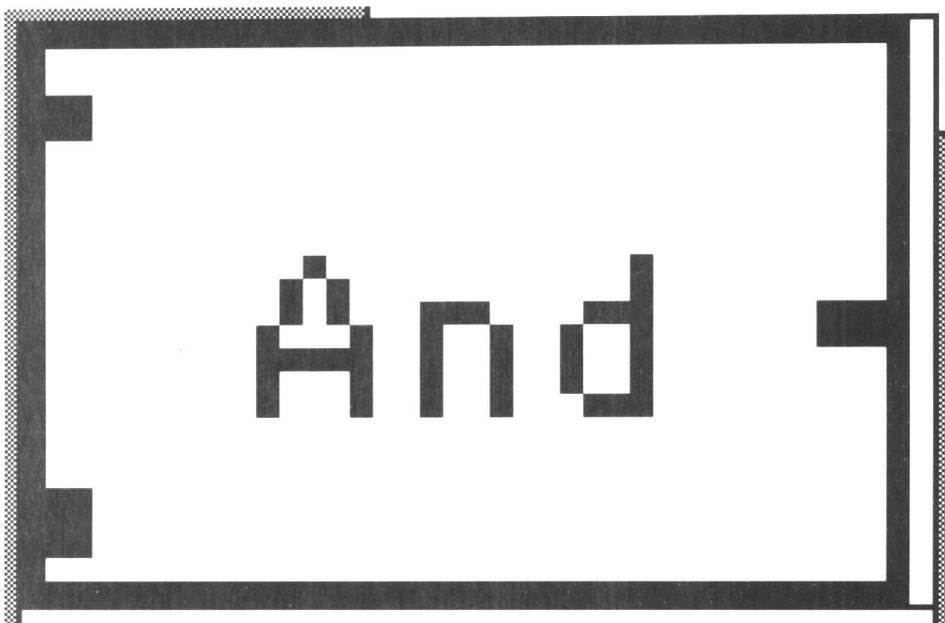


Figure 13.4 A Bit Editor View of the AND Gate Icon (Version 2.5).

to create forms in Version 2.5 using the form and bit editor. Figure 13.4 shows an enlarged picture of an *and* form created. The resolution shown will vary depending on the platform. In this case, a 640 by 480 pixel screen was employed, thus resulting in the fairly coarse bitmap shown. Contrast the form created with the form shown in the form editor displayed in Figure 13.5. As indicated earlier in the text, form and bit editors have been removed from

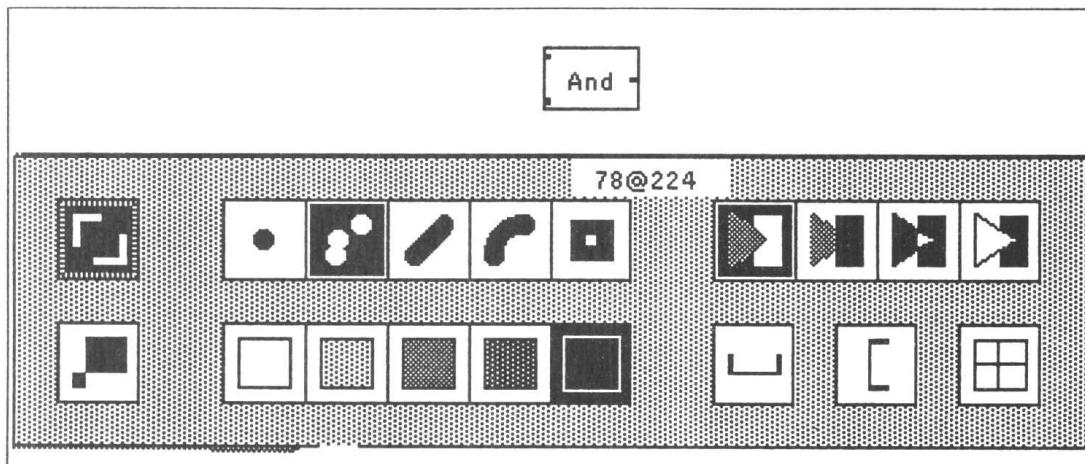


Figure 13.5 An AndGate in a Form Editor (Version 2.5).

Release 4. Hence, users familiar with bit and form editors need to use other methods for creating images. Fortunately, however, the sophistication of commercial drawing packages far surpasses the primitive bit and form editors. Hence, the user is really much better off employing commercial drawing packages and clip art for creating images to use in ST-80-based Release 4 applications.

13.7 Building the View

Determining the View Components

The view for the digital circuit simulator example utilizes only buttons to evoke actions and a view for drawing the circuit. The functionality shown is for demonstration purposes and can be changed quite easily to accommodate additional functionality. The simulator was built several times in several different ways. First, in one version, ViewBuilder was used to implement the view. Figure 13.6 shows the screen view of ViewBuilder with the CircuitInterface constructed. As described in Chapter 8, the method of operation of the ViewBuilder is to first use the create button to size a new view followed by adding other views from the pop-up menu. As shown in this version done in Version 2.5, nine buttons were added to the window. The three buttons at the top left of the window are intended to implement adding a component: *not*, *or*, and *and*. The next two buttons deal with adding and removing wires and the final four buttons are associated with operation of the system created. These last four buttons are completely specific to the half-adder example selected to demonstrate the operation of the system.

To demonstrate how the simulator operates, the standard half-adder circuit is shown in the example views that follow. The four buttons added in the view permit setting the two inputs to 0 or 1, thus allowing the viewing of the operation of the circuit. Other circuit configurations would require a different input technique; for example, one would want to be able to attach a signal to any desired input rather than use the specificity shown here.

The design of the view is remarkably simple, associating each desired functionality with a button. Each button activates a block of code that calls a method associated with implementing the functionality of the named button. Note also that, in addition to the buttons, a view is specified on which to draw the components and wires.

Even without the use of ViewBuilder, the creation of the view is straightforward, that is, creating and adding buttons and other views to a window. By following the code shown in this chapter, a circuit interface that operates the same as the one shown can be constructed without the View-

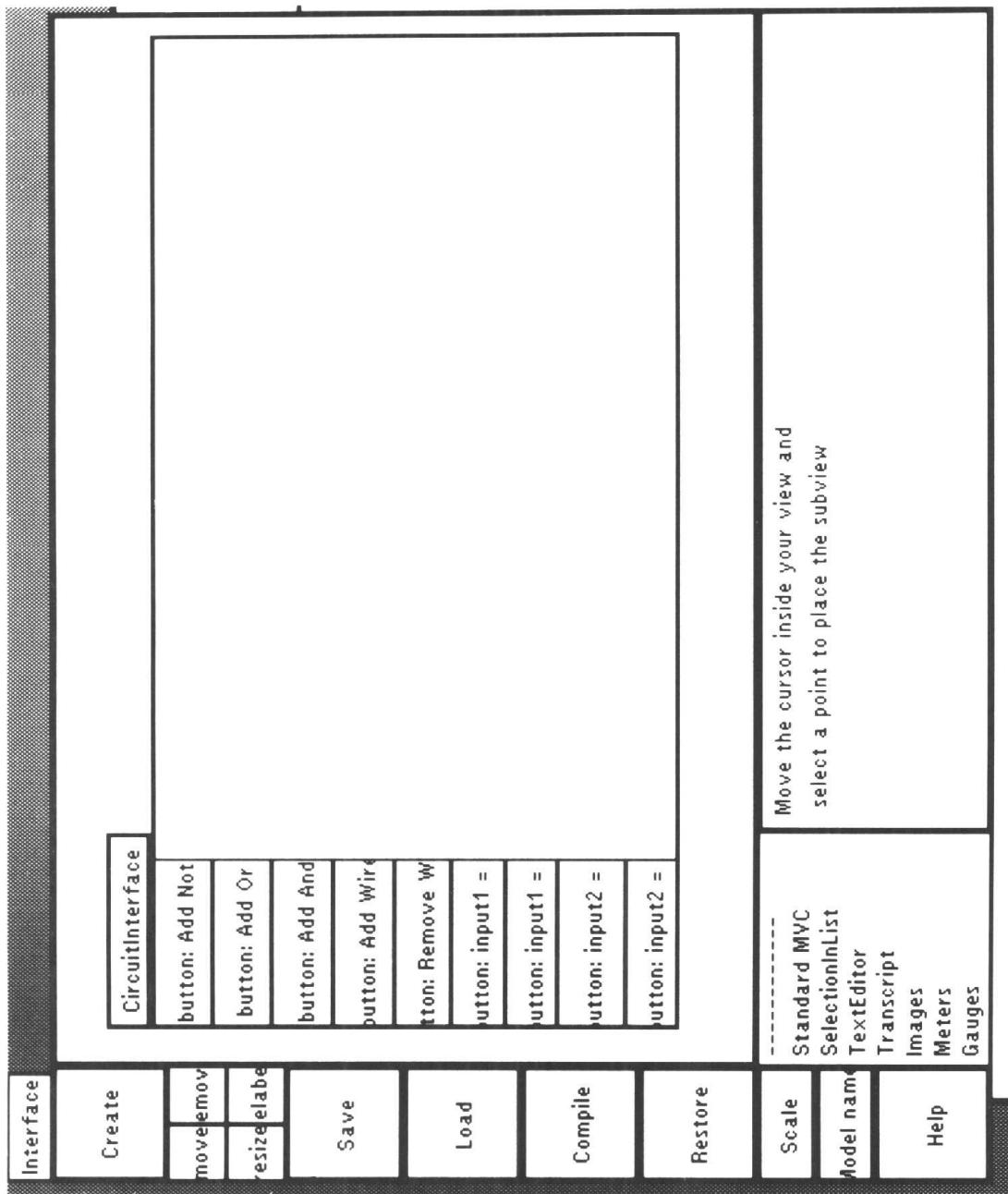


Figure 13.6 The ViewBuilder Interface for the Digital Circuit Simulator. This screen dump is from Version 2.5; the screen dump for Release 4 appears essentially identical.

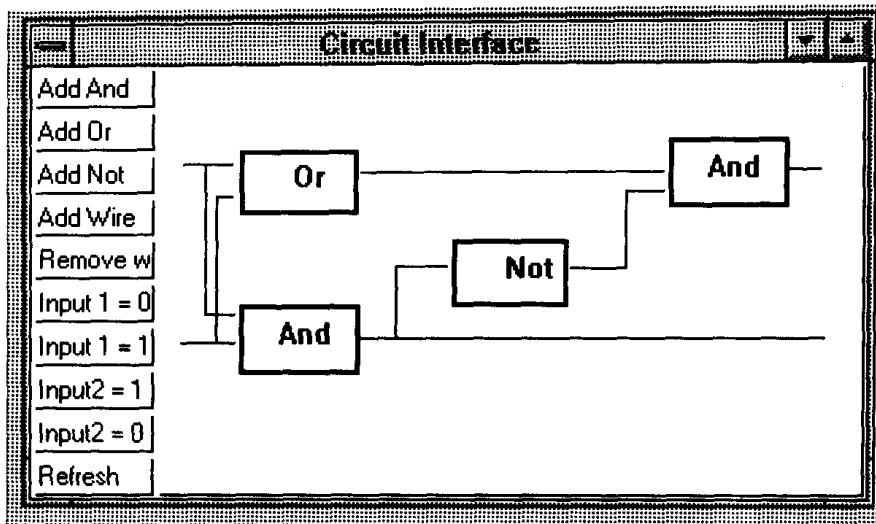


Figure 13.7 The Completed Digital Circuit Interface View (Release 4 screen dump).

Builder by simply creating the needed classes and typing in the code into the browser.

Figure 13.7 displays the completed view, showing the placement of the buttons. The right 85 percent of the view is the area reserved for drawing the circuit. As shown here, the half-adder is drawn, consisting of four gates and six wires. This particular view was created using Release 4; note the slight difference between the way the window outline appears in this example and the other examples written in Version 2.5.

Implementation of the Digital Circuit Simulator

The complete implementation of the DCS is shown in Table 13.4. The major components created are as follows.

1. The opening method, in **VCircuitSimView**, with which one launches the DCS using

VCircuitSimView new open.

A window is created for the application using

```
window := ScheduledWindow new.  
window minimumSize: 300@200.  
window label: 'Circuit Interface'.
```

The value shown as an argument to *minimumSize* was specified arbitrarily. Once the window is opened, the size can be easily changed.

2. Each view is then added to the window by adding views into *aContainer*, an instance of **CompositePart**. For example, a *circuitView* was created by simply sending *new* to **View**. Then, this view was placed in *aContainer* with the following code:

```
aContainer add: circuitView borderedIn:  
(0.15@0.0 extent: 0.85@1.0).
```

where the extent specifies that *circuitView* will take the right 85 percent of the window.

For the buttons, all are added in the same way:

button1 := LabeledBooleanView new.	"make a new booleanview"
button1 beTrigger.	"make it a button that"
	"activates when released"
button1 controller beTriggerOnUp.	
button1 beVisual: 'Add And' asComposedText.	"define its label"
button1 model: ((PluggableAdaptor on: self)	"self refers to this view"
getBlock: [:myself false]	"nothing is done to change"
putBlock: [:myself :value self moveAnd]	"the button"
	"when the button is pressed,"
updateBlock:	"send moveAnd"
[:myself :value :parameter false].	
aContainer add: button1 borderedIn: (0@0 extent: 0.15@0.10).	

In the code shown in Table 13.4, ten buttons are created rather than the nine described in the *ViewBuilder* implementation. The tenth button was implemented to show how to refresh the screen. Each button has an extent specified; in the preceding example code, the button begins in the top left of the window and extends 15 percent in the *x* direction and 10 percent in the *y* direction. Adding many buttons to the code by replicating the code is not a big chore because of the cut-and-paste capabilities that exist in the system browser.

Whenever a window is moved, the views shown need to be updated to redisplay themselves. The buttons contain code for update; *displayOn:* is sent to *circuitView* for redisplay. Also, for illustration, the last button *Refresh* was added to show how the display can be manually updated. Whenever the *Refresh* button is pressed, the *displayOn: aGraphicsContext* method in **VCircuitSimView** is called; the same refreshing of the screen occurs automatically by resizing the view. In the *displayOn:* method, all the objects on the screen are redisplayed using the information contained in the objects displayed. For example, all the icons contain the image to display and the location, relative to the window's top left corner, at which the icon is to be displayed. The only information needed is the *graphicsContext* on which to display, which is passed via the argument to the method *displayOn:.* Because wires are stored simply as a collection of *wirePoints* in each wire, redrawing the wires simply requires passing *wirePoints* to the *displayLineFrom:to:* method of **GraphicsContext**.

An alternative technique for updating the view is to provide a *self changed* message in the model after any changes are made to information stored in the model. The *self changed* method causes an *update* message to be sent to the view which, in turn, ultimately results in a *displayOn: anArgument* method in the view being sent. This linkage works best for selectively redisplaying only parts of the view that have been changed. Specification of the part of the view that is to be redisplayed may be indicated in the *self changed: anAspect* message, where the *anAspect* argument is passed via the *update* method to the view. In the view, different parts of the view can be selectively updated according to the value of *anAspect*.

13.8 Interfacing to the Model

The model remains relatively separate from the view in this implementation of the digital circuit simulator. Some information has been added to the model, such as the changes in the model for supporting animation and the data about the location of the gate and the *graphicsContext*. One task of the view is to interface with the user and to extract information about the contents of the model while the user is graphically building the circuit. When adding an *and* gate, the code shown in Table 13.2 is activated (for the purposes of this discussion, each line is numbered for further reference).

In Table 13.2, the first action in the method named *moveAnd* is to make the cursor blank (line 1) and have the *and* image follow the cursor as long as there is no button pressed. When any button is pressed, the image is displayed at the cursor location on the *graphicsContext* and that location is stored in the local variable *locationOfImage*. In line 7, *gateHolder*, the second local variable, is assigned a new *and* gate which is initialized. Next, in lines 8

Table 13.2 Code for Creating an AND gate

```

1      Cursor blank showWhile: [AndImage
2          follow: [circuitView controller sensor mousePoint]
3          while: [circuitView controller sensor noButtonPressed]
4          on: circuitView graphicsContext].
5      AndImage displayOn: circuitView graphicsContext at:
6          (locationOfImage := circuitView controller sensor mousePoint).
7      gateHolder := AndGate new init.
8      gateHolder
9          input1: (self getWireFrom: circuitModel)
10         input2: (self getWireFrom: circuitModel)
11         output: (self getWireFrom: circuitModel).
12     gateHolder storeLocation: locationOfImage.
13     gateHolder storeGraphicsContext: circuitView graphicsContext.
14     circuitModel gates add: gateHolder

```

through 11, the new *and* gate is wired by permitting the user to connect to the inputs and output of the gate in the same style as in Chapter 3, except using menus. Finally, the location of the gate is stored in the *and* instance and the *and* gate added to the collection of gates contained in the model (line 14). Thus, the model contains all the needed information about each gate, that is, its behavior (created during *init* in line 7), its connections (lines 8 through 11), and where it should be shown to the user (line 12).

The remaining interfaces to the model are basically accomplished in the same way: the *or* and *not* gates have the same construction and the wire operations collect wire points and place collections of wire points in the model. The signal operations simply set the values of the signal on the inputs of the circuit in the model. In this example, *input1* and *input2* are assigned to the first two wires created. When signals are set on the wires, the values propagate through the constraint blocks in the model until reaching a wire for which there is no connection. At this point, the symbol for the wire and its value are printed on the transcript.

Hence, the conclusion about interfacing the view to the model for building a simulated circuit is that the view simply is an interface to the user, passing information to the model. There are a few complexities for determining the locations of objects to store and their relationships to each other. When examining the code in Table 13.4, note how the locations of the icons, wires, and the view itself are determined by sending *circuitView controller sensor mousePoint* to obtain the location of the mouse on the screen. The point to be made here is that by sending *controller* to *circuitView* one gets the controller for *circuitView* and then retrieves the sensor for that controller. This method enables accessing the location of the cursor with respect to the view specified.

13.9 Operating the Simulator

Operation of the simulator is as follows. The user first opens a new circuitView (e.g., by using the message quoted in the open message to **VCircuitSimView**) and presses one of the gate creation buttons. The icon representing that gate will appear and follow the movement of the cursor. However, the cursor is blanked so that the icon will appear to be dragged along as the user moves the mouse. Once a mouse button is clicked, the icon is deposited on the view's graphics context and a menu appears that allows the user to select in sequence the names of the wires that connect to the inputs and output of the gate placed (see the example in Figure 13.8). Each gate can be placed this way and each wire can be drawn on the view's graphicsContext by selecting "Add Wire" and single-clicking the mouse for drawing the wire until a double click occurs. After the wire is placed, the user is asked for the name of the wire from the menu of wires, in just the same way as asked for wire names when the gate was placed.

To operate the system, the user can press one of the signal-setting buttons which will place the indicated value on input wires A and B. Note the contrived nature of this example which uses a designated wire name. It would,

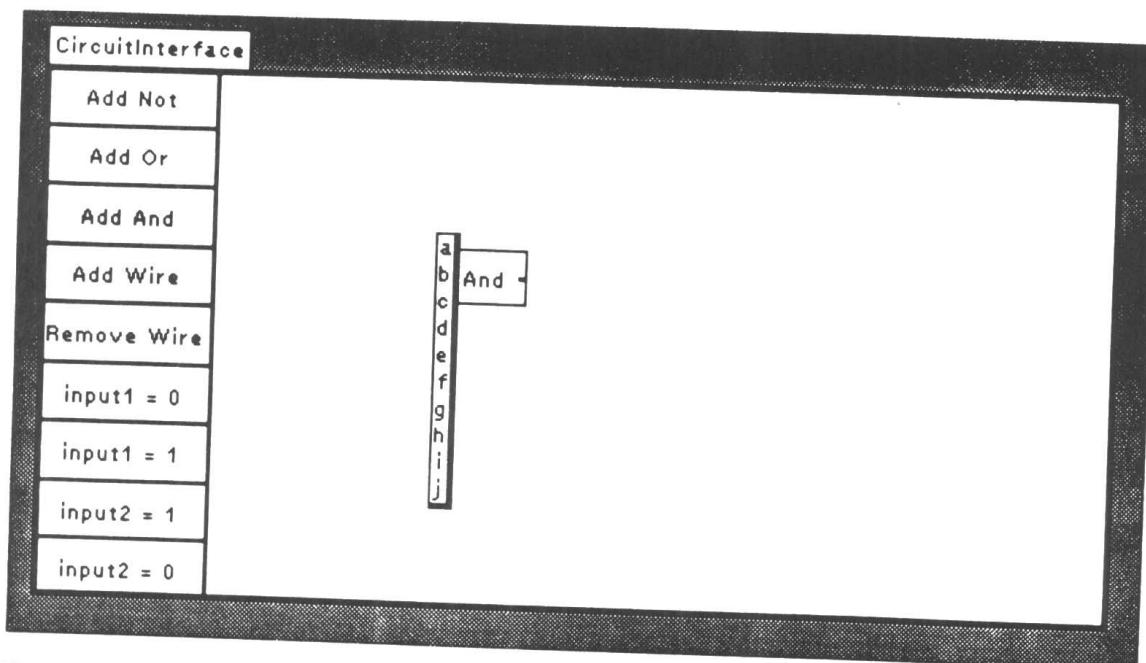


Figure 13.8 Selecting the Wires for a Gate.

Table 13.3 Classes and Protocols of the Digital Circuit Simulator

VCircuitSim View	
display	"redraw the view components"
displayOn:	
wire manipulation	
addWire	"store wire in the model"
checkForDoubleClick	"two clicks in same location"
drawWire	"draw the wire"
getWireFrom: aModel	"retrieve wire from the model"
getWirePositionInWireArray	"wire location in array"
showCursor	"display the cursor"
straightenLines	"ensure vertical / horizontal lines"
button control	
moveAnd	"add an And gate"
moveOr	"add an Or gate"
moveNot	"add a Not gate"
set the inputs	"four different values for input"
instance creation	
open	"open the application"
VCircuitSim	
access	
gates	"return the gates"
wires	"return the wires"
init	"initialize the gates and the wire"

of course, be more general to ask the user which wire to place the signal on. Once the input signal is placed, the model propagates values throughout the circuit; in each constraint, the `BlackImage` and `WhiteImage` images are displayed inside each icon in which a signal is changed to "1." Given the delay included in the constraint block in each icon, the view appears as a series of black dots rippling through the circuit. Furthermore, the output signals are printed in the system transcript in almost the same way as in Chapter 3.

There are many other configurations that could be employed to demonstrate signal propagation. The technique used here is simply illustrative and the reader is encouraged to experiment with other representational methodologies.

13.10 The Digital Circuit Simulator Code

Table 13.3 displays the classes and the protocols used in the digital circuit simulator example. Table 13.4 displays the code for the digital circuit simulator example, and Table 13.5 presents the changed implementations of the

Table 13.4 Digital Circuit Simulator Code

```
Object variableSubclass: #VCircuitSim
instanceVariableNames: 'gates wires'
classVariableNames:""
poolDictionaries:""
category: 'Circuit View example'
```

VCircuitSim holds the model of the example circuit simulation. Specifically, instance variables are specified for holding collections of wires and gates.

VCircuitSim methodsFor: access

gates
"returns the collection of gates"

^ gates

wires
"returns the collection of wires"

^ wires

VCircuitSim methodsFor: instance creation

Init
| testNameArray |
gates := OrderedCollection new. "a collection of gates"
wires := OrderedCollection new. "a collection of wires"
testNameArray := #(A B C D E F G H I J). "for testing, provide for 10 names for Wires"
1 to: 10 do: [:i|wires add: (Wire new init name: (testNameArray at: i))]. "create the 10 wires"

View subclass: #CircuitSimView
InstanceVariableNames: 'circuitView firstCursorPosition secondCursorPosition
circuitModel'
classVariableNames:""
poolDictionaries:""
category: 'Circuit View example'

This is the view class for the example circuit simulation.

VCircuitSimView methodsFor: display

displayOn: aGraphicsContext

"This method displays the wires and the gates on the screen"

```
| wires |
circuitModel gates isEmpty ifFalse: [circuitModel gates do:
[:eachGate |
"show all the gates on the circuitView"
eachGate storeGraphicsContext: circuitView graphicsContext.
"save the graphicsContext"
eachGate getIcon displayOn: circuitView graphicsContext at: eachGate getLocation].
wires := circuitModel wires.
```

Table 13.4 *Continued*

```
wires isEmpty ifFalse: ["get the wires from the model"
wires do:
[:eachWire|
"now display each wire"
|wirePoints firstPoint secondPoint|
wirePoints := eachWire getWirePoints copy.
wirePoints isNil ifTrue:[^ nil].
wirePoints isEmpty ifFalse: [firstPoint := wirePoints removeFirst].
[wirePoints isEmpty]
whileFalse:
['for all the points in each wire draw a line from point to point'
secondPoint := wirePoints removeFirst.
aGraphicsContext displayLineFrom: firstPoint to: secondPoint.
firstPoint := secondPoint]]]
```

VCircuitSimView MethodsFor: wire manipulation**checkForDoubleClick**

"If two successive cursor positions are the same return true"

```
firstCursorPosition = secondCursorPosition
ifTrue:[^ true]
ifFalse: [^ false]
```

drawWire

"This method draws a line from one point to another"

```
firstCursorPosition = nil
ifFalse:
[self straightenLines. "make sure the line is either horizontal or vertical"
circuitView graphicsContext displayLineFrom: firstCursorPosition to:
secondCursorPosition]
```

getWireFrom: aModel

"This example provides for only 10 different wires"

```
^(PopUpMenu labels: 'a\b\c\d\e\f\g\h\i\j' withCRs values: aModel wires) startUp
```

getWirePositionInWireArray

```
^(PopUpMenu labels: 'a\b\c\d\e\f\g\h\i\j' withCRs) startUp
```

showCursor

"MyCursor is an Image, defined globally, containing a cursor. As long as a mouse button is not pressed, MyCursor follows the cursor. When pressed, the cursor position is saved"

firstCursorPosition := secondCursorPosition.

Cursor blank showWhile: [MyCursor

```
follow: [circuitView controller sensor mousePoint]
while: [circuitView controller sensor noButtonPressed]
on: circuitView graphicsContext].
```

secondCursorPosition := circuitView controller sensor waitNoButton

straightenLines

"This method makes sure that the lines are straight before drawing"

```
|pointDifference|
pointDifference := (firstCursorPosition-secondCursorPosition) abs.
```

Table 13.4 *Continued*

```

pointDifference x < 10 ifTrue: [secondCursorPosition := firstCursorPosition
    x @ secondCursorPosition y].
pointDifference y < 10 ifTrue: [secondCursorPosition := secondCursorPosition
    x @ firstCursorPosition y]

```

VCircuitSimView methodsFor: button control**addWire**

“Add a wire to the circuitView”

```

|currentWire targetWire tmp|
currentWire := OrderedCollection new.
firstCursorPosition := 0@0.
secondCursorPosition := nil.
[self checkDoubleClick]
whileFalse:
    [self showCursor; drawWire.
     currentWire add: secondCursorPosition].
targetWire := circuitModel wires at: (tmp := self getWirePositionInWireArray).
circuitModel wires at: tmp put: (targetWire storeWire: currentWire)

```

moveAnd

“Add an AndGate to the circuitView”

```

|locationOfImage gateHolder|
Cursor blank showWhile: [AndImage “show the AndImage until a mouse button is pressed”
    follow: [circuitView controller sensor mousePoint]
    while: [circuitView controller sensor noButtonPressed]
    on: circuitView graphicsContext].
AndImage displayOn: circuitView graphicsContext at: (locationOfImage := circuitView
    controller sensor mousePoint).
gateHolder := AndGate new init. “now get the wires connected to the gate”
gateHolder
    input1: (self getWireFrom: circuitModel) “get the wires sequentially”
    input2: (self getWireFrom: circuitModel) “first the two inputs”
    output: (self getWireFrom: circuitModel). “then the output”
gateHolder storeLocation: locationOfImage.
gateHolder storeGraphicsContext: circuitView graphicsContext.
circuitModel gates add: gateHolder. “store the gate in the model”

```

moveNot

“Add a notGate to the simulation”

```

|locationOfImage gateHolder|
Cursor blank showWhile: [NotImage
    follow: [circuitView controller sensor mousePoint]
    while: [circuitView controller sensor noButtonPressed]
    on: circuitView graphicsContext].
NotImage displayOn: circuitView graphicsContext at: (locationOfImage := circuitView
    controller sensor mousePoint).
gateHolder := NotGate new init.
gateHolder input: (self getWireFrom: circuitModel)
    output: (self getWireFrom: circuitModel).
gateHolder storeLocation: locationOfImage.

```

Table 13.4 *Continued*

```

gateHolder storeGraphicsContext: circuitView graphicsContext.
circuitModel gates add: gateHolder. "store the gate in the model"

moveOr
"add an OrGate to the Simulation"

|locationOfImage gateHolder|
Cursor blank showWhile: [OrImage
    follow: [circuitView controller sensor mousePoint]
    while: [circuitView controller sensor noButtonPressed]
    on: circuitView graphicsContext].
OrImage displayOn: circuitView graphicsContext at: (locationOfImage := circuitView
    controller sensor mousePoint).
gateHolder := OrGate new init.
gateHolder
    input1: (self getWireFrom: circuitModel)
    input2: (self getWireFrom: circuitModel)
    output: (self getWireFrom: circuitModel).
gateHolder storeLocation: locationOfImage.
gateHolder storeGraphicsContext: circuitView graphicsContext.
circuitModel gates add: gateHolder

removeWire
Transcript show: 'Wire removal is not implemented in this example'

setInput1To0
"This and other setInput... messages explicitly set the two input wires to a value
Used for demonstration purposes"
circuitModel gates first inputWire1 set: 0

setInput1To1
circuitModel gates first inputWire1 set: 1.

setInput2To0
circuitModel gates first inputWire2 set: 0.

setInput2To1
circuitModel gates first inputWire2 set: 1.

VCircuitSimView methodsFor: instance creation

open
"VCircuitSimView new open"

|aContainer button1 window button4 button2 button3 button5 button6 button7 button8
button9 button10|
circuitModel := VCircuitSim new init. "define the model"
window := ScheduledWindow new. "define the window"
window minimumSize: 300@200. "the minimum size of the window"
window label: 'Circuit Interface'. "and its label"
aContainer := CompositePart new. "define a container to put the parts of the view in"
circuitView := self. "circuitView is the name of the specific instance of this
simulation"

```

Table 13.4 *Continued*

```

circuitView model: circuitModel.      "view and its model is circuitModel"
aContainer add: circuitView borderedIn: (0.15@0.0 extent: 0.85@1.0).
button1 := LabeledBooleanView new. "All the buttons below are the same, each has a
                                    different label"
button1 beTrigger.                "Each button activates a method in the circuitView"
button1 controller beTriggerOnUp.
button1 beVisual: 'Add And' asComposedText.
button1 model: ((PluggableAdaptor on: self)
    getBlock: [:myself|false]
    putBlock: [:myself :value|self moveAnd] "send moveAnd to this view instance, when
                                                pressed"
    updateBlock: [:myself :value :parameter|false]).
aContainer add: button1 borderedIn: (0@0 extent: 0.15@0.1).
button2 := LabeledBooleanView new.
button2 beTrigger.
button2 controller beTriggerOnUp.
button2 beVisual: 'Add Or' asComposedText.
button2 model: ((PluggableAdaptor on: self)
    getBlock: [:myself|false]
    putBlock: [:myself :value|self moveOr]
    updateBlock: [:myself :value :parameter|false]).
aContainer add: button2 borderedIn: (0@0.1 extent: 0.15@0.1).
button3 := LabeledBooleanView new.
button3 beTrigger.
button3 controller beTriggerOnUp.
button3 beVisual: 'Add Not' asComposedText.
button3 model: ((PluggableAdaptor on: self)
    getBlock: [:myself|false]
    putBlock: [:myself :value|self moveNot]
    updateBlock: [:myself :value :parameter|false]).
aContainer add: button3 borderedIn: (0@0.2 extent: 0.15@0.1).
button4 := LabeledBooleanView new.
button4 beTrigger.
button4 controller beTriggerOnUp.
button4 beVisual: 'Add Wire' asComposedText.
button4 model: ((PluggableAdaptor on: self)
    getBlock: [:myself|false]
    putBlock: [:myself :value|self addWire]
    updateBlock: [:myself :value :parameter|false]).
aContainer add: button4 borderedIn: (0@0.3 extent: 0.15@0.1).
button5 := LabeledBooleanView new.
button5 beTrigger.
button5 controller beTriggerOnUp.
button5 beVisual: 'Remove wire' asComposedText.
button5 model: ((PluggableAdaptor on: self)
    getBlock: [:myself|false]
    putBlock: [:myself :value|false]
    updateBlock: [:myself :value :parameter|false]).
aContainer add: button5 borderedIn: (0@0.4 extent: 0.15@0.1).
button6 := LabeledBooleanView new.
button6 beTrigger.
button6 controller beTriggerOnUp.
button6 beVisual: 'Input 1 = 0' asComposedText.
button6 model: ((PluggableAdaptor on: self)
    getBlock: [:myself|false])

```

Table 13.4 *Continued*

```

putBlock[:myself:value | self setInput1To0]
updateBlock[:myself :value :parameter|false]).
aContainer add: button6 borderedIn: (0@0.5 extent: 0.15@0.1).
button7 := LabeledBooleanView new.
button7 beTrigger.
button7 controller beTriggerOnUp.
button7 beVisual: 'Input 1 = 1' asComposedText.
button7 model: ((PluggableAdaptor on: self)
    getBlock: [:myself|false]
    putBlock: [:myself :value|self setInput1To1]
    updateBlock: [:myself :value :parameter|false]).
aContainer add: button7 borderedIn: (0@0.6 extent: 0.15@0.1).
button8 := LabeledBooleanView new.
button8 beTrigger.
button8 controller beTriggerOnUp.
button8 beVisual: 'Input2 = 1' asComposedText.
button8 model: ((PluggableAdaptor on: self)
    getBlock: [:myself|false]
    putBlock: [:myself :value|self setInput2To1]
    updateBlock: [:myself :value :parameter|false]).
aContainer add: button8 borderedIn: (0@0.7 extent: 0.15@0.1).
button9 := LabeledBooleanView new.
button9 beTrigger.
button9 controller beTriggerOnUp.
button9 beVisual: 'Input2 = 0' asComposedText.
button9 model: ((PluggableAdaptor on: self)
    getBlock: [:myself|false]
    putBlock: [:myself :value|self setInput2To0]
    updateBlock: [:myself :value :parameter|false]).
aContainer add: button9 borderedIn: (0@0.8 extent: 0.15@0.1).
button10 := LabeledBooleanView new.
button10 beTrigger.
button10 controller beTriggerOnUp.
button10 beVisual: 'Refresh' asComposedText.
button10 model: ((PluggableAdaptor on: self)
    getBlock: [:myself|false]
    putBlock: [:myself :value|self displayOn: circuitView graphicsContext]
    updateBlock: [:myself :value :parameter|false]).
aContainer add: button10 borderedIn: (0@0.9 extent: 0.15@0.1).
window component: aContainer.
window open

```

VCircuitSimView class
InstanceVariableNames:"

VCircuitSimView class methodsFor: Icon creation

example

```

" 1. create an icon using a drawing package
  2. Use image fromUser to acquire image from screen
  3. Store in the System Dictionary; for example, AndImage := Image fromUser.
"
```

Table 13.5 Code for the Gates in the Digital Circuit Simulator Example**Object subclass: #Gate**

```
instanceVariableNames: 'graphicsContext icon locationOfImage'
classVariableNames: ''
poolDictionaries: ''
category: 'digital-demo'
```

Gate is an abstract superClass that contains information about the location of the icon of the gate, graphicsContext and the location of the image. It serves as the superclass of TwoInputGate and OneInputGate.

Gate methodsFor: access

```
getGraphicsContext
^graphicsContext
```

```
getIcon
^icon
```

```
getLocation
^locationOfImage
```

```
storeGraphicsContext: aGraphicsContext
graphicsContext := aGraphicsContext
```

```
storeLocation: aLocation
locationOfImage := aLocation
```

Gate subclass: #OneInputGate

```
instanceVariableNames: 'inputWire outputWires constraint delay'
classVariableNames: ''
poolDictionaries: ''
category: 'digital-demo'
```

OneInputGate is an abstract class used for all one input and one output gates.

OneInputGate methodsFor: action**propagate**

```
|constraintOutputValue|
constraintOutputValue := constraint
    value: inputWire value
    value: graphicsContext
    value: locationOfImage.
outputWires do: [:aWire|aWire set: constraintOutputValue]
```

OneInputGate methodsFor: Initialize**init**

```
delay := 0.
outputWires := OrderedCollection new.
```

input: aWire1 output: aWire2

```
inputWire := aWire1.
outputWires add: aWire2.
inputWire outputConnection: self.
aWire2 inputConnection: self.
```

Table 13.5 *Continued*

```
OneInputGate subclass: #NotGate
  InstanceVariableNames:""
  classVariableNames:""
  poolDictionaries:""
  category: 'digital-demo'
```

NotGate inverts the signal entering the gate.

NotGate methodsFor: initialize

init

```
super init.
constraint := [:input :aGraphicsContext :location | input = 1
  ifTrue: [0]
  ifFalse:
    [BlackImage displayOn: aGraphicsContext at: location + (5@10).
     (Delay forSeconds: 1) wait.
     WhiteImage displayOn: aGraphicsContext at: location + (5@10).
     1]].
icon := NotImage
```

Gate subclass: #TwoInputGate

```
InstanceVariableNames: 'inputWire1 inputWire2 outputWires constraint delay'
classVariableNames:""
poolDictionaries:""
category: 'digital-demo'
```

TwoInputGate is an abstract class used for implementing all two input and one output classes.

TwoInputGate methodsFor: Initialize

Init

```
delay := 0.
outputWires := OrderedCollection new.
```

Input1: aWire1 Input2: aWire2 output: aWire3

```
inputWire1 := aWire1.
inputWire2 := aWire2.
outputWires add: aWire3.
inputWire1 outputConnection: self.
inputWire2 outputConnection: self.
aWire3 inputConnection: self.
```

TwoInputGate methodsFor: action

propagate

```
|constraintOutputValue anArray|
anArray := Array new: 4.
anArray at: 1 put: inputWire1 value.
anArray at: 2 put: inputWire2 value.
anArray at: 3 put: graphicsContext.
anArray at: 4 put: locationOfImage.
```

Table 13.5 Continued

```
constraintOutputValue := constraint valueWithArguments: anArray.
outputWires do: [:aWire|aWire set: constraintOutputValue]
```

TwoInputGate methodsFor: access

```
constraint
  ^ constraint
```

```
InputWire1
  ^ inputWire1
```

```
InputWire2
  ^ inputWire2
```

TwoInputGate subclass: #AndGate

```
instanceVariableNames: ""
classVariableNames: ""
poolDictionaries: ""
category: 'digital-demo'
```

This Class implements ‘AND’ behavior; that is, whenever both signals are present on the input, the signal is high on the output, otherwise the output is low.

AndGate methodsFor: initialize**Init**

```
    "initialize the constraint block"
```

```
super init.
```

```
constraint := [:input1 :input2 :aGraphicsContext :location|input1 = 1 & (input2 = 1)
  ifTrue: "display a dot, wait, and white it out"
    [BlackImage displayOn: aGraphicsContext at: location + (5@10).
     (Delay forSeconds: 1) wait.
     WhiteImage displayOn: aGraphicsContext at: location + (5@10).
     1]
  ifFalse: [0].
icon := AndImage
```

TwoInputGate subclass: #OrGate

```
instanceVariableNames: ""
classVariableNames: ""
poolDictionaries: ""
category: 'digital-demo'
```

The behavior of OrGate is that whenever an input signal is high, so is the output signal.

OrGate methodsFor: initialize**Init**

```
super init.
```

```
constraint := [:input1 :input2 :aGraphicsContext :location|input1 = 1 | (input2 = 1)
  ifTrue:
    [BlackImage displayOn: aGraphicsContext at: location + (5@10).
     (Delay forSeconds: 1) wait.
```

Table 13.5 *Continued*

```

        WhitImage displayOn: aGraphicsContext at: location + (5@10).
    1]
    ifFalse: [0]].
icon := OrImage

Object subclass: #Wire
instanceVariableNames: 'inputConnection outputConnections value name
pointsDescribingWire'
classVariableNames:”
poolDictionaries:”
category: 'digital-demo'

Wires are used to connect digital components.

Wire methodsFor: access

getWirePoints
^ pointsDescribingWire

InputConnection: aGate
inputConnection := aGate.

name
^ name

name: aName
name := aName

outputConnection: aGate
“multiple gates can be added to the output of a wire”

outputConnections add: aGate

set: aValue
value := aValue.
outputConnections isEmpty
ifTrue:
    [Transcript show: self name printString. “print my name”
Transcript show: self value printString. “print my value”
Transcript cr.]
ifFalse: [outputConnections do: [:eachGate|eachGate propagate]]
“propagate value to each gate I am connected to”]

storeWire: aWire
pointsDescribingWire := aWire.

value
^ value

Wire methodsFor: initialize

init
value := 0. “initialize the wire to hold a zero value”
outputConnections := OrderedCollection new. “each wire connects to multiple sites”

```

gate classes that were given in Chapters 3 and 10. The reader is encouraged to create a mental organization of the code, according to the class/protocol representation of Table 13.3, then read the code starting with the *open* method of **VCircuitSimView**, and then code and demonstrate the application.

13.11 Additions to the DCS Application

The example described in this chapter could benefit from many improvements, some of which are suggested in the exercises. For example, consider the problem of drawing the wires. The technique selected requires the user to draw wires using the mouse. The linkages between gates are made using the wires specified when placing a gate on the circuitView. Wires are selected from predefined wires created during system startup and stored in a collection. To continue to follow this model, a dynamic extension of the collection should be made to include the automatic addition of wires as needed. Another modification would be to click on a gate and select the terminal of the gate that the mouse cursor is closest to for association with a wire. As the system operates in the example, the name of the wire needs to be remembered, an obviously problematic requirement. A second but more complex technique for drawing wires could be to create wires algorithmically, requiring no user interaction. If it is known, for example, that wire A is connected to the inputs of the *and* and *or* gates of the half-adder, then a routing algorithm should be able to determine where to draw the line showing the connection. Various algorithms are easy to understand but harder to implement. For example, between any two points, a rule might be “draw a line such that it is never closer to any gate icon than x pixels, it is always vertical or horizontal, maintains equidistance between any two gates (except when other lines are there), and takes the shortest distance possible.” One can image a series of rules of this sort that could create wires from point to point.

A simple addition to the simulator would be the changing of the gate images. Simply replacing the icons with more realistic icons would improve the look of the simulation. Furthermore, the ability to rotate icons by 90 degrees would help, as would the capability of snapping wires to the proper location. On this latter point, each icon could have a set of connection points defined; in the *and* gate case, there would be three, two inputs and one output. For each of these connection points, an area around the point could be defined, for example 5 or 10 pixels. When placing a wire, if the point selected was within the 5- or 10-pixel area, the wire could be attached to that point; otherwise the connection would be disallowed.

Hierarchical viewing of circuit elements, as discussed briefly in Chapter 10 (see Figure 10.6), would be another useful addition to the simulation. For example, when a user moves the cursor inside an icon and brings up a menu

with a mouse click, one option could be “expand.” Selection of this option would zoom a window on top of the current circuit which would show the contents of the circuit represented by the icon. Similarly, a menu could be available for whole circuits which would collapse the circuit to an icon and show it as part of a larger circuit. This type of whole-part expansion and collapsing is ideal for viewing a circuit in more or less detail.

13.12 Summary

This chapter has presented a complete example of how to create a rapid prototype of a digital circuit simulation system. Building on a model that uses constraint propagation to simulate the way a digital circuit operates, the primary addition in this chapter was the demonstration of how view components can be easily added to display visual models. The conclusion is that it is straightforward to implement a view that permits the observation of the working of a model. Given the components that have been described in some detail in earlier parts of the text, there are a wide variety of standard components such as pop-up menus, lists, buttons, and so on that make the job of rapid prototyping easy. Rather than create new kinds of representations to show objects visually, it is easier to simply use what is available.

A key concept in the rapid prototyping notion is the idea of creating a framework for the problem and then incrementally modifying the working prototype. As has been discussed above, there are many alternative ways of creating a digital circuit simulator with many attendant issues to resolve such as how to place components and place wires. These issues are hard to resolve *a priori*. It is indeed useful to be able to create a prototype rapidly and see how the concepts work and how the system should evolve into something perhaps more suitable. This capacity for continual evolution of a system is antithetical to top-down design, in which complete specifications for software are written prior to actually coding a system. However antithetical it may be, the fact remains that rapid prototyping is a boon to creativity, permits trying out ideas quickly and easily, and, in the long run, likely results in better and more thoroughly considered software products.

References

- Jiang, Zhihe, and John Bourne (1991). A Visual Programming Environment for Building Smalltalk-80 Views. *Journal of Object-Oriented Programming*, May.
- Morris, William (Ed.) (1981). *The American Heritage Dictionary of the English Language*. Houghton Mifflin, Boston, MA.

van der Meulen, Pieter S. (1989). Development of an Interactive Simulator in Smalltalk. *Journal of Object-Oriented Programming*, Vol. 1, No. 5, January/February, 28–51.

Exercises

- 13.1 Four maxims for rapid prototyping are presented: the analysis phase, the design phase, the implementation phase, and the incremental refinement phase. Describe the steps associated with each phase.
- 13.2 Implement the digital circuit simulator, as presented in this chapter. Discuss any problems that you had while creating the implementation.
- 13.3 As mentioned in this chapter, a wire routing algorithm might work well as an alternative to the user-directed wire placement methodology described. Specify a few wire routing rules and indicate how the algorithms that you propose might be implemented.
- 13.4 In the example given in this chapter, you may have noticed that the “remove wire” switch does not actually remove a wire. Describe how you would implement (1) the removal of a single wire and (2) all the wires.
- 13.5 Note that there is no technique in the example given in this chapter for moving an icon once it is placed on the circuitView. One method is to erase the image at the specified location (perhaps by restoring the background that was behind the image) and then to have the icon follow the cursor. One can imagine defining the rectangular coordinates for each icon placed on the view’s graphics context and storing this information in a dictionary. Then, whenever the cursor is within a rectangle and the mouse button is pressed, the designated icon would be erased and reattached to the cursor. See if you can implement this solution or a similar solution.
- 13.6 Attaching wires to a gate can be accomplished using the method described previously to locate the pins of the gate around certain points on the image. Discuss how to do this operation and propose a way of implementing wire attachment in this fashion.
- 13.7 In the current implementation, the value of the output wire is displayed on the system transcript. Add an icon, such as a light bulb, which will change color (e.g., from white to black) as the signal to which the icon is attached changes from 0 to 1.

Engineering Simulation Methodologies and Simulation Frameworks

14.1 Introduction

A framework is defined as “A structure for supporting, defining, or enclosing something.” (Morris, 1981) or alternatively as “a frame or structure composed of parts fitted and joined together” (Webster’s, 1989). For software, frameworks are shells, environments, or simply programs that support the integration of various pieces of software into a package that assists in problem solving or system operation. Widely used software packages, such as Microsoft Windows (1987), which organize the capabilities of an operating system for improved communication with the user, can be thought of as frameworks. More commonly, the term *framework* is employed to designate a set of software that is integrated to support a special type of application such as design, viz., design frameworks, which typically contain some type of simulation capabilities.

Simulation, as discussed at several points during earlier chapters, is the mimicking of the way that an actual physical system works, that is, the behavioral manifestations of a functional specification. Engineering simulation deals with the simulation of designed artifacts—those things that have been designed for use in our world. Hence, an engineering simulation provides the capability of replicating how something would work if it were fabricated. Indeed, this is the concept behind computer-aided engineering (CAE). Rather than create physical working prototypes of a system being designed, testing, and refining the prototypes, it is more parsimonious to simulate a system, pinpoint and correct design flaws, and fully test a system prior to ever creating a hardware implementation of a design. As an example, there are a number of design frameworks for computer-aided engineering in electronics. Frameworks built by such companies as Daisy System Corporation, Mentor Graphics Corpo-

ration, and Valid Logic (Ohr, 1990) are widely used. These frameworks are bundled packages of software that integrate software for schematic entry, netlist generation (i.e., the structure of circuit connectivity), simulation, printed circuit board layout, ASIC (application-specific integrated circuits) layout, and test programs. Such systems give an engineer or engineering team the complete capability of creating a design for an electronic circuit, simulating the circuit, producing the specifications for fabricating the circuit, and rigorously testing the product. Such frameworks are large, complex, and have normally been quite expensive.

At a simpler level, one can attempt to understand the concept of frameworks for organizing the simulation aspects of design. In much the same way that one can integrate generic software packages for solving electronic design problems, one can also consider general frameworks, which contain no domain knowledge, that permit the building of simulations for use in different engineering domains. In this chapter, the different types of simulation that are currently used will be considered and related to object-oriented thinking and implementation strategies. Examples of two kinds of existing simulation frameworks will be given, as well as an example in manufacturing.

In contrast to the previous chapter, this chapter contains a limited amount of code. Instead, concepts are discussed and pointers to the code are given. Because the text has a strong Smalltalk-80 orientation for implementation, Section 14.5 presents a discussion of a simulation framework in ST-80 code that can be secured from the ParcPlace Systems bulletin board.

14.2 The Kinds of Simulation

Simulation is used for many activities in engineering. Several examples are given in Table 14.1 that show a range of simulation activities across a broad spectrum of application areas. For example, in manufacturing, simulation is used for constructing a model of a manufacturing process. Once the model suitably mimics a real or planned system, various parameters can be varied to determine the influence on behavior of the simulation. An often-used example [see, e.g., Pinson and Wiener (1988) and Goldberg and Robson (1983)] simulates the arrival of customers at a bank. The purpose of the simulation is to assist in designing the characteristics of the bank, for example, how many tellers should be hired, how many customer queues should be allocated, and so on, for different probabilistic distributions of customers entering the bank at different times.

In almost any conceivable area, simulation methodologies can be used to predict behaviors for designed artifacts. The topics listed in Table 14.1 are by no means exhaustive—they are simply a few examples to show the range of

Table 14.1 Example Uses for Simulation

Area	Examples
Manufacturing	Warehouse modeling, printed circuit board manufacturing, rolling mill modeling, chemical plant models
Service sector	Goods distribution systems, mail delivery, medical staffing, planning, food service modeling, banking
Computers	Information packet distribution, memory modeling, communications network models
Civil engineering	Traffic control, hydrology, structural models
Military	Battle simulation, command, control
Aerospace	Flight simulation, airplane aerodynamics

possibilities. Of course, the difficulty with simulation is knowing how close to reality the simulation will be. Obviously, it would be disastrous to simulate a bridge and not be able to believe that the simulated results that dictate how the bridge would be built are close to being accurate. The fidelity of the simulation (i.e., how close the simulation gives answers close to reality) is a critical issue in determining what type of simulation to use.

There are many different styles and methodologies for producing simulation systems. Perhaps the most easily recognizable groups are (1) the traditional engineering simulation methodologies [see, e.g., Bratley, Fox, and Schrage (1987)] which predominantly employ discrete-event simulation and (2) qualitative simulation methodologies which have gained recent prominence due to research in artificial intelligence [see, e.g., Bobrow (1985)]. The differences, advantages, and disadvantages of these two major subdivisions in simulation methodology will be explained in this section.

Four terms relating to simulation need to be identified: *discrete*, *continuous*, *qualitative*, and *quantitative*. Discrete and continuous refer to time: (1) discrete time, such as fixed increments of time or tagged instances of events that occur at a particular time, and (2) continuous time, in which time varies without interruption. Quantitative and qualitative refer to what variables represent. Quantitative refers to the capability of explicitly associating a variable with a specific quantity, for example, $x = 5$, or an enumeration of specific values, for example, $x = \{5, 6, 7, 8, 9, 10\}$. In contrast, qualitative refers to associating a quality with a variable, for example: (1) the variable is not changing, (2) the variable is increasing, or (3) the variable is decreasing, with no specific numeri-

Table 14.2 Variable, Time, and Constraint Attributes of Qualitative and Quantitative Models

Quantitative Models	Variables: Time: Constraints:	Numeric Discrete or continuous Linear or nonlinear
Qualitative Models	Variables: Time: Constraints:	Symbolic, numerical landmarks Symbolic Symbolic, functional relationships

cal association. Often symbols such as $\{0, +, -\}$ are associated with these latter qualities to symbolically represent change.

Table 14.2 summarizes the attributes of quantitative and qualitative models with respect to variable, time, and constraint dimensions. Quantitative models employ mainly numeric variables, whereas qualitative models employ a combination of symbolic and numeric variables. In qualitative modeling, symbolic values represented as symbols or strings can be coupled to numeric values used as critical numeric “landmarks” for constraining ranges or specific values. Time is represented in quantitative models in either a discrete or continuous fashion. Discrete representations often do not use formal mathematics, whereas continuous representations are typically cast into a set of mathematical equations, algebraic or differential. For qualitative models, time can be represented symbolically with no numerical quantities involved. Finally, for the constraint dimension in modeling, various constraints are activated as information is processed by a simulation. In quantitative models, constraints (i.e., the expressible relationships between entities) are normally linear equations; yet, nonlinear relationships could serve equally well. For qualitative simulation, constraints become symbolic functional relationships. For example, a symbolic functional relationship might be expressed verbally as: “if there are two inputs, and both show an increase, then propagate an increase to the output.” In object-oriented representations, constraints could be simply implemented in methods operating on symbolic values stored in object instance variables, in much the same way as quantitative relationships are implemented in the digital circuit constraints given in the examples in Chapters 3 and 10.

Alternatives for Representing Time

There are a number of choices for representing time. One familiar technique is the use of differential equations that permit mathematical representation of system behavior related directly to continuous time. Although equation-based closed-form mathematical descriptions are appealing and com-

plete, discrete representations are more attractive for computer simulation. Hence, discrete time (i.e., representing time as a series of exact discrete steps) has become the dominant means for simulating time-oriented systems. Most discrete-time representations use either a fixed-interval time representation or event-related timing. In the first case, fixed-interval-time models simply move a simulation clock from one time to the next; if an event is associated with a certain time, then it is activated. As a concrete example, imagine that events should occur in a simulation at times 5, 10, and 20 seconds after the start of the simulation. An internal clock could be set to tick and, after the requisite elapsed times, events (perhaps displaying an activity on the screen) would occur. Of course, time could be either real time or time that is either faster or slower than real time. In the second case, discrete-event simulation, events are tagged with a time at which they should occur and placed on a queue ordered by time. After starting the simulation, events are removed one by one from the head of the queue and activated. The activation of each new event may cause other events to be stored on the queue, each tagged with a specific time relative to the current time. The event activation process may continue indefinitely if the queue never becomes empty or will terminate when events on the queue are depleted.

Both fixed-interval-time simulation and event-driven simulation are widely used. Contrasting the two methods does not reveal clear superiority for either method, although event-driven simulation has some inherent advantages. For example, representation of events that are temporally distant (i.e., there is a long time between events) is simpler with event-driven simulation because time tags are the only information that is required to track time. Fixed-interval simulation also contains events that are time tagged; however, the requirement of permitting a clock to "tick" from one event to the next means that a suitable time compression must be found to permit observation of temporally distant events. Hence, watching a system simulated by fixed-interval simulation becomes difficult if there are events that are separated by both long and short times.

Chapter 13 described a form of fixed-interval-time simulation, in which delays were added to constraints. In the example given in that chapter, signals propagated through constraints, waiting an appropriate amount of time at each node. No system clock was involved, however; yet, the effect was precisely the same as if each constraint in each of the nodes had been triggered by a clock.

Table 14.3 shows several examples of continuous and discrete simulation systems. Examples listed under discrete-event simulations are often called "counter" simulations because of their relationship to the model of a customer approaching a counter for service, obtaining service, and leaving. All of these actions are discrete events and can be readily modeled using discrete-event

Table 14.3 Examples of Discrete and Continuous Simulations

Discrete-Event Simulations	
Example	Components
Traffic control	Events: lights changing Entities: vehicles
Commodity transport	Events: commodity transactions Entities: trucks and trains
Aircraft control	Events: aircraft queue handling, positions Entities: aircraft
Operating system	Events: switching processes Entities: users
Bank simulation	Events: transactions Entities: customers, bank tellers

Continuous Simulations	
Example Simulation	Implementation
Simple mechanical and electrical simulations (e.g., flush toilet or analog circuit)	Constraints using equations
Continuous flow (e.g., through pipes)	Constraints / equations
Stresses on mechanical systems or structures	Differential equations
Analog computers	Physical analog components
Models of human expertise	Rule-based systems

simulation. The first three continuous simulations shown in Table 14.3 are equation-based examples. Also shown are two other continuous simulations worth mentioning. First, analog computers physically implement continuous simulation. Analog computers are built from analog electronics hardware; hence, when changes are made in a simulation implemented in an analog computer, the results are immediately observable. Finally, rule systems can conceptually be thought of as organizing models of human expertise; when an observation is made, the inference should be made immediately. Thus, rule systems may be thought of as being continuous.

As a final note, the distinctions between discrete and continuous are often a matter of granularity. For example, as a homely example, putting sugar on cereal is a continuous type of action. Yet, when viewed very closely, discrete individual grains of sugar are being distributed at distinct points in time. A continuous model of the more global view might be implemented with an equation that relates the amount of sugar distributed to the time, and a discrete

model of the microscopic view would link distinct times to the distribution of each grain of sugar.

Alternatives for Representing Quantities and Qualities in Simulation Systems

The word *quantitative* is defined as “Expressed or capable of expression as a quantity,” and *qualitative* is defined as “Of, pertaining to, or concerning quality or qualities” (Morris, 1981). Quantities are describable in terms of variables that hold numeric values, whereas qualities are describable in symbolic terms. Traditional engineering simulation has represented quantities using several types of simulation methods. Table 14.4 lists the two leading classes of simulation methodologies and several simulation languages that have been useful for quantitative simulation. The most prevalent type of simulation is discrete-event simulation; listed in the table are five major types of discrete-event simulation: event scheduling, activity scanning, the three-phase approach, process interaction, and transaction flow (Pidd, 1988; Derrick et al., 1989). Event scheduling deals with the description of a system in terms of events that may occur and event routines that run when the state of a system changes. Activity scanning examines the interactions between entities in a system. The three-phase approach modifies the event scheduling model by segmenting system activities into three phases: a time scan, bookkeeping, and execution of condition-related events. The process interaction method tries to identify processes in a simulation as the main entities of interest, and the transaction flow method models time and state relationships in terms of the arrangement and interaction of transactions. There are a number of other discrete-event simulation methods [Derrick et al. (1989) identifies six more] that are somewhat less popular.

Table 14.4 Examples of Types of Quantitative Simulation Systems and Languages

Discrete-event simulation	Event scheduling Activity scanning Three-phase approach Process interaction Transaction flow
Classical modeling	Optimization theory Constraint programming Bond graphs
Simulation languages	GPSS SIMSCRIPT GASP SimKit

Classical modeling techniques, as displayed in Table 14.4, such as optimization theory, constraint programming, and bond graphs are other tools frequently used in simulation. Constraint programming issues have already been examined in some detail in Chapters 10 and 13. Many simulation languages produced during the last few decades permit a user to construct simulations by using the language syntax. The oldest simulation language is probably GPSS (General-Purpose Simulation System) which was introduced by IBM some three decades ago. GPSS is an ad hoc language that originally introduced the transaction model of simulation. More recent versions of GPSS have added graphical construction and animation (Cox, 1985). Simula (Birtwistle et al., 1973) is an early object-oriented simulation language. Simscript (Markowitz et al., 1963) utilizes English-like statements to permit modeling, and GASP (Pritsker and Kiviat, 1969) is a collection of Fortran subroutines for modeling. SimKit (Drummond and Stelzner, 1989) is an object-oriented modeling system that is part of the KEE (Knowledge Engineering Environment) package produced by Intelllicorp.

The lesson to be learned from these few paragraphs about traditional simulation is that simulation is, by modern computer standards, an old field. Starting with the development of systems based on quantitative methods implemented with Fortran, the field has moved toward refining the initial concepts over several decades. Recently, however, traditional research in simulation has become aware of work on qualitative methods that has arisen in the field of artificial intelligence, and vice versa.

Qualitative simulation has evolved almost as a separate entity with little input from traditional simulation research. Starting from theories proposed by Hayes (1979), qualitative simulation has grown to be a viable interest area within the field of artificial intelligence. The basic idea is that one can replace detailed mathematical descriptions of quantitative relationships with symbolic representations that describe qualitatively interrelationships between entities. For the most part, work on qualitative simulation has been on theoretical constructs; however, as shown in Table 14.5, various investigators have applied their theories to practical problems. Consider the examples given in Table 14.5. Application examples range from the modeling of valves to topics in electrical and electronic circuits to even a description of the behavior of

Table 14.5 Examples in Qualitative Simulation Systems

Valves	de Kleer and Brown (1985)
Analog circuits	de Kleer (1985)
MOS circuits	Williams (1985)
Electrical heating	Farquhar and Kuipers (1987)
Projectiles	Kuipers (1986)

projectiles. The commonality that can be observed in these examples is that the domains are typically ones that can be usefully modeled using differential or algebraic equations. Hence, one might question the utility for qualitative modeling. Several reasons have been advanced for employing qualitative modeling. Fishwick (1989) observes that there are indeed few cases in which qualitative simulation might make sense to use in preference to quantitative simulation. These cases are: (1) modeling human cognition, (2) cases in which quantitative models require excessive computation, and (3) tutoring systems. In the first case, it makes sense to utilize symbolic representations because there are few quantitative methods in cognitive science. The second and third cases are linked; qualitative models can produce a sense of how a system being simulated works, even though precision is lacking. Visualization of how a system operates is of paramount importance for tutoring systems because it is essential to produce simulations that rapidly give observable results to a student using a simulation. It would be totally unacceptable, for example, to require a student to wait long periods of time to observe a change on the computer screen. Of course, if the mathematical relationships between the entities utilized to model a system are sufficiently simple and/or computationally efficient, then quantitative models would be perfectly adequate for implementing models used in tutoring systems.

The thrust of these discussions is the view that qualitative modeling and quantitative modeling attempt to achieve the same goal, that is, providing a behavioral representation of something in the real world. Quantitative discrete methods have been used for this purpose for a long time, and the study of qualitative symbolic methods has only commenced during the last decade. Presumably, the trade-offs between the methodologies and the utility of the capabilities of the different paradigms will gradually become better understood.

The Relationship between Traditional and AI-Based Simulation

Only during the last few years has there been any recognition that it would be useful to merge concepts in artificial intelligence with simulation methodologies. The central concept driving a merger of these formerly separate fields is that the addition of some form of intelligence will make simulation systems much more powerful. Capable of making decisions and interpreting the results of manipulation of simulation parameters, AI-based simulations will likely become widely used during the upcoming decade. Table 14.6 shows some of the efforts in this direction that have been made during the last years. CASM (computer-aided simulation modeling) (Balmer and Paul, 1986) applied expert systems technology to simulation for the purposes of problem formulation and simulation model development. SimKit (Drummond and Stelzner, 1989) is a set of software tools that permit building domain-independent discrete-event simu-

Table 14.6 Artificial Intelligence Systems and Simulation

System Name	Capabilities
CASM	Expert system for problem formulation and simulation model development
SimKit	Modeling kit for KEE
KBSim	Reasoning in simulation
Molgen	Rule-driven simulator for genetics

lation systems. Object-oriented in the sense that KEE (Kehler and Clemenson, 1984) frame structures are used in SimKit, this package permits graphical building of simulation systems.

KBSim (knowledge-based simulation) (Rothenberg, 1989) is a system that assists in reasoning in simulation, including inference and explanation. There are a variety of examples of AI-based simulation. One of the most impressive is the Molgen project (Meyers and Friedland, 1984) which uses a knowledge base of rules about regulatory genetics and facts to attempt to make deductions about specific growth behaviors in genetics.

The common theme in the use of AI methodologies in simulation is to add some level of intelligence to simulation. In work accomplished until the early 1990s, most activity has consisted of encoding human problem-solving knowledge in the form of rules that can be used to assist the user in (1) planning and putting together a simulation and (2) interpreting the results of a simulation.

The Importance of Object-Oriented Methodologies in Simulation

The review of the state of the art of simulation methodologies given in this section permits the making of several assertions about why object-oriented methodologies are useful in simulation. First, most simulation is concerned with models of real-world entities and their interactions. The structures and behaviors of these objects are readily modeled by object-based techniques, as has been shown throughout the text. Second, objects provide a parsimonious representation for real-world objects by permitting specialization and inheritance. Thus, libraries of general simulation objects could exist which would facilitate building of new simulations simply by specializing existing objects in the library. For example, if a pipe existed in a library of simulation objects, a rusty pipe could be used in a simulation simply by specializing the existing class for pipe. This type of capability does not normally exist in traditional simulation packages.

Table 14.7 Examples of ST-80 Simulation Applications

Name	Application
INSIST	VLSI design
SIMFLEX	Flexible manufacturing
GREGG	Military graves registration
SIMBIOS	Continuous models

Although the use of object-oriented methods in simulation is a relatively new field, there are already a number of examples that have been created during the recent past. Table 14.7 summarizes some of these systems (Haden, 1990). The four systems shown in the table are Smalltalk-based (INSIST, van der Meulen, 1989; SIMFLEX, Vujosevic, Antao, and Bourne, 1990; GREGG, Helfman, Ralston, and Suckling, 1987; SIMBIOS, Guasch and Luker, 1990). Haden (1990) in a review of the literature found that approximately half of the 25 AI-based simulation systems reviewed were written in Smalltalk. The remaining systems were other object-oriented languages such as C++, Flavors, Ada, and Simula.

The next sections of this chapter will move away from the relatively abstract concepts presented previously and give some concrete examples of how to create simulation systems using object-oriented techniques. Also, more attention will be given to Smalltalk-80 implementations.

14.3 Multiple Independent Processes

Smalltalk-80 provides the capability for managing multiple independent processes with its virtual machine and virtual image. In essence, the virtual machine and virtual image permit the spawning of multiple processes as if one were able to use multiple physical machines to execute multiple processes. However, the Smalltalk-80 programmer needs to know virtually nothing about the implementation mechanism of the virtual machine and virtual image in order to make use of their capabilities. For simulation, the capability of having multiple independent processes running contemporaneously is significant. For example, consider the implementation of the counter simulations discussed earlier in this chapter in which there are two types of actors in a simulation: customers and servers. In these types of simulations, each type of actor can be made into an independent process and can operate in the simulation much like they do in the real world, that is, waiting on each other for service. The object-oriented communications paradigm works well in this type of simulation

Table 14.8 Some Features of Multiple Independent Processes in Smalltalk-80

Feature	Description
Process support	Supports multiple processes that can have priorities and are managed by the virtual image
Scheduling	Processes may be scheduled, activated, and terminated
Semaphores	Support synchronization of processes
Shared queues	Class SharedQueue provides safe communication between objects
Delays	Class Delay provides real-time delays

in which customers and servers send each other messages to communicate and start transactions.

Table 14.8 shows some of the important features of multiple independent processes that are implemented in ST-80. Processes are simply sequences of actions that are performed to carry out tasks. The ST-80 system permits multiple tasks to run contemporaneously (i.e., at about the same time). Although simultaneous parallelism of tasks is the objective, each process receives a slice of time to run. Also, some tasks can be given a higher priority than other tasks, as specified by the user. A simple experiment to test how multiple processes run in ST-80 can be conducted by the casual user by constructing the following code in a workspace and executing the code.

```
X := 0.                                     "a global variable"
MyProcess := [[true] whileTrue:                  "MyProcess is global"
    [|context|
        context := ScheduledControllers
        activeController view graphicsContext.
        context displayString:
            (X printString) at: 100@150.
        context displayImage: EraseIt1 at: 50@100.
        X := X + 1.
        Processor yield]
    ] forkAt: 4.                                "show the number"
                                                "erase it"
                                                "increment the number"
                                                "make sure that higher"
                                                "priority processes"
                                                "are seen"
                                                "start this independent"
                                                "process at level 4"
```

¹EraseIt is a global variable that holds a blank image.

This code should display a continuously increasing number on the workspace. If you move the cursor, you should still be able to type code in the workspace. Try typing

MyProcess terminate.

and executing this code. The continuously increasing numbers should cease to increment. In this test code fragment, the following points are relevant. First, one can send a message *fork* or *forkAt: aPriority* to a block of code and it will run independently. The code in this test block runs again and again, never terminating on its own, because *[true] whileTrue: [...]* is always true. Fortunately, the process is identified with a name: “MyProcess,” which can be sent the message *terminate* to cause the process to stop running. Other messages such as *suspend* and *resume* are also available. Priorities range from 8 (high) to 2 (low). The normal user interaction priority is 4. By including in the block, the code “Processor yield,” we ensure that other processes with the same priority can gain control. In the preceding example, the number contained in the global variable “X” is displayed on the workspace. To continue to be able to see each number in turn, this example erases a small area on the workspace by displaying a blank image over the area on which the numbers are displayed. Note that the “EraseIt” image can be captured easily by simply assigning *EraseIt* to *Image fromUser* and grabbing a blank area of the screen.

An interesting experiment that can be attempted is to duplicate the code given previously with different process names and with different locations to display numbers on the workspace. By doing this multiple times and forking multiple processes, it can be easily observed that adding more and more processes slows down the time given to each process. By declaring “X” as a global variable in the preceding example and in each subsequent process that is forked, multiple processes referring to “X” will each increment this same variable and display it.

The remaining features of multiple independent processes in ST-80 include classes for scheduling, semaphores (essentially flags—one-bit pieces of information) which permit synchronization of processes, shared queues, and a class called **Delay** which has already been used in Chapter 13. The class **Delay** will be used in the examples that follow to implement delays in a simulation.

14.4 Probability Distributions

A strong capability that is available for ST-80 is a set of classes for implementing probability distributions. In simulating randomly interacting multiple processes, it is essential to be able to characterize the behavior of the

Table 14.9 Probability Distributions Implemented in ST-80

Class Name	Possible Use
Bernoulli	Determining whether an event occurs
Geometric	Determining how long until the next event
Poisson	Determining how many events occur in an interval
Exponential	For Poisson-distributed events, determining the time until the next event
Uniform	Picking equally likely events
Gamma	Determining how long until the N th event occurs
Normal	Describing a process with a central tendency (mean) and a standard deviation

process statistically. The classes, contained in a fileIn `\utils\stats.st`,² provide a rich set of distributions which can be employed by the user with little additional thought. Table 14.9 lists the types of distributions that are available in the ST-80 fileIn `\utils\stats.st`. A detailed discussion of these classes are found in Goldberg and Robson (1983).

An example of the use of probability distributions in simulation is straightforward. Imagine a counter-style simulation, for example, the bank simulation discussed previously. Customers arrive randomly according to a mean and a standard deviation. As indicated in Table 14.9, the Normal (also known as Gaussian) distribution would be suitable for modeling customer arrival. A specific example might be creating a new instance of a **Normal** class, for example:

```
aNormalDistribution := Normal mean: 10 deviation: 1.
```

The meaning of this distribution could be that, on the average, a customer arrives every 10 minutes with a deviation of plus or minus 1 minute. Practically, the use of the distribution instance would be to sample the distribution when a new number is needed, for example, to record the arrival of new customers appearing at the entrance to the bank on the screen. To secure a number, the instance of the distribution is sent the message *next* which, when repeatedly sent, results in this example, in a sequence of numbers such as: 10.34, 9.45, 8.56, 11.09, 12.34, 10.52, 10.98

²Found on the ParcBench Bulletin Board of ParcPlace Systems.

14.5 A Simple Framework for Discrete-Event Simulation

To explain discrete-event simulation, the example of the digital circuit simulator that has been used throughout this text (Chapters 3, 10, and 13) will be recast into a discrete-event-simulation framework. The intention is to explain how discrete-event simulation could be used to solve the same problem that has already been solved in a slightly different way and to compare the methodologies.

Consider the diagram presented in Figure 14.1 which portrays a discrete-event model for a digital circuit simulator consisting of a collection of gates and wires. This system is driven by procedures contained in wire objects which post their procedures on an agenda implemented as a queue. Whenever a wire receives a signal, it posts the procedures to be activated on the event queue at the time at which the event is supposed to occur. This idea is especially useful for posting events at some future time. For example, a signal arrives at an electronic gate and is delayed because of the inherent electrical characteristics of the gate for some fixed time. To simulate this situation, the class **Delay** was used in the simulation given in Chapter 13. In the discrete-event-simulation situation, the amount of time to be delayed is added to the current simulation time and posted to the event queue so that the procedures attached to the wire are run at the appropriate later time. This capability is especially important in simulating large systems of objects, many of which spawn delayed events at varying future times.

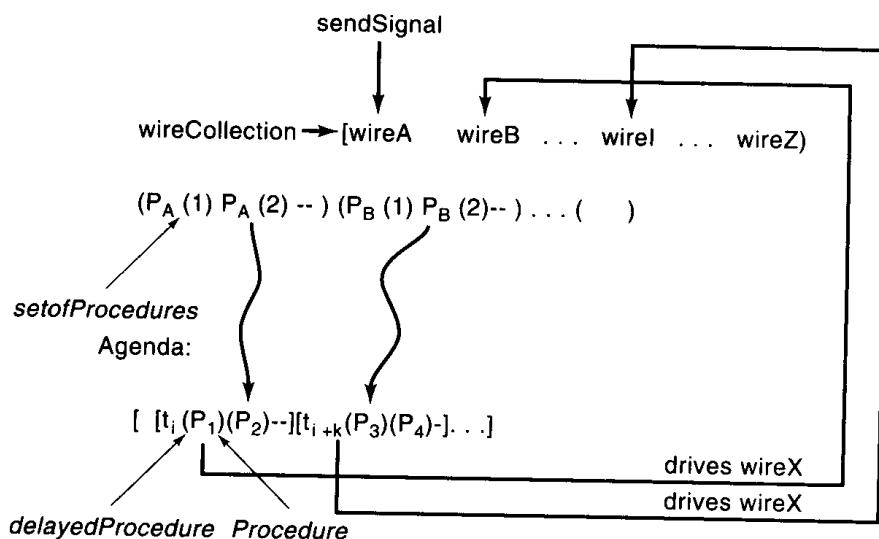


Figure 14.1 Discrete-Event Model for Digital Circuit Simulation.

The model shown in Figure 14.1 attempts to graphically explicate the description given previously. At the top of the figure, a wireCollection is shown which contains all the wires in a circuit simulation. Each wire contains a set of procedures, labeled $P_A(1)$, $P_A(2)$, and so forth, where the subscript indicates the wire with which the procedure is associated. The agenda, also known as the event queue, contains associations of procedures and times. The diagram indicates that the procedures are posted in the event queue at the appropriate time, where times are sequentially ordered. Implementation should be by sorted collection, sorting on the time so that as new events are added to the queue, sorting will create the correct time ordering. Finally, any procedure can send a message to another wire, reinitiating the entire sequence of events, resulting in the effective propagation of events through the simulated system.

14.6 A Discrete-Event-Simulation Framework

Simulation.st is an example file that is available in Smalltalk-80. Goldberg and Robson (1983) in their seminal text on Smalltalk provide a detailed explanation of this code. This section gives an overview of the simulation framework contained in *simulation.st*. Readers without ST-80 can skip this section and those interested in the details of the code are referred to the Goldberg and Robson text. The rationale for this brief presentation is to make students who are interested in simulation aware of this existing framework.

Figure 14.2 presents this author's interpretation of the operation of the code provided in *simulation.st*. Shown at the top of the figure is an object that is used to organize a simulation. This object, class **Simulation**, contains four instance variables: resources, eventQueue, currentTime, and processCount. Resources is a set that contains objects that specify the amounts of various resources in the simulation. EventQueue, a sortedCollection, contains the delayed events of the simulation. CurrentTime is used to maintain the current simulation time, and processCount keeps track of the number of processes spawned. The eventQueue shown in the center of the figure displays a number of delayed events ordered by time. Each delayed event is a simulation object whose execution is delayed until a time specified in the object and sorted on the queue. Implemented as multiple independent processes, the delayed events utilize ST-80's capability of suspending a process and the capability of resuming a process using a semaphore (signal).

Figure 14.3 shows the class hierarchy created to implement the discrete-event-simulation framework. The major subclasses of **Object** created are **Resource**, **Simulation**, **SimulationObject**, and **DelayedEvent**. **Resource** is subclassed with **ResourceProvider** and **ResourceCoordinator** which provide classes that contain resources and implement capabilities for coordinating resources.

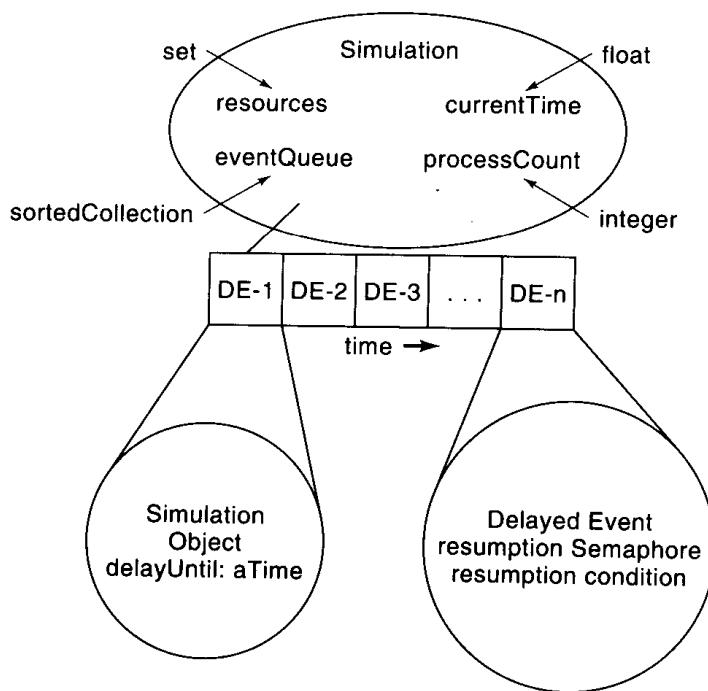


Figure 14.2 Simulation Framework Interpretation of 'simulation.st' code in Smalltalk-80.

The major interesting aspect of this simulation framework is the implementation of the simulation as a discrete-event simulation using multiple independent processes (implemented as blocks of code) which are scheduled on an event queue sorted by time. This idea is essentially the same as the digital circuit discrete-event simulation discussed in the previous section except that, in this case, blocks of code are stored on the queue to be executed. In this case,

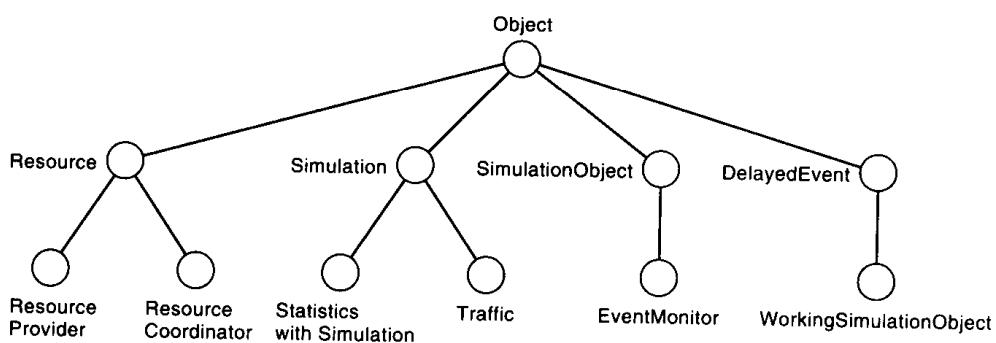


Figure 14.3 Class Hierarchy of Simulation Framework. [From Goldberg and Robson (1983).]

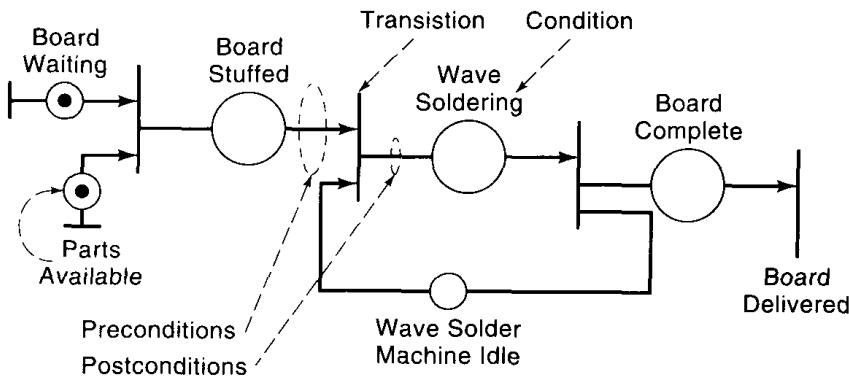


Figure 14.4 Petri Net for Electronic Printed Circuit Board (PCB) Manufacturing.

processes to be run are suspended, waiting to run until a signal is received from the simulation object that directs the sequential execution of delayed events posted on the event queue.

14.7 Simulation Using Petri Nets

Petri nets, first investigated by C. Petri (Petri, 1962), have been widely used during the last few decades for simulation (Peterson, 1981). Petri nets provide an alternative representation to the discrete-event queue-based systems discussed in the previous sections of this chapter. Petri nets provide an easy-to-understand technique for representing many kinds of systems and are easily implemented in the object-oriented style. Figure 14.4 presents a simple example of a Petri net. This figure is a Petri net that models a portion of an electronic printed circuit board (PCB) assembly system. The operation of this part of the assembly process is to fill a board with components and then solder the components on the board. Examining the figure reveals some unfamiliar symbols: (1) a vertical bar with arrows entering and exiting are termed transitions and (2) circles are termed conditions or places. The inputs to a transition are preconditions and the outputs from the transition are postconditions; that is, the transition behaves much like a rule: when preconditions are true, then the postconditions are triggered. The Petri net concept propagates tokens between places in the net, moving tokens between places according to the firing of the transitions. In the example, the first transition is “PCB Arrives,” resulting in a token being placed in the place called “Board Waiting.” The second transition has as input, “Board Waiting” and “Parts Available.” When there are tokens in both of this transition’s input places, then the transition can fire and produce a token to be placed in the output place “Board Stuffed.” Following the flow of tokens through the example shows that the next activity “Wave Soldering” will

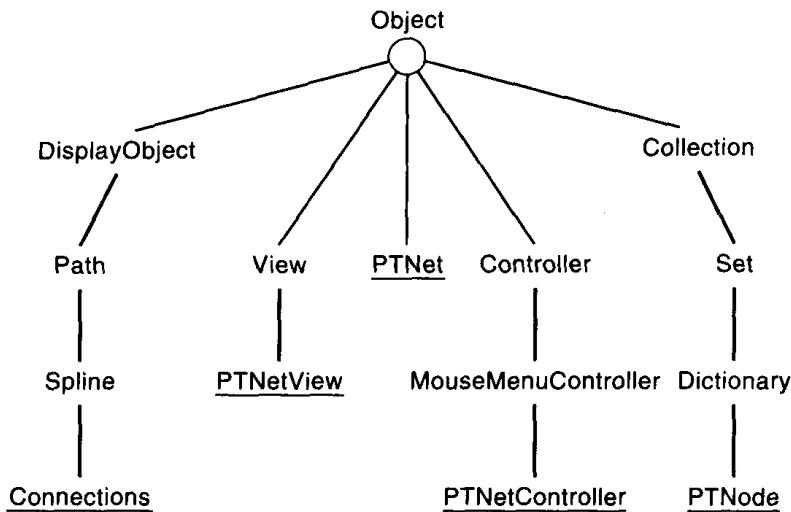


Figure 14.5 Object Hierarchy for Petri Net Implementation. Underlined classes are new.

occur when the wave solder machine is ready to be used. The output of the system is the board with components soldered on it.

The example given is typical of the kinds of graphs created using Petri nets. Most nets, however, are quite complex, yet visually useful for organizing the operation of entities such as machine shops, computer hardware, and software. Petri nets have been shown to be useful, as well, for organizing knowledge about commonplace activities such as the flow of information in a legal system, building a house, and so on (Peterson, 1981). Most applications capitalize on the capability of Petri nets to model parallel activities. Parallel applications can be easily understood by considering the PCB example. Suppose multiple assembly lines were simulated using the same schema as employed for the single line given previously. It is easy to visualize multiple places and transitions that would model the multiple parallel activities of the multiple assembly lines. Tokens could propagate asynchronously through the parallel systems and information on the operation of each line could be maintained separately or as composite data.

Figure 14.5 displays a class hierarchy for implementing a Petri net simulator. The major classes displayed are: **PTNetView**, a subclass of **View**; the **PTNetController**, a subclass of **Controller**; **PTNet**, the organizing model for the system; **PTNode**, the description of the nodes in the network; and **Connections**, a subclass of **Spline**³ used for drawing connections between the nodes on the graph. The code for this application consists of a fileOut of about 50K charac-

³Note that class **Spline** does not exist in Release 4; *displayPolyline*: could be used in a Release 4 implementation.

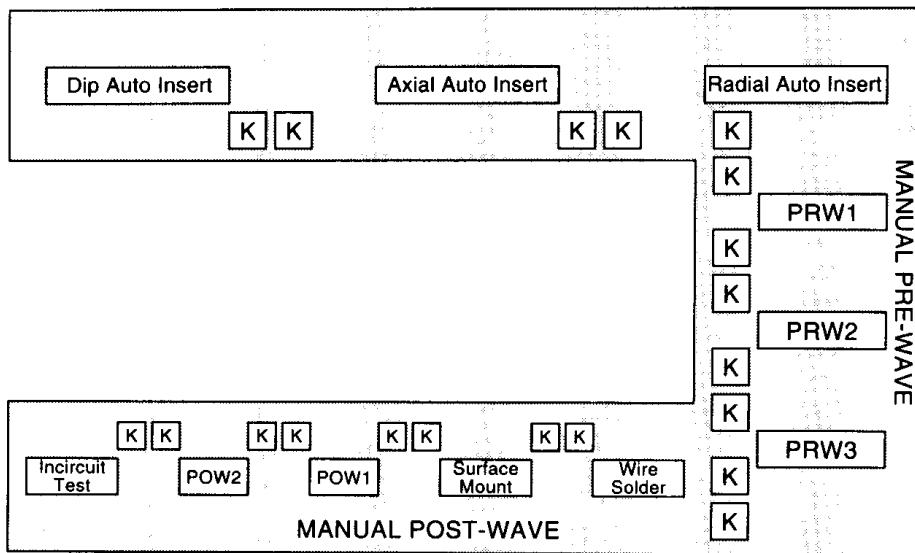


Figure 14.6 A Flexible Manufacturing Line. [With permission; Dehner (1990).]

ters and is available to interested parties as indicated in the Appendix. The use of this ST-80 implementation in a manufacturing example will be discussed next.

14.8 Building an Example Simulation for Manufacturing

This section describes the use of Petri nets for implementing a printed circuit board manufacturing line. Figure 14.6 (Dehner, 1990a), a simplification of Figure 7.17 in which the concept was discussed in the context of user interface design, displays a view from above a manufacturing line. Empty boards enter on the top left and progress through various stations around the U until exiting as a completed product at the bottom left. At each station, a single type of operation occurs, for example, moving from DIP (dual in-line package) insert to axial insert, and so forth. At each station, an operator supervises the operation of machines that insert components, wave solder the components to the board, inspect the boards, and test the boards. The squares marked K are so-called Kanban squares or temporary storage areas for the partially finished product at each stage of the manufacturing process (Dehner, 1990b). As a set of boards is completed and tested, the product is moved from one Kanban square to another, “pulling” the product through the manufacturing process.

Petri nets can be used to simulate this manufacturing process. Figure 14.7 shows the use of the Petri net simulator discussed in the previous section of this chapter. Icons for the transitions are changed to rectangles which contain

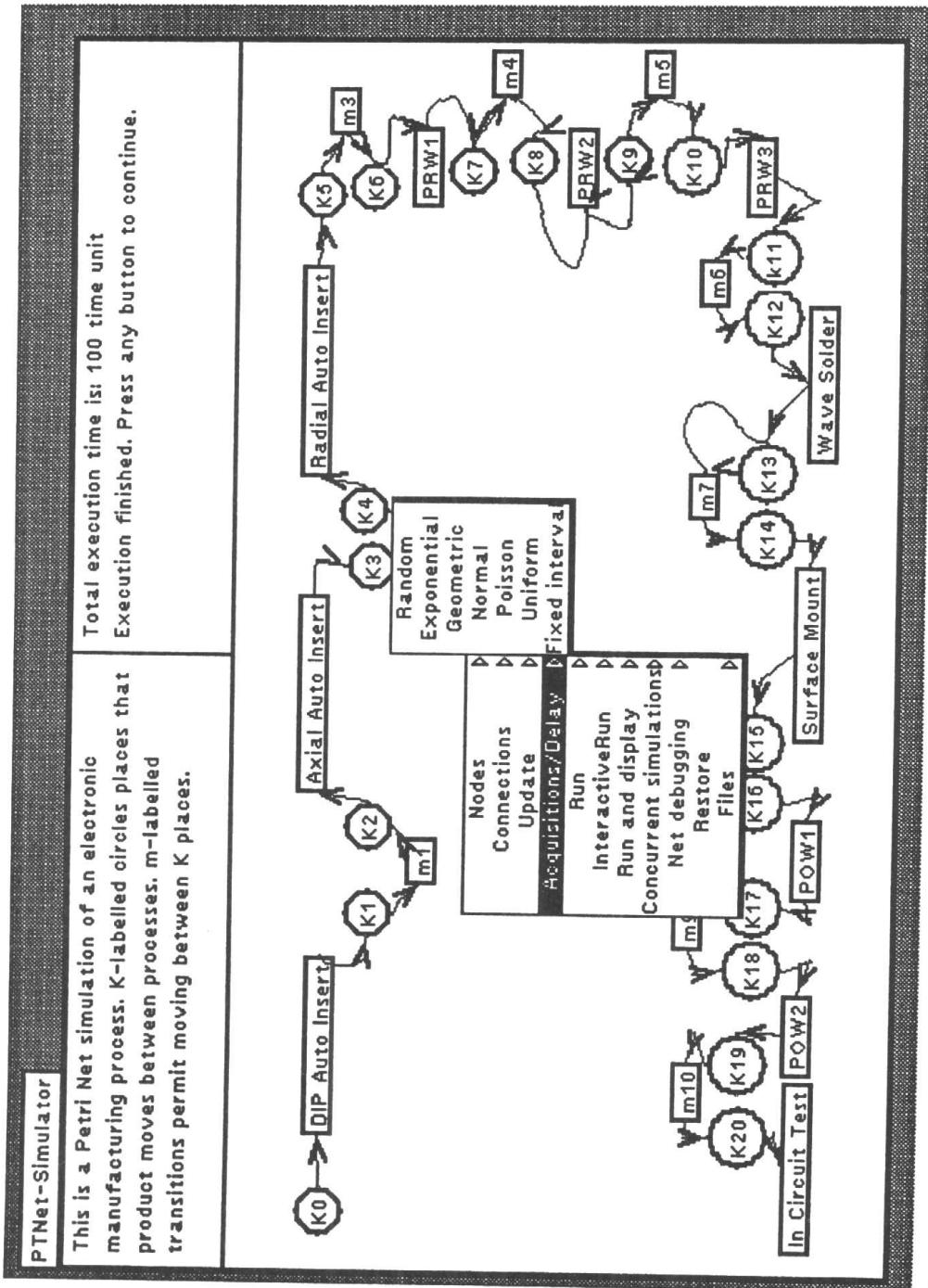


Figure 14.7 The Petri Net Simulator for Flexible Manufacturing: Building the Simulation.

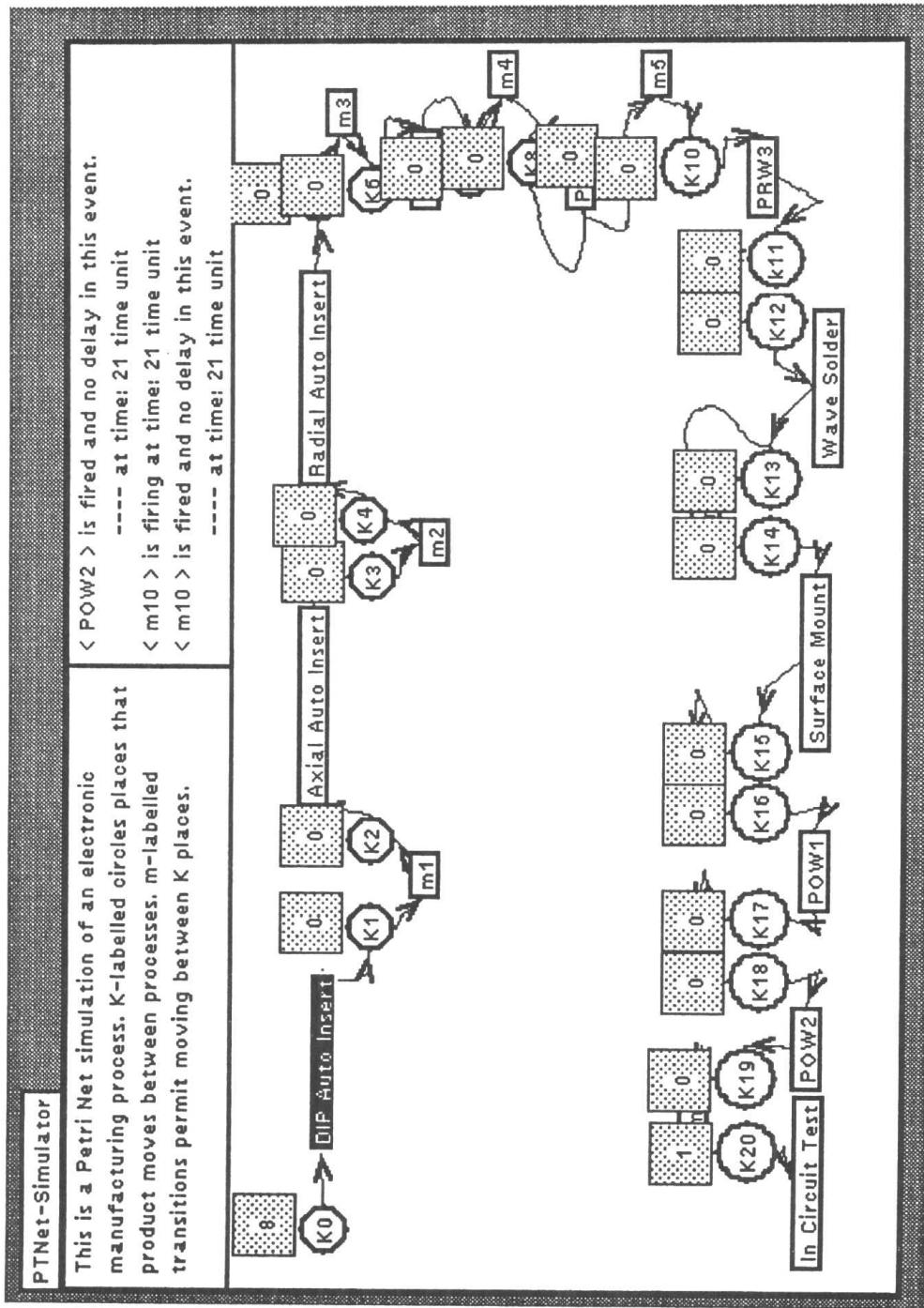


Figure 14.8 The Petri Net Simulator: Running the Simulation.

the name of the operation occurring at the transition. The places remain displayed as circles. Note the inclusion of a transition between each of the Kanban squares. The purpose of this transition is to permit transfer of a token from one square to the next (e.g., K1 to K2) when a predetermined number of items is present on the first square. Relating this statement to the example, each station would process a number of boards (e.g., 10 or 20) prior to permitting the forwarding of the batch to the next station.

Shown on the simulation view is the yellowButtonMenu which permits operation of the simulation. Operations for adding and removing nodes and links, adding tokens to nodes, defining the capacity of nodes, selecting the probability distributions associated with a transition, and other operations are accessible from this hierarchical menu (Jiang, Bourne, and Brodersen, 1990). Actually running the simulation results in the view presented in Figure 14.8 which animates the simulation. The hatched squares above each node show the number of tokens that are in a node (place) at any given time. Each transition reverses color (i.e., white to black) when the transition fires. Hence, one can watch the simulation work as the tokens flow throughout the simulation. Simultaneously, in a transcript window at the top right of the view, a textual set of information that describes the actions of the simulation is displayed.

14.9 Summary

The three major simulation methodologies described in this and previous chapters are: (1) analog constraints, (2) fixed-interval time simulation, and (3) discrete-event simulation. Analog constraints (e.g., the digital circuit simulator without delays) and equations implement instantaneous propagation of relationships. Adding time as a factor can be handled in different ways: by delays and by queues. Furthermore, relationships, time, and variables can be described both quantitatively and qualitatively. Thus, at this point, one must ask how one can decide what type of simulation is most useful for which types of situations. The answers are not easy—but here are a few observations that may be useful.

First, if a simulation system is already available for a specific purpose that will solve a problem, it should be used. Second, if the problem one is dealing with is analog in nature, use of the constraint methodology is very appealing because constraints permit visualization of analog interrelationships. Third, for discrete systems modeling, one has many choices among the myriad of digital simulation systems already created that use fixed-interval discrete simulation or event-driven simulation. Both of the latter techniques will yield the same approximate results; neither appears inherently superior except in

cases in which events are widely distributed in time. In this case, event-driven simulations appear to be more useful.

This chapter has examined a number of issues and alternatives for using various types of simulation methods and has indicated that object-oriented methodologies are a useful vehicle for implementing simulation systems. In several cases, in which entities in the real world are modeled, object-oriented techniques are obviously superior due to the one-to-one mapping between objects and simulated entities. The rapid prototyping capability of ST-80 permits building simulation systems based on the various models described. With the attendant graphical capabilities of the system, ST-80 is an appropriate tool for building simulation systems.

References

- Balmer, D. W., and R. J. Paul (1986). CASM—The Right Environment for Simulation. *Journal of the Operational Research Society*, Vol. 37, 443–452.
- Birtwistle, G., O.-J. Dahl, B. Myhrlaug, and K. Hygaard (1973). *Simula Begin*. Petrocelli/Charter, New York.
- Bobrow, D. G. (Ed.) (1985). *Qualitative Reasoning about Physical Systems*. MIT Press, Cambridge, MA.
- Bratley, Paul, B. L. Fox, and L. E. Schrage (1987). *A Guide to Simulation*, 2nd ed. Springer-Verlag, New York.
- Cox, S. and A. Cox (1985). GPSS/PC: A User Oriented Simulation System. In *Modeling and Simulation on Microcomputers*, R. G. Lavery (Ed.). Society for Computer Simulation, San Diego, CA, pp. 48–50.
- Dehner, G. (1990a). Literature Review, Personal Communication.
- Dehner, G. (1990b). Personal Communication.
- de Kleer, J. (1985). How Circuits Work. In *Qualitative Reasoning about Physical Systems*, D. G. Bobrow (Ed.). MIT Press, Cambridge, MA.
- de Kleer, J., and J. S. Brown (1985). A Qualitative Physics Based on Confluences. In *Qualitative Reasoning about Physical Systems*, D. G. Bobrow (Ed.). MIT Press, Cambridge, MA.
- Derrick, E. J., O. Balci, and R. E. Nance, (1989). A Comparison of Selected Conceptual Frameworks for Simulation Modeling. *Proceedings of the 1989 Winter Simulation Conference*. Dec. 1989, pp. 711–718.
- Drummond, Brian, and Marilyn Stelzner (1989). SimKit: A Model-Building Simulation Toolkit. In *AI Tools and Techniques*, M. H. Richer (Ed.), pages 241–259. Ablex Publishing, Norwood, NJ.
- Farquhar, Adam, and B. Kuipers (1987). QSIM: A Tool for Qualitative Simulation. University of Texas Internal Memorandum.

- Fishwick, Paul A. (1989). Qualitative Methodology in Simulation Model Engineering. *Simulation*, March, pp. 95–101.
- Goldberg, Adele, and D. Robson (1983). *Smalltalk-80, The Language and Its Implementation*. Addison-Wesley, Reading, MA.
- Guasch, Antoni, and Paul A. Luker (1990). SIMBIOS: Simulation Based on Icons and Objects. *Proceedings of the SCS Multiconference on Object Oriented Simulation*. January, 61–67.
- Haden, Gerald L. (1990). The Virtual Engineering Laboratory: A Framework for Intelligent Computer-Aided Simulation, Analysis and Design. A Review of Relevant Literature, Area Paper, Vanderbilt University, Nashville.
- Hayes, P. (1979). The Naive Physics Manifesto. In *Expert Systems in the Microelectronic Age*, D. Michie (Ed.). Edinburgh University Press, Edinburgh.
- Halfman, R. A., M. H. Ralston, and J. R. Suckling (1987). Object-Oriented Simulation for the U.S. Army Graves Registration Service. In *Proceedings of the 1987 Winter Simulation Conference*. 860–869.
- Jiang, Z., J. R. Bourne, and A. J. Brodersen (1990). A Petri-Net-Based Simulation System. Internal Working Paper, Center for Intelligent Systems, Vanderbilt University, Nashville.
- Kehler, T. P., and G. D. Clemenson (1984). An Application Development System for Expert Systems. *System Software*, Vol. 3, No. 1, January, 212–224.
- Kuipers, Benjamin (1986). Qualitative Simulation. *Artificial Intelligence*, Vol. 29, 289–338.
- Markowitz, H. M., B. Hausner, and H. W. Karr (1963). *Simscrip: A Simulation Programming Language*. RAND Corporation RM-3310-pr, Prentice-Hall, Englewood Cliffs, NJ.
- Meyers, S., and P. Friedland (1984). Knowledge-Based Simulation of Genetic Regulation in Bacteriophage Lambda. *Nucleic Acids Research*, Vol. 12, No. 1, 1–9.
- Microsoft Windows, Version 3.0 (1990). Microsoft Corporation, Redmond, WA.
- Morris, William (Ed.) (1981). *The American Heritage Dictionary of the English Language*. Houghton Mifflin, Boston, MA.
- ObjectWorks \ Smalltalk-80*, Release 4, (1991) ParcPlace Systems, Mountain View, CA.
- Ohr, Stephan A. (1990). *CAE: A Survey of Standards, Trends, and Tools*. Wiley, New York.
- Peterson, James L. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ.

- Petri, C. (1962). Kommunikation mit Automaten. PhD dissertation, University of Bonn, Bonn, West Germany. In German.
- Pidd, M. (1988). *Computer Simulation in Management Science*, 2nd Ed. Wiley, New York.
- Pinson, L. J., and R. S. Wiener (1988). *An Introduction to Object-Oriented Programming and Smalltalk*. Addison-Wesley, Reading, MA.
- Pritsker, A. A. B., and P. Kiviat (1969). *Simulation with GASP II: A FORTRAN-Based Simulation Language*. Prentice-Hall, Englewood Cliffs, NJ.
- Rothenberg, Jeff (1989). Tutorial: Artificial Intelligence and Simulation. *Proceedings of the 1989 Winter Simulation Conference*, December, 33–39.
- van der Meulen, Pieter S. (1989). Development of an Interactive Simulator in Smalltalk. *Journal of Object-Oriented Programming*, Vol. 1, No. 5, January/February, 28–51.
- Vujosevic, Ranko, B. Antao, and J. R. Bourne (1990). Object-Oriented Hierarchical Modelling and Simulation of Flexible Manufacturing Systems. *Proceedings of the SCS Multiconference on Object-Oriented Simulation*. January, 87–92.
- Webster's Unabridged Dictionary of the English Language* (1989). Dilithium Press.
- Williams, Brian C. (1985). Qualitative Analysis of MOS Circuits. In *Qualitative Reasoning about Physical Systems*, D. G. Bobrow, (Ed.). MIT Press, Cambridge, MA.

Exercises

- 14.1 Implement the multiple-process experiment described in this chapter. Create several processes of the type specified and determine how many processes need to be started in order to slow down your machine. Next insert a (*Delay forSeconds: 1*) *wait* statement in the continuously executed block. You should see a distinct difference in the behavior of your multiple processes.
- 14.2 Consider the items given in Table 14.1. Now make up your own list of examples. Indicate for each what style simulation is best and why, in your opinion.
- 14.3 Give two examples of the terms quantitative and qualitative to demonstrate that you understand the difference between the two terms.

- 14.4 Describe the differences between classical modeling and discrete-event modeling methods.
- 14.5 Try out the probability distribution class of ST-80.⁴ Plot a Gaussian curve with a mean of 10 and a standard deviation of 1. Secure points as indicated in the example in this chapter for this distribution and plot them on the same graph. Do the points appear valid for the distribution given?
- 14.6 Investigate the Smalltalk classes for scheduling multiple processes. Normally, if a method opens a scheduled window, any code in the method following the opening of the window is not executed. This prevents a block of code from opening several windows at once. Describe a technique using multiple scheduled processes which would allow a single method or block of code to open several ScheduledWindows. [Hint: Help on this question can be secured from Goldberg and Robson (1983).]
- 14.7 Petri nets provide a very general simulation model. Using the concept, draw a Petri net for a process that you understand (e.g., a car wash or multiple serving lines in a cafeteria).

⁴Found on the ParcBench Bulletin Board of ParcPlace Systems.

Conclusions

15.1 Review

In this text, object-oriented methodologies have been studied with a special emphasis on engineering analysis and design. The presentation has been segmented into three parts: (1) an introduction to object-oriented methods, (2) an examination of languages and tools, and, finally, (3) a sequence of engineering-related topics, including several examples. The text has concentrated on explaining how to use object-oriented techniques in engineering; consequently, all examples given have a relationship to engineering. In part, the reason for this concentration is the strong belief that object-oriented techniques will become a favorite methodology for facilitating computer-based engineering problem solving during the 1990s. This concluding chapter will examine trends and issues and attempt to provide a prognosis for the future of object-oriented methodologies.

15.2 Prognosis for Object-Oriented Methodologies

Object-oriented programming methodologies may ultimately be viewed as a turning point in programming or, alternatively, as only one of the major evolutionary changes in programming paradigms during the last part of the 20th century. Due to the very long time required to make assessments of the influence of programming paradigms, it may be well into the 21st century before a realistic assessment can take place. However, from the viewpoint of the beginning of the last decade of the 20th century, a prognosis can be given, arguing from the facts that are now known.

First, object-oriented methods appear exceptionally useful for breaking out of the procedural paradigm which has been taught for the last 30 years and more. Procedural coding (i.e., coding of how to do something) enforces thinking about programming in a sequential manner, that is, in a “do this, do that” format. This style is perfectly adequate for small programs and even acceptable for very large programs if one is careful to create well-structured designs in which to work. Yet, when complexity becomes high, some type of information hiding without loss of understandability is needed. The encapsulation capability of object-oriented methods is useful for reducing complexity because objects are specified as behavioral entities with which one can communicate only by sending messages. The user is not required to know what goes on inside an object, so long as an object carries out the desired task specified when a message is sent to it. In contrast, in nonobject-based languages, code is usually openly observable, thus increasing the complexity with which persons trying to understand code must deal. Of course, some complexity in traditional programming is reduced by the liberal use of function calls and subroutines. Nevertheless, hiding the entire internal structure of an object and communicating only via messages is a powerful conceptual tool that makes programming much more similar to the way humans interact and make transactions in the real world. This last characteristic should ultimately make object-oriented methods much more acceptable to future generations of programmers who can directly relate to the inherent familiarity and conceptual understandability of the methodology.

15.3 Object-Oriented Methods in Engineering

As emphasized in this book, object-oriented programming methodologies are thought to be inherently useful for problem solving in engineering. Many, if not most, engineering problems can be described conceptually in terms of objects interacting with each other. Various examples in the engineering domains have been presented; however, it is difficult to say with certainty that object-oriented techniques are a cross-engineering panacea for all problem solving. There are, after all, many algorithmic techniques, such as finite-element analysis, which are unlikely to ever become object-oriented in the near or long term. Yet, even such techniques might become part of object-oriented problem-solving packages at some point.

The real world and transactions in the real world are easily represented using the object-oriented paradigm. Engineering as a discipline deals with problem solving in the world. Hence, the use of object-oriented methods is, almost by definition, well suited for engineering. The prognosis for the use of object-oriented methods in engineering is that object-oriented solutions to problems will become gradually more used during the 1990s. As more tools for

implementing object systems become widely available and as the understanding of the methodology grows, a broad following for the paradigm should be in place by the end of the century.

15.4 The Importance of Environments

Although there are many arguments about which object-oriented language is “best,” the argument is fairly empty because the constructs on which all the object-oriented languages are based are similar, as outlined in Chapter 2. A better argument is about environments. Robust environments provide the tools needed to make programming simple and understandable. Highly efficient languages are of little use if they are difficult to use and understand. Hence, as discussed at several points in the text, the position taken here is that an environment to support any programming facility is essential to secure any degree of productivity. To date, the best example of a robust environment for object-oriented programming is Objectworks \ Smalltalk which is used as an example throughout the text. The typical argument about languages is “my language is better than your language because my language executes faster.” It is indeed surprising that this type of argument persists because it is fairly obvious that the really critical aspect of a language and environment is how efficiently and effectively one can produce the desired result—not just how fast a particular language runs! The quest for efficiency, effectiveness, and understanding revolves around the creation of more and more robust environments which ease the programmer’s and engineer’s route to a completed product with less difficulty, more insight, and a feeling that the program development environment assists rather than hinders in development.

15.5 Reusability and Productivity

There are critical and unresolved issues in reusability and productivity which should be examined as this book comes to a close. Reusability has been portrayed as a critical element for enhancing productivity. Indeed, this conjecture is almost certainly true because it is always easier to use a piece of working code rather than writing new code from scratch. Because inheritance properties of object-oriented systems make it particularly easy to reuse classes that have already been built, it is nearly axiomatic that the reusability of classes is essential to the proper conduct of object-oriented programming. Indeed, in the examples given throughout the text, the classes created are subclasses of the many classes that already exist in the ST-80 environment. However, there is a significant difficulty that must be addressed—that of rights to code created. In

traditional programming, source code is compiled and executable code delivered to the end user. For most software products, it is essential that the code and the “look and feel” of the systems created be different from any other to protect one’s legal rights. Hence, code production usually starts from scratch—a lengthy and time-consuming process that is completely anathema to our goals of reusability for productivity enhancement. Given the current widespread legalistic view of the world, any company producing a new product based on object-oriented methods might well need to produce all new code rather than subclass code that exists elsewhere, produced, for example, by another company. The producers of environments such as ST-80 and ST/V have come to grips with this problem by selling run-time licenses at a low cost. Yet, to encourage real increases in national productivity in object-oriented methodologies, there should be an open exchange of classes written. Perhaps by working together rather than in competition, richer, broader, and more productive solutions to problems can be achieved.

15.6 The Reduction of Complexity and Improvement in Rapid Prototyping

The attributes of object-oriented programming—encapsulation, polymorphism, inheritance, and abstraction—make it possible to reduce the complexity of programmatic solutions to problems. Abstraction is especially important as a way of thinking because the efficient reuse of class hierarchies requires acceptance and understanding of conceptualizations. Novice object-oriented programmers often simply subclass **Object** to create a new object. More advanced object-oriented programmers look carefully for (or remember) classes that can be subclassed which will give them more of the needed functionality than simply creating a subclass of **Object**. Hence, understanding the abstract characteristics of a class library and how to make use of the representation and behavioral characteristics of these classes can be a true productivity enhancer. With regard to encapsulation, discussed briefly before, the idea of hiding the internal code of an object is essential. For the most part, one should not need to understand the internal functionality of an object. One needs only to know what will be the behavior of the object when it is sent various messages that activate methods in its protocol. This idea is also a productivity enhancer in the sense that the implementer of a new application does not need to understand any of the encapsulated code that is used (i.e., the inherited class behaviors), only the way the object presents itself to the outside. Polymorphism is the final construct that enhances productivity. By permitting different objects to respond to the same message, polymorphism reduces conceptual complexity and enhances understandability.

15.7 Graphics and Visualization

Smalltalk-80 provides excellent graphics and visualization tools. As the original leading environment that employed a rich graphics interface, Smalltalk stimulated the creation of many similar environments such as the Macintosh user interface, Microsoft Windows, and others. Almost without question, the ability to rapidly display graphics and to be able to easily create direct-manipulation interfaces promotes understanding and ease of use of any computer system. The concept of browsing, as formalized in the visually presented classBrowser of Smalltalk, for example, opens code to easy inspection, thereby increasing understandability. Throughout the text, numerous examples of direct-manipulation interfaces have been given in various different engineering fields which illustrate the utility of using graphics-based direct-manipulation interfaces to build engineering applications.

15.8 Language and Environment Trends

As discussed previously, arguments about which language is “better” frequently center around the speed of the code produced by the compiler of a language, rather than which language promotes productivity. The reason that arguments of this sort are pinned to speed rather than usability, understandability, and productivity is probably historical rather than based on reason. Indeed, for many years, languages have been gauged in terms of speed; now, however, because computer speeds have increased significantly while costs have plummeted, it is time to refocus on the real use for programming, that is, solving problems! Given such a refocus, it is essential to consider what attributes of languages and environments will likely come to the forefront in the future. If programming is to assist the large number of individuals solving problems, the programming paradigms used should be simple and easily understood and, even more importantly, the environments in which languages are embedded should be completely supportive and nonthreatening to the average user. The present high level of skill needed and the detailed study of programming languages required should completely disappear. The object-oriented paradigm and the environmental qualities manifested in ST-80, when extended and developed, should ultimately fulfill these requirements.

During the upcoming years, a general prediction is that more programming will be conducted using object-oriented methodologies. The current trend seems to be toward adapting standard languages to the paradigm rather than learning “new” languages such as Smalltalk. It may be immaterial which language ultimately becomes a “frontrunner” in the language wars—the paradigm remains the same in all. Once learned, the cognitive methodologies

can be readily transferred between languages. There are interim dangers, however. The adding of object-oriented capabilities to traditional languages does not encourage a paradigm shift, that is, forcing a user to think in an object-oriented style. Hence, it may be difficult to teach the object-oriented methodology to individuals trained in coding procedural code. Students who do not have a long history of procedural coding are able to learn object-oriented methods quickly and fairly easily. Even more impressive is that high school and junior high students can very rapidly understand the tenets of object-oriented methods and write code. This information would tend to point toward introducing more object-oriented methodologies courses in undergraduate engineering education and in high schools where students can learn new programming habits during their formative years.

Another prediction is the growth of environments to facilitate programming. Such environments are partially already in place as discussed in detail in this text. ST-80 certainly has one of the strongest environmental capabilities of any system today.

With regard to the future of ST-80 and other current leading object-oriented languages (e.g., C++, CLOS, etc.), the prediction is that the features and capabilities of the various language contenders will ultimately look more and more alike. This prediction has the potential for becoming true because the target goal for all languages/environments should be the same, that is, facilitating the creation of applications that solve problems! Given that the methods assist in solving problems, all tools should ultimately have similar characteristics.

15.9 Near and Long-Term Trends for Object-Oriented Methodologies

Although it is always difficult to predict future developments in any field, there are several conjectures about possible trends for object-oriented methodologies that will be examined next.

Currently, the creation of applications in the style discussed in this text requires the careful consideration of model characteristics, choosing a controller or controllers, and the building of views. A likely trend may well be the continued appearance of systems that assist the user in creating applications. One rudimentary system of this type—ViewBuilder—has been shown to be of great utility in creating views in the examples given in earlier chapters in this text. Extensions of this concept seem likely, probably in the form of systems that assist in building views with extensive capabilities, such as animation, or in organizing and understanding how to specialize class hierarchies.

Knowledge acquisition mechanisms for extending and organizing class hierarchies seem like plausible tools to be added to environments. Imagine

describing a problem, perhaps even in natural language, and receiving prompts about which classes are useful for producing the application or even receiving a skeletal outline of the class organization needed for creating the application. These types of ideas about almost automated programming may be more feasible in an object-oriented environment than in other currently existing paradigms. Program understanding systems for object-oriented methodologies may eventually become widespread. For example, systems that analyze classes used in creating an application and make recommendations about how the application could be improved or reorganized would have considerable utility for application developers. The existence of large, well-formed class libraries makes it possible to believe in the feasibility of creating such object-oriented program understanding and assistant systems.

If object-oriented programming methodologies reach their expected potential, the future should see greatly expanded class libraries that are widely shared and sold. In fact, a subindustry that sells knowledge in the form of class libraries is not a too-far-fetched notion. During the upcoming years, the problem of legal rights for knowledge stored in class libraries will surely be solved in some way. The most likely directions are (1) the licensing of object-hierarchies and (2) the creation of freely shared object libraries.

15.10 Concluding Statement

This text has presented the case for the importance of employing object-oriented methodologies for engineering analysis and design. Various concrete and prospective examples have been given. For examples that have not been created, but only outlines given, the intention has been to show the plausibility of the use of object-based techniques throughout the engineering discipline. It is hoped that the material presented will assist both engineering students and practitioners in understanding the role of object-oriented methodologies in engineering analysis and design.

Exercises

- 15.1** Prepare a position paper about what you think is the potential for object-oriented methodologies.

- 15.2** Review the object-oriented languages and environments currently in existence. Determine if these languages and environments show characteristics that are similar to or different from those discussed in this text.

- 15.3 Trace the development of object-oriented languages and environments during the 1990s.
- 15.4 Review tools for object-oriented programming assistance that have been created and compare them with the original set of tools delivered with Smalltalk-80. Have these tools changed significantly?
- 15.5 Determine if any knowledge acquisition and program understanding tools exist for object-oriented systems.
- 15.6 Catalog object-oriented class libraries that are currently in existence, both for Smalltalk-80 and for other object-oriented languages. How do these libraries augment and extend the libraries described in this text?

Appendix

The appendix lists additional materials that are available from both the author and the publisher.

I. The Instructor's manual for the text is available from the publishers by writing:

Richard D. Irwin, Inc.
1818 Ridge Road
Homewood, IL 60430-9986
Faculty Service telephone number: 800-323-4560

and requesting the manual. Available also with the manual are either a DOS 5 1/4" floppy disk or a Macintosh disk (3 1/2") that contains the ST-80 code referred to in the text. These materials are available to purchasers of the textbook. Contact the publishers at the preceding address for information about availability and shipping and handling charges.

The contents of the Instructor's Manual are as follows:

- A. *Getting Started.* This section of the manual describes alternatives for acquiring and setting up a laboratory for teaching the material in this course.
- B. *Planning the Course Structure.* An outline of a course structure is given, including initial handouts, an example exam, and a syllabus that includes the amount of time to be spent on each topic.
- C. *Hints, Help, and Information about Using Each Chapter.* A chapter-by-chapter discussion of how each chapter can be presented is

given. Included are the complete solutions for the exercises at the end of each chapter.

D. *Code Printouts:*

1. canvas.st, Version 2.5
2. canvas.st, Release 4
3. ViewBuilder, Version 2.5
4. ViewBuilder, Release 4

E. *MAC or DOS Disks.* Either DOS or Macintosh disks are available with the following code:

1. canvas.st (Version 2.5) and canvas4.st (Release 4)
2. ViewBuilder classes (both Version 2.5 and Release 4)
3. The simple inference engine (Chapter 11)
4. The CLIPS interface class (Chapter 12)
5. The Petri net example (Chapter 14)
6. The digital circuit simulator (Chapter 10)
(Version 2.5 and Release 4 Versions)
7. The Network Representation Language (NRL; Chapter 11)
(Version 2.5 and Release 4 Versions)
8. gauges.st (Release 4). Gauges derived from the work by Adams (1987). Distributed with permission of Knowledge Systems Corporation.
9. charts40.st: A collection of useful chart builders in Release 4.

II. Copies of enlarged figures are available that are suitable for making overhead transparencies that can be used for lectures.

III. A version of this text written using only examples in Smalltalk-80, Version 2.5, is available from the author for the cost of duplication and a handling fee. Write: J. Bourne, Box 1570, Station B, Vanderbilt University, Nashville, TN 37235.

Reference

Adams, S. (1987) *Pluggable Gauges*. Knowledge Systems Corporation, Cary, NC.

Index

- Abstract classes, examples of, 246
- Abstraction, 245–246
 - definition of, 19, 24
 - contrast with generalization, 24
- ACOM (*see* Application/class organization method)
- ACOM cards, 52
 - application organizer card, 53
 - example, 62
 - class organizer card, 53
 - example, 63
 - class organizer extension card, 54
 - method description card, 55
 - example, 66
 - transferring ACOM example to browser, 129
 - use in DCS example, 260–261
- Active values, 30, 33
- Actor, 92, 94
- A/D (*see* Analog to digital converter)
- Ada, 88, 91
- Adams, Sam, 169, 193, 194, 418n
- Advice-giving system, 37–43
 - architecture, 40–43
- AI (*see* Artificial intelligence)
- Algol, 88
- Algorithms:
 - black box, 285
 - coupling to, 251
 - definition of, 285
- Alternate realities kit, 166–167
- American Standard Code for Information Interchange (ASCII), 318
- Amplifier, 49
- Analog to digital converter (*see* Laboratory instrumentation)
- Analog to digital/digital to analog conversion, 321
- Analysis, definition of, 45
 - general concept, 46
 - maxims, 56–59
 - process, 46–51
- Analyst, 14, 251
- Analytical thinking, 45
- AndGate, 60–61
- Animation, 147–148
 - using *follow:while:on:*, 147
- APL, 88, 89
- Application, spreadsheet, 138
- Application/class organization method (ACOM), 51–56
- Application-specific integrated circuits, 382
- Applications, engineering:
 - biomedical 241–245
 - build-it-yourself method, 179–181
 - chemical, 235–236
 - civil, 233–235
 - creating using ST-80 Environment, 125–130
 - electrical, 236
 - environmental, 241
 - examples of interfacing to the external environment, 329–338
 - mechanical, 236–237
- Pluggable View method for creating, 181, 199
 - tools for Building, 175–215
- ViewBuilder methodology, 181, 203–213
- ARK (*see* Alternate realities kit)
- Array, 73
- ART, 307
- Artificial intelligence (AI), 283
- ASCII (*see* American Standard Code for Information Interchange)
- ASIC (*see* Application-specific integrated circuits)
- AT & T, 14
- Augusta Ada, Countess of Lovelace, 91
- Automatic view building methods (*see* Applications, ViewBuilder methodology)
- Backus, John, 87
- Backward chaining, 304–306
- Backward compatibility (V.2.5), 148

- Bag**, 74
Bank example, 382
Behavior, 47–48
Binary object storage system (BOSS), 341
Binding:
 definition of, 88*n*
 dynamic, definition of, 88*n*
 static, definition of, 88*n*
Black box, 285
Blocks, use of, 263
BorderedWrapper, 179
BOSS (*see* Binary object storage system)
Browser:
 creating a new application with, 125
 screen dump, 118
 tool, 117
Browsing, term defined, 117
Buttons:
 four types, examples of, 162
 types of, 114
C, 88, 90
 interfacing to, 249, 327–328
C++, 15, 73, 91, 92
CAD (*see* Computer-aided design)
Capturing images (*see* Image fromUser), 199
CASM, 389
Car, 26
Causality, 33–35
 examples, 33–35
 half-adder, 34
 hot water heater, 34–35
Changed, 72, 177
Changes, 129
 condenseChanges, 129
Character, 117
Clascal, 89
Class:
 definition of, 20
 hierarchy V.2.5, 149
 hierarchy, Release 4, 116
 how to create, 126
 library, 116–117
CLIPS, 306, 324
 interfacing to, 325–326
CLOS (*see* Common Lisp object system)
CodeView, 187
Collection, 117
Color, use of, 145–147
ColorValue, 145
Command interpretation distance, 133
Command line interfaces, DOS, UNIX, 135
Common Lisp object system (CLOS), 14, 93
CommonLisp, 89
Communications (computer), 317–322
Compiler:
 High-C, 327
 MPW C, 327
CompositePart, 155
Computer language [*see* Languages]
Computer peripherals, 316–322
Computer-aided design, 7–9
Computer-aided engineering (CAE), 381
 example of tool, 7–8
Computing environment, 109
Conclusions, about object-oriented methodologies, 409–416
Concurrent recommended reading, 15
Constraint, definition of, 255
Constraint methods, 255–281
 concepts, 255–257
 Fahrenheit to Centigrade converter, 255–256
 local propagation, 257
 relaxation methods, 257
Constraint systems, general advice, 279–280
Constraints:
 implementation of, 263
 in the digital circuit simulator, 354–355
Controller:
 design of, 78
 objects in, 39–40
Controller maxims, 80
Coupled systems, 249
 architecture, 251–253
CP/M, 109
CRC (class-responsibility-collaborator) cards, 51
Cursor icons, 113
Custom interfaces (*see* Views)
CYC, 285
Daisy System Corporation, 381
DataBase management, object-oriented, 341–342
DCS (*see* Digital circuit simulator)
Debugger, 123
Debugging:
 debugging ST-80 Code, 127
 example of, 128
 “subscript out of bounds,” 128
DEC, 94
Declarative information, definition of, 6
Delay, 265, 269, 279, 392, 393, 395
Delays, implementation of, 265
Dependents, 72, 176
Design:
 conceptual, 69
 implementation, 69
Design scenario example:
 for digital circuits, 80
 interface methodologies, 81
Desktop, 110
Dictionary, used for representation, 75
Dictionary, 74, 79
Digital circuit simulator (DCS):
 additions to, 377–378
 animation, 352
 building the view, 359–360
 code, 367–376
 creating components, 351
 creating wires, 351
 encapsulation, 352
 example analysis of, 59

example of, 260–273
 functionality, 351
 images, 357
 implementation, 361–363
 knowledge organization, 355
 operating, 365–366
 setting inputs, 351
 teleology, 350
 visually connecting components, 351
 wire, 356
 wiring and testing, 269–273

DigitalCircuitSimulation, 46, 61

Digitalk, 14

Discrete-event model, 395

Disks with code examples, 417

DisplayDemo, example, 199

DisplayDemo

- Displaying objects, V.2.5, 149–154
 - animation, 154
 - forms, 150–151
 - paths, 152
 - pen, 152
- Displaying pictures (*see* PictureView example)
- displayLineFrom:to:*, 148

DisplayObject, 117

displayPolyLine:at:, 145

DOS, 109

Draw-80, 14

Drawing package, 14

DrawingController, 195

DrawingView, example, 196–198

DrawingView, 195

EdgeWidgetWrapper, 179

Edit-compile-debug loop, 104

Eiffel, 92, 94

Encapsulation, definition of, 19, 25

Engineering:

- aerospace, 223
- agricultural, 223
- biomedical, 222
- chemical, 222
- civil, 222
- computer, 222
- definition of, 219
- education, 223
- electrical, 222
- environmental, 222
- fields, 221
- intuition, 220
- manufacturing, 223
- materials, 223
- mechanical, 222
- nature of, 4
- problem solving framework, 5–6
- science and design, 220

Engineering knowledge:

- control, 225
- definition of, 283–284
- domain knowledge, 224
- engineering reasoning, 224
- laws, algorithms and methodologies, 224

modeling and simulation, 225
 models, algorithms, rules, 285–287
 presentation and visualization, 225
 representing, 283–313

Engineering problem solving system, 4

Error holders, for DOS, UNIX and Mac, 323

External device interfacing, 323

FFT (Fast Fourier transform), 9, 10, 49, 176, 285

File list, 124–125

FillInTheBlank, 189

Flavors, 89

- programming example, 93
- fork*, 393

Forms (V.2.5), 190–191

Fortran, 87

Forward chaining, 303–304

- ST-80 code, 304–306

Fourier transform (*see also* FFT), 220

Frame representation, 30

- C, 31
- history, 289
- Lisp, 31
- methodology, 290–293
- Pascal, 31
- Smalltalk, 31

Framework, definition of, 381

fromUser, 192

Fuji, 14, 94

Full-adder, 68

Function, 47–48

- definition of, 6

GASP, 388

GDMI (*see* Graphics direct manipulation interface)

Gemstone, 341

GemStone, 14

Generalization, 25

GPSS, 388

Graphics contexts, 144, 191

Graphics direct-manipulation interface (GDMI), 139

- concrete representations of, 141
- evaluation gap, 140
- execution gap, 140

Graphics objects, displaying, table of, 145

Graphics user interface and direct-manipulation interface:

- definition of, 136
- difference between, 137

GREGG, 341

Group work, 333, 342

GUI (*see* Graphics user interface)

Half-adder, example of, 273

Hierarchical design example, 262

Hierarchical menu, 82

Hierarchical representation (*see* Knowledge, hierarchical representation)

- HP/Apollo, 94
- Humble, 14
- Hypercard, 84
- Hypertext/Hypermedia, 296–300, 310

- IBM, 87
- Icon:
 - black image, 366
 - definition, 9n
 - dragging, 363–364
 - white image, 366
- Icons, 162–163
 - building and using, 192
- IEEE-488 bus, 11
 - bus control, example of, 335
 - digital voltmeter commands, 335
 - protocol, 321
 - serial to 488 conversion, 322
- Image fromUser, 199
- Images, 119
 - bit, 119–120
 - example editor (V2.5), 121–122
 - experiments with, 146–148
 - form, 119–120
 - hierarchy, 191
- Inference (*see also* Rule/production systems), 249–250
- Inheritance, 49
 - definition of, 19, 25
- INSIST, 391
- Inspectors, 79
 - using, 198
- Instantiation, definition of, 20
- Instructor's Manual, 417
- Instrumentation interface, example of, 10–11
- Interface design, general issues, 159
- InterLisp, 104
- Interviews, 73, 92
- IO Accessor, 323–324
- I/O port connections, 336

- KBSim, 390
- KEE, 307, 309, 388, 390
- Keyboard access, 165
- Knowledge (*see also* Engineering knowledge):
 - acquisition, 308
 - browsing, 301
 - captured in tutorial system, 302
 - chunking, 301
 - hierarchical representation, 292
 - organization, 220
 - representation, 223–226
 - representing:
 - procedural and declarative, 288–290
 - frames in objects, 293–294
 - types of,
 - design, 287
 - simulation, 287
- Knowledge organization, 220
- Knowledge Systems Corporation, 14, 193

- Laboratory instrumentation:
 - control of, 334
 - control of A/D, 337
 - control of D/A, 336
- Language (computer) prospectus:
 - familiarity, 99
 - graphics issues, 100
 - learning curve, 102
 - memory issues, 100
 - performance issues, 100
 - portability issues, 101
 - reusability, 103
 - task suitability, 99
 - training issues, 102
 - user interface issues, 101
- Languages:
 - object-oriented, 87–107
 - examples, 92
 - issue summary, 104
 - procedural, 87
- LANs and WANs (local and wide area networks), 320
- LASAR, 340
- LaserWriter, Apple, 330
- Launcher, 114
- LauncherView, 156, 192, 193
- Learning systems, 309
- Lisp, 88, 89
- ListView, 155
- Logic, 289
- LOGO, 90
- LOOPS, 104

- MacApp, 70, 72
- MacOS, 109
- MACSYMA, 259
- Magic (*see* ARK)
- Magnitude, 117
- Mentor graphics, 381
- Menus:
 - context sensitive, 113
 - dialogs, 188–189
 - hierarchical, 161, 190
 - list, 161
 - pop-up, 188–189
 - pull-down, 161
 - examples of, 160
- Message, 24
- Method, 24
 - how to create, 127
- Microsoft windows, 112, 328–329
- Microsoft software development kit (SDK), 329
- Model:
 - alternative representations, 73
 - design, 73–76
 - maxims, 79–80
 - objects in, 38–39
- Model-view-controller, 70–73, 175–181
 - conceptual framework, 176
 - controllers, 178
 - models, 177
 - views, 178

- Modeling** (*see* **Simulation**)
Models,
 declarative, 284
 mathematical, 285
 mental, 36, 286
 quantitative, 286
Molgen, 390
Mouse (*see also* **Pointing devices**), 113
 buttons, 113
Mousing, learning, 165
MovingObject, 26
MSWindowsInterface, 328
Multics, 109
Multiple independent processes, 391–393
 example of, 392
Multiple inheritance, 27
MVC (*see* **Model-view-controller**):
 example view using colors, 195

New product introduction, Petri net example, 278
NewWave, 136
NotGate, 60–61
NRL, 296
 hierarchy, 297

Object, 117
Object, state definition, 50
 behavior 50–51
 general definitions 19, 23
 identification, 48–49
Object-oriented analysis, 45–68
Object-oriented analysis and design,
 introduction to, 3
Object-oriented design, 69–84
Object-oriented engineering, definition of, 3
Objective-C, 14, 91
Objectkit \ Smalltalk, 112, 331
Objects, as a representation paradigm, 290
Objectworks \ Smalltalk (*see also* **Smalltalk**, and
 ST-80), 90
 tutorial, 15
 user's guide, 15, 70, 98, 327
open, 180
Open Desktop, 136
Open method, 180, 183
OpenOn method, 180, 182–184
Operate menu, examples, 157
Operating system:
 definition of, 13
 environment, 109
 IO Classes, 324
OPS5, 306
OrderedCollection, 73, 79
OrGate, 60–61
OS/MVS, 109
OS/2, 109, 136
OS/360, 109

Pagemaker, 82
Paint, 145
Palettes, 147

Parallel interfaces, 320
ParcBench (**ParcPlace Systems Bulletin Board**), 382, 394*n*
ParcPlace Systems, 94
Pascal, 88, 89
Pascal, object, 89
Pen (Pen), 149, 152
Pert, 14
Petri net, 14
 use for simulation, 397
 example, 276–279
PictureView example, 201–202
Pluggable gauges, 14, 169, 247
PluggableAdaptor, 186
Pointing devices:
 joysticks, 137
 light pen, 137
 mouse, 137
 new modalities, 138
 tablets, 137
 touchpads, 137
 trackballs, 137
 virtual scenes, 138
Polymorphism:
 definition of, 19, 25
 example of, 28
Pooled dictionary, 126
Pop-up menu, 82, 114
PostScript, 330
Predication, 30
Primitives, 249, 325, 327
Printed circuit board simulation, 397
Probability distributions, 393–394
 types supported in ST-80, 394
Problem solving:
 algorithmic, 219
 concept formation, 226
 constraint satisfaction, 229
 fabrication of result, 230
 heuristic, 219
 problem decomposition, 228
 process diagram, 227
 prototype design, 228
 refinement, 229
 review, 228
 simulation, 229
 steps in, 227
Procedural systems, 289
Procedures, interfacing to, 324
Productivity Products International (*see*
 Stepstone Co.)
Prolog, 90
Protocol, 24
Pull-down menu, 82

Rapid prototyping:
 analysis phase, 348
 definition of, 345
 design phase, 349
 digital circuit simulator, 346–379
 example, 345–379
 implementation phase, 349

- incremental refinement phase, 350
- maxims, 347–350
- Real World** and objects, 315–344
 - classes for accessing, 323
 - concepts, 315
- Representation hierarchy, 25
- Representing objects:
 - behavior representation, 32
 - how to, 28–33
 - structural representation, 29
- ResourceCoordination**, 59
- Reusability, 245, 246–247
- Reusable classes, examples of, 247
- Ritchie, Dennis, 91
- RMG user interface, 169
- Robot, 330
- Rule/production systems, 286, 289, 303–308, 386
- Running ST-80 code, 127

- Scheduledcontrollers, example, 146
- ScheduledWindow**, 156, 179, 182
- Schneiderman, Ben, 136
- Screen interpretation distance, 133
- Scripts, 289
 - implementing, 294
- ScrollView**, 156
- SDK (*see* Microsoft Software Development Kit)
- Self changed (*see* Changed)
- self changed*, 72
- Semantic, definition of, 289
- Semantic network, 74, 289
 - example, 29
 - implementing, 295
- SequenceableCollection**, 73
- Serial interfaces, 318
- Serial port:
 - DOS utilization, 332
 - example of use, 330
 - Unix utilization, 332
- Serpent, 70*n*, 72
- Set**, 74
- SharedQueue**, 392
- Signal analysis, example of, 9–10
- SIMBIOS, 391
- SimFlex, 391
- SimKit, 388, 389
- Simscript, 388
- Simula, 89, 90, 388
- Simulation:
 - and AI, 389
 - and object-oriented methodologies, 390
 - continuous, 383–384
 - examples of, 386
 - contrast with modeling, 36
 - coupling to, 251
 - definition of, 381
 - discrete, 383–384
 - discrete-event, examples of, 386
 - engineering, 381–407
 - event-driven, 385
 - example uses, 383
- example of manufacturing cell, 170
- fixed-interval-time, 385
- flexible manufacturing, 398–403
- framework, 397
- hierarchy, 397
- object-oriented, 12
- qualitative, 383–384
 - definition of, 387
 - types of, 308
- quantitative, 383–384
 - definition of, 387
 - types of, 387
- traffic planning, 13, 22
- using Petri nets, 397
- Simulation.st**, 396
- SKETCHPAD**, 257
- Smalltalk**:
 - class hierarchy (Release 4), 114
 - common display objects, 141, 142
 - interfacing to C, 327–328
 - interfacing to Microsoft windows, 328
- Smalltalk/V**, 12, 14, 91, 94
 - comparison with ST-80, 213
- Smalltalk environment**, 95, 109–131
 - characteristics, 114
 - classes, 103
 - components of (Release 4), 110
 - Release 4, 112
 - saving, 129
 - screen dump, 111
 - Version 2.5, 112
- Smalltalk language**:
 - blocks, 98, 263
 - constructions, 96
 - global dictionary, 96
 - iteration, 98
 - literals, 96
 - messages, 97
 - origins, 12
 - revisions, 112
 - syntax, 95–98
- Smith, R., 166, 173
- Sockets**, 340
- Software engineering, waterfall model, 284
- SortedCollection**, 73
- ST-80 (*see* Smalltalk, and Objectworks)
- startUp*, 79
- Stats.st**, 394
- Steamer**, 257
 - example of, 260
- Stepstone Co., 14, 92
- Storyboards, 248, 348
- Stream**, 117
- Stroustrup, B., 92
- Structure**, 47–48
 - definition of, 6
- SUN, 94
- SunTerminal**, 331
- Sutherland, Ivan, 257
- Symbolics Lisp Machine, 93
- System creation:
 - design frameworks, 232
 - inference, 232

- knowledge organization, 232
- refinement, 232
- Tangled hierarchies (*see also* Multiple inheritance), 292
- Tektronix, 94
- Teleology, 47–48
- Teradyne, 340
- Terminal interfaces, 331, 332
- TextCollector, global, 164
- TextCollectorView**, 187
- Text editor, 118
- TextEditor**, 163–164
- ThingLab, 257
 - example, 259
- Time, representing, 35, 384–385
- TK!Solver, 257
- Tools for object-oriented engineering, 12–14
- Traditional application building, coupling to, 246–253
- Traffic simulation, example of, 13, 233–235
- Tutorial system, a digital circuit simulator, 273–276
- TwoInputGate**, 61
- Typed:
 - definition of, 88n
 - strongly, definition of, 88n
 - weakly, definition of, 88n
- UNIX, 90, 109
- Untyped, definition of, 88n
- Update, 79, 177
- update*::, 257
- User interface software, 138–139
- User interfaces:
 - contents, 136
 - definition of, 133
 - design, 133–173
 - graphics, 135
 - primitive, 134–135
- Valid logic, 381
- VCircuitSimView**, 127
- Video interface, 169, 171
- View:
 - compiler, 77–78
 - design of, 76–78
 - library, 77
 - maxims, 80
 - objects in, 40–41
 - operations,
- basic view types, 156
- displaying, 158
- View**, 117
- View builder interface, 77
- ViewBuilder** (*see* Applications, engineering; ViewBuilder methodology)
- View framework, examples, 181–183
- Views:
 - Bar gauge, 194
 - Buttons and switches, 186–187
 - capabilities of the ST-80 user interface, 166
 - creating, 183–184
 - custom interfaces, 166
 - literalism and magic, 166
 - room metaphor, 168
 - graphically tailoring, 165
 - launcher, 190
 - selection-in-list, 184–185
 - transcripts and text editors, 187
- Virtual machine, 36
- Visual component hierarchy, 144
 - CompositePart, 143
 - DependentPart, 143
 - DialogView, 143
 - EdgeWidget, 143
 - Image, 143
 - View, 143
 - VisualComponent, 143
 - VisualPart, 143
- Visual display methods, 143–148
- VMS, 109
- VT100Terminal**, 331
- Whole-part hierarchies, 30–32
 - example, 32
- Widget, 143
- Windows and views, concepts, 154
- Wire, 60–61
 - example code, 266–268
- Workspace:
 - system, 123
 - use, 129
- Wrapper:
 - example, 340
 - software, 338–340
- Wrapper**, 143, 144
- X-Windows, 72, 136
- Xerox, 12, 14, 94

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):

```
{  
  "filename": "T2JqZWN0LU9yaWVudGVkIEVuZ2luZWVyaW5nXzQwMDUzNTUwLnppcA==",  
  "filename_decoded": "Object-Oriented Engineering_40053550.zip",  
  "filesize": 36280176,  
  "md5": "3ed4b5bd557291640e2e05030f08d867",  
  "header_md5": "79fc5ce3e56a8557bf6b1f9d5154e5b",  
  "sha1": "595a2d8edc43c035bf81a77c9e4a999496286644",  
  "sha256": "400cd2db8b5efcc1b7d9e9b49d9fa009cd5c6cfcc3d5b7e37f19761e6e0bb95d",  
  "crc32": 2168942133,  
  "zip_password": "52gv",  
  "uncompressed_size": 39985322,  
  "pdg_dir_name": "Object-Oriented Engineering_40053550",  
  "pdg_main_pages_found": 425,  
  "pdg_main_pages_max": 425,  
  "total_pages": 441,  
  "total_pixels": 2573598721,  
  "pdf_generation_missing_pages": false  
}
```