SERIES EDITOR SCOTT KNASTER

Macintosh Inside Out

DEVELOPING Object-Oriented Software FOR THE Macintosh®

Analysis, Design, and Programming

NEAL GOLDSTEIN
JEFF ALGER

Developing Object-Oriented Software for the Macintosh®

Developing Object-Oriented Software for the Macintosh®

Analysis, Design, and Programming

Neal Goldstein

Jeff Alger

Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Library of Congress Cataloging-in-Publication Data

Goldstein, Neal.
Developing object-oriented software for the Macintosh : analysis, design, and programming / Neal Goldstein, Jeff Alger.
p. cm. — (Macintosh inside out)
Includes bibliographical references and index.
ISBN 0-201-57065-3
1. Macintosh (Computer)—Programming. 2. Object-oriented programming. 3. Computer software—Development. I. Alger, Jeff. II. Title. III. Series.
QA76.8.M3G643 1992
005.265—dc20 91-29046
CIP

Sponsoring Editor: Carole McClendon
Project Editor: Joanne Clapp Fullagar
Technical Reviewer: Steven Weyl
Cover Design: Ronn Campisi
Set in 10 1/2 point Palatino by Ruttle, Shaw & Wetherill, Inc.

1 2 3 4 5 6 7 8 9-MW-9 6 9 5 9 4 9 3 9 2
First printing, January 1992

To the real objects of our affection, Linda, Sarah, and Evan Goldstein and Cindy, Nicholas, JJ, and Bobby Alger, and in memory of Jack Goldstein.

▶ Contents

▶ Foreword by Scott Knaster

Books of methods and techniques that tell you how to make good programs aren't a new idea. They've been around probably almost as long as the glorious invention of Fortran itself. Anyone who has studied college-level computer programming has gotten an earful of the latest fads that show the "right" way to build software, and any good (or bad) technical bookstore is loaded with volumes containing nifty ideas on programming methodology.

As methodology theories have come and gone over the years, programmers have also had their choice of various new toys and technologies to help them in their work. Of these, the current darling is object-oriented programming, now the star of screen, book, and Apple-IBM joint venture. Though frequently represented as a new idea, object-oriented programming has been around so long that it's just about old enough to drink. This makes it mature enough to be taken seriously in the computer biz.

You've probably read and heard a lot about object-oriented programming over the last few years, and since you're probably a hip Macintosh programmer, chances are that you've even done some real object-oriented programming yourself. As you've learned about objects and passed through what John Barlow calls "the learning curve of Sisyphus," you've probably written code, read books, and pulled your hair out discovering the joys of this nifty technology.

Developing Object-Oriented Software for the Macintosh: Analysis, Design, and Programming represents the harmonic convergence of an old and revered idea (methodologies) with an upstart, relatively new technology (object-oriented programming). This book will help you get a handle on

how you might deal with all the power and freedom that object-oriented programming provides.

In all your object-oriented travels, you probably haven't seen anything quite like this book. Neal Goldstein and Jeff Alger have been to object-land and have spent quite a lot of time there. Having seen more method calls than most people have breakfast cereals, Neal and Jeff devised the rules, methods, tests, and philosophies that they present in this book.

I think that crystallizing their ideas and writing this book has helped to keep Neal and Jeff sane men, but that doesn't mean they went about it sanely. This book was not written from some cold ivory tower of untested theory. The stuff in here is real and field-tested, and Neal and Jeff have the scars and rewritten drafts to prove it. If you follow their recipes, you'll have a good chance of finding your way through the wonderful world of object-oriented programming. Then maybe you, too, can start a joint venture with IBM.

Scott Knaster
Macintosh Inside Out Series Editor

▶ Acknowledgments

Steve Weyl, what would we have done without you? At times when it looked like this book might never be finished, you gave time and energy. Throughout, we could count on praise for good work and frank and sometimes blunt criticism where we screwed up. We are deeply in your debt.

To the good folks at Addison-Wesley, we are indebted for your patience and encouragement. Carole McClendon helped launch this project and was always enthusiastic and supportive. Joanne Clapp Fullagar was a terrific editor and sounding board. Kathy Traynor was also supportive and patient during the trying final days of the manuscript. Scott Knaster, series editor for Macintosh Inside Out, jumped onto our bandwagon at the beginning and provided invaluable encouragement and insights. Without these people, their professionalism and their patience, this book would never have seen the light of day.

Rodney Jew and Jeff Eaton of Rodneys Strategic Design and Communications provided the creative genius behind VDL and put up with our clumsy attempts to explain their craft to them. In the end, we learned that as graphic artists we are great methodologists.

Matt Melmon created many of the illustrations herein. Steve Burbeck and Roger Dunn reviewed drafts of the early chapters and provided valuable criticism and advice. It is impossible to name everyone else individually who has at some time critiqued this book or the ideas behind it, so what follows is a partial list in alphabetic order. If we've left anyone out, rest assured that your contributions were important to us and the omission is unintentional. Here goes: Harvey Alcabes, Harriet Alger, Eric Berdahl, Tom Condon, Steve Friedrich, Lee Harris, Clyde Kelley, Robin

Mair, Tony Meadow, Carl Nelson, Mark Neumann, Keith Rollin, Kent Sandvik, Andy Shebanow, Steve Strong, Dave Wilson, and Hal Wine; just about every client or student we've had in recent years; and the many contributors to MacApp.Tech$. In addition, our friends have put up with general crankiness and unreliability for the past two years and, thankfully, continued to be friends anyway.

We are also deeply indebted to the pioneers and giants, living and dead, of several key concepts in this book. There are others, but the short list includes Gregory Bateson, George Lakoff, Bertrand Russell, Edward R. Tufte, and Jean Piaget.

Whatever good ideas emerged in the form of this book are largely due to the contributions of these and many others who have come before us. As to the authors themselves, this was a true collaboration; alone, neither of us could have written this book and each is grateful for the contributions of the other. Books, like software, usually have bugs and any you may find in these pages are of our own breeding.

Our families endured far too many missed meals and ruined weekends, but remained supportive anyway. Anyone who has ever written a book can tell you that you do it because deep inside you have to, not because you want to, and our wives and children somehow seemed to understand and accept that. Vacation, anyone? Our parents also provided consistent support and encouragement. We are very sorry that Jack Goldstein is not here to see this book reach publication.

PART ONE

▶ Object-Oriented Software Development for the Macintosh

1 ▶ Introduction

▶ Solution-Based Modeling for the Macintosh

One of the authors of this book recently taught an advanced seminar to a group of some two dozen highly seasoned object-oriented Macintosh software developers. As an opener, he asked, "How many of you have ever worked on a Macintosh software project where you felt you used a development methodology?" Not a single hand went up. Everyone felt that they had made up their strategies as they went along. Many of these developers would have been appalled at proceeding this way in other environments. They clearly did not lack familiarity with software engineering principles and all wanted to better structure their projects. Nevertheless, none had yet found any techniques that really worked in the Macintosh environment.

The motivation for this book came from the author's experience at that seminar. The pages that follow describe a software development methodology, Solution-Based Modeling (SBM), that is specifically designed for use with object-oriented programming and tailored for the Apple Macintosh. SBM is a recipe resulting from years of experience with and research into the unique needs of this most demanding of environments: a dash of original material, a pinch of cognitive science, and a gallon of carefully chosen siftings from the best of many other methodologies. The best advances are those that take place in small increments and make original use of old material. So it is with SBM.

Among development methodologies designed for object-oriented software, Solution-Based Modeling is unique in an important respect. Unlike

the flood of recent books and articles on object-oriented analysis (OOA), object-oriented design (OOD), or object-oriented programming (OOP), SBM is a complete life cycle methodology that deals with all phases of software development and maintenance. We use the term *object-oriented software development* (OOSD) to encompass all that goes into creating an object-oriented program, not just the programming. One Macintosh developer had this reaction to an otherwise very popular book on OOD: "It had a lot of good material, but I felt like I was missing pages from the beginning and the end." SBM covers the entire process, from setting requirements through maintenance. It begins with a model of the business in the absence of the proposed program, proceeds through analysis, design, programming, testing, and implementation, then continues to follow the program through its useful life of enhancements and corrections.

SBM is based not only on sound principles of object-oriented design and programming, but on the way end users, managers, analysts, programmers, and the myriad other players in the game interact to create software. Among the claims that have been made for an object-oriented approach to software is that it is a "natural" way to describe the world. As you will see, that assumption, which underlies much of OOA today, is suspect at best and harmful at worst. SBM deals with cognitive science (the way people really think and communicate) and OOP (the way they program) as similar but not identical activities, with clear bridges between the two. The result is a single framework that can be discussed and understood by everyone—not just programmers and other experts, but end users and management as well.

Although SBM can be used on any computer and with any object-oriented language, there is a good reason the authors have tailored this book specifically for the Mac. Since its first release, the Macintosh has been a groundbreaking machine. Breakthrough programs—those that are exciting, innovative, and have the slick look and feel we Mac enthusiasts have come to know and love—are developed first for the Mac. Some are later ported to other machines, but they remain in their hearts Macintosh products.

At the same time it has been making life easier for computer users, the Macintosh has brought to the forefront many of the most daunting problems facing the software community today: modeless operation, graphical user interfaces, WYSIWYG ("what you see is what you get"), and copy/paste between applications. While other platforms now have similar features, the Macintosh Toolbox and most Mac development environments are in their third or fourth generation in dealing with these

problems; other graphical environments are still getting off the ground. It is altogether appropriate to launch Solution-Based Modeling the same way the authors would launch a new program: on the Mac first.

▶ Who Should Read This Book

This book is designed to appeal to a wide spectrum of Macintosh software professionals. To get the most out of it, you should have average or better experience in Macintosh software development and be familiar with, but perhaps not expert in, one or two object-oriented programming languages like C++ or Object Pascal. Although fascinated with the potential of object-oriented programming, you have probably already decided for yourself that it is not as easy as some purport it to be and you have some tough, skeptical questions. Why was it so easy at first, why is it so hard now, and why do object-oriented solutions frequently seem so artificial? Why don't your old techniques of organizing projects and programs work any more? This book answers these and many other questions.

We assume that you are already familiar with Macintosh programming fundamentals and the basic concepts of object-oriented programming. If not, you may wish to consult one of the many fine books available on these subjects. If you know little or nothing about object-oriented programming, you should be able to read and understand this book on a first reading, but will probably return to it again after using these principles to develop a project or two. If you are already an object-oriented programming expert, we hope that this book will resonate with truths from your own experiences, put object-oriented programming into a fresh perspective, and provide practical, hands-on ways to organize your projects.

▶ How to Read This Book

This book is divided into two parts. Part One provides background information about object-oriented software development, compares it to other techniques, and discusses problems and misperceptions in object-oriented software development as it is widely practiced today. Part One concludes with a brief review of research in cognitive sciences and what lessons it provides for developers of object-oriented software. Part Two presents Solution-Based Modeling (SBM), a methodology that comprises the best of both worlds by combining an intuitive, natural way to perform analysis based on sound cognitive principles with techniques that lead to good object-oriented designs and programs.

Part One is intended to be read straight through. Because each chapter depends on the previous, jumping around is not recommended. Part Two is intended to become your reference guide to SBM for the Macintosh. It is organized to facilitate its use as an on-going reference, and we hope that you will refer to it again and again.

▶ Macintosh Software Development Today

Before proceeding, it is important to answer the question no doubt already in your mind: Why bother with yet another software development methodology? Put simply, current methods are not working. If you cannot accept this at face value, we invite you to take The National Object Programming Test. If you can answer "true" to each statement below, you probably don't need this book. The rest of you are part of the silent but vast majority.

The National Object Programming Test

- I am consistently on time and within budget in software projects.
- I look forward to requests for useful enhancements to my software.
- Bonus: Users of my software love me!

The fact is that software development today remains largely a hit-or-miss affair. Despite decades of experience with top-down, bottom-up, inside-out, flowcharts, pseudocode, structured walk-throughs, software development life cycles, data flow diagrams, entity-relationship models, structured analysis and design, and an alphabet soup of acronyms, software engineering as we know it today receives at best a "D" grade in the one area that counts: bottom-line results. The Macintosh is no exception; you need only look at any edition of *MacWeek* to read of yet another major Macintosh product that is over budget and behind schedule.

Suppose you wanted to have a house built from scratch. Would you hire someone to build that dream home who had a consistent track record over several decades of not finishing the job? Yet, that is exactly the track record of the software industry. *Fortune Magazine* recently reported that 75 percent of all software projects are either never completed or are never used even when complete. This is remarkably consistent with similar surveys conducted ever since the days of coding pads and punch cards. One has to conclude that the "progress" in software engineering during that time has made no real difference in delivering quality software on time and in budget.

Suppose further that you were told that it would cost $335,000 to build the house: $85,000 for construction and *$250,000 to fix errors or omissions in the design!* Multiply those numbers by 1,000 and you have the software budget for the U.S. Air Force's F-16 fighter. Nor is this an isolated case: typically 60 percent to 85 percent of the overall cost of software is spent on maintenance. Yet, this is not the popular perception. Most people assume that the lion's share of software dollars goes to development, as shown in Figure 1-1.

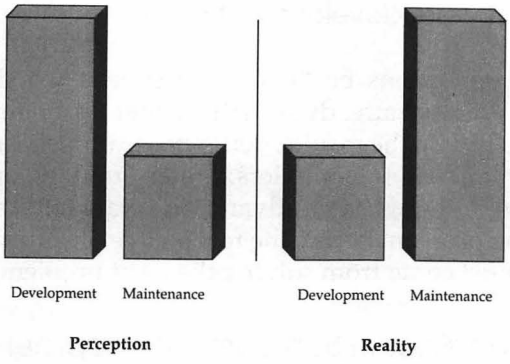


Figure 1-1. Software costs: perception versus reality

▶ Where Does the Money Go?

Start by lopping off about 30 percent for development costs. What do we spend the remaining 70 percent on? 14 percent is spent on taking corrective action. Another 14 percent is spent on adaptive changes, making the software keep up with changes in the software, hardware, and environment within which it is used. A whopping 42 percent is spent on perfective changes that make the software better fit the problem at hand. Taken together, this means that the problem does not lie with poor programming. It is our poor understanding of the need that is the real problem.

Figure 1-2 recaps these numbers. Still think high software maintenance costs are the result of errors made by programmers? Even if we completely eliminate them, the net effect will be a mere 14 percent of the total! *Forty-two percent* of total costs are due to the software not performing the right job once it is completed (if it ever is). There is also evidence to suggest that a good chunk of that 30 percent initial development cost can be traced to a poor understanding of the problem.

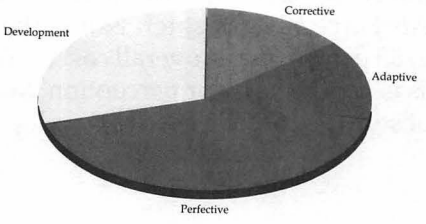


Figure 1-2. Software development costs

Can these proportions be "fixed," or should we simply adjust our expectations? We have already seen that better programming can have, at best, a minor effect on the results. Better programmers are not the answer, nor are better languages, compilers, linkers, editors, or structured code walk-throughs. Although such advances are valuable, they do not help us understand the problem better, the real source of software costs. "Fixes," if they exist, must come from solving the right problem.

▶ Myths and Realities in Software Development

Traditional software development methodologies are based on the linear or "waterfall" model shown in Figure 1-3.

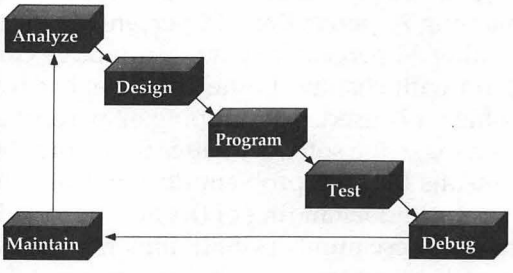


Figure 1-3. Linear ("waterfall") model

First one writes the requirements, then specifications, then design takes place, then programming, and finally implementation and on-going maintenance. None of these tasks overlaps the others; they are performed end-to-end. In theory, this is great. By having distinct hand-off points among the different groups involved in the project (management, end

users, analysts, designers and software architects, programmers, trainers, installers, and maintainers) we can easily manage such a process. In practice, however, the result is more like Figure 1-4.

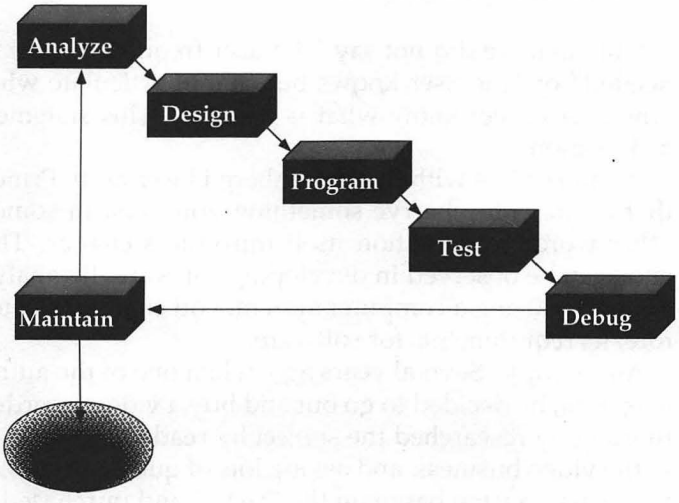


Figure 1-4. Black hole model

Once the software is completed (if it ever is) the software is so complicated and obscure that no one understands how it is built. This is the "black hole" model. Before we talk about new ways of doing things, it is important to understand why the linear model fails.

The Software Uncertainty Principle

In an ideal world,

- the user knows what is needed,
- the world does not change (at least, not once we start!),
- we fully understand the user's expressed needs, and
- we implement the user's needs flawlessly.

In reality,

- the user cannot know what is needed,
- the world changes—in fact, in addition to normal change, it changes *because* of the system we are developing,

- either the user does not communicate well, we don't listen and take notes well, or both, and
- we make mistakes.

Note that we did not say "the user frequently does not know what is needed" or "the user knows but cannot articulate what is needed" but "the user *cannot* know what is needed." This statement requires some justification.

Physicists live with the Heisenberg Uncertainty Principle, which states that in order to observe something you must in some way affect it. In other words, observation itself introduces change. The same phenomenon can be observed in developing software. By analyzing the need for and introducing a computer system, you change the business and, therefore, its requirements for software.

An example: Several years ago, when one of the author's first children was born, he decided to go out and buy a video recorder and camera. He thoroughly researched the subject by reading reviews, talking to friends in the video business, and asking lots of questions of the sales people. He made his decision based on the "facts," and purchased the "best" unit.

What happened? To those of you who owned the old style separate video camera and recorder, this will come as no surprise. He used it once an hour for the first day, once a day for the first week, once a week for the first month, once a month for the first year, and annually on birthdays after that. Why? The unit was too big, lacked autofocus, had a 20-minute battery, and required lots of light.

A few years later, he decided to buy a new video camera. This time, he knew exactly what he needed. It had to be lightweight, have a long battery life, focus itself, and be able to take movies in very low light. Was he upset? Not really. He realized that no matter how much research he had done the first time, he could not possibly know features that were important to him until he actually used a camera.

Software development, especially for graphical user interfaces like that of the Macintosh, is very much like this. At the beginning of the software development process, people understand their needs in the context of their existing environment. Once the system is implemented, in fact, once the analysis begins, the environment itself changes largely as a consequence of the system. Work flow is redirected, responsibilities are shuffled, costs and revenues change, and old bedrock assumptions about the business are undermined. In short, the business adapts to the system as the system adapts to the business.

Figure 1-5 shows the real cycle of software development. Every change results in new perceptions, which lead to more change. There is no way to

stop it. We must simply accept that there is no final equilibrium in software development; there are only passing phases which approach equilibrium. In Greek mythology, Sisyphus was condemned to push a rock up a mountain, only to have it roll back down whenever he got close to the summit. If Sisyphus were alive today, he would be in the software business.

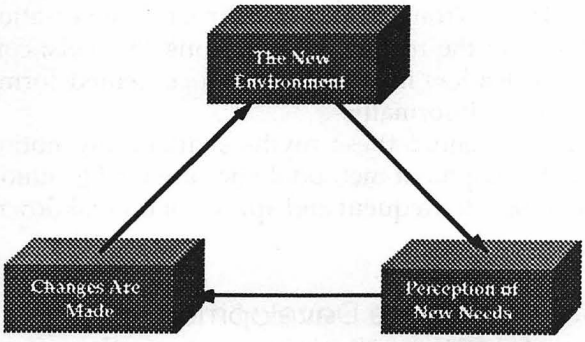


Figure 1-5. The cycle of software development

Faulty Assumptions

At the Ninth Annual IEEE Conference on Software Engineering, a talk was given exposing four common myths of software development. Taken together, they strongly suggest that traditional software development methodologies are out of touch with the way software development is really practiced.

Myth #1: The software team understands the requirements. The researchers found that, although this assumption was commonly made, the collective software team rarely had a good understanding of the problem it was supposed to solve. The damage was as much in the wrong assumption as in the fact.

Myth #2: There are fixed specifications to guide programming. The assumption was that the traditional linear model of software development could be relied on. First you define requirements, then specifications, then do the design, then program. No task overlaps any other; they are performed end-to-end. In reality, the written specifications were never correct and were constantly changed. Again, the false assumption was as damaging as the phenomenon itself.

Myth #3: Decisions on a project team are made by a process of reasoned analysis. Wrong! One or a small few dominant individuals seemed to always have their way through force of personality, not force of logic. Thus, the results depended solely on how well those people blessed with forceful personalities happened to understand the problem.

Myth #4: Information flows between project teams from team leader to team leader. Wrong again. Lunchroom conversations between team members were the real communications channels; communications between team leaders invariably just documented formally what had already occurred informally.

The reality behind these myths shatters any notion that traditional software development methodologies are well grounded in reality. That they are subject to frequent and spectacular breakdowns should surprise no one.

▶ Traditional Software Development for the Macintosh Is Even Worse

If these problems exist for software development in general, they are particularly severe in development for the Macintosh. This is not just another pretty machine. Macintosh applications tend to be sophisticated, complex, and altogether different compared to programs on other machines. There are several distinctive characteristics of the Mac that give developers fits.

Graphical User Interface

The Macintosh uses a Graphical User Interface (GUI, pronounced "gooey"). Instead of typing in cryptic commands, the user manipulates graphic images on the screen by using a mouse to point a cursor, then clicking with the mouse. The cursor and mouse together mimic one's index finger pointing to a piece of paper. Another cornerstone of Macishness is WYSIWYG, "what you see is what you get." The image on the screen should match precisely the printed result.

Figure 1-6 shows a fairly typical user interface for the Macintosh. Although the overall effect is one of simplicity, myriad details are in fact presented to the user.

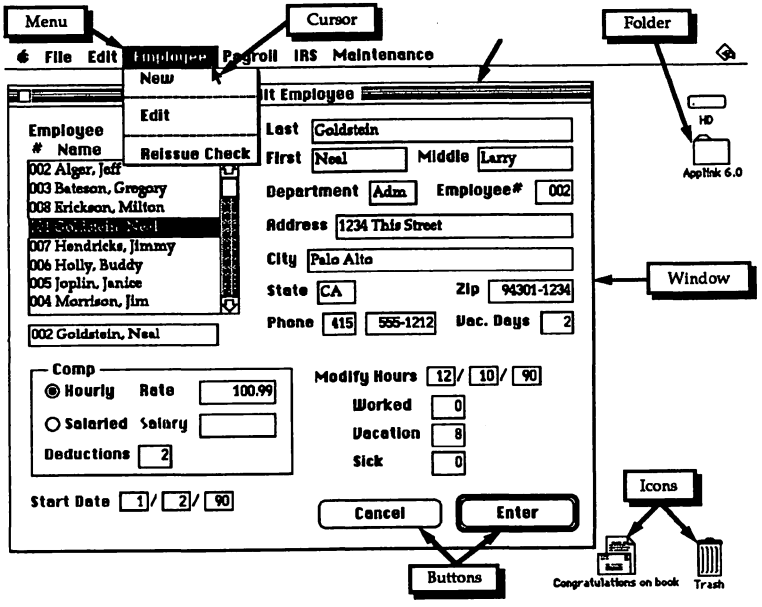


Figure 1-6. Typical Macintosh user interface

As anyone who has written a Macintosh program knows, this may look simple to the user, but is not at all simple to implement. Traditional, linear development does not work well where the look and feel of the interface is important. Why? The Software Uncertainty Principle at work. The user cannot possibly know what he or she wants until there are tires to kick and doors to slam. In order to tell the programmer what he or she wants, the user must conceive of a metaphorical, graphical interface and grasp details of data flows and algorithms, all while keeping track of the original objective. This is simply asking too much of any human being, no matter how well trained. A Macintosh programmer recalled some of the earliest advice he got when learning to program the Mac: "Remember: The trash can is your friend."

Macintosh development requires instead a process of prototyping and cyclical refinement. Study a little, think a little, develop a little, review a little, revise a little, then loop. At all stages, intermediate results guide the remaining work. This is generally true for GUI. The better the look and feel of the interface, the greater the need to abandon traditional, linear techniques.

Modeless Operation

In a traditional computer program, the program is in control. The user is guided through a hierarchy of choices, one at a time, with very few options at any one point. In the Mac's so-called "modeless" interface, the user is allowed to roam about the program almost at random, choosing the order in which to accomplish tasks and issuing commands willy-nilly to the program. For example, the user may click in a text editing window, double click on a word, copy it to the clipboard, pull down the Apple menu and launch a desk accessory, paste the text, then return to the application in an entirely different window. This plays havoc with traditional top-down approaches to software specification because there really isn't any hierarchy to the flow from step to step. Traditional methodologies are generally organized to mimic the hierarchy of choices available to the user. The Macintosh interface begs instead for an approach based on small, independent modules whose sequence of firing is not important to the design.

Attention to Detail

The Macintosh environment requires meticulous attention to a tremendous amount of detail, including each element of the user interface, the clipboard, cooperation with desk accessories and other applications, and the usual file and data management. The software environment of the Macintosh is complex enough that everything becomes closely interrelated. It is a standing joke in the Mac community that in order to thoroughly understand any chapter of the six-volume *Inside Macintosh* reference manuals, you must first thoroughly understand all of the others. Some of these "details" are not minor considerations and cannot simply be postponed until it is convenient. Yet, somehow they seem out of the mainstream of what the program is about. In a traditional top-down approach, they would not be considered candidates for top-level treatment, but would have to be jammed into bottom layers. By the time you reach those layers, you may have introduced a serious error that undoes a lot of high-level work.

Ambition

The Macintosh simply encourages more ambitious projects and attracts more ambitious programmers than other computers. A "simple" word processing program can easily consume 100,000 lines of code. The Macintosh is not inherently more difficult to program, but the standards for Macintosh software are the highest of any computer today. As in the song, if you can make it here, you'll make it anywhere.

▶ Macintosh Software Development As It Should Be

Let us start over. Forget for a moment limitations of technology and concentrate on what characteristics we would like to see in a software development methodology.

▶ What Is a Good Model?

A model is an approximation of some system. A computer program is a model of a real-world situation. Software development methodologies are models as well, of how people think and behave in a software project. A large part of the software developer's task is to create accurate, useful models of the business, while the software methodologist seeks to create models of the software development process that conform to the way good software is created. Thus, we must deal with models in both contexts.

In general, a good model helps you and others understand what a system contains, how it works, and why it works that way. It provides structure as an aid in learning about the system and, therefore, in refining the model itself. To support this, a good model should provide a natural way to create and test hypotheses about the not-as-yet understood details of the system it mimics. The model of the world used by physicists is not the real world, yet it allows us to explore the real world in a structured, systematic manner, updating our physical model as we go. We can derive apparent details from the model, test them against the real system, then update the model accordingly. Similarly, we can deduce cause and effect relationships within the model, then check whether the real system also behaves according to the same laws we derive from the model.

Models are not inherently good because they are accurate; models should also facilitate this kind of discovery and feedback. This means that human factors in understanding the model must be taken into account. This, in fact, is the real problem with traditional software. You may or may not have an accurate model, but no one can tell whether you do because the model itself is too complex or too far removed from human cognition!

To cope with human abilities, good models allow us to perceive a system at varying levels of detail. Human short-term memory only holds 7 ± 2 "facts" at once. Any model, in order to be useful, must, therefore, allow us to deal with about seven pieces of information at a time. We should be able either to temporarily ignore the detail in order to understand the whole or to focus on details without regard to the big picture. Complete accuracy at all levels of detail is neither necessary nor desirable.

We should be able to explore only "differences that make a difference," viewing the model from many different directions with only the appropriate facts visible.

A good model should also be stable. Think of a model as a road map. One can add additional cities and routes in the real world without invalidating the map. Similarly, in a good model of a business system, one should be able to add new functions and facts without changing the underlying characteristics of the model. This implies that the model is somehow based on that which is relatively unchanging and that elements of the model are as independent of one another as possible. With this independence comes the flexibility to change things easily without dramatically upsetting prior results. In the real world, the characteristics of ice cream do not really depend on the characteristics of the cone, and the whole arrangement does not depend much on the flavor chosen. Our model should have similar independence.

Finally, a good model should have *high fidelity,* introducing only the minimum possible distortion. A simple question serves to measure fidelity: to what degree can you and others recognize the real world in the model? If you must do a good deal of explaining and translating, distortion has been introduced. High fidelity models allow our users to be able to understand the structure and content of programs we create for them.

Traditional software development results in programs that are not good models in the above sense. The structure of the program has very little in common with the natural system it simulates, and a great deal of distortion results. It does not allow us to quickly focus on the relevant 7 ± 2 facts. And experience has shown that software is notoriously unstable. Linear methodologies are similarly out of synch with what we require of models. They introduce distortion, as the four myths so clearly demonstrate. Since end users are taken out of the loop early, discovery and feedback cycles are discouraged. At each stage, all of the detail must be dealt with before the next stage can begin.

▶ Five Characteristics of a Good Methodology

There are five characteristics we can use to judge whether a new software development methodology is a good one. These will guide us as we construct such a methodology for use with object-oriented software development and the Macintosh.

1. It Has to Work

   The methodology has to work. It should consistently provide high quality software that is on time, within budget, and meets the needs of the users of the software. No other measure of success matters.

2. It Must Allow for Continual Evolution

   Since we know that it is futile to try to develop fixed specifications, let's not try. Instead, our methodology should correspond to the way people think and businesses operate: by cyclical refinement. There will be no fixed ending point. Rather, there will be an on-going evolution that never ends, but with points of equilibrium along the way. Release of software will be based on utility of the "intermediate" result, not any notion of finality.

3. There Must Be Rapid Turnaround

   Because we know that the software itself will change the need, it is doubly important to start reaping results quickly. No six-month to two-year turnaround here: initial results should be available in days or weeks.

4. It Must Minimize Distortion

   Figure 1-7 illustrates a popular childhood game called "Telephone," in which a person at one end of a long line of people whispers some complicated story into the next person's ear. That person quickly turns to the next one in line and repeats the story, and so on until the last one in the line tells the story—or such of it as has survived—to the prolonged laughter of the group. Seldom does the end result even resemble the initial story.

Figure 1-7. "Telephone" and the communications gap

This game illustrates an important point: The more people we have involved in developing the software, the more important smooth communication becomes. There once was a time, long, long ago, when software development was considered to be a task relegated to a few specialists in the back room. Today, software projects require multi-disciplinary teams. Programmers alone don't have all of the answers, nor do systems analysts, end users, management, support personnel, or any one other group. Each has some unique contribution to make. In this environment, communication, not program-

ming technology, is the single biggest problem in delivering quality software. Each community has a collectively different perspective, background, and agenda.

Ideally, everybody involved should be able to explain perceptions and needs in ways that translate directly into software, and should be able to understand the structure of the resulting software without knowing much about computers. Put another way, we need a common lexicon and structure throughout the process of creating software, from analysis through code.

5. The Process Must Be Stable

   We should be free to roam through the needs and design of the software, confident that short-term mistakes will be picked up and corrected in due course. It should matter little the order in which we explore the relevant topics. If the methodology is stable, comparable results will be produced regardless. In particular, we should be able to focus first on central issues. Stability also implies—assuming that the right people are brought into the team—that the order or manner in which we talk to various participants will at worst affect the time it takes to produce results, not the accuracy of the results. The methodology should not present a canvas on which we paint, but a chalkboard, portions of which can be erased and modified at any time.

▶ Benefits of Object-Oriented Programming

Few would dispute the value of this list. Unfortunately, the technology to achieve these goals has traditionally not been available. Programming languages are still much closer to the way a computer processes data than to the way people think. Because the structure of programs has been so far removed from the way users perceive their world, it has always been necessary to interpose systems analysts and programmers between users and their programs, thereby creating a high-tech game of telephone.

As soon as you get so many people into the picture, with no one really understanding what anyone else is saying, the need arises for a great deal of structure. Development must proceed in stages and be handed off from one community to the next at specific points in time. Otherwise, we would spend all our time translating between programmers, analysts, end users, management, operations, and everyone else who has a hand in the creation of the software. Each group uses a different jargon and we can handle at most two different lexicons at one time.

Traditional analysis, design, and programming techniques fold, spindle, and mutilate users' observations into forms so completely unnatural to the non-programmer that it is a wonder that software ever comes close to the "right" solution. The situation is somewhat akin to translating poetry: It never retains quite the same meaning in another language. Except that we have several languages, one for each community involved.

It is widely claimed that object-oriented software development rewrites the rules. This is not because object-oriented programming languages are faster, bigger, smaller, or more expressive than other languages. People think in objects, so a language oriented around objects can allow programs to mimic much more closely the way people, especially end users, perceive their world. In a sense, OOSD is important precisely because we no longer need care very much about the programming language; we can and should concentrate on the problem, confident that a particular solution in a particular language will follow smoothly. OOSD provides a common lexicon that can be understood by everyone involved.

Yet, for many software professionals, these strategic goals quickly become obscured in a haze of object-oriented technobabble (OOTB): dynamic binding, polymorphism, objects, classes, inheritance, messages, encapsulation, and so on. There are, in fact, so many concepts, techniques, and technologies in object-oriented software development that it is easy to equate OOSD with technology. Granted, the technology has to exist, but object-oriented programming languages such as C++, Smalltalk, or Object Pascal are not the same as object-oriented software development. The languages are simply a means of implementing object-oriented software concepts and designs. Languages alone do not accomplish sweeping results. In fact, the game of software development has already been won or lost before the first line of code is written!

Used correctly, the object-oriented paradigm matches our natural ability to make distinctions; that is, we naturally perceive the world as divided into objects with specific behaviors. It closely models our tendencies to classify objects into wholes and parts or into types.

Look at an ice cream cone. What do you see? "Cold" described as a Boolean condition or perhaps a 32-bit integer? Algorithms?

```
for i := 1 to 10 do drip;
```

No. Chances are that your first glance will reveal a single "object," in the common usage of the word. Closer inspection reveals *parts* of the object: a

scoop of ice cream (two on a hot day), a cone, a napkin. Reflection may cause you to assign *types*, perhaps based on other desserts or with "things you eat after a softball game." Extended observation yields some *behavior* as well: it is cold at first, but if you don't eat it fast, it drips!

With object-oriented programming, these ideas can translate pretty directly into a computer program. In fact, the process described in this book is largely based on such intuitive notions: objects, wholes and parts, types, and behaviors. Thus, we cut out much of the potential for faulty translation. Put another way, object-oriented software development allows us to construct models of the problem and the solution that bridge the way people think with the way object-oriented programs are constructed. Used properly, OOSD allows a quantum leap forward.

▶ Problems With Object-Oriented Programming

Somehow all of this seems too good to be true. More seasoned readers have probably already thought back to similar claims for any of the software development methodologies introduced over the past thirty years. What makes object-oriented software development different? What are its problems? Can OOSD really be so revolutionary? Have the authors already started lying through their teeth right here in Chapter 1? The answer to the last question is . . . yes and no. What we have talked about so far could be called the "folk theory" of OOSD. As we will see in later chapters, the folk theory is not wrong, just oversimplified. Before proceeding, let's look at three skeletons in the object-oriented closet.

▶ Where Are the Methodologies?

Any software manager who has managed an object-oriented software development project knows the single biggest problem in using OOSD: Object-oriented *programming* by itself is not enough. There is still a need for a well-defined process that converges on the right results and fully exploits the power of object-oriented programming. Done poorly, object-oriented software development can be just as bad as traditional techniques. Unfortunately, methodologies for use with OOSD are still emerging. This book and Solution-Based Modeling are modest attempts in that direction.

▶ The Sheer Cliff Principle

The second problem is that object-oriented software development is easy to use only for small, simple projects. It is easy to learn object-oriented programming basics and to write simple programs. In fact, it is easy to write some fairly complex programs. However, it is a myth that object-oriented programming remains "natural" in most large, complex systems. Listening to a roomful of OOSD experts discussing what the "right" design is can remind you of a roomful of economists discussing whether the economy is going to grow or shrink a year from now: ten experts, twelve opinions. If the experts cannot agree, how can a neophyte get it right? In the real world, object-oriented software development is not always simple and is seldom obvious. We call this the "sheer cliff" principle of OOSD: Problems tend to be either as simple as a stroll in a meadow, or as difficult as a sheer cliff, but are seldom in between.

▶ "It May Be Obvious to You, But It Isn't to Me!"

Compounding this problem is the mystique that surrounds the "experts." Yes, they do produce better results. But how? Despite the dozens of books on the subject, no one yet has clearly articulated just how the expert's approach differs from that of mere mortals. It seems that the object-oriented software development expert is tuned in to some hidden channel of understanding and that, "the problem isn't OOSD, it's *you*." If only you turn the problem over in your hand and view it from different angles, maybe shake it a little and listen to the sounds it makes; if only you concentrate so hard your puzzler starts hurting; if only you can just *see* it in the right light, the solution will become obvious to you, too. Unfortunately, real software managers and real programmers need more than just faith backing their basic tools. They need blueprints and tools that produce consistently good results without a two-year learning curve.

▶ Object-Oriented Programming Is Still Worth the Effort

OK, we've said it. Object-oriented software development is not as easy as it seems. What is? OOSD is still one of the major advances in computer software of the past twenty years. It is, indeed, a better way to create

software. It is closer to the way people organize their own thoughts and, therefore, likely to produce better results than other approaches. It just isn't perfect. Fortunately, perfection isn't a requirement. OOSD need only be significantly better than other approaches in order to be useful, and that it most certainly is.

We will reconcile the problems outlined previously as this book unfolds and show how they can be recognized and overcome. No, we will not make OOSD simple to use for all problems, just simpler than other approaches and yielding substantially better results. In the real world, that is what counts. There is a difference between excellence and perfection: excellence is achievable. Let's set excellence as our goal.

▶ Summary

- Traditional software engineering is failing to produce good results, especially for the Macintosh. Forty-two percent of all money spent on software is spent to make a program fit the problem at hand better. Only 30 percent is spent on initial development and 14 percent on fixing bugs. Dramatic improvements can only result from doing a better job of solving the right problem.
- Software development is essentially an exercise in model building. Both software methodologies and the programs they produce are models of real systems. A good model
  - — Helps you and others understand what a system contains, how it works, and why it works that way.
  - — Provides structure as an aid in learning about the system you are studying.
  - — Takes into account human factors in understanding the model.
  - — Enables one to explore only "differences that make a difference."
  - — Is stable.
  - — Has high fidelity.
- A good software development methodology works, accepts continual change, provides rapid turnaround, minimizes distortion, and is stable.
- The biggest potential benefit of object-oriented software development is in bridging the gap between highly trained experts and end users in describing problems and solutions, not in pure technology.

For all of its promise, OOSD suffers from several problems today.

— Methodologies have been slow to emerge.

— There is a sheer cliff phenomenon with OOSD: simple problems are simple to solve, but complicated problems seem insurmountable.

— There is too much "magic" to OOSD and OOSD experts.

Despite these problems, OOSD remains a considerable leap forward in software technology.

2 ▶ Object-Oriented Programming: The Technologist's Perspective

▶ What This Chapter Is About

This book focuses on the object-oriented approach primarily as a great way to organize software development and only secondarily as a collection of programming languages. However, it is important to understand the technology in order to grasp how concepts and models get turned into real programs. This chapter does not replace the many fine books on object-oriented programming basics. Instead, it explains the differences between object-oriented and conventional software and how to capitalize on those differences to produce great programs. Because there seem to be as many definitions of OOP as there are experts on the subject, an authoritative list of the concepts and terms involved isn't possible. What follows is a synthesis of opinion that you should, over time, modify to suit your own experiences.

We use C++ as the language for most discussion in this book, with examples in Object Pascal where there are significant differences between the languages. Why C++? Why Object Pascal? Why not? It is a central message of this book that it shouldn't really matter what language you are using; in fact, you can apply the techniques of this book without an OOP language at all! The methodologies are as applicable to Smalltalk, Object Pascal, Lisp, or AMOL (Aunt Millie's Object Language.) C++ is very popular and can be understood by anyone with a reading knowledge of C or, with a little effort, similar high-level languages like Pascal. Do not expect a rabid defense of C++ as "the" OOP language. The authors admit to having a soft spot for C++, but don't much care which language you use. There are too many other things involved in software development that have a far greater impact on the bottom line.

▶ Objects

There is vigorous disagreement over which languages are and are not object-oriented and what features simply must be present in order for an object-oriented language to qualify for the term. We thus start with only the most basic notions of objects on which there is general agreement.

▶ What Is An Object?

Everyone can agree on one point: an "object" is a combination of data and program code. This in itself is a radical departure from traditional programming practice. In traditional programming, data structures are designed separately from the software modules that access and modify them. In OOP, the two are developed in lockstep.

Consider the following code fragment in C for a simple calculator application. Each "node" of a binary tree is either a number or a binary arithmetic operator (+, -, *, or /.) The field `node_type` tells us which class of node we are dealing with. We will declare both the data structure (node) and a function that operates on nodes. The interface looks something like this:

Interface

```
struct node {
  enum {value,plus,minus,times,divided} node_type;
  int a_number;    /* applies only when node_type == value */
  struct node *left, *right;  /* operands of a binary operator */
  };
int eval(struct node *node);
```

`struct node` and the material that follows inside the braces `{...}` declares a template for construction of and access to data of a certain type. `eval()` operates on data of that type. Note that the data is described separately from the code. The implementation of `eval()` might be as follows:

Implementation

```
int eval(node)
struct node *node;
{
  switch(node->node_type) {
      case value:
         return node->a_number;
      case plus:
```

```
        return eval(node->left) + eval(node->right);
    case minus:
        return eval(node->left) - eval(node->right);
    case times:
        return eval(node->left) * eval(node->right);
    case divided:
        return eval(node->left) / eval(node->right);
    }
}
```

In a real program, one might use the following code to call the function `eval()`:

Usage

```
the_result = eval(a_node);
```

Now, let's try the same thing using objects in C++, then Object Pascal. (What follows would make OOP purists blanch, but be patient. We will return with a better implementation later.)

C++ Interface

```
class node {
  private:
     enum {value,plus,minus,times,divided} node_type;
     int a_number;
     class node *left, *right;
  public:
     int eval();
  };
```

Object Pascal Interface

```
TypeOfNode = (value,plus,minus,times,divided);
node = OBJECT
  node_type : TypeOfNode;
  a_number : INTEGER;
  left, right : node;
  FUNCTION eval: INTEGER;
END
```

This simultaneously declares that we have a new type of data structure called `node` (a *class* of object) that contains *fields* or *data members* `node_type`, `a_number`, `left`, and `right`, as well as a function that

operates on occurrences of that data structure, `eval()`. A function declared as part of the declaration of a class in this way is called a *method*. Data members and methods are called *members* of a class and its objects.

As with a normal C structure, simply declaring a class *interface* does not create any objects; it simply provides a template for constructing and using objects. Think of an object class as a blueprint that tells you how to build a house of a certain design. The house itself is an object created from that blueprint. This relationship between class and object is sometimes difficult to grasp, but it is very important. An object is variously called an *instance* or *member* of its class. A class is also sometimes called a *type* of object. To create an object is to *instantiate* one using some class as the blueprint. Any time this many terms swirl around one concept, you can bet that it is a frequently used concept!

Note that `eval()` does not have any arguments. How does it know what structure to operate on when called? The answer is that the object is implied by the way a method is called. You cannot refer to a field in a data structure directly, but must somehow name the data structure first. Consider the following:

```
struct house {
  char address[20];
  ...
  };
...
the_address_I_want = my_house->address;
```

The variable `my_house` tells us which `address` we are talking about. The name `address` may have been used extensively elsewhere in the program, perhaps in other data structures for offices or as itself a data type describing speeches. By naming the structure with `my_house->`, we focus on the use of the symbol `address` within the structure type of the variable `my_house`. We also know that each copy of a data structure has a copy of each of its fields. There may be hundreds of `house` structures in our program at one time, each of which has its own copy of the field `address`. By naming the particular structure with a variable, we identify which of the many copies of `address` we want.

OOP languages take this idea one step further. Conceptually, copies are made of the methods named in the class for each instance we create. As with a field in a structure, you must identify the object you are talking about in order to call one of its methods or refer to one of its data members. For efficiency, OOP languages have clever ways of maintaining this illusion without really making copies, but it is the illusion that counts. We will return to this illusion in a moment. For now, consider the follow-

ing code which calls our method `eval()`. The resemblance to the way one references a field in a structure is more than coincidence.

```
the_result = a_node->eval();
```

The keywords `public` and `private` illustrate another fundamental concept in OOP. Members that are declared `private` are accessible only within the implementation of methods of the class. To every other part of the program, they are syntactically hidden. `public` members constitute the entire interface to the object. This ability to hide implementation details, while publishing the interface, is a fundamental tool of OOP called *encapsulation*. This kind of encapsulation is not available in all OOP languages. In Object Pascal, for example, all methods and data members are public; in fact, no counterparts to the key words `public`, `private`, and `protected` exist. However, it is still considered good programming practice to document and use methods according to these conventions, even where the language does not enforce the concepts.

It is very common, in fact, considered good practice, to make all data members in the object private, as was done in the C++ example, basing the entire interface to the object on methods. This technique of hiding data structures behind a functional interface is known as creating an *Abstract Data Type* (ADT). ADTs are very useful tools for creating maintainable, reusable code, even in non-OOP environments. OOP languages carry on this trend and make it an explicit part of the language. As with methods, in a language that does not enforce privacy, it is still a good idea to adopt conventions that call for accessing data members entirely through methods.

C++ Implementation

```
int node::eval()
{
  switch(node_type) {
     case value:
        return this->a_number;
     case plus:
        return this->left->eval() + this->right->eval();
     case minus:
        return this->left->eval() - this->right->eval();
     case times:
        return this->left->eval() * this->right->eval();
     case divided:
        return this->left->eval() / this->right->eval();
     }
}
```

Object Pascal Implementation

```
FUNCTION node.eval: INTEGER;
BEGIN
     CASE node_type of
        value: eval := self.a_number;
        plus: eval := self.left.eval + self.right.eval;
        minus: eval := self.left.eval - self.right.eval;
        times: eval := self.left.eval * self.right.eval;
        divided: eval := self.left.eval / self.right.eval;
     END
END
```

`this` is the name of the object for which the method was called (`self` in Object Pascal). This is the magic for which we have been searching: a method can refer to the rest of the object by using `this`. As we have already discussed, the function `eval()` is every bit as much a part of the object as `a_number`, `left`, `right`, and `node_type`. In the implementation of `eval()`, note that `a_number` and the other fields are directly accessible as if we were dealing with a normal structure. Also note the way that the left and right branches of the tree were evaluated:

```
this->left->eval()
this->right->eval()
```

This is the way one calls a method: by giving the variable which is, or points to, the object (in this case, `this`,) then the name of the method.

In this example, `this->` and `self.` are actually not necessary. C++, Object Pascal, and most OOP languages prefix names of members with a reference to the object for which the method was called. One uses `this->` and `self.` in situations where there may be some ambiguity. For example, suppose that there was a global variable called `left`? We could not tell whether the data member or the global variable was meant without some way to identify the target. `this->` is the identification. Absent some such problem, the `eval()` method could be rewritten as follows:

```
int node::eval()
{
  switch(node_type) {
      case value:
          return a_number;
      case plus:
          return left->eval() + right->eval();
      case minus:
```

```
            return left->eval() - right->eval();
        case times:
            return left->eval() * right->eval();
        case divided:
            return left->eval() / right->eval();
        }
}
```

C++ Usage

```
the_result = a_node->eval();
```

Object Pascal Usage

```
the_result := a_node.eval;
```

Note the distinction between the two usages:

```
the_result = eval(a_node); /* straight C version */
the_result = a_node->eval(); /* C++ version */
```

Both accomplish exactly the same thing.

▶ Terminology Review

Let's quickly review the concepts and terminology introduced so far.

- An *object* is a combination of a data structure and program code that accesses and/or changes that data structure.
- A *member* of an object is the equivalent of a field in a data structure. A member can be either a *data member* or a *method*.
- A *method* is a function or procedure attached to an object.
- An *object class* is the description of a generic type of object; all of the data and methods will be the same for objects in a given class. `node` is a class in the above example; `a_node` refers to an object of that class.
- Members of an object are *encapsulated* by the interface to that object. That is, we need not make visible all of the details of an object in order to allow the rest of a program to interact with the object. The `eval()` example used `private` and `public` to encapsulate all objects of the `node` class within the public interface.

Not everyone agrees on even this much terminology. For example, it is possible to have an OOP language in which there are no classes. All

objects are constructed from scratch by painstakingly adding in one member at a time. In some languages, one can shortcut the process by *cloning*, or copying, an existing object. There are also hybrids in which classes function as starting points, but objects can be modified on the fly to add to or subtract from from their class. *Calling a method* is frequently described as *sending a message* to the object. The term "member" is popular in C++ circles but "field" and "instance variable" are more common with other languages, including Object Pascal. Many OOP languages, Object Pascal among them, do not allow true encapsulation since all members are public.

One fine day, the clouds will part, the sun will radiate true enlightenment, the Cleveland Indians will win the World Series again, and all of the terminology used with OOP will settle down into something on which we can all agree. But don't hold your breath. Especially for the Indians. Until then, concentrate 90 percent on the concepts and 10 percent on the names and you will not go far wrong.

▶ Anthropomorphism

The species *Homo sapiens* is incredibly egocentric. We like to ascribe human characteristics to everything: animals, cars, food, whatever ("Martha, this fool car is just too lazy to start this morning" or "What do you think of Mozart, Rover?") This *anthropomorphism* explains much of the appeal of OOP as a way of organizing software. The idea of a "thing" that can remember things, take actions, and exhibit behavior corresponds pretty closely to the way we as humans tend to organize our perceptions of the world.

From this point on, think of an object in a computer program in those terms. How about this for a paycheck object?

> *First, he asks the employee object how much her salary is. Then, he asks the deductions to compute themselves. Finally, when he's all done thinking about it, he computes the amount to pay and packs himself off to the printer.*

Seem a little foolish? Perhaps. But didn't it seem much more natural than a data flow diagram? You can almost imagine a smile on the face of the paycheck object as he smells the fresh ink being deposited on his face. On the other hand, what vision does a structure chart produce? The flow chart template you had to buy from the college bookstore years ago?

Which image is more useful in discussing software and understanding what it does?

The nice thing about objects is that they really do lend themselves to implementations of these seemingly absurd descriptions of programs and systems. By combining data and algorithms into one neat package, we can turn this kind of anthropomorphic thinking into software. Objects are things that operate on their own data; they have an awareness of themselves and of other objects and the rest of their environment. This smacks of consciousness, the notion of "self," which is the hallmark of human intelligence. No, objects are not really intelligent, but neither are a lot of other things that we describe as "thinking this" or "feeling that." It is the fit of the analogy that counts. If it walks like a duck and quacks like a duck, it might as well be a duck, even though a biologist might beg to differ. No wonder that the OOP industry gravitates toward human terms like "responsibilities" to describe objects and their interactions!

Objects are the things that you would see if you could open up the computer and peer inside as the program runs. Classes are merely ways of categorizing objects and templates for creating them. If people are objects, then "big people," "people who live in Milwaukee," and "people who read books on OOP" are examples of classes of people. Even though much of the art of creating software using objects is to choose the right classes, it is important to keep focused on the objects themselves as the ultimate program. To paraphrase Shakespeare, "the object's the thing." Classes that do not make it easier to create and understand the objects in a program are not much good. We will have much, much more to say on this subject.

▶ Inheritance and Polymorphism

So far, we have talked about classes as blueprints for creating objects. Most object-oriented languages, however, allow classes to be used as a way of grouping other classes based on the data members and methods they have in common.

▶ Inheritance

Take another look at our implementation of the `eval()` method above. You have to admit, there is something a little distasteful about the big, ugly, `switch-case` block. Let's try again by describing several different classes of objects, one for each kind of node. Bear with the syntax until we have a chance to explain.

C++ Interface

```
class node {
    /* This class is a placeholder only. We don't expect to really
     create one. */
    public:
         virtual int eval() = 0; /* a dummy eval() placeholder */
    };
class value_node: public node {/* a leaf node */
    private:
         int a_number; /* the only field which applies to a leaf */
    public:
         virtual int eval(); /* a custom eval() for leaves */
    };
class plus_node: public node { /* a + operator node */
    private:
         node *left, *right; /* the only fields for interior nodes */
    public:
         virtual int eval(); /* a custom eval() for + nodes */
    };
/* similarly for minus_node & etc. */
```

Implementation

```
int value_node::eval() /* the leaf version of eval() */
{
    return a_number;
}
int plus_node::eval() /* the + version of eval() */
{
    return left->eval() + right->eval();
}
/* similarly for minus_node & etc. */
```

Usage

```
node *a_node;
...
the_result = a_node->eval();
```

What does all this mean? Let's walk through the interface first. The line

```
class value_node: public node { /* a leaf node */
```

means that we have declared `value_node` to be a *subclass* of the class

node. This means that it has all of the features of a node, plus the custom features listed in the definition for `value_node`. The colon (`:`) separates the class being declared (`value_node`) from its *base classes* (`node`). Base classes are also called *superclasses*. This technique, which combines two or more classes into a hierarchy in which subclasses automatically assume the characteristics of their base classes, is called *inheritance*. Inheritance can span multiple levels or generations. A base class may, itself, inherit from another base class and so forth. Base classes of a class, perhaps indirect, are called *ancestors* of that class; similarly, subclasses, perhaps indirect, are sometimes called *descendants*. Figure 2-1 shows a simple notation for the classes in our example.

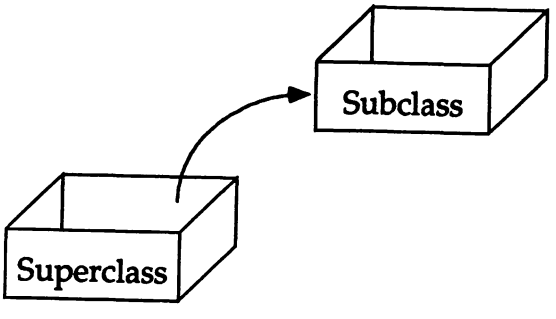


Figure 2-1. Notation for class inheritance

Another image to carry with you: Objects can put on disguises. Any object can appear to be any of its ancestors. In this example, a `value_node` can appear to be either a `value_node` or a `node`. A `plus_node` can appear to be either a `plus_node` or a `node`.

▶ Polymorphism

The keyword `virtual` says that the method `eval()` can have a meaning that varies from one subclass to another. In this example, `eval()` for `value_node` is clearly a different operation from that of `eval()` for `plus_node`. In effect, they are two different functions that share a common interface and a common purpose, operating out of sibling object classes. This ability to use the same name, in this case, `eval()`, to represent several different functions is called *polymorphism* (Poly is no relation to Anthro). When a subclass changes the meaning of a method inherited from its base class(es), it *overrides* that method. `value_node` and `plus_node` both override the method `eval()` in this example.

Polymorphism is one of the most powerful tools in the OOP arsenal for simplifying the structure of a program. Essentially, it allows you to rely on the object being able to figure out what you're talking about based on its own context. The line

```
virtual int eval() = 0;
```

means that there is no meaning for the method `eval()` in the base class. It is declared in the base class only as a placeholder. Because it is in the base class, we know that some definition of it will be available in objects of all subclasses. Remember: the class `node` is now an abstract category of nodes. We do not really expect there to ever exist an object of that class, just its subclasses. Thus, you can think of the class `node` as a *prototype* for its subclasses.

Note that in the usage we declare `a_node` to be a pointer to an object of type `node`. In practice, it is set to point to an object of one of the subclasses of `node`, such as a `plus_node`. This is OK because an object can be referred to as if it were an object of any of its superclasses. Because `eval()` is declared to be `virtual`, the right version of `eval()` will be called in the line

```
the_result = a_node->eval();
```

regardless of the real class of `a_node`.

C++ is somewhat unique in requiring a keyword—`virtual`—to identify methods that can be overridden. In most OOP languages, no such distinction is made and every method that is not private can be overridden by any subclass. Let's look at the same example in Object Pascal.

Object Pascal Interface

```
node = OBJECT
  {This class is a placeholder only. We don't expect to really create
  one.}
  FUNCTION eval : INTEGER;
END;
value_node = OBJECT (node) { a leaf node, inherits from class 'node' }
  a_number : INTEGER; { the only field that applies to a leaf }
  FUNCTION eval : INTEGER; OVERRIDE; { overrides the version in 'node' }
END;
plus_node = object (node) { a + operator node }
  left, right : node; { the only fields for interior nodes }
  FUNCTION eval : INTEGER; OVERRIDE;
END;
{ similarly for minus_node & etc. }
```

Implementation

```
FUNCTION node.eval : INTEGER; { a placeholder }
BEGIN
  write ('warning: pure virtual function node.eval called');
END;
FUNCTION value_node.eval : INTEGER; OVERRIDE; { the leaf version of eval }
BEGIN
  eval := a_number;
END;
FUNCTION plus_node.eval : INTEGER; OVERRIDE; { the + version of eval}
BEGIN
  eval := left.eval + right.eval;
END;
{ similarly for minus_node & etc. }
```

Usage

```
the_result := node.eval;
```

Since there is no such thing as a pure virtual (=0) method in Object Pascal, we have to add error-checking code for a method we never intend to call. No keyword is needed to identify a method as overridable, but you must use the keyword OVERRIDE when you override a method. And Pascal does not support the concepts of public and private interfaces. Otherwise, the two implementations are very similar.

Why did we bother? Take another look and see how much simpler the code is now! No more massive switch-case blocks. We have used polymorphism and the ability to set up hierarchies of classes to decompose one big problem into a bunch of small, simple problems. Each of the subclasses contains a single method with a single line of code doing the work. We can explain the whole program by concentrating on only one type of node at a time, along with our prototype for all nodes.

In addition to being simpler to write and understand, the program is more maintainable as well. If we now want to add more operators to the calculator, for instance, modulo division, we need only declare more subclasses of node and figure out how to create them in the enclosing program. Not a single line of code in the existing interface need change to accommodate new types of nodes!

This, in fact, is characteristic of well-designed OOP programs: They are simpler to write, easier to understand, and easier to maintain. But notice the caveat: well-designed. This example does not have any other obvious ways to categorize the objects. In any project of reasonable size, that will not be the case. You will be inundated with the possibilities and have no

way to prove that any one is right and the others wrong by sheer force of logic. Identifying objects can be tricky. Properly categorizing them into classes, armed only with logic, is like trying to eat soup with a fork. You must take into account very human factors in choosing from the alternatives. The question is not "Is this the right set of classes?" but rather "Is this a natural way to categorize my objects?"

▶ The Two Roles of Inheritance

We asserted earlier that classes are not strictly needed in order to have objects. Since inheritance is a class-based concept, it follows that inheritance is not strictly necessary in order to have objects or object-oriented software. What reasons are there for using inheritance?

- Inheritance directly expresses a way of classifying things, which is natural for people in describing their environment. Words like "ancestor" and "inheritance" convey a strong intuitive sense of a program's structure based purely on their conversational meanings. And, what could be more anthropomorphic than to ascribe lineage or role-playing to objects? In other words, *inheritance is a convenient way to describe things.*
- As with classes, inheritance is a convenient way to save a lot of time with objects that have a lot in common. We have already seen a good example of this in the use of the class node as a prototype for its subclasses. That is, *inheritance is an implementation convenience.*

Both of these are strong reasons to include inheritance in almost everyone's short list of must-have features for an OOP language. However, it is very important to keep the two uses of inheritance straight. Inheritance used to capture requirements must correspond to human cognition; inheritance used as a nifty way to write a program must be nifty but well hidden from the real definition. Keeping these two uses straight is not as easy as it seems, but is absolutely critical to producing quality results.

▶ Multiple Inheritance

In an old segment of "Saturday Night Live," two actors in a television commercial spoof argue over a new consumer product. One insists that it is a dessert topping; the other claims it is a floor wax. A third actor, obviously a representative of the manufacturer, comes in the kitchen door

and declares that it is both! How would you classify that object in the world of items in your kitchen? It is correct to say that it is a dessert topping, but it is equally correct to call it a floor wax. Yet, those two classes are as different as can be. Neither is an ancestor of the other, nor are they siblings. They are not even seventh cousins eleven times removed. There is really only one solution: use both dessert toppings and floor waxes as base classes of the new product! Figure 2-2 shows this dual inheritance.

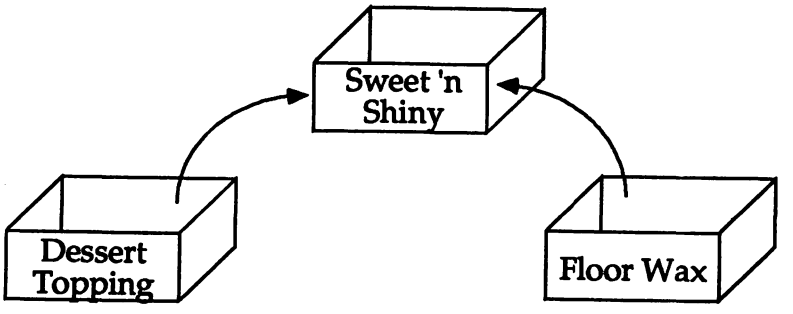


Figure 2-2. Inheritance of sweet_n_shiny

Here is a fragmentary interface for this class in C++.

```
class sweet_n_shiny: public dessert_topping, floor_wax {
    ...
    };
```

This declares class **sweet_n_shiny** to have the characteristics of two alternative lineages: that of dessert toppings and floor waxes and their respective ancestors. Objects of this class will have all of the members of both base classes.

Whenever a class inherits from more than one base class, it is said to use *multiple inheritance*. Many OOP languages, Object Pascal among them, do not support this idea and there is even some controversy as to whether it is ever appropriate to use it. After all, there are alternatives, diagrammed in Figure 2-3 and outlined in the following code.

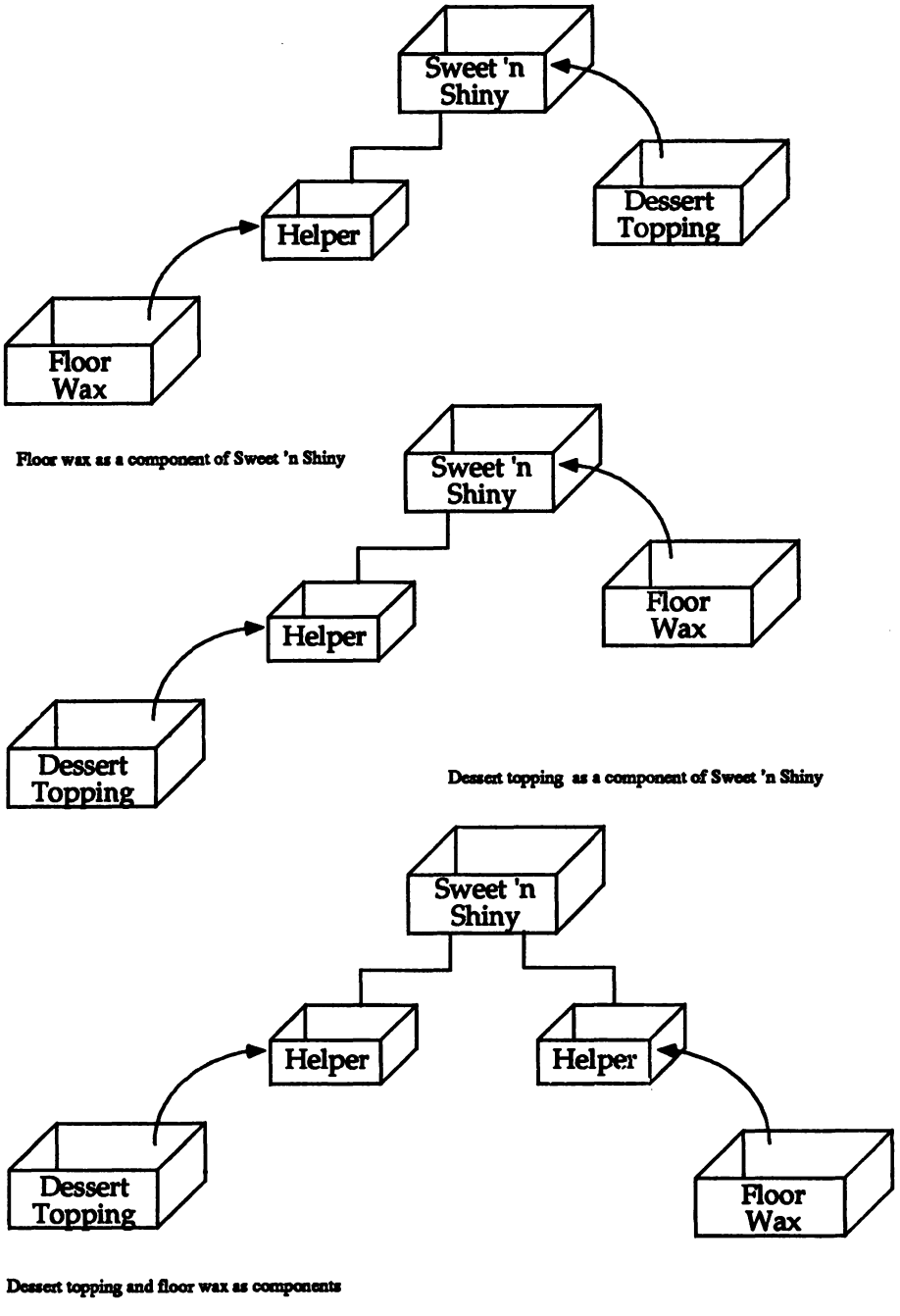


Figure 2-3. Multiple inheritance work-arounds for sweet_n_shiny

```
class sweet_n_shiny: public dessert_topping {
  private:
     floor_wax as_wax;
  ...
  };
```

or

```
class sweet_n_shiny: public floor_wax {
  private:
     dessert_topping as_dessert;
  ...
  };
```

or even

```
class sweet_n_shiny {
  private:
     dessert_topping as_dessert;
     floor_wax as_wax;
  ...
  };
```

Of course, each of these requires some extra overhead. Methods of class `floor_wax` must in some way be replicated by methods of `sweet_n_shiny` in the first version. In the extreme of the last version, all members of the base classes are `private` and, therefore, must exist under the umbrella of methods of `sweet_n_shiny`. The technique of using a member to indirectly "inherit" characteristics of a class is frequently called the multiple inheritance work-around.

Later, we will have a great deal to say about when multiple inheritance is a good idea and when it is not. For now, a few general observations will do.

1. As with classes and inheritance, multiple inheritance can be used either to *express a concept* or to *make implementation easier*. Even more so than with single inheritance, it is critically important to keep the two straight.
2. Multiple inheritance is seldom used properly. It is a powerful technique used well; it is a disaster used poorly.

3. The use of multiple inheritance as a concept in specifying objects is more important than the actual implementation. Without it, you can't create natural descriptions of consumer products like `sweet_n_shiny`. In the end, it doesn't really matter whether you use multiple inheritance in your program or not, since you can always simulate multiple inheritance using the work-around.

▶ Class Libraries

In a normal programming environment, libraries of software are organized into data structures and program modules that use them. Let's take a look at a hypothetical library.

```
typedef struct {
  int foo;
  char bar[10];
  } Foo_Bar;
void do_something_to_a_foo_bar (a_foo)
Foo_Bar *a_foo;
{
  a_foo->foo = 17;
}
void do_something_else_to_a_foo_bar(a_foo)
Foo_Bar *a_foo;
{
    /*set a_foo-> to a meaningful value */
    do_something_to_a_foo_bar (a_foo);
    /* convert foo to a string store in bar */
    sprintf(a_foo->bar, "%d",a_foo->foo);
}
```

This example may be trivial, but it still points up a big problem with conventional libraries. Suppose that we have a situation that differs slightly from what the designers of this library intended. `do_something_else_to_a_foo_bar` does exactly what we want, but `do_something_to_a_foo_bar` does not. Instead of setting `foo` to `17`, we want to ask the user for the value. Seems simple enough, but even this simple change forces us to throw out the entire library! We cannot use `do_something_else_to_a_foo_bar` without getting the wrong behavior from `do_something_to_a_foo_bar`.

Since it is simply not possible to consider all of the permutations of usage at the time you create a library, this problem can seldom be blamed on the writer of the library; this sort of problem can arise whenever you have one library function calling another.

Let's assume that our `do_something` functions are all right, but that we want to add data to the structure `Foo_Bar`. We can do this in a backhanded way as follows:

```
typedef struct {
  Foo_Bar a_foo;
  char my_data[10];
  } Slightly_Different_Foo_Bar;
```

This leads to messy code. We have added another level of indirection to the fields: `my_foo->a_foo.foo` replaces `my_foo->foo`. More ominously, what if some library function does something like this:

```
write (fd, a_foo, sizeof (Foo_Bar)); /* write the foobar to a file */
```

There is no convenient way to communicate the impact of simple additions to structures to the library functions that use them. Because of these and other problems, libraries can actually act to stifle otherwise beneficial creativity by forcing programmers to live within the structure of the library.

Now let's try this as a *class library,* which is a collection of object classes.

```
class Foo_Bar {
  private:
     int foo;
     char bar[10];
  protected:
     virtual void do_something ();
  public:
     virtual void do_something_else ();
  };
void Foo_Bar::do_something ()
{
  foo = 17;
}
void Foo_Bar::do_something_else ()
{
  do_something ();  /* set foo to a meaningful value */
  sprintf(bar,"%d",foo);  /* convert foo to a string, store in bar */
}
```

Now let's make both of the changes that caused such trouble in the conventional version.

```
class My_Foo_Bar : public Foo_Bar {
  private:
     char my_data[10];
  protected:
     virtual void do_something ();
  };
void My_Foo_Bar::do_something ()
{
  scanf ("%d", &foo);
}
```

Not bad! We recovered everything except the one behavior we wanted to change anyway. Now, when you create a `My_Foo_Bar`, the library method `do_something_else` automatically calls *your* version of `do_something`, not the version in `Foo_Bar`.

Handling problems like `sizeof` depends on the language you are using. Many languages have a way to ask for the actual size of an object, not the size of the class you think it is. In other languages, you would tend to use a polymorphic method `Get_Size()` to return the actual size, which you can override for each subclass. Either way, the problem is easily solved.

Class libraries take libraries of software out of the closet. Instead of spending most of your time working around the limitations of the library, you can spend most of your time leveraging its benefits. However, despite their power, it is a mistake to assume that all class libraries are alike for a given computer and language. As we will see, simply using object classes to build a library does not completely free the programmer from dependence on architectural decisions of the library's authors. Nor is the language a trivial issue when using a class library. Features such as encapsulation (private and protected interfaces), multiple inheritance, and dynamic behavior in the language can have a dramatic impact on the ease of use and power of a class library. We will have more to say about such language and platform dependencies in the appropriate chapters on the Solution Based Modeling methodology.

▶ Variations on a Theme of OOP

There are two fundamentally different ways of handling OOP at the language level, nicely represented by Smalltalk and C++. Smalltalk allows new classes to be created and old ones modified while the program is running. It is so flexible that your program can even change the way Smalltalk builds and uses objects. Since Smalltalk looks up methods as

they are called, it is not necessary to know the true class of an object in order to use it; you need only have a reference to the object and Smalltalk will take care of the rest. Smalltalk programs are highly dynamic. C++, on the other hand, is a compiled language. (Actually, most C++ "compilers" actually translate into standard C, which is then compiled.) This means that all classes must be explicitly declared before they can be used, and no new classes can be created at run time. The mechanisms for dispatching to methods are built into the language and cannot be changed. In order for an object's methods to be properly called, your program must tell the compiler enough for it to deduce the class to use. C++ programs are static.

In Smalltalk, everything is an object—integers, records, even program code. All objects are encapsulated behind a procedural interface. To add two integers, you call the + method of one with the other as argument. In C++, all data types available in C are at your disposal. In practice, at some level of detail you stop using objects and start using C data types such as integers and arrays of characters. These are not encapsulated.

When you use Smalltalk, the language interpreter takes over the machine. In C++, the language is compiled and is easily integrated with other programs or fragments. Smalltalk is interpreted and comes with a rich set of development tools for creating and debugging your programs. C++, like C, is a compiled language to which one must separately add software tools.

Virtually all OOP languages can be placed somewhere on the spectrum between Smalltalk and C++. Why the variety? Put simply, Smalltalk is built for elegance and features, C++ for speed and portability. Other languages have made their own tradeoffs in the above areas and each is targeted to a particular niche. It is silly to argue over which one is "best." They all have more in common with each other than with conventional languages, and for each there is some application out there crying "AMOL is the only language for me!" Here is a sampling of major OOP languages for the Macintosh:

- C++ is a standard widely available on other computers, making C++ programs more portable than programs written in Object Pascal. However, in order to make C++ compatible with MacApp and Object Pascal, Apple had to modify the standard C++ grammar. Some of these changes, if you choose to use them, remove good points of C++, among them multiple inheritance and private interfaces.
- Smalltalk V/Mac. A solid Smalltalk that carries all of the penalties of Smalltalk.

- Object Pascal. The grandparent of OOP for the Mac. Many Mac programmers will not even consider anything else. However, C programmers are equally fervent in their dislike for Pascal. "Pascal makes you say 'please,' but C makes you say 'I'm sorry.'"
- Macintosh Common Lisp. If you like Lisp, a great environment. Be prepared for large, slow programs.
- AMOL. There are always a few dozen new OOP languages in the wings. Most are being written by small companies trying to do spiffy things with OOP that C++ and Object Pascal do not support and that require better performance than Smalltalk; or that need to run their products across several different computers. It is also no great trick to throw together a simple object system from scratch tailored for a specific application.

Again, most of our examples are in C++ because of its popularity, not because the authors consider it substantially better than other languages. There are more C programmers of small computers out there than any other kind and more books and classes on C++ than all other OOP languages put together.

▶ Object-Oriented Programming on the Macintosh

Why were Macintosh programmers among the first to seize on OOP languages and practices? Principally due to the graphical user interface (GUI) presented by the Mac. In conventional user interfaces, a control program guides the user through a hierarchical menu of actions. The only real choices left to the user are to move up and down the hierarchy and to decide when to take a coffee break. Boring and not too productive.

On a Macintosh screen, one sees windows, buttons, menus, icons, and other "things." Each thing responds in a certain predictable way to mouse clicks; some will also respond to keystrokes on the keyboard. Each thing also has some current state: highlighted or not, open, closed, and so forth. The user is free to roam about, clicking with childish delight on anything that catches the eye, unfettered by a control program's dictatorial sequencing of activity.

Figure 1-6 showed a number of these typical features in the Macintosh user interface. These are very naturally implemented as objects. Each holds its state in its data members. Each responds to certain stimuli through methods. Objects in the user interface can know about and send messages to other objects. Nothing happens until the user injects an

outside stimulus into the system, typically a mouse click. Little wonder that all the way back to the Lisa (may it rest in peace) Apple engineers saw OOP as a natural way to program such interfaces. There are lots of other reasons to use OOP in non-GUI environments, but when creating a GUI program the choice is obvious.

But hold on a minute! GUI does not equal OOP; it is simply a good fit for problems-for-which-OOP-is-well-suited-as-a-nifty-way-to-implement. Simply having a GUI does not take away the need to analyze one's business requirements carefully and implement them using sound, non-GUI practice! Remember that we said that inheritance can be used in two ways: as a way of expressing concepts and as a trick of implementation. The use of OOP specifically for GUI definitely falls into the trick category: important, but not central to what this book is about.

We can more fruitfully turn the case around: GUI is a very good way to paint objects on a computer screen and allow the user to interact with them. We have asserted that the central contribution of OOP is to make computers think more like people think; GUI is a good way of allowing people to see their objects and control them. OOP has been around ever since GUI was invented and it is arguable which has driven development of the other over the years; however, we believe that objects are the more significant concept and that GUI is simply the natural expression of objects. We will deal with GUI, as it is practiced on the Macintosh, in this context throughout this book.

▶ Summary

- An *object* is a combination of a data structure and program code that accesses and/or changes that data structure. A *member* of an object is the equivalent of a field in a data structure. A member can be either a *data member* or a *method*. A method is a function or procedure attached to an object.
- An *object class* is the description of a generic type of object; all of the data and methods will be the same for objects in a given class. Members of an object are *encapsulated* by the *public interface* to that object.
- *Anthropomorphism* is commonly used to ascribe human qualities to objects in a program, particularly behaviors. This leverages our human abilities to usefully form such metaphors.
- When one object class assumes by default the characteristics (members) of another, it is said to *inherit* from the other class. The ability to use the same name to represent several different functions is called

polymorphism. This is closely related to inheritance, since inherited methods have the same names as overridden ones. Inheritance has two distinct uses in object-oriented programs: as a natural way to describe things, and as an implementation convenience. Whenever a class inherits from more than one base class, it is said to use *multiple inheritance*. As with classes and single inheritance, multiple inheritance can be used either to express a concept or to make implementation easier.

- OOP languages come in all flavors and sizes. One key difference is the tradeoff of performance versus dynamic changes at run time. C++ is at one extreme with all classes and behaviors compiled in. Smalltalk is at the opposite extreme. Almost any behavior of the Smalltalk language and your application can be changed at run time. Most other languages, such as Object Pascal, fall somewhere in between.
- *Class libraries* are reusable sets of object classes that serve the same basic purpose as conventional libraries of data types and functions that operate on them. Class libraries tend to be much more usable than their non-object counterparts due to the use of polymorphism. The structure and utility of a class library can be heavily influenced by the platform and language chosen for it.
- Macintosh programmers adopted OOP early on because it is a natural way to model the Macintosh graphical human interface. Icons, scroll bars, and other things-you-can-click-on are easily implemented as objects.

3 ▶ The Folklore of Object-Oriented Software Development

▶ What This Chapter Is About

Few people would argue that computer programs are "natural." "Natural" for a computer, perhaps, but not for people! Yet, there is something about object-oriented programming that seems to have great intuitive appeal. Even people new to programming seem to grasp object-oriented software development (OOSD) in a fraction of the time it takes to learn about conventional software. There is a commonly held explanation for this which we have already discussed: OOSD corresponds to the way people perceive and organize their thoughts about the world. This chapter explores whether this explanation is true and what the implications are. In particular, this chapter explores the *objectivist* approach to object-oriented software development. The objectivist approach is based on the commonly held belief that the world consists of objects, grouped into classes, that correspond to the kinds of objects and classes used in an object-oriented programming language. If the objectivist approach is correct, creating object-oriented software should be tremendously easier and have better results than conventional techniques. All we need do is carefully observe the real world, then create classes that implement what we see.

We call this the *folklore* of object-oriented software development. Like most folklore, the objectivist approach handles many simple situations quite well but breaks down when confronted with complicated or subtle problems. Yet, because objectivism resonates so deeply in the human psyche and because it maps so well to object-oriented programming, objectivism remains one of the dominant philosophies of object-oriented

software development. This chapter explores the basic premises of the folklore. The next chapter applies the folkloric methodology to two sample applications. As we will see, the folklore is not wrong, but it is simplistic.

▶ Software and the Human Psyche

In Chapter 2 we talked about anthropomorphism, thinking of an object as an autonomous "being" capable of remembering certain facts and interacting with other objects. Although anthropomorphism explains some of the appeal of OOP, there are much deeper reasons rooted in the way people perceive the world around them. People come prewired with certain abilities to organize perceptions of their world, including

- making *distinctions* between "things,"
- creating mental images of *objects* to represent those distinctions, and
- perceiving *relationships* between objects, between *parts* and *wholes*, and between *members* and *classes* of objects.

These concepts are not drawn from computer science, but from cognitive science. They are well supported by what we know of the way people think and perceive. The folklore springs from two additional assumptions which are difficult to support:

- It is not just our perception that the world is composed of objects and classes; the world really *is* that way. *This is the objectivist philosophy.*
- The objects and classes of the real world are easily modeled using object-oriented software. *This is the objectivist approach to OOSD.*

The evidence is against both of these assumptions, but they remain strong undercurrents in OOSD. They have a strong intuitive appeal that must be carefully explored if we are to build a methodology for OOSD that truly works. Let's start by tracing these ideas back to their philosophical roots, remembering that we are exploring a myth, albeit one commonly mistaken for reality. Later, we will explain what is and is not valid in the folklore.

▶ Objectivism

Objectivism, which dates at least to the Greek philosophers, has two major tenets.

1. There is a real world "out there," independent of any one person's perceptions. If two people have different perceptions of the world, either one of them is wrong, both of them are wrong, or they are talking about different things.
2. That world is composed of discrete objects, each of which has *properties* that characterize it. Objects do not overlap; there is a sharp boundary that separates each object from everything else. Properties can be *attributes*, such as color, or *behaviors*, such as a tendency to bite.

Corollaries to these tenets include the following.

3. There are *classes* of objects based on their shared properties. If objects A and B are both members of a class, they share the properties of the class.
4. Classes can be part of *superclasses* which have the shared properties of the classes.
5. Classes exist in the real world. The taxonomy used by biologists is a good example: family, genus, species, and so on, represent a natural division of the class of living things into a hierarchy of subclasses based on shared properties.
6. The properties of an object do not depend on the observer or the context of the observation. We may not see everything correctly, but the real object does not depend on the way we see it.
7. Since the real world is independent of the observer, the principal job of someone attempting to understand the world is to transcend human perceptions and capture the world "as it really is," not as that person thinks it is.

▶ Object-Oriented Analysis, Design, and Programming

Objectivism has been at the core of much of the field of *semantic modeling*, a discipline within computer science that seeks to simulate the structures of the mind with computers. Although object-oriented languages are not specifically designed for this purpose, many authors have pointed out the striking parallels between OOP structures and the way we deal with objects in our minds. They have therefore suggested that OOP is a good basis for semantic modeling. But this goes beyond what we normally think of as programming. Is object-oriented *programming* enough?

In traditional data processing, we more or less organize our programs around procedures that operate on data. The data is passive and cannot do anything to or by itself. Of course, in the natural world things are not passive but exhibit behaviors. A procedure, pure action and no substance, is at best a clumsy approximation of the way real-world objects behave. As a result, most people have some difficulty getting used to this separation when they are learning to program. OOP abandons this arbitrary separation of data and behaviors and replaces it with something we can all understand: objects that combine both.

▶ Creating High Fidelity Software

Remember from Chapter 1, though, that the principal challenge in creating great software is to create the right solution to the right problem; programming is only a small part of what you need to achieve that goal. Of more importance is creating software with high fidelity, software that better fits the real world. Key to achieving this goal is finding better ways to foster communication between users of software, systems analysts, software designers, programmers, and management.

It seems obvious that a good deal of the problem is the dramatic difference between the real world and traditional computer programs. Enter the twin disciplines of Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD). OOA seeks to capture requirements and specifications in terms of objects on the premise that objectivism is the one common thread among all people involved. OOD also expresses software designs in terms of objects and their properties, with the hope that non-programmers will be able to understand and comment on the design.

Clearly, we should be able to integrate all of these into one cohesive methodology so that the program is based on objects of the design, which are based on objects of the requirements, which are, in turn, a very good model of the real world objects present in the problem. The authors feel so strongly about the need for such a synthesis that we refuse to discuss object-oriented *programming* by itself. We use the term *object-oriented software development* (OOSD) in this book to represent the combination of all three practices, not just programming.

▶ Discovering Objects

This approach to object-oriented analysis and design makes us feel comfortable since we are leveraging what we already do naturally. All we have to do is select the real world objects, model in software those behaviors that will achieve whatever we want the system to do, and place

them in class hierarchies according to their natural groupings. Of course, this is not quite as simple as it sounds, but since it based on the real world it should be easier than other methods. Recall the basics of objectivism: there is a real world "out there," independent of any one person's perceptions, composed of objects that are naturally members of classes. This means that our goal in creating object-based software is not so much to create objects and classes as to discover them.

Let's apply these principles to constructing a computer program for building things (electronically) with snap-together blocks. Each block has distinct boundaries and properties such as color, smoothness of the surfaces, sizes and locations of connectors, and weight. Seems pretty straightforward, doesn't it? So obvious, in fact, that the natural way to organize a computer program is according to these objects.

The objects in our program are blocks and their parts. Presumably, all blocks are pretty much alike and therefore share all properties. Let us, then, create a single class called block to represent all BLOCK objects. What about parts? We have used the term "connector" to represent a snap arrangement, so let's also create a class called CONNECTOR. That's it for objects and the most basic level of classes; now for the superclasses. Blocks and connectors seem to have some characteristics in common if one really stretches, but they are more different than alike. There don't appear to be any pertinent superclasses. Figure 3-1 shows our blocks world.

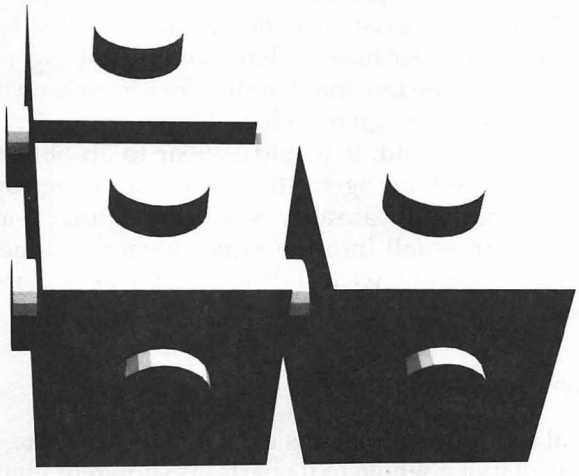


Figure 3-1. Blocks world

Now that we have classes, let's talk about properties. We have already talked about a few attributes: color, surface characteristics, and so on. Clearly dimensions will be needed. But where do we draw the line? We could start talking about the date and time of manufacture, the retail store that sold the block . . . where do we stop? At some point, properties become irrelevant to the problem at hand. Time to come back to earth and take into account what the program is to accomplish, not just what objects are in it. We don't want all properties, just relevant properties.

Assume that our program allows placement of blocks either on a surface, such as a table, or connected to one another. Color is important because the program draws the configuration on a color monitor. However, we don't need photorealistic drawings, so the smoothness property doesn't really matter. Certainly we need to know the locations of all connectors on each block in order to tell when connections are possible. As to properties of connectors, we need only know enough information to say whether a given pair actually connects.

We now need methods to go with these attributes. Since all objects should draw themselves, a draw method is in order. For a connector we would like a method that tells us what, if anything, it is currently connected to. We might want to create methods that allow our blocks to move themselves around or connect and disconnect with other blocks.

This, of course, is just a quick sketch. We didn't invent any objects or classes but picked from those already in the problem. We also picked and chose properties based on what our program is to accomplish rather than elaborating everything possible. We also engaged in a little projection. In real life, there would exist an agent of some sort that would cause movement to take place. Because we have an invisible agent in this program, we merely project the illusion that the block is moving itself. At the risk of stretching a point, imagine a block blown about by the wind. Absent knowledge of the wind, it would appear to an observer that the block moves itself. Since some agent still causes the change by calling the move method, our model of causality is still the natural one. For that matter, methods like draw fall into the same category. Nonetheless, we really didn't invent anything; we just discovered objects, classes, attributes and behaviors.

▶ Discovering Relationships

In the real world, relationships exist between objects. One kind of relationship is that of a whole to its parts, as shown in Figure 3-2.

We have already seen an example of this in the relationship between a BLOCK and a CONNECTOR. We can take this one step further. When we

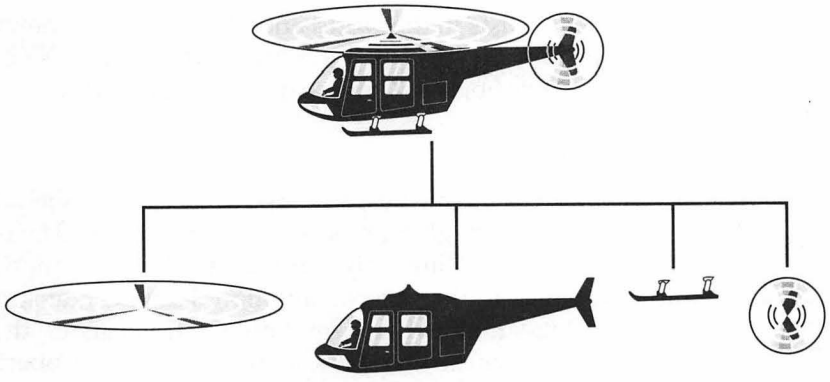


Figure 3-2. Wholes and parts

snap several blocks together, we create a single object whose parts are the individual blocks. This suggests that there is another class of object in the problem that we can call an ASSEMBLY. By examining relationships, we can discover further objects and classes.

Recall that objects communicate or interact with one another. We said that calling a method is commonly called "sending a message" to the object. This suggests another type of relationship between objects, based on the type of communication taking place. For example, one block may send a message to another that it is now disconnecting, perhaps as part of a move operation. These are relationships that depend on the objects; the relationships generally have some name. Some examples follow.

1. An object can have OWNERSHIP of another object. This means that if the owning object is deleted, the owned object is deleted as well. This frequently is a special case of WHOLE-PART relationship.
2. An object can be an ANTECEDENT of another. This may apply if we are interested in different versions of an object over time.
3. In spatial applications, such as our blocks world, we can have other relationships such as IN FRONT OF or ABOVE.

▶ Discovering Classes

Objects group into classes based on their shared properties. Similarly, classes group into superclasses based on their shared properties. We can take advantage of this fact in creating object-oriented programs by mapping natural world classes onto object classes. This can be viewed as a special kind of relationship between objects and classes: membership.

Each object exists in the world and also exists as a member of some class. The real objects, such as Joe Smith, President of XYZ Corporation are called *concrete* objects. They exist in the real world.

Abstraction

Concrete objects can be combined into classes such as "presidents of companies," through a process called *abstraction*. The dictionary defines abstraction as dealing only with relevant information, ignoring details not important to the present situation. However, abstraction has a more restricted meaning in the formation of object classes: the identification of classes, all of whose members share some set of properties.

Abstraction is a fairly mechanical process, as shown in Figure 3-3. Suppose we have two objects, one with properties A, B, and C, the other with properties B, C, and D. There is thus a natural class encompassing both objects defined by properties B and C.

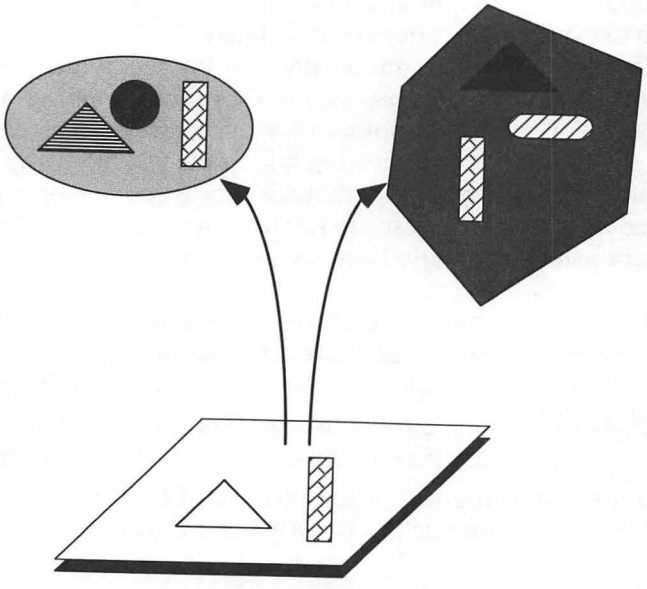


Figure 3-3. Abstraction

Specialization

Specialization is the opposite of abstraction and always starts with a class. Let's take our above example in reverse. We have a class with properties B and C. We notice an object with properties A, B, and C and see that the

overlap is not complete. The object is a specialization of the class, with one additional property, A. A cornerstone of objectivist methodologies is, when faced with a class, to use specialization to derive subclasses and so forth until the objects themselves pop out.

Concrete Classes

This leads to an interesting side note. In most OOP languages, it is impossible to directly describe a single object. Instead, the entire description of an object must be contained in its class. This means that we cannot map real world objects to program objects without using a *concrete class* as an intermediary. In an object-oriented program, a concrete class is like a template or mold for creating objects. The concrete class is a complete description of one or more concrete objects and forms the bottom of our inheritance hierarchy. Since we naturally classify and our classes naturally occur in the real world, all we have to do is use the natural way we think to judge our objects as concrete or abstract, then find abstractions for concrete objects and derive concrete objects from abstractions.

Three Ways to Discover Classes

Let's recap the ways we have discussed to discover classes.

- We intuitively recognize the existence of a class. Typically, we then derive subclasses and so forth until reaching objects through specialization. In our blocks world example, we concluded without much trouble that there is a class of objects called BLOCK. We told ourselves that we were really discussing objects, but we already knew that the class existed. This ability to perceive classes is part of our inborn cognitive ability and much of the success of the object-oriented approach rests upon it.
- We derive a class through abstraction of one or more objects or subclasses, by comparing their properties.
- We derive a subclass as a specialization of some other already-known class.

▶ Objectivist Methodology

Although methodologies for object-oriented software development are still in their formative stages, a good deal of commonality has already emerged.

▶ Basic Steps

In *Object-Oriented Software,* Ann Winblad lists the following common sequence of steps.

1. Identifying and defining objects and classes.
2. Organizing relationships between classes.
3. Cultivating frameworks in a hierarchy of classes.
4. Building reusable classes and application frameworks.

In Object-Oriented Design, With Applications, Grady Booch, one of the most widely-respected experts on object- oriented design, follows the same general lines.

1. Identify classes and objects at a given level of abstraction.
2. Identify semantics of objects and classes.
3. Identify relationships among classes and objects.
4. Implement these classes and objects.

These and other writings all urge the same basic three-step approach: Discover the objects and classes, discover their relationships, then implement. Of course, the cornerstone is the discovery of objects and classes. This remains a relatively informal process; some would call it a black art. For example, a currently popular technique is the *lexical* approach. Write a verbal description of the problem; the nouns will be your object classes, the verbs the methods. This approach depends heavily on the analyst's writing skills and knowledge of the problem, but despite the lack of formality, practitioners continue to flood the trade press with good results.

Once the low-hanging fruit of a problem has been harvested, specific techniques can be brought to bear on the hard-to-reach yield. Abstraction and specialization can be used to extend class hierarchies and encapsulation and various measures of what is a "good" class can be used to refine the design. Also, there is no end of notational conventions for communicating design ideas once they are discovered. Yet, most current methodologies are still based upon the assumption that we are intrinsically capable of discovering the objects and classes of the real world. The methodologies simply try to capture this intrinsic ability and put it to practical use.

▶ The Comfort of the Objectivist Approach

Let's recap the objectivist approach. Object-oriented programming is natural because the world is made up of objects that are parts or wholes and members of classes. People have been using these ideas for several thousand years in areas such as biology, zoology, and mathematical logic. We build on this long tradition, our innate abilities, and the nature of the physical world to find objects, define relationships, build class structures, and ultimately build a program.

This approach is based on the way that most people think that we think—the objectivist philosophy we spoke of earlier. Objectivism is what George Lakoff, in *Women, Fire and Dangerous Things,* has called a "folk theory" of human thought. It explains to a large degree why object-oriented software has become so popular so quickly. Object-oriented programs are, after all, organized along just these lines: distinct objects, grouped into classes according to shared properties. Methods and data members are the properties, object classes the classes. This concept of software organization resonates so deeply because it is a direct metaphor for the way we perceive our own perceptions of the world. Thus, people new to OOP find themselves making remarkable progress when they first approach it because they need do little more than mimic their own perceptions of the world in software.

This is the approach that natural scientists have used to develop taxonomies. Biologists and zoologists place an animal or plant in its appropriate subspecies, species, genus, and so forth, based on the properties of the organism. Each level of the taxonomy becomes more and more complex, until we reach a point were we have defined particular plants or animals.

The objects in our programs are analogous to real world objects, the relationships between objects analogous to the real world relationships. Taking a page from the success of the natural world scientists, we use this taxonomic approach and take the shared properties that objects have and turn them into classes.

▶ Program Evolution and the Four Itys

So far, we have talked only about the success of modeling the real world using objects when we start from scratch. There are, however, a number of side benefits that extend beyond the initial implementation. Let's call these the *Four Itys: modularity, extensibility, maintainability,* and *reusability.*

Modularity

We talked a little in Chapter 2 about the advantages of coupling procedures and the data on which they operate. For purely technical reasons, this, in itself, yields better modularity than traditional programming. However, remember one of our earlier corollaries about objectivism: A given object or class in the real world is independent of all others. If I pick up a rock lying in a meadow and put it on top of a mountain, the properties of the rock do not change. Objectivism and OOP draw on this to achieve a much deeper level of modularity. The real world has been around longer than any computer and has had a lot longer to settle issues of modularity. We observe and learn and, in the process, divide our program according to the relatively stable principles of nature.

Extensibility

This is a purely technical advantage that results from encapsulation, especially if we make all data members private. Because every class is defined entirely by its public interface, we can adapt our program to changes and extensions by changing the implementation of methods internally to existing classes. In theory, this creates no side effects because the implementation is hidden.

Even better, we can always create a subclass that adds any additional methods and attributes we may need without changing the existing class. By using inheritance, we can extend classes while leaving the existing classes alone.

Maintainability

This is largely the result of three factors: modularity and extensibility, as discussed above, and the stability of real world objects and classes. The real world seems to have a good deal of stability and consistency. By directly modeling it, we provide a sounder foundation for our programs. Recall from Chapter 1 that a large proportion of changes to programs are adaptive and perfective in nature; that is, they are refinements that make the program better fit the real world. If our software is already structured similarly to the real world, the changes should amount to simple extensions, not changes. We expect new methods, more capability in old ones, new classes, and new objects. We do not expect major changes or wholesale deletions of old methods, objects, and classes; most changes will be extensional. Since traditional programs are too far removed from the structure of the real world, their structure depends on the particular problem you are trying to solve. The structure is likely to come tumbling down when the problem changes or expands.

Reusability

Code reuse is one of the time-honored goals of software. Ideally, we can build a library of software over time and reuse it for new projects. We should be writing less and less code for new projects as we reuse more from our library. In traditional software, this goal is seldom achieved, despite the prominent position it holds in development methodologies. Can we do better with object-oriented software development?

The independence of objects and classes looms large here. If we have done a good job of describing a real world object or class in one application, the result should be reusable in any other application that requires the same object or class. After all, since each object and class stands on its own, it should be like moving a rock to a new setting. We would hope, for example, that an employee class created for a payroll application would be reusable for a later personnel scheduling application. Reusing incomplete objects and classes should be simply adaptive or perfective. Over time, our object library should come to approximate the real world better and better. As long as our projects have some degree of overlap, we should be able to write less code for each new project. Finally, on a purely technical level inheritance and polymorphism are very powerful tools for reusing stock libraries; we can keep what we want and throw out the rest.

Of all of the benefits cited for OOP, code reuse is one of the most talked about. It has the potential to drastically reduce costs and improve quality over time and can go a long way toward recovering the cost of the shift from traditional to OOP technologies—if it works.

▶ Problems With Objectivism

This is the folklore, but what is the reality? Somehow the objectivist approach seems too easy. All you have to do is pick out the objects, implement them, and out pops an object-oriented program with all the benefits of good design and the Four Itys. As we will see, this view is not really wrong, but it is simplistic. The folklore does, in fact, work very well for simple programs and simple problems. That it breaks down on more complex projects will surprise few; the reasons, we suspect, will surprise many.

In the next chapter, we will see how this naive view works on two realistic applications representing computer-aided design and business systems. As we will see, the simple approach can work well, but often does not. In Chapter 5, we will locate the source of the problem: *People may think they think in objects, but in reality they don't!* We will explain when to expect the folklore to work well and when to expect trouble. Chapter 5 is

followed by a series of chapters that reconcile the appeal of the folklore with the realities of OOSD through the authors' Solution-Based Modeling (SBM) methodology.

Have the authors wasted their time and yours with this chapter? Not really; remember that the objectivist approach does work for simple projects and many projects are, in fact, simple. Even when it doesn't work, objectivism is far from a complete failure. There is a strong parallel here to the difference between Newtonian and Einsteinian physics. Newtonian mechanics says nothing about time itself being affected by speed, but in most circumstances, who cares? How many of us will ever travel close to the speed of light in the ordinary course of affairs? Newtonian mechanics is not wrong, just imprecise in some reference frames. So it is with objectivism: It is a useful approximation that deserves to be understood as such.

Most of the goals and claims for object-oriented software development that are based on objectivist philosophy are achievable using a more accurate model for the way people think. The authors have had a great deal of success in object-oriented software projects. Our experience has shown that OOSD can provide better results both on initial development and during the maintenance life cycle. It is absolutely essential before suggesting a new way of developing software to thoroughly explore the current state of the art. Because objectivism is so ingrained in our thinking about our thinking, it is necessary to contrast it point by point against any alternative.

▶ Summary

- *Objectivism* is the commonly held belief that the world consists of objects and that those objects naturally group into classes based on shared properties. This corresponds closely to the way object-oriented software is organized. There are two tenets of objectivism: that there is a real world "out there," independent of any one person's perceptions, and that that world is composed of discrete objects, each of which has properties that characterize it.
- People come prewired with certain abilities to organize perceptions of their world, such as making distinctions, creating mental images to represent those distinctions, and perceiving relationships. The notion of an object as a mental building block underlies all of this.
- Object-oriented analysis (OOA) seeks to capture requirements and specifications in terms of objects and classes. This approach makes us feel comfortable since we are leveraging what we already do

naturally. According to the folklore, our goal is not to create objects and classes but to discover them. Similarly, we discover relationships among the objects and classes. We need not capture everything there is to know about the real world. We don't want all properties, just those that are relevant to our program.

- Objects group into classes based on their shared properties, and classes group into superclasses based on their shared properties. Abstraction is the process of discovering new classes based on observing the sharing of properties among multiple objects or other classes. Specialization is the opposite process in which we discover subclasses that contain additional properties not present in their superclass.
- There are several ways to discover classes. We can intuitively recognize the existence of a class, use abstraction, or use specialization. In an object-oriented program, a concrete class is like a template or mold for creating objects when the program runs. The concrete class is a complete description of one or more concrete objects.
- Object-oriented software methodologies commonly consist of three generic steps:
  1. Discover classes and objects.
  2. Discover relationships among classes and objects.
  3. Implement the classes and objects.
- The objectivist approach is based on the way that most people think that we think. This explains to a large degree why OOP has become so popular so quickly. Although it has problems, the objectivist approach can work very well for simple programs and simple problems. Even on larger projects, it is not really wrong, but oversimplified. Beyond these cognitive reasons to use object-oriented approaches, there are four classical goals of software engineering that are well satisfied by the folk theory of OOSD: modularity, extensibility, maintainability, and reusability.

4 ▶ Sample Applications (Why Aren't They Easy?)

▶ What This Chapter Is About

This chapter outlines the development of two applications as object-oriented programs. The first application concerns computer-aided design (CAD) for creating model railroad layouts. In theory, this application should fit quite well with the folklore because it involves physical things with well understood behaviors in the real world. In practice, it is not so clear how to proceed. The principal challenges are knowing where to stop in the modeling process and how to organize the classes into an inheritance hierarchy.

The second application is a case study of a relatively naive programmer tackling an object-oriented payroll application, armed only with his talent and the folklore of object-oriented software development. This is decidedly less physical than the CAD application, because it deals with more abstract business entities and concepts. The principal challenges in this application are picking out the objects and deciding where to place the behaviors.

We will not go through the entire development process. Instead, we will skip around and highlight the ways in which the folklore succeeds and fails. As you will see, much of the objectivist folklore starts to crack when used for more complex projects.

▶ Model Railroad Computer-Aided Design

Here's the letter Sandy sent to Jean asking her to create a program that could be used to design model railroad layouts. Figure 4-1 shows her drawing of the model.

Jean,

You know, I've been thinking about your suggestion the other day to use the Macintosh to design model railroad layouts. For my more experienced customers, it would be a great way to design complex layouts. It would also make it easier for beginners to get going, since they could make all their mistakes on the computer where it doesn't cost anything. If you're interested in creating the program, I'm interested in funding it, provided it doesn't cost too much.

Here's what I have in mind. The user should be able to start with a blank table top of whatever size he wants. The program should let him add scenery, track, controls, trestles, and so forth; the results should display on the screen. At any time, the program should be able to print out the diagram currently on the screen, along with a bill of materials ready to bring to my store. I have included a mock-up of what the screen might look like.

The program has to be able to handle tracks that pass over one another on trestles and bridges, or through tunnels. It should allow for mountains and lakes made out of papier-mâché or clay or whatever, not just finished goods off my shelf. I want it to include all the different kinds of track and scenery I sell. That's a long list and it changes all the time. I need some way to send customers a new catalog on a floppy disk or something. I've enclosed a current catalog (not that *you* really need another one). It would be nice if the user could choose from a library of standard designs that I will supply. Even better if more than one design could be merged together!

Did I mention trains? I want the program to include the cars themselves. You should show an actual picture of each type of car I sell, but only for the gauge of track the designer has chosen for the layout. The ultimate would be to simulate the train running around the track, with switches and controls operating in the program the way they do in real life. If you can do that, I can sell a lot of equipment to people who otherwise might not buy.

Remember: Since the people who will use this probably don't know much about computers, the program has to be really easy to use. I like the Macintosh for this. If the user sees something on the screen, he or she can use the mouse to point to it and move it around. Be sure you keep that way of interacting in mind.

What do you think? Can you do it, how much will it cost, and how long will it take?

Sandy

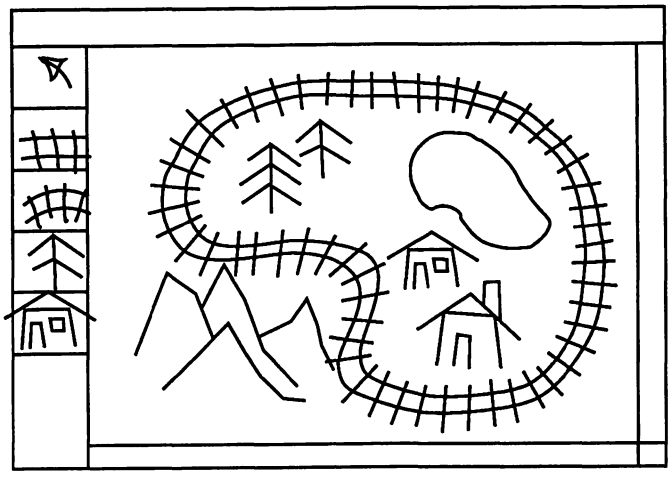


Figure 4-1. Model railroad layout

▶ First Try: Lexical Analysis

This sounds pretty easy. The program is concerned with physical things such as tracks, cars, and scenery. Let's see how well the lexical approach discussed in Chapter 3 works here.

Finding Objects and Classes

We start by identifying the objects and classes in the letter by underlining the first instance of all nouns that imply some requirement for the program (this requires exercising a little judgment.)

> You know, I've been thinking about your suggestion the other day to use the Macintosh to design model railroad layouts. For my more experienced customers, it would be a great way to design complex layouts. It would also make it easier for beginners to get going, since they could make all their mistakes on the computer where it doesn't cost anything. If you're interested in creating the program, I'm interested in funding it, provided it doesn't cost too much.
>
> Here's what I have in mind. The user should be able to start with a blank table top of whatever size he wants. The program should let him add scenery, track, controls, trestles, and so forth; the results should display on the screen. At any time, the program should be able to print out the diagram currently on the screen, along with a bill of materials ready to bring to my store. I have included a mock-up of what the screen might look like.

The program has to be able to handle tracks that pass over one another on trestles and bridges, or through tunnels. It should allow for mountains and lakes made out of papier-mâché or clay or whatever, not just finished goods off my shelf. I want it to include all the different kinds of track and scenery I sell. That's a long list and it changes all the time. I need some way to send customers a new catalog on a floppy disk or something. I've enclosed a current catalog (not that you really need another one). It would be nice if the user could choose from a library of standard designs that I will supply. Even better if more than one design could be merged together!

Did I mention trains? I want the program to include the cars themselves. You should show an actual picture of each type of car I sell, but only for the gauge of track the designer has chosen for the layout. The ultimate would be to simulate the train running around the track, with switches and controls operating in the program the way they do in real life. If you can do that, I can sell a lot of equipment to people who otherwise might not buy.

Candidate Objects and Classes

LAYOUT	RESULTS	CATALOG
CUSTOMER	DIAGRAM	LIBRARY
BEGINNER	BILL OF MATERIALS	DESIGN
MISTAKE	STORE	TRAIN
USER	BRIDGE	CAR
TABLE TOP	TUNNEL	PICTURE
SIZE	MOUNTAIN	TYPE (of TRAIN CAR)
SCENERY	LAKE	GAUGE
TRACK	PAPIER-MÂCHÉ	DESIGNER
CONTROLS	CLAY	SWITCH
TRESTLES	FINISHED GOODS	EQUIPMENT

Even before looking at the catalog and other sources of information, we have quickly identified 33 candidates. Obviously, there is some overlap within this list and there are a few terms that we can obviously discard. For example, CUSTOMERS are the same as end USERS and DESIGNERS, and BEGINNER is a type of CUSTOMER. There are also some relationships among these nouns. MOUNTAINS and LAKES are specific kinds of SCENERY, EQUIPMENT covers a lot of other terms, and a SWITCH is a kind of TRACK. After some sorting and sifting, the list can be arranged in an outline in which a slash (/) indicates synonyms.

LAYOUT/DESIGN/RESULTS/DIAGRAM

USER/CUSTOMER/DESIGNER/BEGINNER

FINISHED GOODS
- EQUIPMENT
  - TRACK
    - SWITCH
  - CONTROL
  - TRESTLE
  - BRIDGE
  - TRAIN
  - CAR

SCENERY
- MOUNTAIN
- LAKE
- TUNNEL

MISTAKE

TABLE TOP

BILL OF MATERIALS

PAPIER-MÂCHÉ

CLAY

CATALOGUE

LIBRARY

PICTURE

TYPE (OF TRAIN CAR)

Notice that we made a few somewhat arbitrary calls. For example, do we classify BRIDGE as EQUIPMENT or SCENERY or both? What about a TUNNEL? A TRESTLE? Is a TYPE OF CAR an object? Is a SIZE an object or just an attribute of the TABLE TOP? Is GAUGE an object or an attribute of a piece of TRACK? The right answers are not at all obvious.

The more experienced OOP analyst recognizes some groupings, or abstractions, that are implicit in this list but not named in the requirements. For example, since PAPIER-MÂCHÉ and CLAY are clearly related, we might want to group them under MODELING MATERIAL. They share certain properties: both can be formed into an arbitrary shape, then hardened to retain that shape. Even though we have not yet listed such properties, it is useful to grab the low-hanging fruit at an early stage. The same analyst also forms relationships among classes that are not abstractions of one another, based on one-to-one and one-to-many associations. One TRAIN has many CARS and one LIBRARY has many DESIGNS. These are all well within the realm of what we perceive as real-world characteristics.

Finding Methods

In order to discover methods, we start with the verbs in our requirements and associate them with the nouns they reference.

You know, I've been thinking about your suggestion the other day to use the Macintosh to design model railroad layouts. For my more experienced customers, it would be a great way to design complex layouts. It would also make it easier for beginners to get going, since they could make all their mistakes on the computer where it doesn't cost anything. If you're interested in creating the program, I'm interested in funding it, provided it doesn't cost too much.

Here's what I have in mind. The user should be able to start with a blank table top of whatever size he wants. The program should let him add scenery, track, controls, trestles, and so forth; the results should display on the screen. At any time, the program should be able to print out the diagram currently on the screen, along with a bill of materials ready to bring to my store. I have included a mock-up of what the screen might look like.

The program has to be able to handle tracks that pass over one another on trestles and bridges, or through tunnels. It should allow for mountains and lakes made out of papier-mâché or clay or whatever, not just finished goods off my shelf. I want it to include all the different kinds of track and scenery I sell. That's a long list and it changes all the time. I need some way to send customers a new catalog on a floppy disk or something. I've enclosed a current catalog (not that you really need another one). It would be nice if the user could choose from a library of standard designs that I will supply. Even better if more than one design could be merged together!

Did I mention trains? I want the program to include the cars themselves. You should show an actual picture of each type of car I sell, but only for the gauge of track the designer has chosen for the layout. The ultimate would be to simulate the train running around the track, with switches and controls operating in the program the way they do in real life. If you can do that, I can sell a lot of equipment to people who otherwise might not buy.

Candidate Methods

DESIGN (LAYOUT)

MAKE (MISTAKE)

START (TABLE TOP)

ADD (SCENERY, TRACK, CONTROLS, TRESTLES)

DISPLAY (RESULTS)

PRINT (DIAGRAM, BILL OF MATERIALS)

HANDLE

ALLOW FOR (MOUNTAINS, LAKES, things made of PAPIER-MÂCHÉ and CLAY, FINISHED GOODS)

INCLUDE (FINISHED GOODS)

CHANGES (CATALOG)

SEND (CATALOG)

CHOOSE (DESIGN)

MERGE (DESIGN)

SHOW (PICTURE)

CHOOSE (GAUGE OF TRACK)

SIMULATE

RUN (TRAIN)

OPERATING (SWITCHES, CONTROLS)

It looks like this list isn't going to be very useful. INCLUDE as a method? ALLOW_FOR? HANDLE? This last is even more vexing because a literal reading is not the correct interpretation. To "handle track that . . ." does not mean that it is TRACK that must be handled. It is overall patterns of use of TRACK, TRESTLES, BRIDGES, TUNNELS, and the like that must be HANDLED. There is no object to represent entire scenarios of this sort, nor is there an intuitive one available. And just what is it that we SIMULATE? Again, it is the operation of an overall system of TRACK and so on that is the subject. The same is true of the verb RUN. We must associate these verbs either with the DESIGN as a whole (SIMULATE, RUN) or with the program itself (INCLUDE, HANDLE.)

Problems with Lexical Analysis

Still think it's easy and intuitive? Proceeding purely from our user-supplied requirements, we have already seen examples of the following common phenomena.

1. It is often not clear whether a noun is relevant or not. Is a BEGINNER somehow different from a USER or DESIGNER and, therefore, worthy of separate consideration?
2. It is often not clear whether a noun represents an object, a class, or just an attribute of some other class. SIZE and GAUGE are good examples.
3. Using verbs as templates for methods can range from useless to a rough starting point, but that is about it.
4. Verbs often refer to implicit objects not named by any noun.
5. Those implicit objects are often complex and non-intuitive, which is exactly the opposite of what we expected of the lexical approach! For example, a DESIGN is certainly an overloaded concept since it can have different meanings depending on the context.

Taken as a whole, these problems mean that our user's description of the problem does not lend itself well to being modeled as an object-oriented program. This does not condemn the lexical approach or the intuitive nature of object-oriented analysis; perhaps it means we haven't gone far enough yet.

▶ Second Try: Top-Down Analysis

Clearly, the requirements we have been given are not enough and more study is in order. But how do we proceed? Building from the bottom up is not the answer. Fully enumerating all possible objects is just too time consuming. The catalog would be huge, with hundreds of thousands of items, each a candidate object. Dealing with every possible item and attempting to build a class hierarchy on top we would never finish. There are so many objects that discovering relationships among all of them at once will be like trying to take a sip from a fire hose. Bottom-up methods won't work.

Top-down methods work a little better. We can start with high-level classes such as EQUIPMENT and SCENERY and drill down into more specific classes. Within EQUIPMENT, we have TRACK, CAR, SWITCH, CONTROL, and so on. But how do you break down SCENERY into more specific classes? Is a LAKE different from a MOUNTAIN (both are fashioned out of some modeling material)? For that matter, is PAPIER-MÂCHÉ a superclass of a MOUNTAIN or just somehow related to it?

How Do You Know When to Stop?

How do we know when to stop reaching into the treasure chest of detail? Do we really care enough about the differences to distinguish a class TREE from a class HOUSE? After all, we can readily come up with a long list of attributes of each, most of which are not shared.

Tree	*House*
HEIGHT	STORIES
DIAMETER	AMPERAGE (FOR LIGHTS)
FOLIAGE	COLORS
HEIGHT TO LOWEST BRANCH	TYPE OF ROOF SHINGLE
	SQUARE FEET (LIFE SIZE)
	SCALE

All of these are differences, but there is nothing in the catalog or in our intuition that tells us what is relevant and what is not. Nor do the questions stop with objects, classes, and attributes; it is not always obvious what methods are relevant and where methods stop and attributes begin. For example, consider connections between EQUIPMENT such as TRACK and TRESTLES. We can describe the precise geometry of the CONNECTORS and thereby capture the real-world characteristics pretty well, but

that is not enough. If a CONNECTOR sits empty, we want to take notice of that fact as a probable shortcoming in the design (we can't have a train run off into oblivion!). Yet, there is no "I'm not connected" property to a piece of track in the real world; it springs solely from our program and our knowledge of its intended use. How can we account for the concept of *desired* behavior when we are armed solely with the ability to "see the world as it really is?"

Simulating the Real World Is Not the Answer

This is a hole in the folklore of OOP big enough to drive a truck through. The folklore holds that we need merely simulate real-world characteristics. However, we can't simulate *everything* if we expect to get any project done on time and in budget! There is always infinite detail available, even on casual examination, and we simply must choose a very small subset of that which is possible. Since even considering detail that is later thrown out is very expensive, we must be able to focus in on the "right" properties and objects quickly. *Yet, there is nothing in the real world that tells us where to prune or even how to tackle the issue. In other words, there is no methodology for OOP development that can be based solely on "discovering the real world."*

▶ Third Try: Put It in Context

Perhaps we are going overboard. Clearly there is a role for experience here. A skilled analyst should be able to look at requirements through the lens of the application's needs and discard detail from the real world that does not directly relate to actions of the program. Thus, the requirements of the program act as a filter against the real-world characteristics and tell us what to keep and what to discard. A good analyst should spot the kind of implied classes and relationships we discussed earlier. A good analyst should also be able to draw out more detail from the user and comb other sources such as the catalog for more information.

This helps a great deal with physical properties. It might help us discover, for example, that we really don't care about the differences between most kinds of scenery, such as TREES and PARK BENCHES since the user does the same thing with both items: ORDERS them, PLACES them somewhere in the DESIGN, or VIEWS a picture on the screen. Other properties, which distinguish one from the other, certainly exist in the real world, but just aren't relevant to this program. One class may do for all.

Programs Do More Than Simulate

Context alone, however, does not tell us about how to handle our CONNECTOR problem. "Disconnectedness" is not a real-world property of a piece of TRACK, unless you stretch the point. It depends on the intended use of the product and the activity—layout design—of the observer, not purely on physical properties. *There are always going to be artificial features of a program that do not spring directly from the real world, and the folklore provides no theory to account for them or methodology for discovering and designing them.*

The clear message is that discovering real-world objects and classes is not enough to formulate our design. There must be an infusion of other principles as well.

▶ Fourth Try: Ask an Expert

Let's jump in here with a typical "expert" treatment of this problem. One expert faced with the problem of "disconnectedness" in this sample application developed the following three-prong approach:

1. Virtually everything in the layout is a subclass of EQUIPMENT, including TRACK, TRESTLE, and CONNECTOR. This expert even made SCENERY a subclass of EQUIPMENT.
2. A single EQUIPMENT object can own one or more CONNECTOR objects.
3. Each EQUIPMENT object has a CHECKFORERRORS method that reports all anomolies to the screen. The default method simply calls the CHECKFORERRORS method of each of its owned CONNECTOR objects. The CONNECTOR class overrides this method to report an error if and only if the CONNECTOR object is not connected to a compatible CONNECTOR object.

This is probably as good as anything, but where did it come from? There is no such error-state property in the real world! And what of this idea of "ownership?" The pin in the end of a piece of track has the same characteristics in the real world whether you leave it in the track or remove it from the track and throw it on the ground. This is our "rock on the top of the mountain" principle from the folklore. The real connector pin does not depend on the track for its existence, yet in the expert design the CONNECTOR object *does* exhibit dependence on the track.

The expert may also develop concepts such as TRACK that LAYS itself, DESIGNS that VERIFY themselves, EQUIPMENT that knows how to DRAW itself on a computer screen, TABLE TOPS that know how to ask the user for their

SIZE . . . you get the idea. Although this would generally be considered "good" design, these are not concepts drawn from the real world! Show them to a typical end user and claim they are the "real world" and you might just get locked in a padded room. The "things" are real, but the behavioral properties are artificial.

Enter Anthropomorphism

In Chapter 2 we introduced the concept of anthropomorphism whereby human qualities are ascribed to non-human things. This is a much closer description of what the expert does intuitively than real-world analysis alone. Real-world analysis can help yield the "things" and certain properties, such as physical dimensions. Many properties, particularly behaviors, must come from elsewhere. If we go back to the drawing board with this in mind, we will find the going much easier. The real world yields the "things," and we project onto those things the abilities to do that which the user of the program wants done. Suppose the user clicks on a piece of track and moves the mouse. The intention is to move the track along with the cursor. We can project this onto the track as a behavior: mouse tracking. The user wants to make sure that the overall design is without major flaws, so the design itself can be empowered with the ability to self-examine.

Outline of a Methodology

This suggests the outline of a methodology.

Step 1. Discover the real-world objects and classes, along with relevant real-world properties such as dimensions, color, and so on.

Step 2. Enumerate the behaviors expected of the program.

Step 3. Project those behaviors anthropomorphically onto the objects.

Of course, we expect to apply these steps iteratively. This approach keeps much of the intuitive appeal of the objectivist school because we still base our objects and classes on the real world. At the same time, we take into account the specific requirements of the program and allow real-world objects to absorb artificial behaviors that derive from those requirements.

The key word "relevant" slipped into the first step and requires further exploration. Just how do we determine what is relevant and what isn't? The answer is that we can't make that determination until we have considered the desired behaviors of the program. A property is relevant

when it is used to accomplish something. Even such "obviously relevant" properties as dimensions are relevant only if they are needed to draw an image, verify connections, or carry out some other behavior that derives from the requirements. Thus, we need to use the program requirements as a filter, not just for objects, classes, and behaviors, but for all other properties as well. The result may well be to combine classes that are distinct in the real world but have only common properties and behaviors for the purposes of the program, as in our example of TREES and PARK BENCHES.

Consequences of the Expert Approach

Using this methodology, we can make great progress provided that everyone understands what we are doing. We can avoid confusion by not portraying the design as the real world but rather as a part of the real world plus a metaphor (anthropomorphism.) Since anthropomorphism and metaphor are things we all do well (OOP programmers and model train designers alike), this should be an acceptable explanation. It is intuitive, but not necessarily natural; that is, the process is comfortable, but the results are not a direct reflection of nature.

One painful loss, however, is the idea of reusability. Remember that we tied reusability to the autonomous nature of objects in the real world. Projecting behaviors that do not really exist in the real world onto our program objects compromises this autonomy. As we project more behaviors our classes become less stable and reusable. This is unfortunate because the largest projects usually have the most requirements and, therefore, the least reusability.

▶ Designing for the Macintosh and Its User Interface

Hold on: we have only begun! So far, we have discussed the relationship of the real world to program objects and classes, but we have yet to consider the impact of our chosen computer and class library. Neither of these is part of the real world that we seek to automate: they are tools that *we* bring to the party. The "simple" problem is figuring out how to draw all of the things in the program since a good Macintosh program visualizes its contents. This is a quite natural extension of the folklore in this example since we are dealing with real things that we can see and, therefore, reproduce visually. The more complex problems arise from conforming to the rest of the rules separating "good" from "bad" Macintosh applications.

Macintosh User Interface Features the Folklore Never Told You About

The Macintosh owes much of its popularity to Apple's published human interface standards. Things like windows, scroll bars, icons, and menus are just the beginning. There are metaphors that must be supported, such as direct manipulation of items on the screen. Clearly, we wish to implement these features using object classes, but where do these classes come from and how should we discover them? Certainly not from the real world we are modeling! Without a real world to classify, the folklore is no help. In its place, we must rely on experience with designing Macintosh user interfaces—yet another black art mastered by only a select few.

Just how many classes are we talking about here? Is it mere icing on an application which otherwise is based heavily on the real world, or is it a dominant aspect of programming the Macintosh? Take a look at Figure 4-2, which shows just a few user interface-related classes of MacApp, and judge for yourself. The folklore cannot, by its very assumptions, give us much help in laying out the user interface, yet clearly that is one of the most important things we must do to create this application.

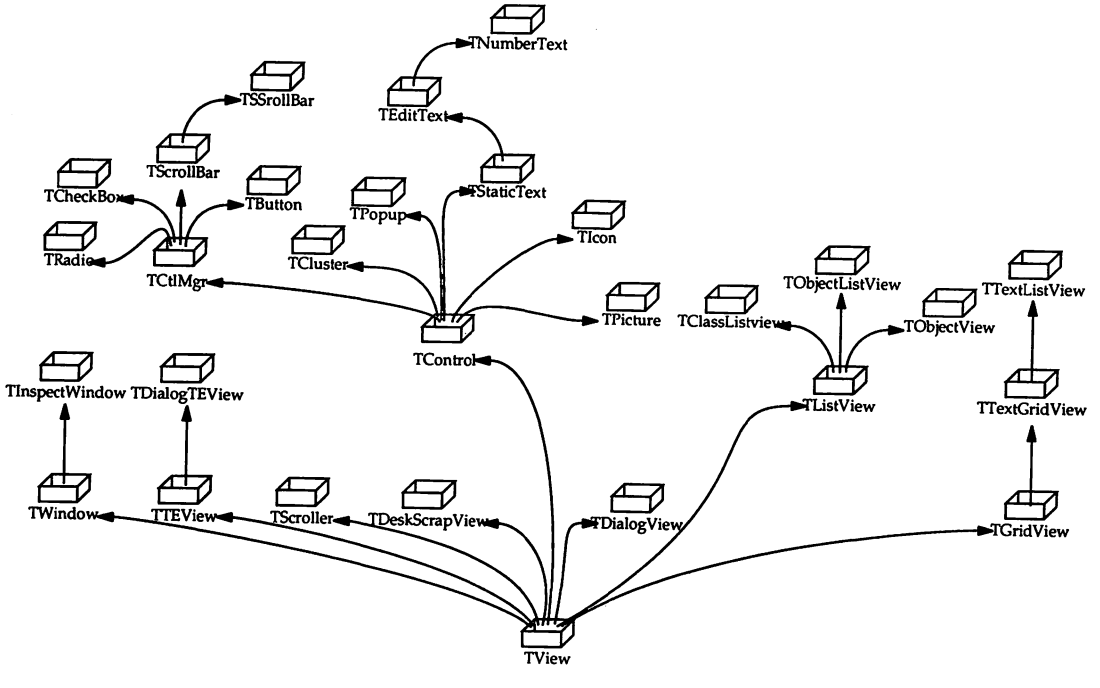


Figure 4-2. MacApp user interface classes

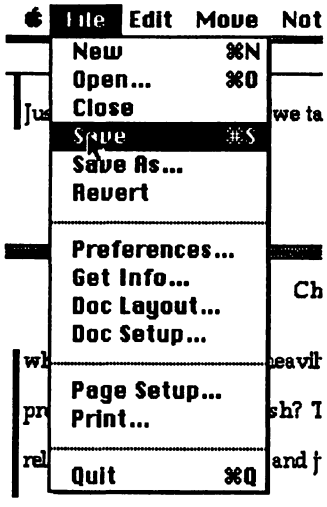


Figure 4-3. The file menu

Documents

Nor is the problem limited to visual direct manipulations. There are organizational issues, such as the concept of a *document*. Using a Macintosh, one should be able to double click on a file in the Finder, launch the file's creator (a program), and bring the document up in a window. As Figure 4-3 illustrates, at all times we have options like "Open," "Close," "Save," and "Revert," which apply to the frontmost window and the data it represents, its document. Thus, rather than being another name for a file, a document is a fundamental metaphor that all "good" Macintosh applications must support through their user interfaces.

In the simplest cases, a document in the Macintosh is simply an analog to a piece of paper. Thus, in the railroad program, we will probably choose to make a layout into a document, so that the user opens, closes, and saves entire layouts. We will also need another kind of document to represent catalogs and perhaps others as well. But what about a window like that of Figure 4-4, which shows a single railroad car from the catalog?

Is this a document, or is it merely part of the catalog document? What about the BILL OF MATERIALS? Is that a separate document or part of the LAYOUT? Again, the folklore is no help.

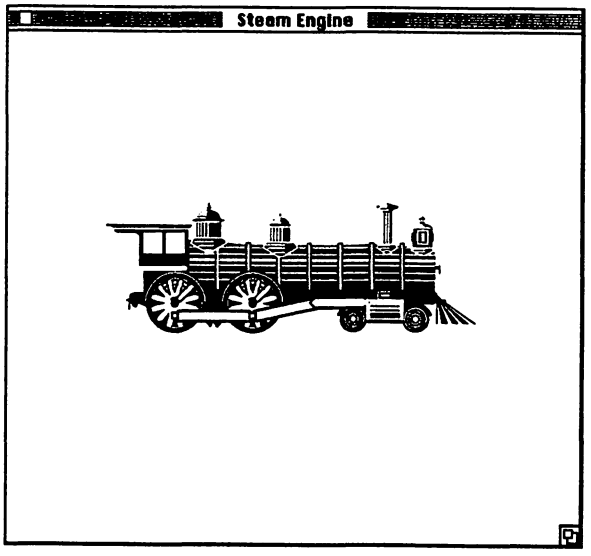


Figure 4-4. Is this a document?

Editing and the Clipboard

Everyone who has developed large applications for the Macintosh knows that supporting the clipboard—Cut, Copy, and Paste—as shown in Figure 4-5 is not trivial.

You would not, for example, allow someone to copy a catalog to the clipboard and paste it into a bill of materials. It simply doesn't make sense. Yet, even some odd combinations can seem sensible when viewed in the right way. Pasting one design over another might merge the designs. Pasting a picture of a railroad car over a bill of materials might add an order for one of that car to the list. Cutting does not always make sense, either. Do you really want to allow the user to be able to select something from the catalog and cut it? And how will you decide what to place on the clipboard for the use of other applications like Excel or HyperCard? Since these are not questions that spring from the problem and its real-world environs, the folklore does not cover them. The folklore does not account for the degree to which software development activities are dependent on the Macintosh, its user interface standards, and its software.

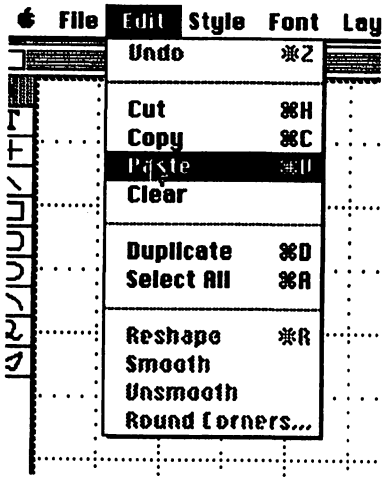


Figure 4-5. The edit menu

▶ Payroll

This is the story of Ace and his efforts to create an object-oriented payroll system. Ace is one of the best programmers in the business, though relatively new to object-oriented programming. There is little that he hasn't done in his fifteen years in the business: systems programmer on large IBM mainframes, programming manager for a Fortune 500 company responsible for distributed executive information systems, consultant, lecturer. Ace has extensive experience with conventional software engineering, from structured analysis and design to information modeling. In short, a pretty capable fellow.

Ace became enamoured of the Macintosh about four years ago after tiring of the mainframe world. Since then, he has mastered the Macintosh Toolbox, C, Pascal, and 68000 assembler for the Mac. He has garnered impressive experience in almost every aspect of programming the Macintosh, from specifications to designing and programming user interfaces. About a year ago, Ace decided to turn his talents to object-oriented software and, true to form, quickly devoured all the material he could find on the subject: Object Pascal, C++, MacApp, OOA, OOD, OOP. To Ace, object-oriented programming was no more than a convenient way to do what his training told him he should always do: Write programs in small, highly cohesive and only weakly linked modules, reusing code wherever possible. He had several small object-oriented projects under his belt when his story begins.

"My cousin Frank called and asked me to help him automate payroll for his company. He has about fourteen employees and his manual system was beginning to worry him. Frank looked at this as an opportunity, the first step in developing a company-wide information system. I suggested he simply buy an off-the-shelf accounting package, but he wouldn't hear of it. He had looked at all of them and none did exactly what he wanted, especially regarding company-wide integration, his eventual goal. You've seen it before: Users have no systems at all, then suddenly it's critical to have everything on line. I'd used object-oriented programming, but always for basically graphical programs, not a strictly data processing problem like payroll. It sounded like a challenge, so I talked to him."

▶ Current Business Model

Frank's survey of the way payroll was at that time done went like this.

1. Daily, employees fill out time sheets.
2. Every Friday, time sheets are collected. On Monday morning, Rachel takes each time sheet and places it in the employee's payroll file. This is a manila folder in a filing cabinet, one per employee. In this folder is the employee's employment application, other personnel information such as name, department, date hired, supervisor, and type of employee.
3. Frank has two types of employees, salaried and hourly. A salaried employee's payroll file contains a weekly salary; an hourly employee's file contains an hourly rate. As Rachel files each time sheet in the hourly worker's employee file, she adds the totals to a report summarizing hours worked, which, when finished, is given to Frank. For each employee, this report lists the hours that employee worked that week. For salaried employees she simply lists 40 hours, less any sick time. All hours worked by hourly employees are listed, separated into regular and overtime.
4. Frank allows unlimited sick time for salaried employees (they've all been with him for a long time and have earned this trust), but he does not pay hourly employees for hours not actually worked. Hourly workers are paid time and a half for overtime hours, but salaried workers do not receive overtime. Each employee is entitled to 1 day of vacation for each 160 hours worked, plus 1 day for each year of service, up to 28 days. Vacation can be accrued if not taken in a given year. It is up to each employee and his or her supervisor to keep track of vacation and to let Rachel know if an employee will be on

vacation. Usually this consists of a Post-it note in the employee's file. If an employee is on vacation, Rachel simply enters "vacation" on the hours worked report.

5. On Tuesday, Christine, who is in charge of computing payroll, takes each employee's file, computes wages, and using the IRS and state circulars, computes the deductions. She fills out a payroll check request form and puts it in the payroll check request file. She also computes the necessary information for the IRS bank deposits and places that in the IRS file.
6. Every Thursday, John who is in charge of actually writing the checks, takes the payroll check request file and writes a check for each payroll check request. In addition, John writes checks for the IRS and the state and fills out the IRS and state bank deposit slips. Quarterly the accountant fills out the necessary reports.
7. As he writes the checks, John creates a weekly payroll report with employee name, hours worked, and the amount of the gross pay. There is no blank for hours worked on the check request form, but Rachel is nice enough to put it there. This saves John a trip to the employee files to get the hours.

In addition, Ace wrote down all of the details of the calculations involved and a number of other factors.

▶ System Objectives

1. Frank is a little worried that vacation tracking is not being done accurately since it is a very informal process. Better accounting for vacation is a major objective.
2. Frank also wants more management reports regarding overtime. This, in turn, will be used to determine when to hire more people rather than paying time and a half.
3. He is also worried about the overall accuracy of the payroll computations. There are a lot of people involved and some redundancy, but Frank worries about dishonesty and wants to have at least two people looking at all computations.
4. Finally, this is to be the anchor tenant in a larger office automation effort. Eventually, Frank wants to be sitting at home next to his pool with his laptop Macintosh and a cellular modem, able to dial in to his computer system and get an immediate picture of his business.

▶ First Try: Simulation

"I thought, piece of cake, right? The first thing I did was to try and pick out the objects. It seemed obvious that the best thing to do was to map the process onto a series of objects. I decided to start with the objects in the current system then just add the ones I needed to implement the new features. I thought about what payroll was supposed to do and picked out the objects that would do that."

Ace developed an object candidate list with the following set of objects and their behaviors:

EMPLOYEE

- Fills out time sheet

TIME SHEET

- Holds hours worked

PAYROLL RECORD

- Holds salary or hourly rate, vacation information, hours worked, and so on

TIME SHEET FILER

- Takes the hours worked and puts it in the payroll record
- Creates Hours Worked Report and tells it to print itself

HOURS WORKED REPORT

- Knows how to format and print itself

COMPENSATION COMPUTATION OBJECT

- Gets the hours worked and salary or hourly rate, then computes compensation and deductions
- Creates a check request
- Creates IRS and state bank deposit forms

BANK DEPOSIT FORM

- Knows how to format and print itself

CHECK WRITER

- Takes the check request and writes out the checks
- Creates the Payroll Report and tells it to print
- Tells the bank deposit form to print

PAYROLL REPORT

- Knows how to format itself
- Knows how to print itself

CHECK

- Knows how to format and print itself

New Objects

OVERTIME REPORT

- Created by CHECK WRITER

Enhanced Objects

PAYROLL RECORD

- Keeps track of vacation

TIME SHEET FILER

- Computes accrued vacation

Simulation Yields Few Benefits

It was at this point that Ace realized that something was wrong, though he couldn't quite put his finger on it. "The way I wanted this to work was to have the check writer get the information it needed from the payroll record and compute the pay. This seemed like a pretty straightforward interpretation of the way it was done, but somehow it didn't seem quite right. It felt more like an old-fashioned procedural data processing program rather than an object-oriented one. What was the benefit of creating objects?"

What Ace had created was essentially a simulation of the real world, with one object in the real world corresponding to one object in the program. In place of TIME SHEET FILER, read "Rachel." In place of SALARY COMPUTATION OBJECT, read "Christine." For CHECK WRITER, read "John." Ace found out the hard way that this approach doesn't, in general, yield very good results. Objects have to know a great deal about other objects, which violates a basic rule of modularity. Furthermore, there is overlap between objects, leaving little or no chance of reusing code for more than one class. Ace intuitively understood that there must be a better way and set out to find it.

▶ Second Try: Shuffling Responsibilities

"It was unclear to me what should be an object and what shouldn't. Did I really need a check request object? Couldn't I just have the check writer ask the employee object what its pay is? But if I did that, I'd have to store the computed pay in the employee object. Why couldn't the check writer

do the pay computation itself? I followed that train of thought for a while and ended up with one object—the check writer—computing the pay, creating the check, creating the payroll report, and creating the bank deposit forms. Where did my modularity go? Furthermore, this was nothing like the real world process.

"Then I thought that the check might compute its own pay. That felt a little better, but how? All I came up with were exotic solutions I didn't really want to implement. For example, one idea was to have the program create a blank check that would search for the first unpaid employee and ask it for its rate, hours, and other information, then compute the pay and print itself. But then the check object would need to know all about both kinds of employees, salaried and hourly, in order to do the computations. That didn't seem right; shouldn't salaried and hourly be subclasses of employees to hide this kind of knowledge? I could have had the employee compute its own pay, but it wasn't clear whether that was any better.

"I realized that a literal interpretation of the real world in objects wasn't good, but nothing I had read told me how to do anything else." Ace experienced a lot of basic problems.

1. Should real-world objects like a check request stay around in the implementation, given that they serve only to pass information from one real-world object to another?
2. Where should computations performed by people in the real world reside in an artificial world of program objects?
3. Ace created a number of objects that store and return data, but are otherwise passive. How do you turn a passive collection of data in the real world into objects that exhibit non-trivial behavior?
4. Who arranges for timing? For example, check objects must be created at a specific point in the payroll cycle. Who has that responsibility?
5. How do you limit type knowledge (for example, the knowledge that there are different ways of computing payroll for different types of employees)? It is messy to pass that information around. Furthermore, even though the program consists of lots of small modules (good), the modules are very dependent on one another (bad).

Ace was not alone with these problems. After hearing Ace's story one of the authors has used this problem as an in-class exercise in his seminars on object-oriented design. The problems Ace experienced are absolutely typical of beginning OOP programmers. In frustration, Ace solved the problem in the usual way by turning to an OOSD expert for help.

▶ Third Try: Ask an Expert

The object-oriented software development expert has none of Ace's problems. Within a few minutes, her list of candidate classes and behaviors might look something like this:

EMPLOYEE

- Computes its compensation and deductions on demand
- Stores its name, address, date of hire, and other information, and returns them on demand
- On payday, creates a PAYCHECK object and tells it to print itself
- Maintains a list of TIME SHEETS
- Keep a running balance of vacation hours accrued but not taken
- Returns daily hours on demand
- Returns summarized hours by week on demand (totals only)

SALARIED EMPLOYEE

- Subclass of EMPLOYEE
- Stores its weekly salary
- Computes its compensation and deductions on demand using the salary computation

HOURLY EMPLOYEE

- Subclass of EMPLOYEE
- Stores its hourly rate
- Computes its compensation and deductions on demand using the hourly computation and the time sheet

TIME SHEET

- Tracks regular, overtime, vacation, and sick hours for each day for one EMPLOYEE
- Prints itself

PAYCHECK

- As part of creation, EMPLOYEE supplies amounts, name, address, and so on
- Formats and prints itself

HOURS WORKED REPORT

- Asks all employees for their summarized hours
- Prints itself

Foundations of the Design

More information is required to handle quarterly and other reports, but this is a pretty good foundation. There are really two cornerstones of this design.

- Employees keep track of data about themselves and perform computations on that data.
- Computations of compensation and deductions are handled by setting up a pure virtual method in the EMPLOYEE class, then overriding it in the SALARIED and HOURLY subclasses.

Experts Don't Simulate the Real World

This is certainly far removed from the "real world" of payroll. Frank would be horrified at the thought of allowing his employees to pay themselves and keep track of a history of their own hours. One of the reasons for automating in the first place was that he didn't trust them to keep track of their vacation hours. Furthermore, reports that print themselves—as opposed to having someone *cause* them to *be* printed—and reports that know how to ask intelligent questions of employees are decidedly unnatural. Where did these concepts come from? Certainly not from the folklore! OOSD experts leap to these sorts of designs in their sleep, yet otherwise talented people like Ace are left scratching their heads. That which is natural is not workable, while that which is workable is not intuitive to the common folk. We need to look further for an explanation of how these designs come about.

▶ What Are the Macintosh Documents?

Ah, yes. Lest we forget, this design must be made to run on a Macintosh. Remember the Macintosh-specific problems with the railroad example? They are redoubled here, for the "documents" are anything but obvious.

Faced with the need to define documents, Ace initially had the idea that all printed reports should be treated as documents: TIME SHEET, PAYCHECK, HOURS WORKED REPORT. He set up a separate file for each document, but found that the idea quickly fell apart. Consider the following scenario: (1) The user double clicks on a TIME SHEET file, launching the payroll application, changes the employee's address and quits the application then (2) the user double clicks on a PAYCHECK file for the same employee, relaunching the application for this separate document.

Will the new address provided in the TIME SHEET document show up here in the PAYCHECK document? Not unless Ace takes extreme measures to propagate changes across files. Keep in mind also that this will generate hundreds or thousands of files, all interdependent and all subject to the whim of a casual user browsing through the Finder. All in all, not a very workable design.

A properly designed application should store an employee's name once, then reference it from everywhere it is needed. Since this was clearly a database problem, Ace chose a database management system to use in storing and retrieving data. Since objects in an object-oriented program and records in a database do not, in general, match up very well, Ace was forced to spend a great deal of effort gluing the two together.

Ace also had to figure out the meaning of Open, Save, and Revert when windows correspond not to files but to records in a shared database. Suppose you as a user change an employee's address in one window and his hours worked in another? The windows are shown in Figure 4-6.

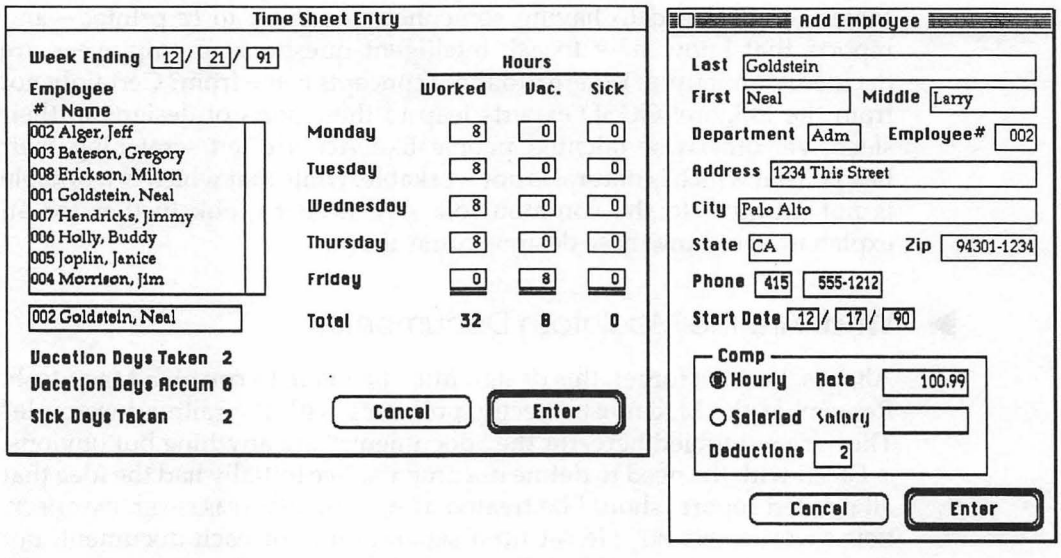


Figure 4-6. Payroll application windows

Bring the employee general information window to the front. What does it mean when you choose Save? Save only the changes made in that frontmost window (the address) or save all changes made to employee

information in all windows (address plus hours)? What about Revert? Should the program undo changes made in a window that is not on top? What options should be presented when the user chooses Open? Database files? Record types? Reports? Specific employee numbers?

These questions cannot be answered from the real world. They derive from the world of the Macintosh, its software, and its standards.

▶ Summary

- Analyzing the "real world" is fine, but you must first decide how to observe it and record your observations. One popular technique is lexical analysis in which the programmer writes a description, then uses the nouns to create candidate lists of objects and classes and verbs for methods. This technique suffers from a number of common problems that render it suspect at best and very dependent on the skill of the practitioner.
- Top-down methods work a little better but cannot tell you where to stop, where to prune, or how to tackle the issue. Instead, the programmer must judge the relevance of each object and class. An additional problem is that there are always going to be artificial features of a program that do not spring directly from the real world. The folklore provides neither a theory to account for them nor a methodology for discovering and designing them.
- Simulation of the real world in general does not yield benefits in designing object-oriented software. Instead, behaviors should be assigned to objects in the program in ways that often don't make sense in the real world. OOSD experts are able to do this intuitively and achieve great results in little time, even though they may be unable to articulate exactly why their approach is better.
- The folklore cannot account for the unique characteristics of the Macintosh, its operating system, Toolbox, and user interface standards. In fact, the folklore presumes that such considerations are mere implementation details, not the dominant factors they are. As a result, the folklore, which bases object-oriented software development on objectivist approaches, is frequently unworkable.
- The folklore ties reusability to the autonomous nature of objects in the real world. Projecting behaviors that do not really exist in the real world onto our program objects compromises this autonomy. The more behaviors we project, the less stable and reusable are the classes. The largest projects usually have the most requirements and, therefore, the least reusability.

5 ▶ The Way We Think

▶ What This Chapter Is About

We have spent a great deal of time so far tearing down the simplistic, objectivist approach to object-oriented software development. This chapter starts the process of building a replacement methodology that is based on sound principles of the way people actually perceive their world. The central message of this chapter is that people do not perceive their world in terms of classes based on shared properties. Although the folklore assumes that the world is naturally organized into classes, people really organize their thoughts and perceptions into *cognitive categories*. Although cognitive categories resemble the classes we use in object-oriented programs, there are important differences. If we are to create a truly natural, intuitive way to develop software, we must start by reconciling the world of human thought and the world of the program. It is only in the simplest situations that categories are the same as classes; in those cases, the folklore works. In all other cases, we must have a framework for software development that takes into account both categories and classes *without insisting that they correspond*. A secondary message of this chapter is that neither top-down nor bottom-up approaches are natural. Instead, people naturally gravitate first to the *solid center*. A sound software development methodology should do likewise.

▶ Categories

So far, we have assumed the objectivist view of the way we perceive our world. However, there is a serious problem with objectivism: The evidence doesn't support it! Objectivism may be the way we think we think, but it isn't the way we really think. Instead, humans categorize in far more intricate ways than by simple sharing of properties. We will present supporting evidence from the cognitive sciences that focuses on implications for the way we create object-oriented software. (For the curious, an in-depth, fascinating study of this phenomenon can be found in the book, *Women, Fire and Dangerous Things: What Categories Reveal About the Mind*, by George Lakoff. This and other sources are listed in the bibliography at the end of the book.)

To avoid confusion, we are adopting a strict convention from this point on in which *category* refers to groupings in the real world as perceived by people, and *class* refers to groupings of objects based on shared properties in an object-oriented program. It is a common point of confusion to think that classes naturally correspond to the way we form categories; in fact, we have deliberately propagated this idea up to now. As we saw in the discussion of our sample applications, however, this assumption does not hold. As a start toward unraveling this problem, let's look at the way people form and structure categories. We will return to class formation only after a thorough discussion of categories and the human mind.

▶ Basic Level Categories

People are not computers. We do not run computer software in a mental digital machine. Instead, we have a complex physiology in which certain categories are formed *preconceptually* (that is, before any process of reasoning takes place). Take perceptions of colors. Human beings recognize eleven basic colors for which signals of recognition are sent from the eye to the brain: black, white, red, yellow, green, blue, brown, purple, pink, orange, and gray. This recognition is not according to some process of reasoning; instead, neurons fire in response to parts of the spectrum of visible light that determine this response. These categories of colors are determined by physiology before they even reach the brain.

This example is not an isolated case. In fact, people have built-in wiring that leads them to form many such *basic level categories*. These are neither the lowest nor the highest level categories people form, but they are the *natural* level. By this we mean that people have a physiology that assists in the formation of this basic level. There is no process of reasoning that supports forming basic level categories; it is just something we can do by

virtue of being human. People from different walks of life, even from dramatically different cultures, are remarkably consistent at this basic level. Basic level categories are the first ones we learn and the first ones to which we assign names. Names tend to be shortest at the basic level and are the names used most frequently. Most importantly, basic level categories can be discerned without picking out details; basic level categories are perceived holistically as *gestalts*.

Basic level categories are determined in large measure by the distinctive ways we as humans interact with their members and, for this reason, tend to be *functional* and *visual* in nature. Put another way, basic level categories are not intrinsic to the "things" in the real world, but are determined in large measure by what we do with them. They are the elementary building blocks of our understanding of the world. Thus, something that we can pick up, shake, perhaps eat, is inherently easier to categorize than an abstract concept like "event." *Basic level categories are formed, not based on shared attributes, but due to the firing of the right neurons.*

Preconceptual categorization results in basic level categories as *gestalts*. We do not find them as primitives, nor are they composed of primitives. They are simply taken as a whole. These basic levels are things that all human beings, even from radically different cultures, can share.

▶ Not-So-Basic Categories

People from all backgrounds seem to be able to distinguish one genus of tree from another (for example, oaks from maples). However, when we drop down to the level of species (for example, sugar maple), the ability to distinguish one category from another is very dependent on culture and experience. Surprisingly, going up the ladder to the category "tree" seems to be somewhat difficult for everyone because of tree-like bushes and bush-like trees. We need to pick out specific characteristics and apply a process of reasoning to answer the question, "Is this a tree?" For example, are both of the plants shown in Figure 5-1 trees?

Higher levels like "plant" and "life form" are even more uncertain. In fact, it is generally true that the genus level of biological taxonomy is formed of basic level categories, but species and subspecies (lower level) and families and so on (higher level) are not. Categories like "emotion," and "event," which are not even composed of things in the real world, are far less likely to be recognized in the same way by different people, especially at the fringes of the category, because there is no built-in wiring to preconceptually recognize members of the category.

Certainly people do not stop categorizing when the basic level has been exhausted. Reasoning extends our categories from the basic level, both up

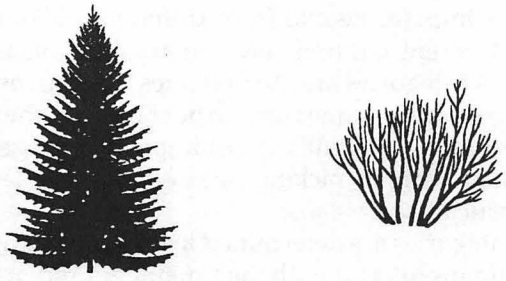


Figure 5-1. Are both of these plants trees?

and down, in detail. However, non-basic categories are fundamentally different: They are formed through a complex and somewhat unpredictable, but nonetheless consistent, cognitive process. Classification, based on shared properties, does not begin to do justice to cognitive categories.

Cognitive categorization is used to create other than basic level categories, extending above and below the basic level. It is in nonbasic categories that we find the remarkable differences between cultures or members of a culture. It is here that individual cultures make their unique contributions to understanding the world around us.

Consider the category GAMES. Games really have no properties common to all members of the category. Some games use boards, others do not. Some use dice, others cards; then there are games like leapfrog that don't use any equipment at all. Some are individual, others involve pairs or teams of people. Some, like "baseball games," are not limited to participants ("fans are part of the game" is commonly said of most professional sports). Yet, the category is recognizable, at least in our culture. If we try to use abstraction, or any of the other classical tools for modeling based on shared properties, we fail to properly model GAMES. What is it that allows people to so readily form such categories without apparent use of shared properties?

Non-basic categories are often subject to what psychologist Eleanor Rosch labeled *prototype effects*, which means that membership in a category is not always clear. Some members or subcategories are better fits for the category than others. Although people have traditionally tried to write this off to a certain degree of unavoidable inconsistency or randomness in human reasoning, there is a deeper structure to categories that makes this anything but random. Membership in a category is not a simple yes/no matter. Rather, a category itself may well contain a rich structure describing the relationships of its members to the category and to each other. A

category is a complex cognitive model, which some things fit better than others. As illustrated in Figure 5-2, Joe Montana is a better example of a quarterback than a high school player playing the same position. And certainly both are better than one of the authors as examples.

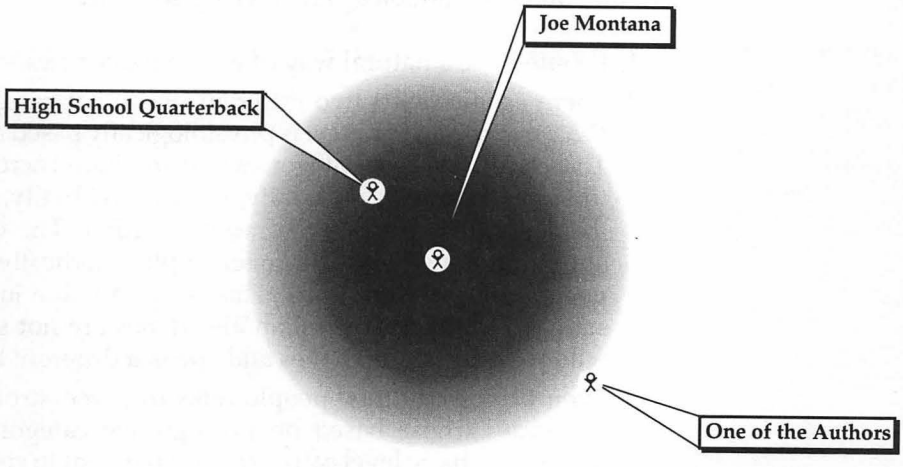


Figure 5-2. Who is the best example of a quarterback?

One type of category that exhibits prototype effects is a *radial category*. The title of Lakoff's book comes from the category BALAN from the language of the Dyribal aboriginal tribe of Australia. BALAN contains, among other things, the subcategories WOMEN, FIRE, and DANGEROUS THINGS. What on earth can these have in common and why put them in the same category? It appears at first glance that they have nothing at all in common, and that the category is just a random, culturally derived grouping. But it starts to make sense when one takes into account not just the language but the Dyribal culture and mythology. BALAN has as its *central member* the subcategory WOMEN. According to Dyribal mythology, the sun is a woman and therefore belongs in the same category BALAN. The sun causes sunburns, which, by a similar chain of links, leads to the inclusion of spears and other DANGEROUS THINGS. While individual links may represent shared properties—WOMEN to SUN, SUN to DANGEROUS THINGS—it is not true that there are any shared properties throughout the category. In a radial category consisting of A, B, and C, A may be linked to B, which is linked to C, but A and C need have nothing whatsoever in common beyond membership in the category.

▶ Categories and Classes Are Not the Same

Let's pause and reflect on what we can conclude already about the relationship between categories and classes. These are the realities we must take into account when using object-oriented software development (or any other methodology) to develop software.

1. People have a natural way of forming *categories.*
2. There are (at least) two different ways that categories are formed. *Precognitive categorization* is physiologically based and non-cognitive. These are basic level categories that are shared across cultures. *Cognitive categorization* is culturally, even individually, based, and it is at this level that cultures can radically differ. The way an Australian aborigine views the world, for example, is radically different from our perceptions as Americans. This is not a value judgment, merely a statement of fact. Australian aborigines are not simply Westerners who wear different clothes and speak a different language.
3. Even across cultures, people tend to agree strongly on *basic level* categories (those based on precognitive categorization). Unfortunately, few basic level categories are relevant to computer programs, especially business systems.
4. We should expect people to differ with equal vigor over categories at other levels (that is, levels based on *cognitive categorization*). These differences occur even among those in the same culture and with similar backgrounds.
5. Membership in a cognitive category is not necessarily a simple yes/no proposition but may be *graded* by prototype effects. Some members are better examples of (that is, more central to) a category than others. Thus, we should be surprised to find authoritative answers to many seemingly simple questions.
6. Categories cannot be modeled by mere classification using shared attributes. Put another way, *categories do not equal classes.*
7. The natural way in which people form categories does not correspond to either of the dominant ways of constructing software: top down and bottom up. There is a natural center (basic level categories) and context- and culturally-determined super- and subcategories. The only "natural" way to form hierarchies of categories is through a *center-out* process.
8. In light of all of the previous observations, it is rare when two people agree on a single definition of the "natural" way to organize a program, let alone an entire project team. Based on the authors'

experience teaching a class on object-oriented design, people working in teams of three or more spend most of their time arguing over categorizations and make little or no progress. Those who work independently or in groups of two make progress.

▶ Reconciling Categories and Classes

If people think in categories, and object-oriented software uses classes, and the two do not correspond, we have then cut the very foundation out from under the objectivist approach to object-oriented software development. It is already clear where many of our difficulties in using object-oriented programming in large, complex projects come from. Processes we assumed were natural and universal turn out to be anything but. Is there a reason for pushing forward?

The easy way out is to say "Forget about categories; forget about analysis and design; I'll develop object-oriented software based on the strength of object-oriented programming language features." This is, in fact, a popular idea in theoretical computer science circles. It is the practitioners, not the theorists, who have latched on to OOSD as a tool for analysis and design. But remember, there must be *something* good about mixing up categories and classes, or so many people wouldn't be reporting such great results. Furthermore, classes may not be an exact match for categories, but they are certainly closer than data flow diagrams and functional decompositions! As we will see in the next two chapters, it is indeed possible to create methodologies that capitalize on the best of both worlds by learning to include both categories and classes in software development. But first, we must gain some further understanding of the way we categorize. We can then start the process of reconciliation.

▶ Schemas and Contexts

Lakoff has proposed a convincing model for cognitive categorization. It is by no means the only model, although it is one the authors find appealing. To account for this rich structure of categories, Lakoff argues that non-basic categories exist within the scope of *schemas*. Each schema represents the following.

1. A set of categories
2. Relationships among the categories
3. Background assumptions of the schema
4. Relationship of the schema to other schemas

There are four kinds of relationships within and among the schemas.

1. *Image-schematic:* based loosely on visual image structures
2. *Propositional:* logical relationships, such as shared properties
3. *Metaphoric:* one category or schema is analogous or similar to another
4. *Metonymic:* one subcategory or member is used to represent an entire category

Because categories are defined in terms of schemas, no category can be completely defined in isolation from some schematic backdrop or *context*. Both the relationships and background assumptions are critical, not just to using a category, but to its definition as well. Discussion of each of these types of relationships follows.

▶ Image-Schematic Relationships

Since people receive much of their information about the world through sight, it is no surprise that much of our knowledge about the world is structured in spatial terms. Figure 5-3 illustrates some common image schemas.

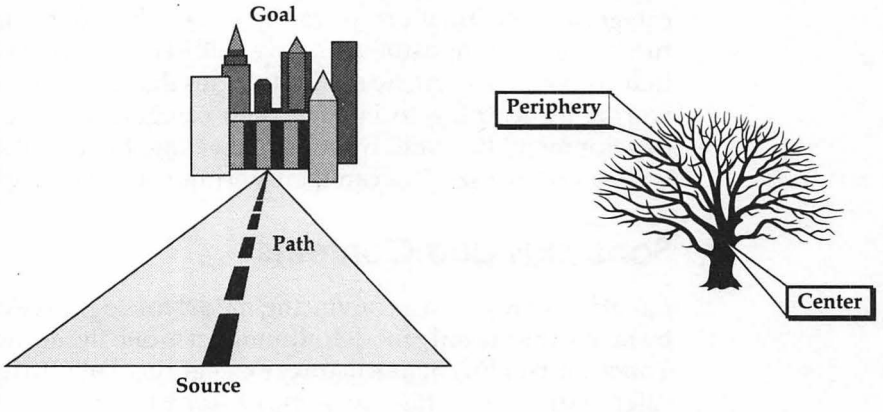


Figure 5-3. Image schemas

Although this way of categorizing and comparing categories is extremely common, it is little used in computer programming. Although human beings may have the built-in ability to handle spatial information, our computers do not! It is hard work to adequately represent these and other image schemas in a computer.

In addition to perceiving images that really exist, we also tend to use images to represent many complex concepts. A home run in baseball is a complicated concept; part of that concept is not of just the ball itself, but also of the trajectory traced by the ball as it travels from home plate to the stands. That arc is itself an image schematic way to think of a home run.

▶ Propositional Relationships

This is the type of relationship with which most object-oriented programmers are familiar. Examples include WHOLE-PART relationships (e.g., CAR and DOOR) and SUBCATEGORIES (that is, CAR and 1979 FORD STATION WAGON).

These are the most common but by no means the only relationships. It is common for propositional relationships to be based on shared properties and, therefore, to be easy to represent in a computer program. This is, in fact, the way in which database schemas are designed to represent relationships within the data in a system.

Propositional relationships are based on properties of the categories and rules for using them. They might be used, for example, to represent control of one object by another: LIGHT SWITCH and LIGHT, or ancestral order: PARENT and CHILD, or more general temporal order: EARLIER-LATER. Several distinct relationships might exist between PERSON and DOG:

- PERSON owns DOG
- PERSON is VETERINARIAN of DOG (clearly there are background assumptions about PERSON here!)
- DOG bites PERSON
- PERSON sold DOG
- PERSON feeds DOG

▶ Metaphoric Relationships

Metaphors are everywhere in human thought and discourse: "He lit up (like a light bulb)"; "We bowled 'em over!"; "My car is like Uncle Al's corns: It aches on a wet day".

Metaphorical relationships among categories are equally common. Whenever we allow one situation to stand for another for purposes of reasoning, we are using metaphor as a tool for reasoning about concepts. When we talk about categories and relationships among them, metaphor becomes a tool for forming and reasoning about categories.

▶ Metonymic Relationships

Metonymic relationships sometimes exist among the members of a category when some member can be used to stand for the entire category. Asked to describe the category, SANDWICH, you might be tempted to say "A SANDWICH has two slices of bread separated by some edible stuff." But what about open-faced sandwiches? Sandwiches made from crackers, rather than bread? What about an important letter "sandwiched among junk mail?" The category SANDWICH includes a lot more than a simple ham on rye! Yet, for most purposes, you use the simplest type of SANDWICH to answer questions about all sandwiches. This is metonymy: one member or subcategory is used to represent the entire category.

▶ The Importance of Context

Recall that we stated that categories are defined in terms of schemas, which in turn have background assumptions. The background assumptions are important because they determine when a given schema and its cognitive categories are a good fit for reality and when they aren't. Put simply, *the categories you choose are highly dependent on their context.* For example, one can create a category CAR and in a schema associate with it categories DRIVER, DOOR, GARAGE, and so on. However, this schema can lose much of its meaning if we try to apply it to a TOY CAR. The category CAR for most people includes both the real thing and the toy, but the relationship of CAR to the other categories is simply not valid for all kinds of CAR. In the context of driving someplace, a toy car is not that kind of a car. The TOY CAR is not categorized as a CAR when you examine it in a context in which CAR implies ability to go someplace.

Categories as cognitive models shoot a gaping hole into our Four Itys. Remember the fundamental assumption that classes and objects are autonomous entities? It is not, in fact, true that a category is independent of all other categories; instead, it is completely defined within and intricately interwoven into the fabric of context, those complex and ill-defined background assumptions that we cannot hope to capture with any rigor. It is the meaning of a category in a context that gives meaning to other categories in that context; the other categories affect its own meaning as well. There are some critical implications from this.

- Category formation and definition is a recursive process. Refinement of one category can lead to refinement of others, which can affect the original category, and so on. *This destroys any notion of exploring and defining each category in sequence.* The category PROGRAMMER is a good example of this. You might start by thinking of the category in terms

of either Cobol business programmers wearing suits and ties or perhaps systems programmers in hiking boots and T-shirts working late into the night. The more you examine the category in different contexts, the more it changes. Is someone who writes Excel macros a programmer? How about 4th Dimension programmers? HyperCard programmers? Each time you introduce the context of a new product, you use the category PROGRAMMER as it is then understood to help categorize that product according to its intended use; as products are categorized, PROGRAMMER itself is refined. You might, for example, rebel against calling a user of HyperCard a PROGRAMMER until you distinguish someone who writes scripts from someone who merely adds cards to existing stacks. As the context of the question "What is a PROGRAMMER" expands, the category grows as well. In short, there is no fixed answer to the question "What is a PROGRAMMER?"

- Background assumptions allow us to place the "same" thing in different categories based on the context. For example, a hug can mean many things. From a child, it is a sign of love. From a burly stranger wearing a mask in a dark alley, it is quite something else. Did you catch the implication of those simple statements? *There is no one, unique way to categorize the things in the world!*
- So much for the Big R: Reusability. We should expect it to be the norm, rather than the exception, when a category is applicable only to the problem at hand.

▶ The Myth of Reusability

The preceding statement is worth repeating:

We should expect it to be the norm, rather than the exception, when a category is applicable only to the problem at hand.

This explains why most reusable object-oriented code is in the form of application-independent libraries such as MacApp: They carry only the background assumptions required of the Macintosh, its operating system and Toolbox, not those of the application. The further we drift away from our little machine- and operating-system-defined island, the less relevant our previous categories, and therefore, our code, will be.

Earlier, in discussing the folklore, we hoped that an employee class in a payroll application would prove highly reusable in a manpower planning application. We can now recognize the ugly truth: it isn't likely to happen, no matter what our exertions. The manpower planning application requires a whole new set of extensions to the class and its relationships to

other classes. A few of these might include employee skill levels and experience; advance scheduling, rather than merely logging past hours; and contingencies involving positions not yet filled or even defined. Who can predict what will be needed without first taking a good, hard look at this new context? Some of these are likely to conflict with our original class, such as positions without specific people in them. Even if we get lucky with the manpower planning program, we likely face an uphill struggle in building a human resources application or an office-wide calendar and appointment book system. There is no easy way out. Such situations have always been and continue to be difficult, even using object-oriented techniques. The problem, however, lies not with the technology, but with the dominant role of contexts that are not yet known.

For applications, it is far better to concentrate on the other three Itys—Maintainability, Extensibility, and Modularity—and on techniques for lowering the costs and time required to create an application's new code. These three Itys are limited in scope to the application at hand and therefore depend only on the context of a single application. We can construct good arguments for the benefits of OOSD for these three Itys without having to make the dubious claim of reusability.

This is not to say that application-independent class libraries are not useful or important, but they are not the major problem. There will always be many more applications than libraries. Otherwise, a library wouldn't be much of a success, would it? Class libraries are the low-hanging fruit of object-oriented software development: juicy, but a small fraction of the fruit of the entire tree. Even so, we can learn an important lesson for code that is specifically required to be reusable: we must be careful to stick to only those categories which, if they carry any dependence on context at all, directly and unambiguously derive from the platform. A sorted list class carries no context; a window class carries (one hopes) only the context of the Macintosh and its user interface features; a "person" class is probably too context-sensitive to be truly reusable.

▶ The Sheer Cliff Principle Explained

By now, it is clear why objectivism does not work for complex projects. It's worth taking a look at why it works for simple ones.

▶ When Does the Folklore Work?

We earlier stated that the folklore is not wrong, just oversimplified. The assumptions that make it oversimplified are that categories and classes are the same thing and that categories or classes are independent of one another and of their context.

The explanation for the success of objectivism in some situations and its failure in others is simple: Under certain conditions, the previous two assumptions do, in fact, hold. Some of these conditions are listed below.

1. The "things" in the problem have a physical and visual reality that is familiar to the programmer and the target audience. Under such conditions, the chances of differences in categorization from one person to the next are drastically reduced.
2. The categories/classes are simple enough so that there are very few choices available in classifying/categorizing. Certainly this is the case in most sample applications designed for teaching purposes. How many ways are there to categorize the tools in a simple drawing palette: a rectangle, an oval, a line, and a polygon?
3. The relationships between the categories/classes are simple and unambiguous and do not depend on complicated, unstated assumptions. In other words, we are dealing with only one or a very few contexts and the background assumptions are either obvious or irrelevant to that program. In a simple drawing application, it is hard to find more than one way to describe the relationship between a rectangle and the electronic page it sits on.

▶ Why the Sheer Cliff Exists

Under most combinations of these conditions, classification corresponds quite accurately to categorization and the process seems "natural." In complex projects, however, these conditions do not generally hold and the gnawing feeling starts to rise that something is not quite right. In fact, sticking to objectivism in such circumstances makes things harder as you search for a "natural" classification that does not exist. The source of the Sheer Cliff Principle is that something that works beautifully for a certain kind of project can quickly become irrelevant or even damaging in another. *It is the assumption that the folklore always works that results in the sheer cliff.*

▶ Avoiding the Sheer Cliff: Solution-Based Modeling

Is this a reason to give up on a cognitive view of object-oriented software development? No, and the reconciliation between the points of view forms the major motivation behind Solution-Based Modeling and the following chapters. We will show that the two can and should coexist inside a single model of software development. We must allow categories and human

thought to be what they are and objects and classes to be what they are, always taking care not to confuse the two. Categories will form the basis of our understanding of what a program is and does—the program's *meaning*. Classes will be used to implement that understanding in an object-oriented programming language.

▶ Categories and Image Schemas in Macintosh User Interfaces

This chapter has explored some facts and theories about how people perceive their world, information that has very interesting implications for the design of graphical user interfaces yet seems to have been left out of the mainstream literature on the subject. Specifically, the concepts of cognitive categories and image schemas can be applied to create intuitive, "friendly" and, above all, approachable interfaces for Macintosh programs. If people organize their perceptions in terms of categories, and if image schemas are a dominant way of understanding our world, interfaces that make use of those innate abilities of the user will be better than interfaces created for their technical merit. Put simply, we seek to design for the benefit of the user, which requires understanding how the user perceives the program.

Categorical User Interfaces

Ever wonder why menus are organized the way they are? Why do certain menu items end up in the Edit menu, rather than, say, the File or Whatever menus? It is very common for the items in a menu to have very few properties or actions in common. Figure 5-4 shows an example of this, the Edit menu from Ashton-Tate's FullWrite word processing program. What do the clipboard operations, outlining, sorting, a glossary, a thesaurus, spell checking, and hyphenation have in common? Not much. Yet, there is a sense of common purpose: Applying some sort of well-defined change to the text. In the actual implementation, these could hardly be more different as command objects, but in the user interface it makes a lot of sense to group them in this way. In other words, they form a category, but not a class.

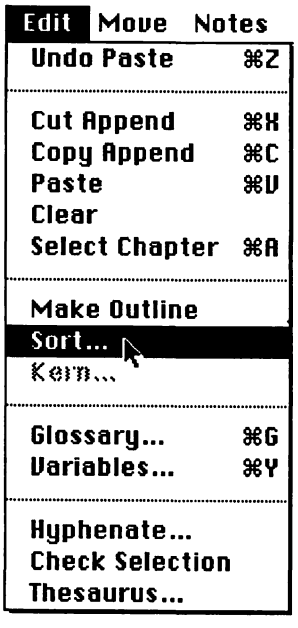


Figure 5-4. FullWrite "Edit" menu

Categories occur throughout user interfaces. Palettes are categories, as are dialog boxes and, more generally, windows. Files and documents represent categories of information, largely determined by the user. Dialog items that are physically grouped together are often categories. Consider the dialog box shown in Figure 5-5.

See the check boxes arranged close to one another? These all act independently of one another, but the grouping makes sense—it's a cognitive category of otherwise unrelated check boxes.

One of the most common user interface design mistakes the authors find among their clients is confusing categories with classes in user interfaces. One of the authors was guilty of this in a past project in designing a palette of different kinds of furniture. There were too many types of furniture to appear on the screen at once, so the palette had to switch among several

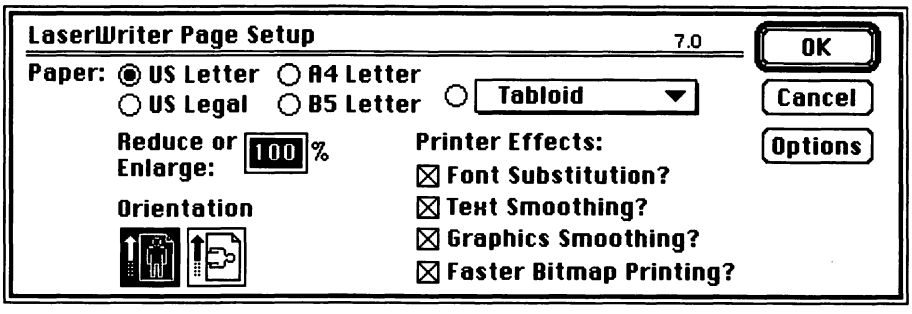


Figure 5-5. Categories in a dialog box

groupings of furniture. The author tried in vain to explain abstraction and classes to the client in justifying why some pieces belonged together in a group and others did not, but in the end the groups were formed in ways that left little behavior in common within each group. One group contained some furniture that rolled under work surfaces, others that fastened in place there, other pieces that attached above the worksurface, and others that simply stood alone. Some pieces, in fact, showed up in more than one group! Yet, these groups made perfect sense to interior designers and architects. In fact, they corresponded to the major headings in the manufacturer's catalog. Categories were the correct, user-centered way to group the furniture, not according to classes of shared properties.

The lessons here are to leave the classes behind when designing user interfaces and to seek out the categories that users naturally form when dealing with the sorts of problems to which the computer is being applied. Classes, based on shared properties, will be used to *implement* the user interface, but are irrelevant when *designing* it.

Image Schemas

Image schemas are important in user interface design as well. Up/down, foreground/background, source-path-goal, bigger/smaller, left/right, inside/outside, and other image schemas are powerful ways to convey information to the user without having to resort to text. The Macintosh user interface is largely based on image schemas. One of the best but most trivial examples of this is the trash can—put something in it and it bulges. Size denotes quantity in visual terms. Windows use the foreground/background image schema to indicate the current context of the program—the foreground window. The modal dialog that appears when you copy files uses a left/right image schema to denote the passage of time, as shown in Figure 5-6.

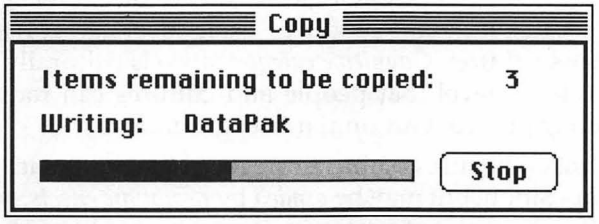


Figure 5-6. Passage of time while copying files

Relative size, line weight, shading, center-periphery organization, color, and other visual tricks can go a long way toward conveying the meaning of the program visually. Perhaps most powerful are simulated 3D interfaces, as with System 7's Finder. People do not perceive their world in two dimensions, but three. Flat interfaces are disorienting and cause the user to have to think about how the objects behind the interface are organized. Now, let's see . . . what does a button that turns black mean? Ah, that's right, the button has been "pushed."

Image schemas are the meat and potatoes of graphic design. User interface design can benefit greatly from the techniques developed over centuries by that discipline. The result when this is done is not just a more pleasant-looking program, but a more effective one. The user is given visual cues about what can be done and how to proceed. If you don't happen to have a degree in graphic arts, don't despair; the basic principles are not that difficult to master. Two works by Edward R. Tufte which should get you on the right track are listed in the bibliography.

In the next chapter we will apply these principles using the authors' notational system for object-oriented analysis and design, the Visual Design Language (VDL).

▶ Summary

- People organize their thoughts and perceptions in terms of *cognitive categories*, not classes based on shared attributes. Categories cannot be modeled by mere classification using shared attributes. Categories do not equal classes. A software development methodology should make provision for both categories and classes without insisting that they be the same. Categories correspond to the way people understand the program; classes are a convenient way of implementing that understanding.
- Although people have a natural way of forming categories, forming and choosing categories is highly dependent on their *context*. There are at least two different ways that categories are formed. *Precognitive categorization* is physiologically based and non-cognitive. Precognitive categorization produces basic level categories that are shared across cultures. *Cognitive categorization* is culturally based. It is at this non-basic level that people and cultures can radically differ, even among people with similar backgrounds.
- Membership in a cognitive category is not necessarily a simple yes/no proposition, but may be *graded* by *prototype effects;* some members are better examples of a category than others. Although pairs of members of a category may share attributes, it is not generally true that all members of a category must have attributes in common with all other members.
- The natural way people form categories does not correspond to either of the dominant ways of constructing software: top down or bottom up. There is a natural *center* (basic level categories) and context and culturally determined super- and subcategories. Rather than being defined in isolation, categories are defined based on the context(s) in which they are used. The dependence of categories on their context undermines Reusability for all but application-independent class libraries.
- The folklore of object-oriented software development has a strong intuitive appeal, partly because classes have a strong correspondence to categories in the simple cases encountered by the beginner. In more complex projects, the correspondence between classes and categories does not hold.
- The principles of this chapter apply to the design of graphical user interfaces. Image schemas, categories, and simulated 3D all contribute to the user's perception of how the program operates.

PART TWO

▶ Solution-Based Modeling for the Macintosh

6 ▶ The Visual Design Language

▶ What This Chapter Is About

This chapter sets the stage for Solution-Based Modeling (SBM) by introducing the principal language of discourse we use with SBM, Visual Design Language (VDL). The specific use of VDL to construct solution-based models will be illustrated in subsequent chapters.

VDL was developed to use the principles of human cognition discussed in the previous chapter by harnessing images as fundamental tools for communicating ideas. VDL provides a rich set of symbols for representing object-oriented software development concepts, together with standards for their use in relation to one another.

VDL has symbols for all of the concepts discussed in earlier chapters—categories, objects of the natural world, program objects and classes, requirements and constraints, and a wide variety of relationships among these elements. VDL is the cornerstone of our effort to bring analysis, design, and programming for object-oriented software under one roof. These symbols, and the standards for their use, are intended to appeal to the intuition as much as the intellect.

▶ Overview of VDL

▶ Visual Communication

The primary purpose of any notation should be to communicate. Unfortunately, diagrams and formal notations are frequently used only to document what has already been developed. What we present here is a working

tool to be used at all phases of development. Ideas are explored using images whenever possible, and those images form the basis of communicating ideas to others. Of course, it is not possible or even desirable to use images for everything. There will always be a role for text. However, all people involved—end users, systems analysts, programmers, and management—should have an intuitive grasp of what the model contains and how it works before resorting to text. This, more than anything, is the objective of the VDL.

▶ Escaping Flatland

Notations have a long history in computer software, starting with flow charts and proceeding through structure charts, data flow diagrams, logical data models, and, in the present object-oriented world, various ways of visualizing objects and classes. However helpful these notations have been, they are quite crude by graphic design industry standards. The authors know this first-hand. They enlisted the help of a professional design firm to help develop a notational scheme for use with Solution-Based Modeling, only to endure their amazement and, at times, benevolent laughter over our early, two-dimensional attempts at graphic symbols. Once we managed to break out of our combined thirty-plus years of indoctrination with flat, boring rectangles and arrows, the results became VDL.

In his landmark book, *Envisioning Information,* Edward R. Tufte states that the principal job of the designer in conveying information is "escaping flatland." People visualize in three dimensions, not two. Yet, all major notational systems for software analysis and design are based on two dimensions. Furthermore, these systems are separated from the real world by a gaping chasm. Look at the flow chart in Figure 6-1.

Although this diagram contains much data about the program fragment it represents, it is useful only to a person trained in the meanings of rectangles, ovals, and diamonds. Furthermore, this "language" is focused squarely on the programming, not on the business. There is nothing in this diagram to suggest the relationship between people and processes in the business on the one hand to conditional branching and steps of the program on the other. Even worse, it has a "techy" look that might scare off people who do not have a formal background in computers.

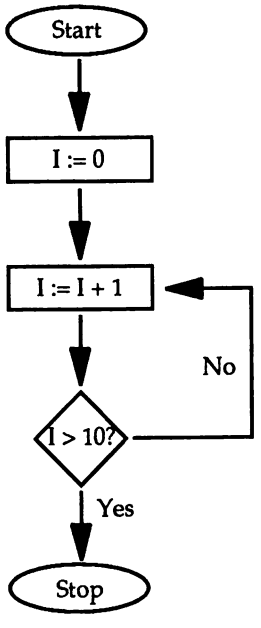


Figure 6-1. What does this mean?

Now look at Figure 6-2, which is a diagram illustrating the relationship between a class BUS and a class PERSON in an object-oriented program.

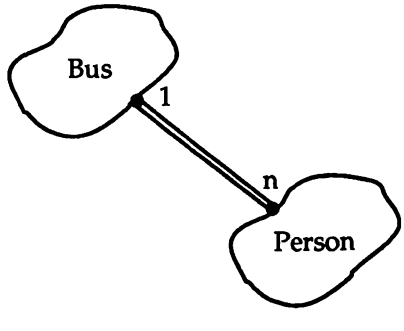
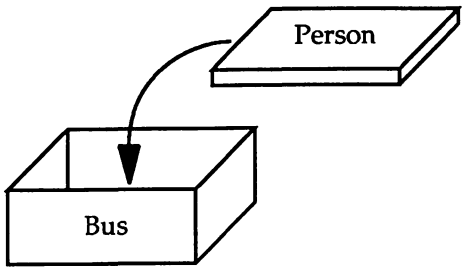
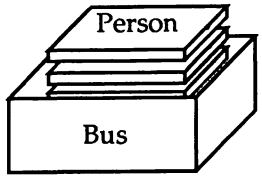


Figure 6-2. What does this mean?

It takes a great deal of explaining to communicate to someone else that this simply means that a bus contains some people. We have to explain that each blob represents a class of objects, that the double line means that BUS uses PERSON in its implementation (which, in turn, requires some



Good



Even better

Figure 6-3. People in a bus

explanation!) and that the 1 on one end and n on the other means that there are one or more people in the bus. Now consider Figure 6-3, which shows how to present the same concept in VDL.

Figure 6-3 conveys much more information than the previous diagram. The three-dimensional character communicates that these are things, not just abstract shapes. This corresponds to the intuitive notion of an object as a thing that has three dimensions. Second, we use an image schema to represent the fact that one is inside the other. The arrow clearly indicates that the PERSON is inside the BUS. This is still not ideal. The best version would show a bus with people inside it, but we need a notation that can be quickly sketched by hand in order for it to be useful as a working tool of communication. This is a quick glimpse at the kind of notation we will present: one that uses to advantage people's ability to grasp more information from image schemas than from words. As a result, we can communicate much more information at a glance.

The illusion of three dimensions in the design is a key element of this strategy. In two dimensions, people must make the mental translation from shape to "thing"; in three dimensions, the idea of a "thing" is apparent. The third dimension also provides a critical boost to the amount of information we can convey. In two dimensions, we can organize things only in horizontal and vertical dimensions using left/right, front/back

and foreground/background image schemas. Adding the third dimension adds above/below as well. The addition of the third dimension dramatically increases the amount of detail we can present without clutter. Perhaps most important is that the results are more pleasing to the eye. A diagram that looks better also communicates better. It encourages exploration and doesn't scare people off. Because it is a more intuitive, comfortable way to communicate, everyone's confidence in the process increases.

▶ Using Image Schemas

We will talk more about the use of specific kinds of image schemas later, but it is appropriate to give some examples now. Western culture uses many visual cues to compare things. For example, flows from left to right are interpreted as a time sequence, and items behind other items are considered ancestral. The larger an element is drawn, the more important it is in the context of the diagram. This can also be true of above/below schemas in which the more significant information is contained in progressively higher schemas. Likewise, foreground/background schemas convey context by using the foreground for the topic of discussion and the background for the context.

Grouping of related elements can be represented by using containers, as we did in Figure 6-3; using center-periphery schemas; by physical proximity in the diagram; and by separation into layers (a form of container). Communication of information from one element to another is understood as a flow or movement along a conduit or path. This is a particularly natural thing to do on the Macintosh, which is already rich in image schemas. A window is a metaphor for a container of information. The trash can and other icons have a three-dimensional look. It is becoming increasingly common, especially with System 7.0, to use simulated three-dimensional buttons and other interface features in Macintosh applications. Why not apply the same principles we use in Mac interfaces to our process for developing the software behind them?

▶ Constraints on the Notation

In order to make it a practical tool, the authors placed the following constraints on the development of VDL.

1. Users must be able to quickly and easily sketch all symbols. One of the authors—never in serious danger of being mistaken for an artist—sketched the diagram in Figure 6-4 in under ten seconds.
2. We do not rely on the use of color or any computer-aided tool. A pencil and paper should be the minimum configuration required.

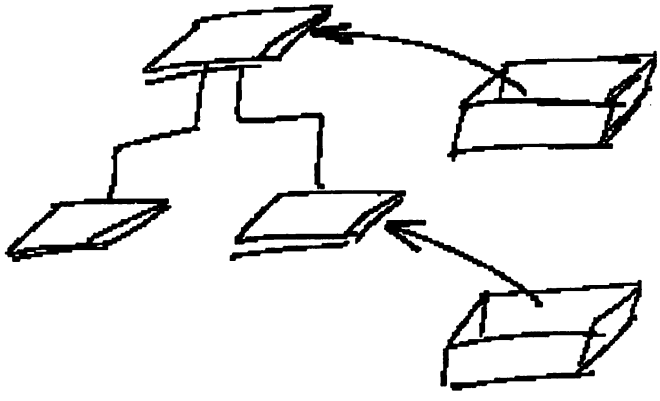


Figure 6-4. Hand-drawn sketch using VDL

3. At the same time, the notation should become even more powerful when automated. Color, information hiding, shadowing, and other advanced techniques should smoothly integrate with the notation where the right tools are available.

The end of this chapter contains a section on extensions to the notation to include color, shading, and other advanced techniques. We hope that others will be encouraged by this book to develop computer-aided tools for use with VDL.

▶ Contents of the Models

The models consist of *elements*, *relationships* among those elements, and *frames*. Elements include natural world and program objects, categories, classes, and a few other related pieces of information we will describe. Relationships between a pair of elements might describe the relationship of a whole to its parts or the sending of a message from one object to another. Frames are far less formal and represent constraints and requirements that are outside the model. For example, a constraint that response time to all actions must be under ten seconds becomes part of a frame.

Certain types of relationships are represented by lines and arrows. Others involve more subtle techniques of organization that we call *spatial effects*. We use a *plane* to organize elements into a single altitude, thereby implying, rather than explicitly stating, that the elements are somehow related to the topic of the plane. Planes are divided into *regions* for similar reasons. We use various other image schemas to represent relationships: front to back and other orderings, layering, relative size, and stroke weight of lines, among others.

▶ Examples of VDL

The best way to introduce VDL is to show some examples of its use. Figure 6-5 shows a simple *scenario*, or diagram, in VDL.

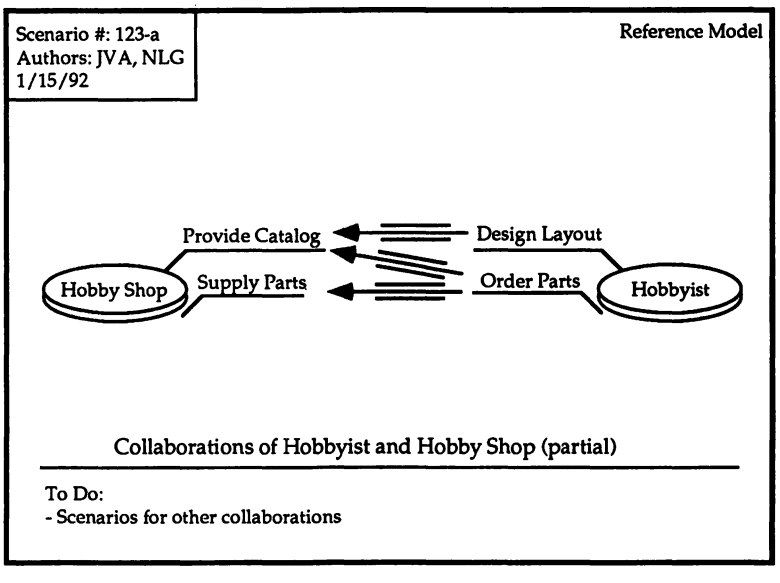


Figure 6-5. Relationship between model railroad designer and hobby shop

This scenario tells a simple message: The designer designs layouts and orders parts and is aided in those efforts by the hobby shop. This scenario describes a business situation with natural world objects and their responsibilities as the elements. Figure 6-6 shows a new scenario in which the Macintosh is now an element.

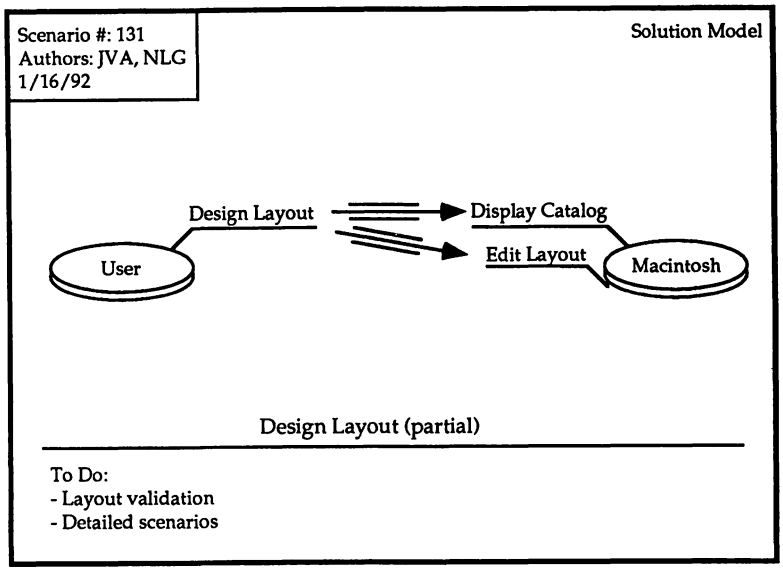


Figure 6-6. Using a Macintosh for model railroad layout

This is still at the level of business modeling with the Macintosh now included as an element of the business world. Figure 6-7 leaves the world of business modeling and presents a storyboard of the model railroad design program.

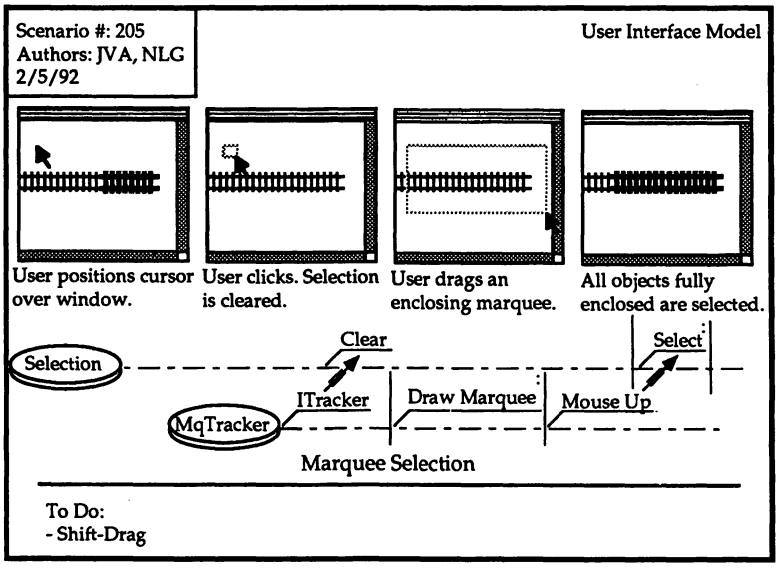


Figure 6-7. Storyboard of model railroad design program

The elements in this diagram are conceptual objects that represent the features found in the user interface snapshot shown. The left-to-right ordering reflects progression of time. Responsibilities are called in the order shown. This is still a conceptual model, not yet constrained by the specific technology of object-oriented software. That is, this is a descriptive *model*, as opposed to a technical *architecture* (design) or *implementation*. We will have much more to say about these three terms—model, architecture, and implementation, in later chapters. Figure 6-8 contains a much finer grain of detail. This is an example of the architectural level of a solution-based model.

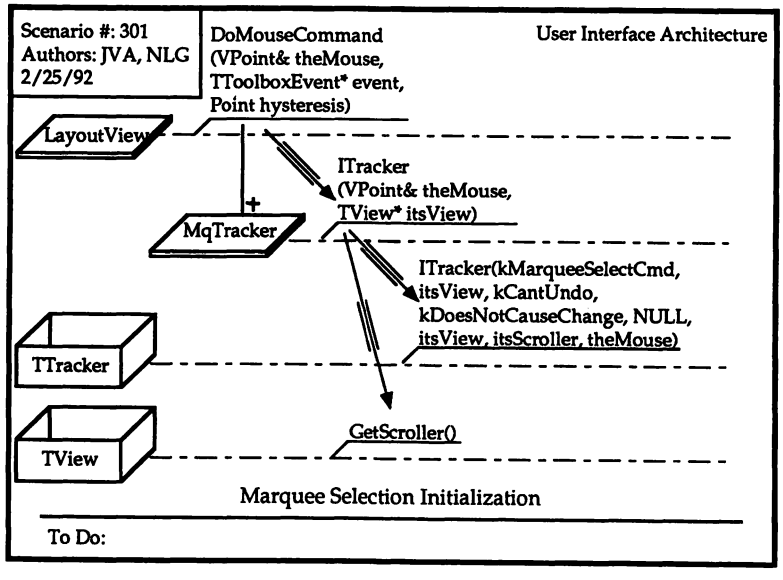


Figure 6-8. Architecture of model railroad design program

The conceptual description of the previous figure has been considerably expanded to include calling sequences for responsibilities and relationships between features of the user interface and the class library, in this case MacApp. What had been one object in the previous scenario has become two objects: a view object and a mouse tracking object. Figure 6-9 shows the relationships between these two scenarios by specifying how the objects and responsibilities of the conceptual model correspond to those of the architecture. The double-headed arrows mean "implements." Item by item, we compare elements of the two levels of detail to make sure that nothing has been lost in the translation from conceptual model to software architecture. Finally, Figure 6-10 shows the final level of detail, in which objects have been implemented using inheritance from specific classes in a class hierarchy. The box-like objects are classes; the arrows emanating from them are inheritance.

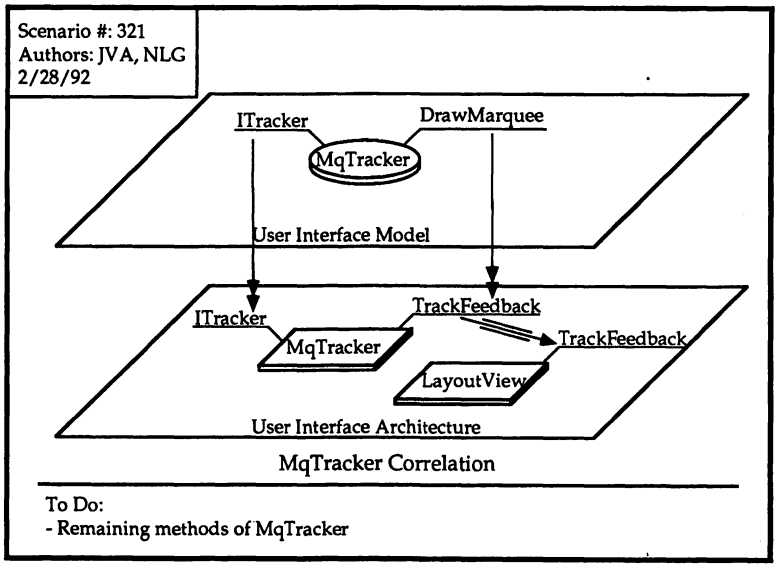


Figure 6-9. Correlation of model and architecture

This brief tour of VDL is intended only to whet your appetite. Now let's take an in-depth look at all of the symbols and conventions that make up the notation. Later chapters will use VDL almost exclusively to present examples and concepts.

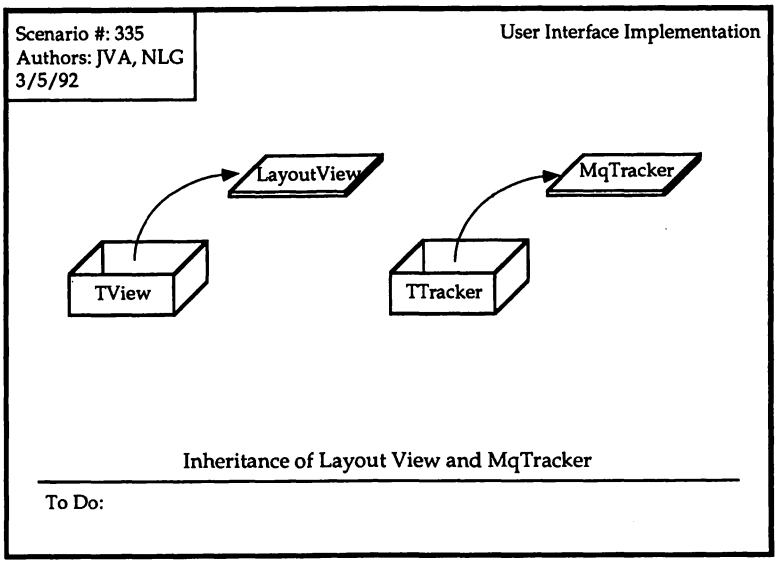


Figure 6-10. Implementation

▶ Elements

Elements are the individual items that provide the content of a model. VDL has as its elements natural world, or conceptual, objects and categories; program objects and classes; attributes; and responsibilities.

▶ Natural World Elements

We use the term "natural world" to describe objects and categories of the real world as well as conceptual objects or categories that describe the program. Natural world and conceptual elements are drawn using curves, and program elements are drawn using hard angles. This simple convention clearly indicates when we are talking in the user's terms and when we are using the technician's concepts. The symbology for natural world elements is shown in Figure 6-11.

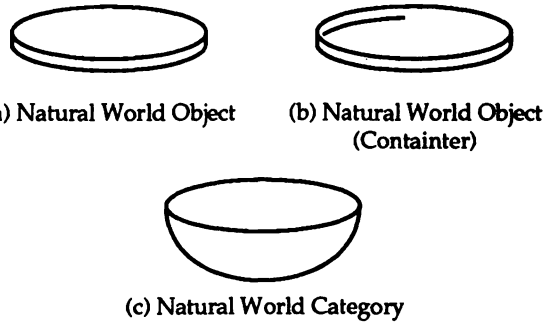


Figure 6-11. Natural world elements

Natural World Objects

A natural world object can be "John Smith" or "the Macintosh with CPU Serial Number 12345." A natural world object may or may not be capable of containing other objects. For example, a bus can contain passengers. If the object is a container, it appears as an open-topped disc, as in Figure 6-11(b); if it is not a container, it appears as a solid disc as in Figure 6-11(a).

Natural World Categories

Natural world categories are represented by the "bowl" shape shown in Figure 6-11(c), whose members are natural world objects. Note that it is open-topped, since every category is by definition a container of its members.

▶ Program Elements

Program elements include program objects, abstractions of program objects, and object classes. These are directly analogous to, respectively, natural world objects and categories. We use hard angles with program elements to identify them as technological creations rather than the real world or concepts. Figure 6-12 shows the symbols we use.

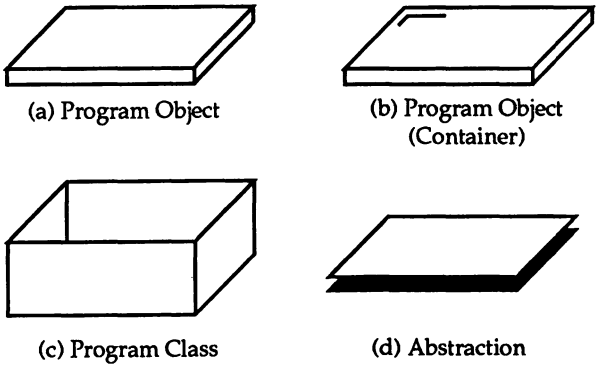


Figure 6-12. Program elements

Program Objects

Program objects are the objects created at run time by the object-oriented program. Where natural world objects can be somewhat imprecise, program objects must conform to strict rules imposed by the language of choice and our design. As with natural world objects, some program objects can also be containers. Those that are not containers appear as rectangular solids, as in Figure 6-12(a). Containers are open-topped, shallow boxes, as in Figure 6-12(b).

Program Classes

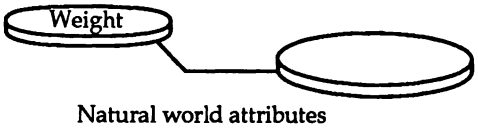
Classes are the substance of the program at compile time. They correspond to classes we directly implement in an object-oriented language. There are two kinds of classes, concrete and abstract. Concrete classes are those that we will actually instantiate as the program runs. Abstract classes exist only to share properties between other classes. For example, in MacApp the class TObject is the ultimate ancestor of all other classes. Although one never creates an instance of a TObject, all classes share its methods. TObject is an abstract class. Figure 6-12(c) shows the symbol for classes.

Abstractions

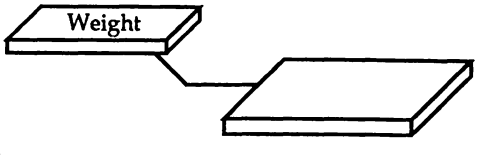
Closely related to program classes are abstractions of run-time objects. An abstraction is an assertion that certain objects share the attributes and responsibilities listed for the abstraction. Think of an abstraction as a shortcut that avoids the need to separately describe each run-time object. The symbol for an abstraction is shown in Figure 6-12(d). Abstractions are discussed in considerable depth in Chapters 10 and 11.

▶ Attributes

An attribute is a quantity or other piece of data about an object, category, or class. For example, for a freight train we might have attributes of gross weight, carrying capacity, and so forth. Creating symbols for attributes is a little tricky for two reasons. First, we generally want to hide them. Representing attributes as data goes against the idea of behavioral modeling, a cornerstone of SBM. If an object has the attribute "weight," we want to express that fact in terms of relevant behaviors: tell me your weight, change your weight, compute your weight. The second problem is language-specific. In some object-oriented languages, notably Smalltalk, everything is an object, including numbers. Thus, in those languages there is no such thing as an actual attribute. Nevertheless, it is useful to draw attributes where they apply and we need symbols for them. Figure 6-13 shows how to do this.



Natural world attributes



Program attributes

Figure 6-13. Attributes

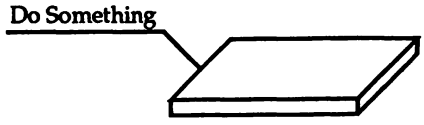


Figure 6-14. Responsibility

Notice that attributes take on the shape of the object or class they belong to. A natural world attribute, is a flat, oblong disc, and a program attribute, is a rectangle. This reflects the idea that an attribute is a small piece of an object or class. Notice also that we move attributes onto their own layer, slightly above the plane of objects, categories, and classes. This avoids confusion over what is an attribute and what is an object, category, or class.

▶ Responsibilities

Simplicity is the key for responsibilities because we have many of them. Furthermore, responsibilities do not have a ready visualization. For both reasons, we use text over a line connecting the responsibility to the element to which it belongs, as in Figure 6-14. As with attributes, we achieve a layering effect by use of the diagonal line that connects the responsibility to the element.

▶ Relationships

Most relationships between elements are visualized by drawing lines and arrows connecting the elements.

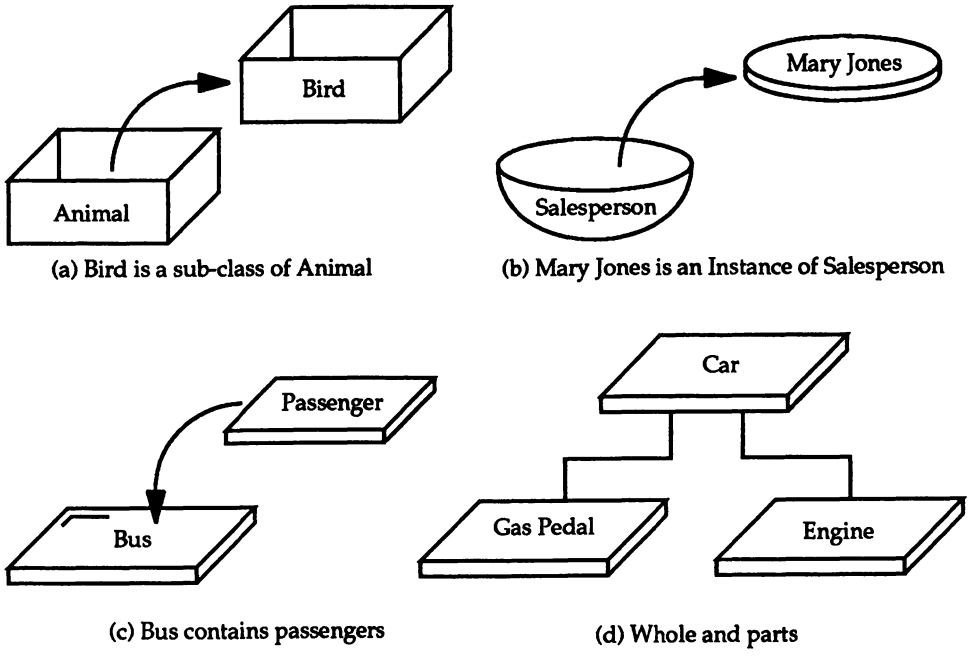
▶ Structural Relationships

Most of the relationships with which we are concerned are behavioral, but there are three important and common types of structural relationships to handle. Their symbols are shown in Figure 6-15.

Membership and Instance

Categories, classes, and abstractions have *members*. A category can have as its members any combination of objects and other categories. A class or abstraction can have as members any combination of objects and other classes. When an object is a member of a category, it is an *instance* of that category; similarly, program objects that are members of a class are instances of that class. We represent both membership and instances using

an arrow emanating from inside the enclosing category, class, or abstraction and pointing to the member. This shows that the member springs from or is derived from the category or class or is described by the abstraction. Figure 6-15 (a) and (b) show the symbols for these relationships.



(a) Bird is a sub-class of Animal

(b) Mary Jones is an Instance of Salesperson

(c) Bus contains passengers

(d) Whole and parts

Figure 6-15. Structural relationships

Containers

Containment is visualized by drawing an arrow from the contained element into the open top of the container, as shown in Figure 6-15 (c).

Whole/Part Relationships

The relationship of a whole to its parts is modeled on the kind of parts explosion diagram you might struggle with when assembling a bicycle on Christmas eve. We use right-angle lines coming from the top of the part and into the bottom of the whole, as shown in Figure 6-15 (d). It is not necessary that the whole be spatially above the parts, as in this example, but the right-angle lines must be used as described.

▶ Behavioral Relationships

There are three basic kinds of behavioral relationships: messages (also called collaborations, for reasons discussed later), creation of one object by another, and destruction of one object by another. The symbols for these relationships are shown in Figure 6-16. Figure 6-16 uses program objects throughout, but the same symbols apply to behavioral relationships involving categories and classes as well.

Messages/Collaborations

Communication between objects, whether described as a message or a collaboration, is represented by an arrow passing through a conduit, as in Figure 6-16 (a) and (b). In order to send information or commands from one object or class to another, there must be a responsibility for the sender that sends the message and one for the receiver to receive and act on the message. Figure 6-16 (a) shows a collaboration between unnamed responsibilities. Figure 6-16 (b), on the other hand, specifically names the responsibilities on each end. Eventually, Figure 6-16 (a) must be made more

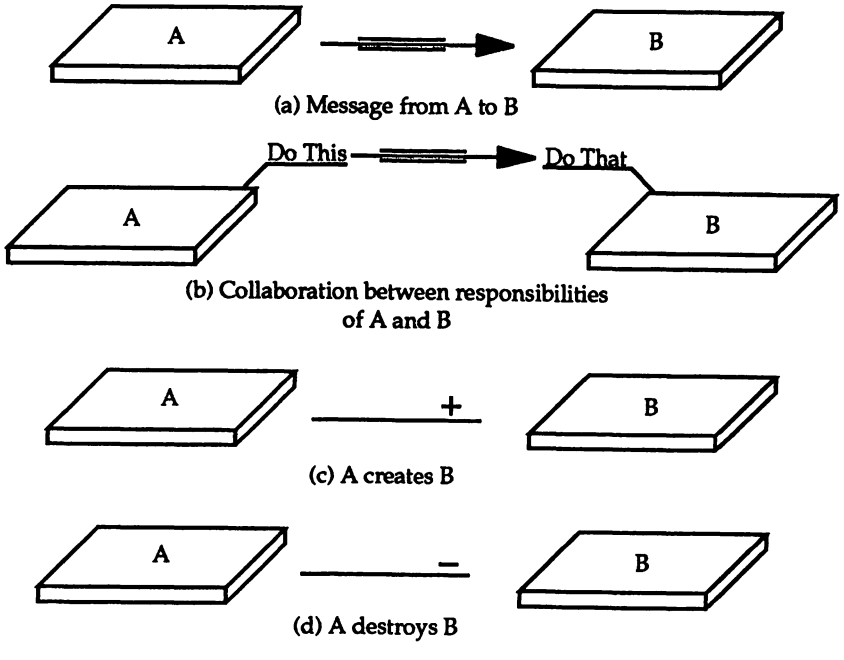


Figure 6-16. Behavioral relationships

specific by naming the responsibilities, but it is perfectly acceptable to suppress that information at an early stage of the project.

Creation

Objects are created by other objects. To show this relationship, draw a simple line with a plus sign (+) on the end toward the created object, as in Figure 6-16 (c). This line can either be drawn from objects, categories, and classes or it can be drawn from specific responsibilities, depending on the level of detail desired.

Destruction

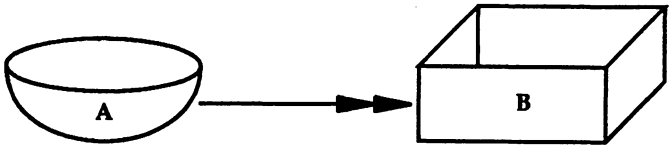
Objects can choose to destroy themselves, but this generally happens in response to a specific request to do so from some other object. To represent destruction of one object by another, draw a simple line with a minus sign (–) on the end toward the destroyed object, as in Figure 6-16 (d). As with creation, this line can either connect a pair of objects, categories, or classes, or it can connect a responsibility to an object.

▶ Calibration Relationships

In SBM, we frequently deal with overlapping descriptions of concepts. Categories can be used to capture some grouping of objects in one plane, but on the next plane one or more classes or abstractions can be defined to represent the same thing (though in terms we can implement in a program). Natural world objects and categories used to describe the way the business runs today may be reused, extended, or made obsolete by objects and categories used to describe the way the business will run with the system in place. For example, in a payroll application, we may learn that today Rose computes deductions, but the computer will do so in the new environment. We call these *calibration relationships*, since they are designed to validate that different slices of the same pie are consistent with one another. Figure 6-17 shows the two kinds of calibration relationships used in SBM.

Implements

An *implementation relationship* most often exists between a natural world element and a program element. We do not insist that our natural world elements maintain the level of rigor required of program elements. If a program element is the realization in an object-oriented program of a



(a) A is implemented by B

(a) A is replaced by B

Figure 6-17. Calibration relationships

natural world element, we say that it implements the natural world element. Figure 6-17 (a) shows the symbol for an implementation relationship, a double-headed arrow from the implementee to the implementor. This arrow can be drawn between objects, categories, classes, responsibilities, and attributes.

Replaces

It is often the case that something becomes obsolete in the process of developing software. Natural world elements describing the way the business functions today may not be relevant in the new, automated scheme of things. In later chapters, we will discuss the importance of not allowing *anything* to simply "fall off the face of the earth," even if it is made obsolete. Every change must be accounted for. To account for obsolescence, we use a *replacement relationship,* in which one element renders another obsolete. For example, in a payroll application, the need for certain staff functions—represented by natural world objects or categories—can be made obsolete by the computer.

The line between implementation and replacement is subtle. Implementation is used to indicate satisfaction of some requirement, and replacement is used to indicate that something is *no longer* required or relevant. Figure 6-17 (b) shows the symbol for a replacement relationship.

▶ Same As

There are only so many ways to organize symbols on a page, even in three dimensions. At times, whether for clarity or necessity, it is convenient to have the same element appear in two or more places in a single diagram. To indicate that two elements are, in fact, the same thing, use the symbol shown in Figure 6-18.

▶ Spatial Effects

The kinds of relationships discussed so far are formal parts of the models you build using Solution-Based Modeling. For each of these we use a symbol. Other types of relationships are better illustrated by position, size, and other *spatial effects* than by lines and arrows. These are techniques of graphic design that organize and present the models, often in ways that indicate emphasis as well as content. You can use layering and separation, relative positioning: left/right, front/back, above/below, foreground/background, and size and line weight or any other technique that clearly presents the information.

▶ Planes and Regions

A solution-based model is organized into *planes* that roughly correspond to the activity required to construct that part of the model. One plane represents the way the business runs today and the way it will run with the new system in place. Another represents the objects that exist as the program is running, and another includes the classes, both concrete and abstract, that make up the program itself. Separate *regions* exist within each plane. We will talk more about the specific planes and regions of solution-based models in Chapter 7.

Note the altitude effect in which planes appear to be above or below other planes. Regions are simply subdivisions of planes; the choice of ovals or lines to delimit regions is up to you. By placing other elements on these planes and regions, we communicate a great deal about the overall

Two copies of A

Figure 6-18. Same-as relationship

organization of the model. We can put a lot more detail onto a single page without generating clutter.

We also use planes in more limited ways. For example, we drew attributes and responsibilities as if they are slightly above the plane of the elements they belong to.

▶ Time Sequence

Western culture interprets left to right as a time sequence. We use that automatic interpretation to describe the time sequence of messages between objects in VDL. Figure 6-7 shows an example. Rather than draw the same element repeatedly, we extend a dashed *time line*. Responsibilities involved in the interaction are placed along these lines at the point in time at which they are called. If an object is created at some point, it first appears at that left/right location; likewise, if an object is destroyed, its time line disappears. Figure 6-17 also shows a variation of this technique that communicates repetition. The convention used is borrowed from musical notation. The segment between vertical lines repeats.

▶ Relative Importance

Not all elements and relationships in a model are equally interesting. There are several ways in which we can at once draw the viewer's attention toward some symbols and away from others. Remember the critical role centrality plays in human categorization. We dramatically increase the viewer's intuitive grasp of the model by communicating what is central and what is peripheral in importance.

Size

Since things that are bigger automatically attract more attention than things that are smaller, the simplest way to emphasize elements is to draw them bigger than others. This effect is demonstrated in Figure 6-19.

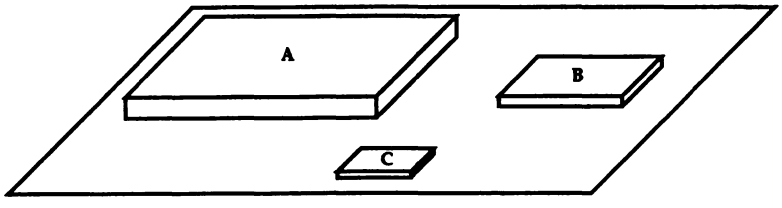


Figure 6-19. Use of size to emphasize importance

Line Weight

We have avoided using line weight for any purpose in VDL so that it can be used as a tool of emphasis. Look at Figure 6-20. To what is your attention immediately drawn?

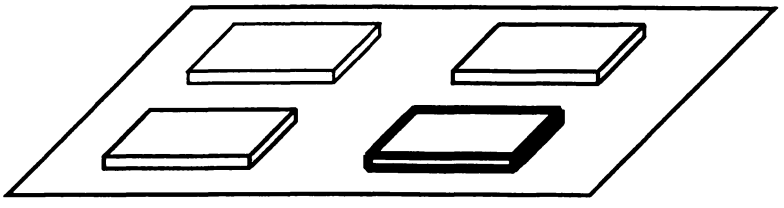


Figure 6-20. Use of line weight for emphasis

Center-Periphery Organization

Figure 6-21 illustrates that an element in the center is somehow more . . . well, central to the diagram than the other elements are.

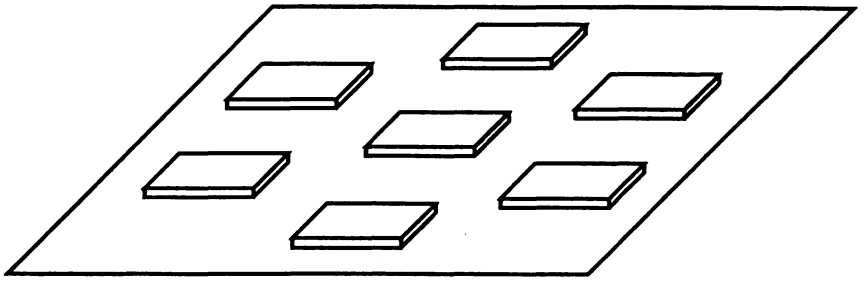


Figure 6-21. Use of center as a cue

Foreground/Background Organization

Foreground/background organizations can also help focus attention. We naturally concentrate on things in the foreground, looking at the background only to establish context. Our use of regions is based on a foreground/background organization in which the elements and relationships above the plane or region are foreground and the plane or region is background.

▶ Frames

Elements, relationships, planes, and regions are internal to the model. Frames represent external considerations, principally constraints. Examples might be, "it has to be implemented using a Macintosh"; "response time must average less than two seconds"; or "overall labor must be reduced." These are more like notes than "things."

We use a foreground/background schema to place the model frame in the *negative space* surrounding the model itself, as shown in Figure 6-22. Constraints and other features of the frame are simply noted in the background using text.

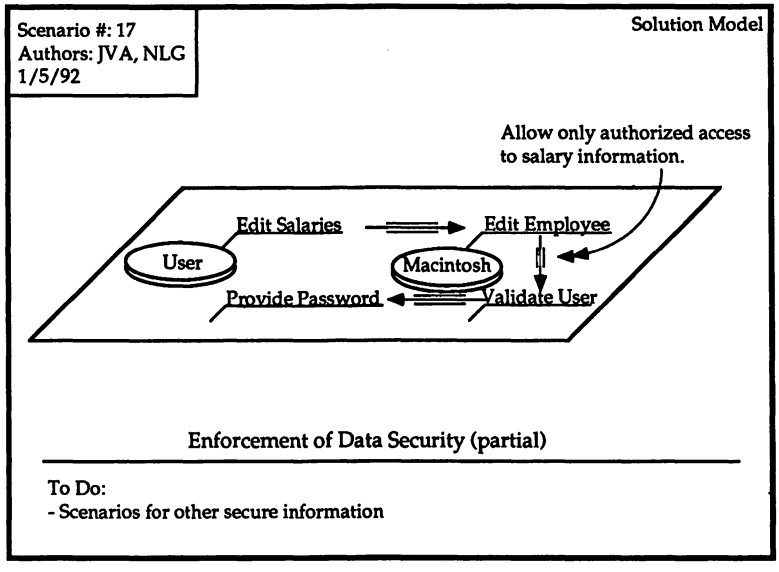


Figure 6-22. Drawing frames in the background

▶ Scenarios

Any useful program is complicated enough so that it is not comprehensible in a single model. Even with a large piece of paper, a model encompassing all of the detail of an application would look like a city viewed from an airplane at 20,000 feet. People need to deal with a small piece at a time. In addition to human cognitive limits, there are practical considerations. Organizing elements into a time sequence is frequently at odds with the

way we organize them structurally. For that matter, two different sequences of events might require entirely different left to right orderings of the same elements! Emphasizing relative importance, centrality, and other spatial effects are usually only meaningful in relationship to some limited topic of interest. An element may be important in one sense and unimportant in another.

For all of these reasons, we try to deal with small models called *scenarios*, which collectively make up the overall solution-based model. Each scenario has a single topic, generally consists of fewer than a half dozen elements and a subset of their relationships, and makes its own use of spatial effects, independent of the overall model and other scenarios. Ideally, a scenario should be a gestalt, a whole that is taken by the viewer as being more than just the sum of its parts. Figures 6-5 through 6-10 and 6-22 are all examples of scenarios. Notice the characteristic features that provide document control: an identifying scenario number, including a version suffix if appropriate; author initials; date; title, and a "To Do" list at the bottom. For most scenarios, the part of the model addressed is indicated, as in the top right corner of Figure 6-5.

At times it is useful to organize scenarios hierarchically so that a single scenario as a whole is represented as a single element in a larger scenario or in the overall model. For example, we might wish to represent an entire car as a single "element" in the overall model and break out the relationships to its parts in a smaller scenario. Figure 6-23 shows the symbol for this.

Figure 6-23. Scenario element

This symbol is allowed in the overall model or in any scenario drawn from it. A special use of the scenario symbol is to indicate if-then or switch-case logic. Figure 6-24 shows an example of this. Only one of the indicated paths will be followed.

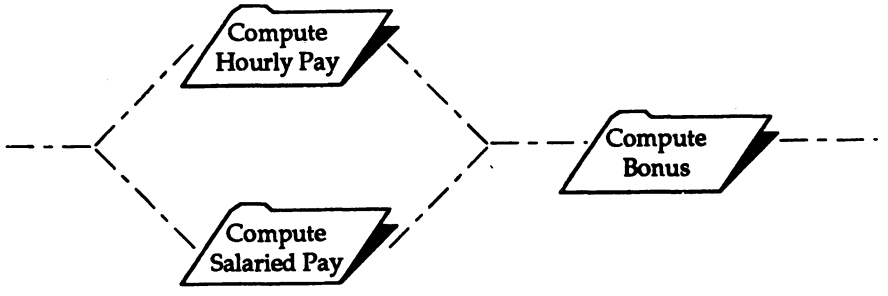


Figure 6-24. If-then logic using scenarios

▶ Vertical Slicing

Planes are the primary spatial organization of elements in our models, but there are times when our interest is in some topic that spans planes. We call this *vertical slicing,* to contrast it with the horizontal organization into planes. Vertical slicing can be represented using shading to visually connect regions or elements from different planes, as illustrated in Figure 6-25.

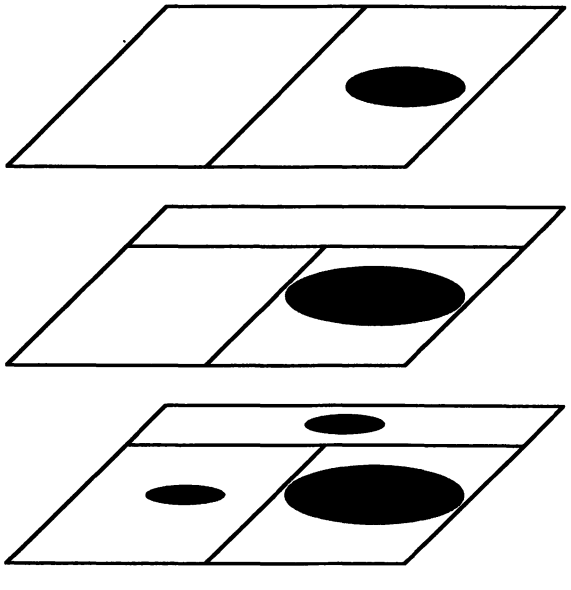


Figure 6-25. Vertical slicing: the shaded regions are associated

▶ Extensions

So far, we have touched only on what could be called "standard usage" of VDL, but there is certainly much more that could be done. Making the symbols appear more realistic is a great aid to understanding, but requires the use of a computer to do the copying and pasting. Similarly, the power of front/back, above/below, and other spatial schemas is tremendously enhanced if perspective is introduced, and shading amplifies the effect of layering. These are techniques that are amenable to automation, but are too time consuming when done by hand.

Used wisely, color is also a helpful tool. For example, drawing important features—elements and relationships—in the normal black while casting everything else in a light red can impart a remarkable sense of separation. Color can also be used to group features to much the same effect as planes and regions, but without having to worry about spatially organizing them. Progressions of colors from left to right can also amplify the role of left/right schemas to represent time sequences. However, it is important not to go overboard with color. It needs to be used sparingly as a supplement.

Finally, as costs of the technology continue to fall, photorealistic rendering and full 3D editing of scenarios promise to make VDL even more powerful and expressive.

▶ Summary

Figure 6-26 recaps all symbols and conventions in VDL.

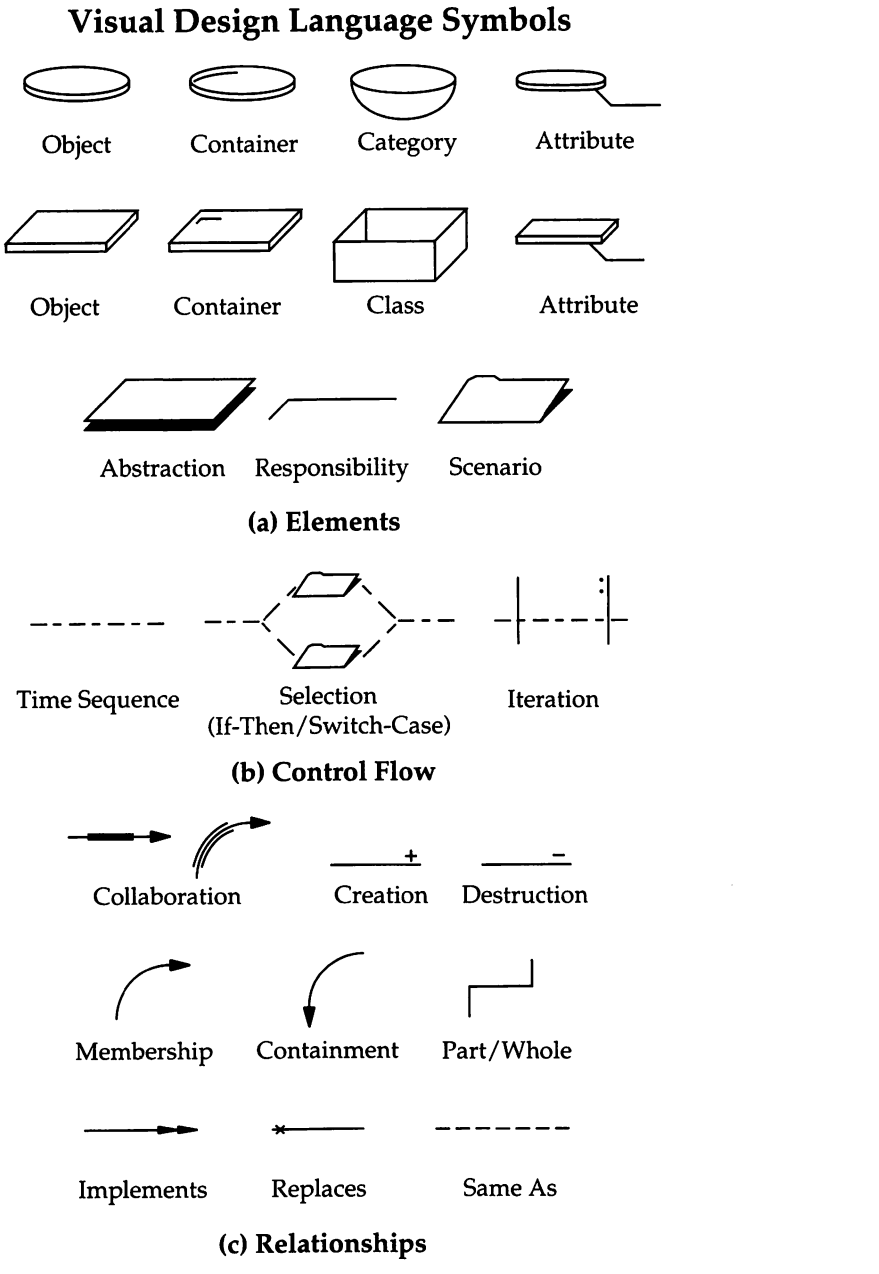


Figure 6-26. Summary of all symbols and conventions in VDL

7 ▶ Solution-Based Modeling

▶ What This Chapter Is About

This chapter introduces Solution-Based Modeling (SBM) for the Macintosh. We start by clearly stating our objectives, then proceed through solution-based models and Solution-Based Modeling, covering the content and process, respectively. At the end of this chapter you will understand how to use the methodology and what you will produce as a result.

This chapter is divided into three parts: objectives motivating SBM, discussions of the models, and the process used to create them. The models consist of four planes and eleven regions that together run the gamut from business analysis to design to program code. The process uses a technique called Center-Periphery-Calibrate (CPC) to build the models. CPC starts with the center of the problem, explores it at several levels of detail, then expands outward toward peripheral issues. Along the way, new work is constantly calibrated to old to ensure consistency.

A Solution-Based Modeling project has four phases: analysis, design, programming, and ongoing evolution. However, these phases are not the same as in a traditional, linear process. During each phase, activity spans business analysis, design, and programming. The phases differ only in the relative mix of these activities.

▶ Objectives

The two fundamental objectives of Solution-Based Modeling are solving the right problem and creating reliable, maintainable programs. These objectives address the findings presented in Chapter 1, namely that

(1) software projects fail most often because they solve the wrong problem or create the wrong solution; and (2) most software dollars are spent on maintenance, not development.

▶ Solve the Right Problem

In order to solve the right problem, programmers and non-programmers must each be able to understand what the other is doing and saying. Neither programmers nor end users alone have all the answers. A team effort is needed with good communications. Two specific objectives follow.

1. Create requirements and designs that non-programmers understand. This allows end users and others to contribute new ideas and critique work done to date.
2. Create business models that software engineers understand. Remember one of the realities of software development from Chapter 1: The project team seldom has the necessary knowledge of the problem to be solved. In order to arrive at the right program, the problem and its solution must be expressed completely and in a way that makes sense to the technical staff.

Several tactics can be used to achieve these twin objectives.

- We rely heavily on the techniques of visualization discussed in Chapter 6 because people best understand abstract concepts like business problems and computer software visually.
- A solution-based model is built on a foundation of categories, not classes, because people organize their perceptions in terms of categories.
- Solution-Based Modeling starts with models of the natural world around us before plunging into software and other abstract concepts because people agree most when discussing the real world and the things with which they interact.

Solution-Based Modeling also recognizes the impossibility of getting the solution right or even knowing all the right questions the first time. This is one of the worst-kept secrets of software development: Seasoned software professionals, even those supposedly using structured, linear methodologies, know that good software is not really created in an orderly, linear manner. That emperor has no clothes. Good software is

only built in a series of incremental steps. Instead of the traditional handoff from analysis to design to programming, SBM combines all three, using a procedure called Center-Periphery-Calibrate. CPC corresponds to the way expert designers work, not just in software, but in all creative fields—you address a few central problems first at several levels of detail, then expand outward toward the periphery. As you add more detail, constantly calibrate the new to the old to maintain consistency throughout the model. This is the natural way to develop software.

▶ Create Reliable, Maintainable Programs

Solution-Based Modeling uses sound software engineering principles specifically adapted for the world of object-oriented programs. This, plus its grounding in the relatively stable natural world, yields efficient, reliable, and, above all, maintainable programs. Modularity, independence, code reuse, and other traditional software engineering concepts have their counterparts in SBM, even though object-oriented software does not lend itself well to the traditional interpretations.

▶ Solution-Based Models

Solution-Based Modeling is based on the idea that any software development project is a process of constructing models. In SBM, we build models of the business before automation (the Reference Model), and then project them into the future, after the program is put in place (the Solution Model). We build architectures for the program that, when implemented, achieve the intended business solution. The program itself is an implementation of that architecture. The combination of all of these is a single Solution-Based Model, which describes the business today, where it must be tomorrow, and the technology used to get there.

Four planes divided into a total of eleven regions comprise a single Solution-Based Model.

- *Business Plane.* The way the business runs today and the way it will run with the new program in place. The regions of this plane are models of the business.
- *Technology Plane.* A conceptual model of the program. The regions of this plane are models of the user interface and contents of the program.
- *Execution Plane.* The objects that exist in the computer as the program executes. The regions of the Execution Plane together provide a detailed architecture for the program.

- *Program Plane.* The program itself. We call the regions of this plane implementations.

Figure 7-1 shows all planes and regions in a solution-based model. Note the parallelism of the regions across the planes.

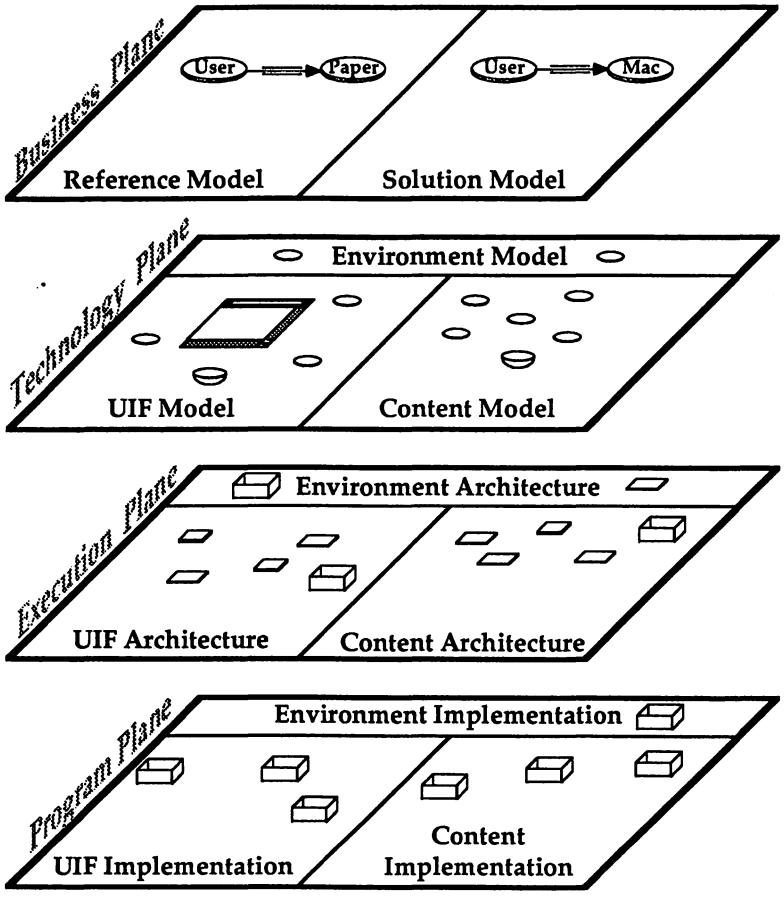


Figure 7-1. Planes and regions in a solution-based model

Elements of the planes and regions were described in Chapter 6 and are summarized in Figure 7-2.

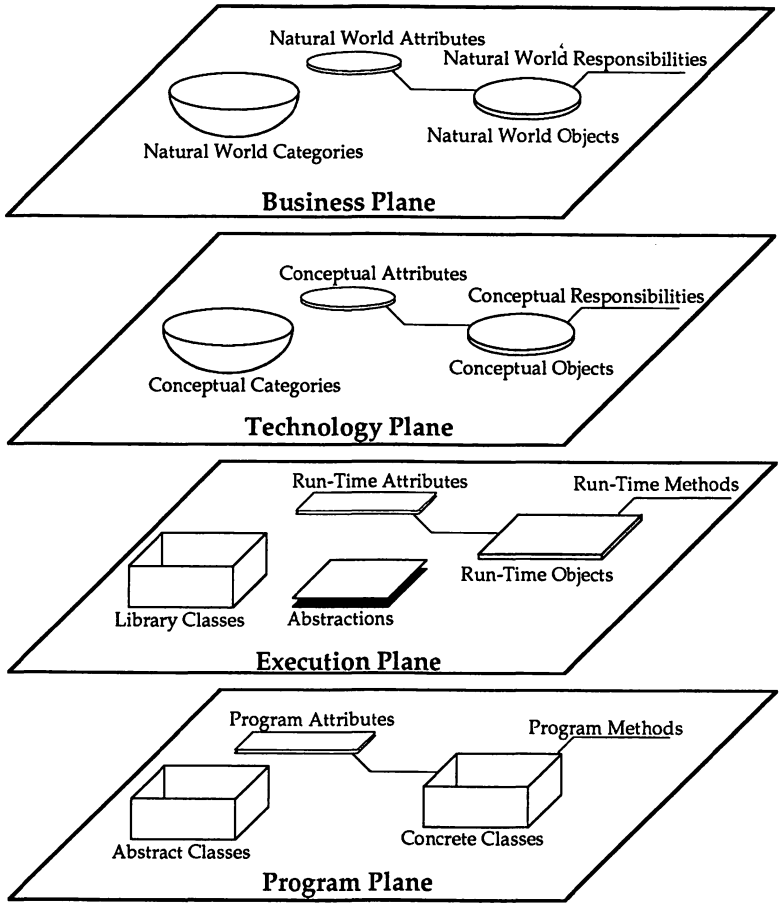


Figure 7-2. Elements of solution-based models

▶ Business Plane

The business environment that surrounds the computer system is modeled in the Business Plane, which contains the *Reference Model* and the *Solution Model* as regions. In order to know your objectives, you must first understand the business as it exists before automating. This is captured in the Reference Model. The Solution Model describes the way the business will run with the program in place.

Natural World Categories and Objects

The Business Plane has as its elements categories of the world around us and natural world objects—people, machines, processes, existing computer systems. As categories, they represent groupings that make sense to the people in the business. They need not share properties other than the simple fact of membership. Properties that are assigned need not correspond to data types or calling sequences in a programming language. They are descriptive of the business, not statements in a programming language. There are several advantages to using natural world categories in the Business Plane.

- They are meaningful to people who have knowledge of the problem but are unfamiliar with computers or object-oriented software.
- They are based on the relatively stable "things" of the real world rather than abstractions of computer science.
- Since natural world categories and objects model the real world, they allow us to easily demonstrate how things will change in the business once the new program is put to use.

Both models seek to capture the real behaviors of the world. As discussed in Chapter 5, there are often conflicting, overlapping ways to form categories or perceive objects. It is not necessary to rely on objectivism, we just need to form models that are meaningful to the people involved.

Reference Model

The Reference Model is critically important. It is grounded in the things that really exist and the way things really are today. This makes it the easiest model for all parties to understand in the same way. All other parts of a Solution-Based Model are hypothetical. The Reference Model is our stake in the ground, the "you are here" on the map of the development project.

Solution Model

The Solution Model also consists of real things and real behaviors, but projected into a future in which the program is in use. This, too, is a model that everyone should be able to comprehend and use. Two kinds of elements are especially important here: people who use the computer and the computer itself. Users have behaviors that require use of the computer to achieve objectives, and the computer has behaviors that allow it to collaborate with those users in carrying out their responsibilities.

The Solution Model must be consistent with user requirements, human factors, capabilities of the technology, comparisons of costs to benefits, and many other factors. The Solution Model sets the scope of the project and drives the creation of the other planes.

Impact Analysis

Differences between the Reference Model and Solution Model are collected into an *Impact Analysis*, which accounts for all changes to the business as the result of installing the new program. Especially important are changes and additions to the responsibilities of people. If the computer is already in use, the Impact Analysis also accounts for changes and additions to the responsibilities and behaviors of the computer. The Impact Analysis serves as an important cross-check that helps to locate errors or omissions in either model through an item-by-item comparison.

The Impact Analysis is as important an outcome of Solution-Based Modeling as the software itself. Impact Analysis allows you to take into account the fact that introducing a new computer system changes the way you do business. Analysis and design should take into account all changes for everyone in the organization who will be affected in any way by the application. Put another way, the entire organization, not just the technical staff, must be involved in order to deliver critical software projects. The Impact Analysis and the use of categories are both tools to facilitate reaching out beyond the walls of the software department to include the entire organization.

▶ Technology Plane

To speak meaningfully about the role of the computer in the business requires having a model of what the computer contains and how it interacts with its users. This is the *Technology Plane*, and it contains three regions: a *Content Model*, which describes the interior of the program; a *User Interface Model*, which describes its exterior; and an *Environment Model*, which describes how the program interacts with other programs, hardware devices, and networks.

Cognitive Categories and Objects

Like the Business Plane, the Technology Plane is composed of categories and objects, but these elements might not really exist in the world. The program isn't yet written or in use, so its content and user interface are not yet real. We are taking part of the real world and replacing it with a computer. We make up what the computer's "inner world" looks like.

Thus, the content of the program is not a real world structure, but a creature of the mind.

This does not mean that we have to abandon categories just yet. People form new cognitive categories all the time in response to need and experience. Eighteenth-century English farmers did not have a category "Graphical User Interface." We rely on this ability to form new categories in building models of the content and interface of the computer.

Some of the new categories will be metaphors for "things" in the real world. Metaphor is a familiar technique on the Macintosh. For example, the Macintosh "desktop" simply describes some of the contents and operations of the computer. We don't insist that the Content Model be the real world. To the extent that we can metaphorically project the real world, we can create models that are easily understood.

We have already seen a few examples of the use of metaphor in the expert solutions of Chapter 4. In the payroll program, employee objects compute their own pay and paycheck objects format and print themselves. In the model railroad example, track lays itself and layouts validate themselves. By extending the use of categories down to the Technology Plane, we keep end users and other non-programmers in the loop as long as possible. The Technology Plane is understandable by end users because it is created and described in terms of metaphor and categories, not technobabble. It is parallel to the structure of the program. From here, it is simple to derive program classes and program objects. The Technology Plane brings together people who understand the problem and people who understand the technology.

Content Model

The Content Model contains an idealized model of the objects, categories of objects, and categories of categories that the computer system contains. It is perhaps easiest to describe the Content Model by outlining the way it is built: (1), collect the responsibilities of the computer from the Solution Model; (2), create a series of conceptual objects and categories based as much as possible on the metaphors for natural world; (3), map the computer's responsibilities onto those objects; (4), refine the objects and their categories.

User Interface Model

The Solution Model contains the specifications for the User Interface Model. It lists all responsibilities of the people who use the computer, as well as the responsibilities of the computer itself. For each responsibility

of a computer user in the Solution Model that calls for use of the computer, there must be one or more corresponding features of the user interface that allow that responsibility to be carried out. For each responsibility of the computer in the Solution Model, there must be some way, through the user interface, to cause that responsibility to be executed.

It is in the User Interface Model that we start to introduce dependencies on the Macintosh platform. The Macintosh human interface guidelines and Toolbox provide much of the available user interface "language" such as radio buttons, scroll bars, icons, windows, menus, and so on.

Using categories, not classes, to build the User Interface Model is a critically important decision. Let's consider an example to see why. In the model railroad application, various things have been lumped into the category "Scenery." These items include buildings, trees and shrubs, modeling material, and a long list of other kinds of objects. To the user, it may make perfect sense to create a palette that represents this category, as shown in Figure 7-3.

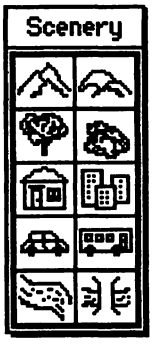


Figure 7-3. Scenery palette

However, this grouping is not based on any shared properties of these subcategories. What does modeling material have in common with a building? Not much. If we really reach, we can say that they both draw themselves on the screen and respond to mouse clicks, but that is equally true of components in other palettes, like those in the track palette shown in Figure 7-4.

In fact, if we built the palettes based on shared properties, we would probably arrive at a totally different arrangement. Based solely on shared properties, it is not unreasonable to put buildings that have electric lights in the same palette with controls for switches because both require power.

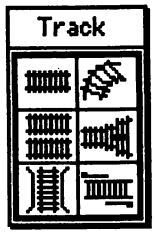


Figure 7-4. Track palette

This arrangement would be convenient for the programmer, who could use a class for two purposes, but not for the user, who will perceive this as an arbitrary, jumbled collection of unrelated things.

This is only one example of the importance of using categories rather than classes sharing properties to create the user interface. One of the trickiest decisions in designing a user interface is how to group things into windows, palettes, menus, lists, radio buttons and other controls, and views within windows. These decisions become easier when you stop looking for mathematical purity and accept that the categories that make sense to the user may share no properties whatever, may overlap, or may appear to be totally arbitrary. It is the user's perception that counts. The user interface's sole purpose is to communicate with the user. It need not be a direct reflection of underlying program structure, but rather a direct reflection of the way users think about their jobs. It is the developer's job to provide the most natural medium of communication possible.

Environment Model

Some programs stand alone, while others interact with other programs, specialized hardware devices, or computer networks. These external entities are represented by objects in the Environment Model. These objects and their responsibilities are used to describe interactions with the program being developed. They are not literal models of the entities. There are strong analogies between the User Interface Model and Environment Model; both describe external ways of interacting with the underlying content of the program.

Separating Content from Interface

Why do we separate content from user interface? If you need to explain how to drive a car to someone who has never seen one before, you would start not with the basic principles of carburation and hydraulics but with the basic function of a car—transportation—then describe its most impor-

tant controls, gas pedal, steering wheel, and brakes. A car contains thousands of parts, a small few of which the typical driver sees or uses in order to drive. Those controls are collectively the user interface to the car, directly analogous to the user interface of a computer program.

Once the basics have been mastered, you can move on to other subjects, such as fuel. You can stick to directions like, "When the needle on the fuel gauge approaches E, unscrew the cap on the rear fender and pour gasoline into it." This deals only with external features of the car. More likely, you would say, "The car has an engine that burns fuel. It also has a gas tank that holds the fuel. The gas gauge on the dashboard shows how much gas is left in the gas tank. When it shows that the fuel is getting low, you have to put more gas into the tank, and here is how." You have now created a model of the contents of the car, engine and fuel tank, in order to explain how to use its interface.

The Technology Plane is organized along the same basic lines of use and content. The User Interface Model explains in great detail how the computer can be used. The Content Model describes the inside of the program in terms of objects that hypothetically exist behind the user interface.

It is untrue that the Content Model exists only to describe the User Interface Model. If anything, the opposite is true: The Content Model is more stable and a more direct reflection of the Solution Model than the User Interface Model; it is a short step from being a business model. It contains as elements conceptual objects and cognitive categories that are derived, perhaps metaphorically, from the natural world. The behaviors assigned to those elements are taken directly from the Solution Model, which, as a pure business model, is relatively independent of platform and technology. In fact, it is common to build ahead in anticipation of future changes by putting objects and responsibilities into the Content Model for which there is initially no control in the User Interface Model. We can do this because the elements in the Content Model are familiar and, therefore, relatively easy to flesh out.

The User Interface Model, on the other hand, is heavily dependent on the Macintosh and, to a lesser extent, the class library you choose. This grounds it more in technology than in the natural world. In order to understand it, you must have a good command of how a Macintosh works and what makes for good and bad interfaces. As a creature of technology, the User Interface Model is relatively unstable. One of the most common changes over the life cycle of a program is to change its user interface while retaining the same basic functionality. Changes in system software can also cause changes in the interface. By separating the content and user interface, you minimize the side effects and complexity of such changes.

The biggest motivation for separating content from interface is to separate clearly what is grounded in the natural world from what is purely technological. To the extent that we rely on the natural world, we build in stability and build models that are easily understood by all parties. The further we move from the natural world, the more difficult it is to build stable, understandable models.

▶ Execution Plane

Any object-oriented program can be viewed either as run-time objects scurrying around doing useful things or as a set of statements written in a specific programming language. The objects that exist at run time and their characteristics are in the *Execution Plane* of the model, and the program itself is in the *Program Plane.*

In Chapter 2 we pointed out that inheritance and polymorphism are not strictly required to call a program object oriented, although few people would want to write programs without them. As a run-time concept, each object contains only data members for attributes and methods that carry out its behaviors. Although inheritance and polymorphism are concepts that can implement data members and behaviors in different ways, they are only tricks of the implementation. The Execution Plane sticks to objects and behaviors, the bare essentials of object-oriented software. Because it strips away all but the final result of specific objects in the computer carrying out specific tasks, the Execution Plane serves as the overall architecture for the program. The architecture described in the Execution Plane is independent of any specific programming language and implementation. The Execution Plane has another, subtle benefit. Because it relies on only the bare minimum of object-oriented concepts, it is accessible to people who are neither programmers nor fluent in object-oriented languages.

There are three regions in the Execution Plane.

1. *Content Architecture.* Run-time objects that together implement the Content Model.
2. *User Interface Architecture.* Run-time objects that together implement the User Interface Model.
3. *Environment Architecture.* Run-time objects that implement the Environment Model, plus the structure (in objects) of the program itself, including the main event loop, event dispatching, file handling, and the like.

Classes and Program Objects

The Execution Plane has as elements program objects and abstractions of objects. It is the objects themselves in which we are really interested. Ideally, we would list each run-time object individually, but it is usually not possible to do this. Therefore, we use abstractions as a shorthand notation for sets of run-time objects that share all or part of their features. We do not worry about inheritance or polymorphism.

In the payroll example, all EMPLOYEE objects have a responsibility to compute their gross compensation. This saves saying, "The object representing Maribelle Fernwilder computes its gross compensation, the object representing Pete Peterson computes its gross compensation," and so on. In this abstraction, we describe an entire set of objects (EMPLOYEES) in terms of strictly shared properties.

Where the Technology Plane has responsibilities, the Execution Plane has interfaces in either pseudocode or whatever object-oriented language you are using. Each element is described in terms of its interface.

Content Architecture

The Content Architecture specifies the run-time objects that together implement the Content Model of the Technology Plane. It is not necessary that the objects in the Content Architecture correspond one for one to objects of the Content Model. However, every object in the Content Architecture should be accounted for in some way, either as the implementation of an object in the Content Model, replacing some object in the Content Model, or simply as something new. Similarly, responsibilities in the Content Model must be accounted for in the Content Architecture, in many cases as method interfaces.

User Interface Architecture

The User Interface Architecture maps the user interface elements of the User Interface Model onto classes of whatever class library is being used. It is in this area that class libraries are the most help. Class libraries like MacApp or the Think Class Library provide a wide variety of classes to ease the implementation of standard Macintosh user interface features. The better the class library, the less effort is expended in this mapping.

Environment Architecture

Strip away all drawing and all data content from a Macintosh application and you are still left with a control structure that glues everything else together and to the Macintosh platform. This is the Environment Architec-

ture. The Environment Architecture refines and expands the Environment Model much as the Content Architecture refines the Content Model. To features derived from the Environment Model we add those control structures common across Macintosh applications:

- Main event loop
- Event handling and dispatching
- Interrupt handling
- File management
- Networking
- Other operating system and Toolbox services

These are the features of the Macintosh, its operating system, and Toolbox. As with the User Interface Architecture, it is likely that this Architecture consists of subclassing off-the-shelf classes from a class library. For example, in MacApp, the classes TApplication and TDocument handle most of the details one normally needs in this area.

▶ Program Plane

Viewed through a text editor in a development environment like the Macintosh Programmer's Workshop or Think C, our object-oriented programs contain classes, abstractions, polymorphic methods, and a host of other technological tricks intended to make the program easier to write, more reusable, faster, smaller, and cheaper. More is involved than simply breaking down the system into objects: there are optimizations such as using inheritance to reuse code and classes to act as templates for object creation. The program itself, with all its technobabble, is the Program Plane. The Program Plane is an attempt to optimally implement the architecture created in the Execution Plane. Its regions parallel those of the Execution Plane: Content Implementation, User Interface Implementation, and Environment Implementation. The Program Plane is responsible for producing at run time the objects specified in the Execution Plane.

Attributes and Abstractions

The Program Plane contains classes as elements. Concrete classes are instantiated at run time to produce the objects of the Execution Plane. The Program Plane also contains abstract superclasses, which are classes that exist only to pass along their interface and/or implementation to their subclasses. An abstract superclass is the opposite of a concrete class. It is

never instantiated directly to create a run-time object. We make decisions in the Program Plane about when to use multiple inheritance vs. the work-arounds discussed in Chapter 2. The actual program code is considered part of the Program Plane.

Although it is generally true that each object and abstraction in the Execution Plane becomes a class in the Program Plane, this need not be the case. In the Execution Plane, we are concerned with describing real objects; in the Program Plane, we are concerned with a program that generates those objects. Many decisions regarding the inheritance hierarchy are made in the Program Plane, including the use of multiple inheritance, optimization and code reuse, that may impact how classes in the Program Plane are determined based on those in the Execution Plane. However, regardless of the decisions made in the Program Plane, the description in the Execution Plane must still hold at the level of the run-time object itself.

▶ Relationships

The relationships in the models are drawn from Chapter 6 and reproduced in Figure 7-5.

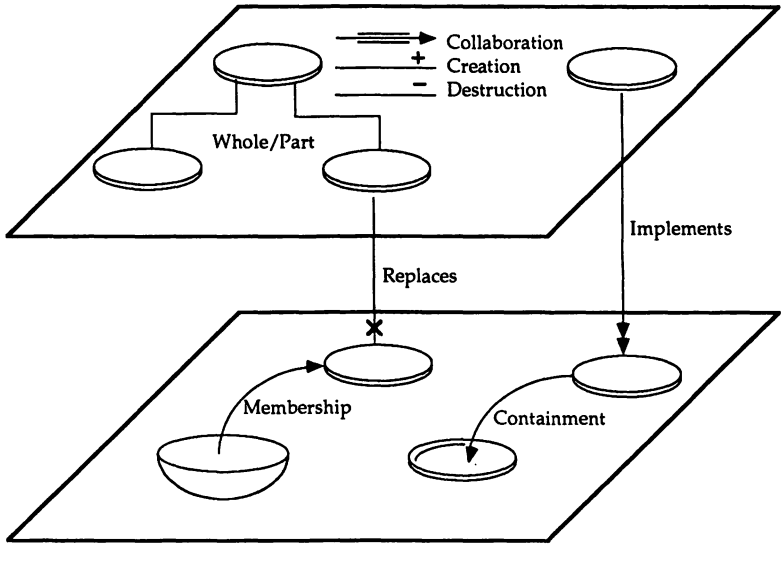


Figure 7-5. Relationships in solution-based models

Relationships of all kinds—structural, behavioral, and calibration—can exist between elements within any one region. Relationships between elements of two different regions of the same plane tend to be behavioral in sending messages, creating, and destroying. An object of the User Interface Architecture may need to send messages to an object of the Content Architecture to record or retrieve information. The objects must exist on each end. Both must agree on the protocol, or interface, for the messages.

Relationships between elements in different planes are always calibration relationships in which the element on the lower plane implements or replaces the one on the upper plane. The Macintosh in the Solution Model must be implemented by the collective elements of the Technology Plane. Elements of the Content Model must be implemented or replaced by elements of the Content Architecture which, in turn, must be implemented or replaced by elements of the Content Implementation.

▶ Frames

Certain information such as constraints and some user requirements are difficult to represent as elements in a formal model. For example, it is difficult to visualize the constraint that there must be a net manpower savings through the use of the new program or that the program must exhibit sound software engineering principles. These factors are collected into the *frames* of each region. As noted in Chapter 6, elements of the frames are defined in simple text. Following is a brief outline of each frame.

Reference Frame

The Reference Model is constrained in two ways. It must be an accurate reflection of reality and it must focus on the problem at hand. The frame of the Reference Model consists of these constraints.

Solution Frame

The Solution Model contains in its frame the following four groups of constraints.

- User requirements
- The Macintosh platform and other technology to be used
- Human factors, such as ease of use, frequency of use, and access to the computer
- Business factors, such as comparisons of costs and benefits; time and

budgetary constraints on development; organizational culture; company policies and procedures; and strategic goals of the organization

Technology Frame

The Content, Environment, and User Interface Model, are framed by the technology being used—the Macintosh platform, any specialized hardware, and, to a lesser extent, the capabilities of the class library chosen (more aggressive designs are used in places where the class library makes it easy). The User Interface Model is also framed by standards for user interfaces on the Macintosh. Constraints that are implicit in the choice of technology need not be formally noted.

Execution Frame

Regions of the Execution Plane are constrained by the technology of object-oriented programming. Objects must be expressed in terms of interfaces containing methods and data members; objects classes must strictly share their interface with their instances. Most of these constraints are implicit and need not be formally noted.

Program Frame

The regions of the Program Plane have as their frame the technology in use, particularly the programming language and class library used for implementation. These are implicit constraints.

▶ Scenarios

A solution-based model for even a small program contains a tremendous amount of detail. It is simply not productive to work with it as a single, overall model. Instead, solution-based models should be organized into small, overlapping *scenarios*, with each scenario dealing with a single topic or concept. Ideally, a scenario is a gestalt in which the whole is immediately recognizable as a single unit and is more than the sum of its parts. A scenario should always fit onto a single page. As noted, scenarios are not mutually exclusive. They can overlap in planes, regions, elements, relationships, and topics—in fact, overlap is an asset. The more scenarios blanket a given part of the model, the easier it is to explore and understand the model, one scenario at a time. Here are a few examples of the kinds of topics suitable for scenarios.

- A single element's responsibilities and the collaborators in carrying it out. Figure 7-6 is a scenario showing how a layout's responsibility to validate itself is carried out with the help of collaborators.
- A whole/part assembly hierachy. Figure 7-7 is a scenario showing the parts of a layout.
- A container, some or all of the elements it contains, and perhaps elements that use the container to store and retrieve objects. Figure 7-8 shows the use of a "portfolio" object to hold layouts and its partial implementation using the MacApp class TSortedList.
- Calibration relationships across planes for a given topic, particularly "implements" relationships. In Figure 7-9, the employee object implements the computer's responsibility to compute gross compensation.
- Time sequences of the execution of responsibilities and creation and destruction of objects. Figure 7-10 shows the sequence of certain payday events in the payroll program.

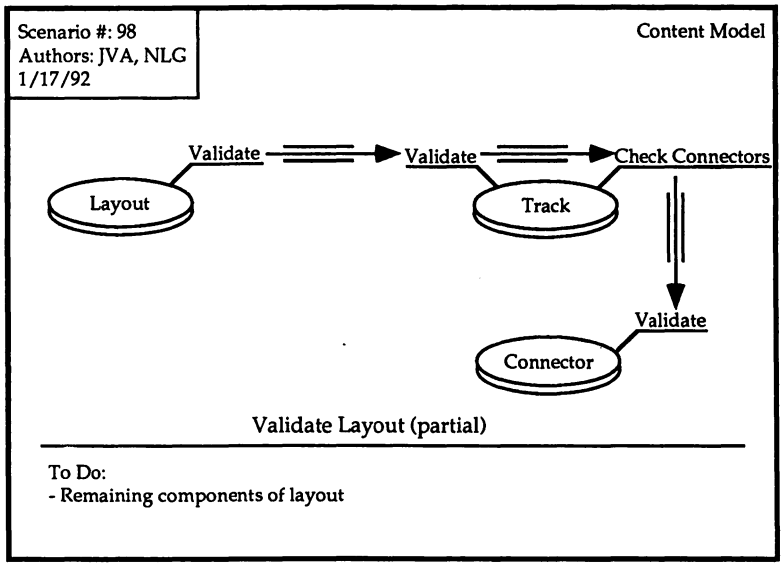


Figure 7-6. Responsibility and collaborators

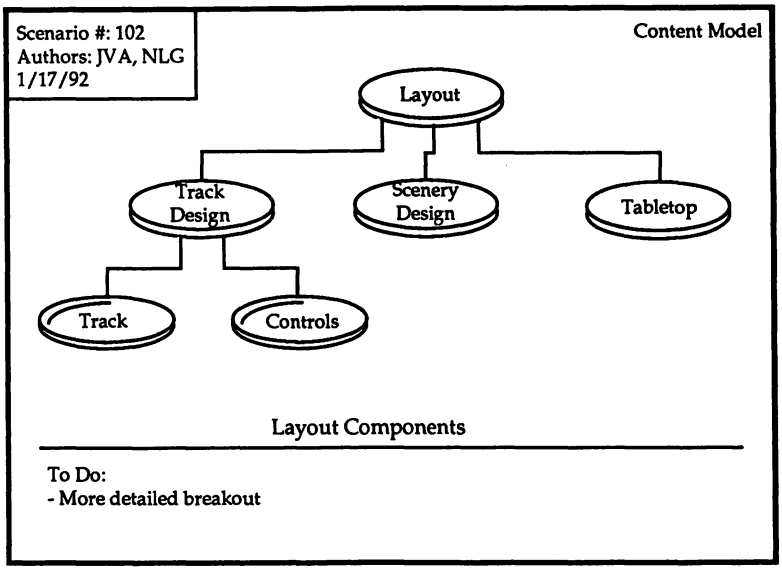


Figure 7-7. Whole and parts

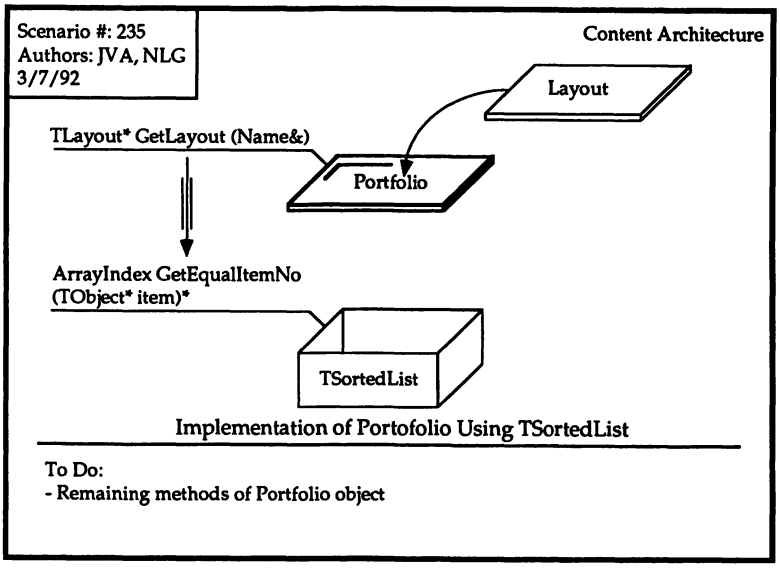


Figure 7-8. Containment

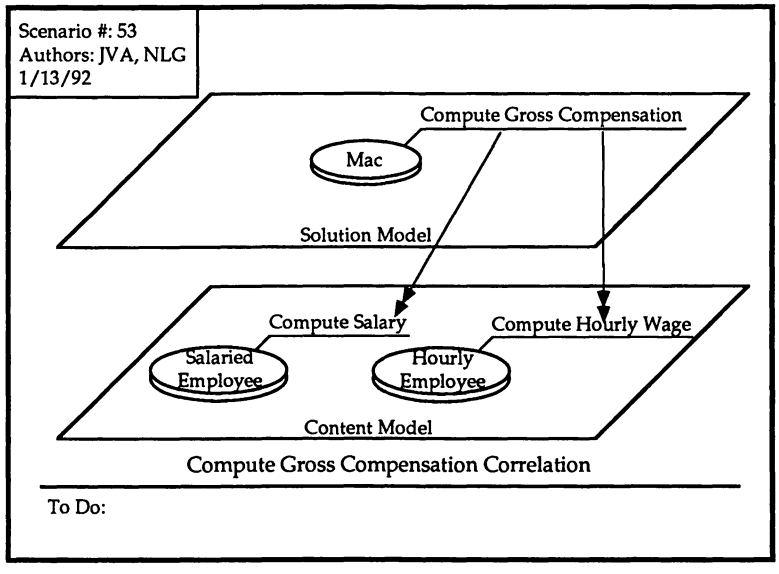


Figure 7-9. Calibration

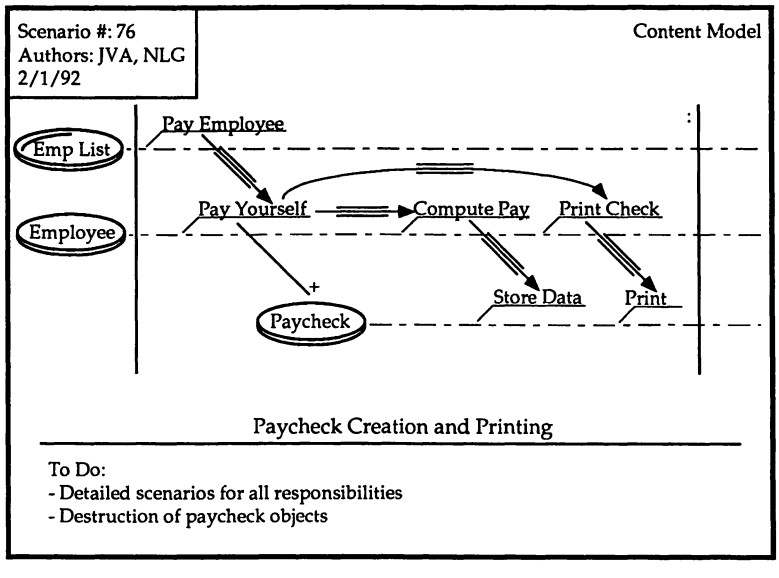


Figure 7-10. Time sequence

This is not a comprehensive list. The variety of types of scenarios is limited only by the need to be useful and communicative. A solution-based model is the sum total of its scenarios, each of which is a small slice of the whole. Just as we use planes and regions to view parts of the model, so do we use scenarios on a smaller scale.

▶ Solution-Based Modeling

We have finished our overview of solution-based models, and it is now time to describe how to build them using Solution-Based Modeling. Just as the models are based on how people's perceptions of the world are organized, the methodology is based on the way people really work on complex projects. The process for building these models is not unique to software. It is the same process that people use to design buildings, paint masterpieces, build models, and develop theories. We merely observe how people naturally work on such projects, left to their own devices, and incorporate it into the process of developing software.

There are two basic activities in Solution-Based Modeling: *forming scenarios* and *calibrating*. Forming scenarios uses the notion of centrality, which holds that some things are more central to a project than others and should be dealt with first. Taken together, these activities form a sequence of steps called Center-Periphery-Calibrate (CPC). CPC is an iterative approach to modeling that emphasizes dealing with what is important or central first, then expanding outward to the periphery. As work proceeds, we constantly calibrate new work to old and old to new to ensure consistency and completeness of the results. CPC is consistent with the basic cognitive principles laid out in Chapter 5. The remainder of the methodology takes into account the myths and realities of software development discussed throughout this book.

▶ Processes

There are three principal processes involved in Solution-Based Modeling.

1. *Forming Scenarios.* This is how new information is added to the model.
2. *Calibrating.* This ensures consistency as new scenarios are added.
3. *Center-Periphery-Calibrate.* This is the process followed to systematically explore the problem, its solution, and the implementation of the solution.

Forming Scenarios

One benefit of using scenarios to organize the model is that they are also available to build it. People tend to deal with a single, limited topic at a time. Complex models are built not by adding in one element at a time to a single, ever-growing model, but by constructing sets of small scenarios, then combining, or synthesizing, them. The result is a single model together with a set of logically consistent scenarios.

There is a direct analogy here to the way database analysts design data dictionaries. Databases are designed by interviewing various people associated with the topic of the database. Their perceptions of the data are captured in the form of "views" of the data, each of which has perhaps a half-dozen record types and their relationships to one another. Once gathered, the views are synthesized into a single data dictionary. Each original view is still a valid way to describe some part of the database. The views have not been synthesized out of existence but simply made consistent with one another.

Scenario formation is used in much the same way in Solution-Based Modeling as the fundamental technique of gathering new information to be added to the model. The topics used to form new scenarios vary according to the type of scenario and the planes and regions that are involved. In the Business Plane, the principal skill is asking good questions and being a good listener (in other words, being a good analyst). In the Technology Plane, analysis still applies, but so too does experience with Macintosh human interface guidelines. Issues of language, operating system, Toolbox, and class library are good sources of scenario topics in the Execution and Program Planes. Software engineering considerations, always present, become especially important in the Program Plane. Each plane and region also suggests topics for other planes and regions. The Reference Model suggests central topics for the Solution Model. The Solution Model suggests topics for the User Interface and Content Models, which, in turn, each suggest topics for the other.

Scenarios should be relatively small and each should deal with a single topic. Ideally, a scenario should contain about two to four elements and a small number of relationships. This leverages the capabilities of human short-term memory to take in an entire model of small size at a single glance.

Calibrating

People do not understand complex things or phenomena as wholes, but as a myriad of small, overlapping perceptions. This is also the way they conceive of and build complex things: not one "piece" at a time, but one concept or topic at a time. Accordingly, solution-based models are divided

coarsely into planes and regions and, on a finer scale, scenarios. Scenarios, planes, and regions are like windows on the model: multiple, overlapping views of a single entity. The more views you have of the problem, solution, and technology, and the greater the overlap between them, the higher the odds of getting the overall model right. Overlap, however, carries with it the potential for contradictions. If two views of a model disagree with one another, the differences must be ironed out; this is as true of overlapping scenarios as it is of planes and regions. The differences are addressed by calibrating. There are three techniques of calibration in Solution-Based Modeling.

1. *Synthesis.* This process takes two scenarios, or a scenario and a model, and creates a single model consistent with both. Synthesis expands the scope of the model.
2. *Correlating.* This ensures logical equivalence of those parts of the model that represent the same thing in different ways. Specifically, correlating ensures that a given plane is consistent with the planes above and below.
3. *Synchronizing.* This ensures that protocols are agreed to and followed for sending messages and creating and destroying objects. Synchronizing assures consistency and completeness within the Execution Plane.

Center-Periphery-Calibrate

In addition to working on a small piece of the puzzle at a time, we also deal with the most important topics first, then add in more detail in small increments. We choose a *central* topic, blanket it with scenarios covering a variety of planes, regions, and angles on the topic, then expand to less central, or *peripheral*, topics. This, too, corresponds to the way people naturally work on complex projects. As we drill down and expand outward, we constantly calibrate the new with the old and the old with the new. The resulting process is called *Center-Periphery-Calibrate*, as illustrated in Figure 7-11.

CPC is used within each phase, each plane, and each region of the project. Whenever there is new material to be explored, CPC provides an orderly way to proceed. The Center-Periphery-Calibrate process is outlined by the following steps.

1. *Pick a central topic.* Distinguishing center from periphery is something people do quite well. Central topics are generally easy to spot. For example, parts of the problem that cause the greatest headaches or that seem to have the greatest potential for improvement; parts of the

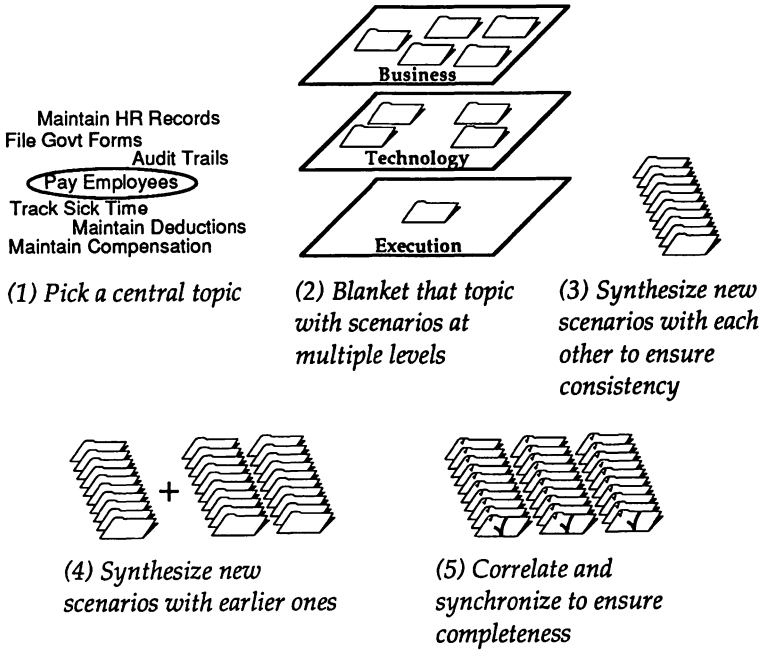


Figure 7-11. Work flow in Center-Periphery-Calibrate

solution that yield the greatest benefit for the least apparent effort; cornerstone techniques or technologies, without which nothing can work; and major features of the user interface. Selecting the definitive, most central of all central things is not necessary; using centrality is a strategy, not a specific algorithm. Everything will be picked up in due course. If you pick what turns out to be a broad central topic, pick something central about that central topic and so forth until it is of manageable scope.

2. *Blanket that topic with scenarios.* If dealing with the Business Plane, interview experts in that topic. If in the Execution Plane, run through time sequences of messages needed to carry out each responsibility. Don't be concerned that you will lack topics. Scenario formation is like eating potato chips—it is hard to stop once you start. Continue forming new scenarios, sticking to the central topic, until it seems to be adequately covered. Do not be constrained to one-dimensional thinking; scenarios can span regions and planes. When they reach the Program Plane, they can result in working prototypes of parts of the program.

3. *Synthesize the scenarios with each other.* This probably suggests more scenarios and changes to the ones just formed.
4. *Synthesize the scenarios with the model.* Changes to scenarios and ideas for more new scenarios will result.
5. *Correlate and synchronize what has been newly added to the model with other parts of the model.* Make sure that the parts of the model that represent the same thing actually do and that the objects that are supposed to be included are there. Make sure that responsibilities have sufficient collaborators to be carried out.
6. *Return to step 1 until all central topics have been exhausted, then start working outward toward less central, or peripheral, topics, repeating the same sequence of steps.* As you do, less and less time will be spent on expanding the upper planes and more time on expanding the lower planes. This corresponds to a shift from a primary emphasis on analysis to a primary emphasis on design and, eventually, programming. Peripheral topics are usually chosen by picking some aspect of the work already performed and saying, "What is central to that which is left to do?" or "What is central to expanding this one topic I dealt with earlier?" In other words, continue using centrality, but on a finer and finer scale.

▶ Project Organization

The linear model of software development provides the illusion of a controlled, orderly process, which may explain its perennial appeal. Of course, we know that this is not the way things really work.

Project Phases

Each stage of a Solution-Based Modeling project uses a combination of analysis, design, and programming skills, spread over more than one plane of the model. However, there is a great deal of underlying structure to the process that is not immediately apparent. In fact, despite this overlap in activities, it is still possible to identify phases of a project.

Not all planes expand at the same rate at any given point in the project. At first, you will do a lot of work on the Reference Model, slightly less on the Solution Model, a little on the Technology Plane, and little or nothing on the Execution and Program Planes. Soon, work is taking place on all planes at once. At some point, the Reference Model starts to approach equilibrium and the Solution Model expands faster than all other models. When that starts to stabilize, the Technology Plane expands the fastest for a while, and so on until the Program Plane is the scene of most of the activity. The effect is shown in Figure 7-12.

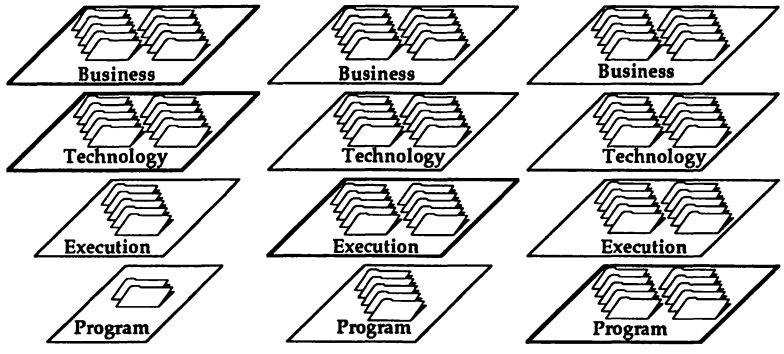


Figure 7-12. Development of a solution-based model

Although they broadly overlap, we can describe four phases of a Solution-Based Modeling project:

1. *Phase I: Analysis.* Most effort is expended on the Business and Technology Planes. Work done in the Execution and Program Planes is primarily proof of concept and prototype. Phase ends when the scope of the project is firmly established in the Solution Model.
2. *Phase II: Design.* Most effort is spent in the Execution Plane, with a great deal of calibration to the Technology Plane and significant work in the Program Plane. Revisions continue in the Business Plane as the result of calibrations. Prototypes in the Program Plane become more sophisticated. Phase ends when the Technology Plane and Execution Plane reach a state of equilibrium and cover the complete scope defined in the Business Plane.
3. *Phase III: Programming.* Almost all work is in the Program Plane, with calibrations to the Execution Plane. Changes to the Execution Plane result in calibrations to the Technology and Business Planes as well. Phase ends when the program covers the defined scope.
4. *Phase IV: Evolution.* The initial scope of the project has been implemented and the software goes into production use. The Solution Model now becomes the core of the new Reference Model for subsequent work, closing the software lifecycle.

It is important not to confuse the names of the phases with the names of the activities taking place in them. During all phases you perform analysis, design, and programming activities and work is performed on all planes

and regions. The differences between the phases lies in which plane is expanding most quickly at that point in time.

Center-Periphery vs. Top-Down

Most software methodologies are organized around a top-down strategy that starts with a high level description of a system and then breaks it down into finer and finer detail. This process of breaking a system's complexity into smaller and smaller pieces can be based on either control or structure. If it is based on control, you start by defining the highest level of flow of control within the program then refine each step. If it is based on structure, start at the highest level of abstraction or modularity of the program then refine each component.

As discussed in Chapter 5 and seen in the examples in Chapter 4, neither top-down approach works very well with object-oriented programs. People naturally deal at the basic level or close to it first, then go both up and down in level of detail. It is better to start at the basic level in developing software. The problem is that this basic level is too large to attack all at once. A targeted approach is needed and CPC fills the need.

Prototyping

Solution-Based Modeling emphasizes the role of prototyping as a development tool. There is a smooth continuum from the Business Plane, which is pure business analysis, to the Program Plane, which is pure design and programming. However, SBM is different from many "rapid prototyping" methodologies in two key ways. First, it does not insist that the prototype be based on a high level overview of the overall program. SBM is oriented toward prototyping limited topics that arise from the CPC strategy. Second, it does not presume that prototypes, once built, are retained through subsequent iterations. Most of the time you spend is in analysis and design, not programming, and most of the risk is in solving the wrong problem. It is best to get quick results in code to provide more feedback for analysis and design, even at the expense of throwing that code away. Don't beat your head against the wall trying to perfect analysis and design before coding starts. You can't do it, and the cost to throw out a prototype and replace it is considerably less than making a big mistake in specifying the solution.

Testing

The fundamental purpose of testing a program is to make sure it has the right behaviors to solve a specific problem in a specific way. Only part of that involves searching for bugs in the software. The majority of the work

in testing is, in fact, what we have been calling calibration: comparing the program with the design and the underlying business requirement. Because we constantly calibrate our model from program code through business issues, testing is an integral part of the process.

There are more specific techniques for testing the program itself that arise out of the division of the Execution Plane into three separate Architectures—Content, User Interface, and Environment. We can leverage this architecture to build in testability across the interfaces, which is where the most serious problems tend to arise in software of any kind. We also use scenarios as a fundamental tool of quality assurance.

Project Management

Solution-Based Modeling emphasizes a team approach to developing software. There is room in the process for end users, analysts, software architects, programmers, managers, marketers, and support personnel at every stage of development. During all phases, calibration results in changes to all four planes and all regions. The Business and Technology Planes cannot be created or changed in the vacuum created by letting non-programmers walk away too early. The relative amounts of effort shift as the project proceeds, but everyone must stay involved to some extent throughout in order to get maximum benefit from the methodology and models.

Since the models are the chief medium of discourse among the people involved, it is important to make sure that information gathered in the development process is captured in the solution-based model. The volume of detail that is gathered and the number of scenarios formed points to the advantages of a project librarian who should work full time for large projects and part time for smaller ones. The librarian is more than a paper pusher. He or she is the key person in the calibration process and has access to the documents that can verify or deny the consistency of the different parts of the model. The librarian need not personally resolve problems, but should be able to trace relationships to expedite and catalyze calibration.

At all times, the Solution Model directs estimates and schedules and defines the scope of the project and the expected outputs.

To a lesser extent, the Technology Plane bounds the technical approach and, therefore, costs and schedules. Significant expansions to the Solution Model or Technology Plane have significant impacts on estimates. Changes in other regions tend not to be as closely associated with variations in costs and time.

▶ Summary

The two fundamental objectives of Solution-Based Modeling are solving the right problem and creating reliable, maintainable programs. In order to achieve these objectives, SBM uses the techniques of visualization discussed in Chapter 6; it uses categories, not classes, as the foundation of the models; it grounds the model in categories and objects of the natural world wherever possible; and it suggests software development be done in an incremental, iterative fashion.

- A Solution-Based Model contains four planes, divided into the regions shown in Figure 7-1. The elements and relationships of the model are those of the notation introduced in Chapter 6. The Business Plane contains as elements categories and objects drawn chiefly from the natural world: people, processes, machines, and other "things." The Technology Plane also uses objects and categories, but these are conceptual, created for the purpose of the project. The Execution Plane has program objects and abstractions, and the Program Plane adds concrete classes and abstract superclasses, polymorphism, and a variety of other optimizations. Because categories are used both for business modeling and designing the interface and content, non-programmers can meaningfully participate through the development of the Business Plane and Technology Plane and, to a lesser extent, the Execution Plane as well. Certain information, such as constraints and some user requirements, are difficult to represent as elements in a formal model. These factors are collected into the *frames* of each region.
- Solution-Based Models are organized into small, overlapping *scenarios*. These scenarios are refined and calibrated following the Center-Periphery-Calibrate process. There are three techniques of calibration in Solution-Based Modeling: synthesis, or combining scenarios; correlating, or making sure that parts of the model that are supposed to be the same, are; and synchronizing, or making sure that protocols are established and followed for sending and receiving messages and creating and destroying objects.
- A Solution-Based Modeling project can be described as being in one of four overlapping phases: analysis, design, programming, and evolution. The phases are identified according to which plane is expanding fastest at that point in time.

8 ▶ Analysis Part I: The Business Plane

▶ What This Chapter Is About

The objectives of the analysis phase are to establish the scope of the project and produce a work plan for completion of the remaining work. At the end of this phase, the Business and Technology Planes are substantially complete, although subject to refinement later in the project. The Execution Plane is underway and the Program Plane may also have been initiated. Work in the Execution and Program Planes during the analysis phase assumes a support role, since the top two planes yield the most benefit in this phase.

Because analysis is a large topic, the subject is divided into this chapter and the next. In this chapter, we talk about how to build the Business Plane. In Chapter 9, we explore the Technology Plane in depth and briefly talk about how and why to descend to the Execution and Program Planes as part of the analysis phase. These two chapters should be read as if they were one long chapter because the material of both chapters is intermingled in a real project. The material has been divided into two chapters solely to provide this information in more manageable chunks.

This chapter centers on the two regions of the Business Plane: the Reference Model and Solution Model. As we discuss how to build each model, we will pause from time to time to explore some of the skills introduced in Chapter 7 in greater depth. Specifically, this chapter covers synthesis, the first of the three basic forms of calibration.

The best way to learn a software engineering methodology is to observe its use in a real project. As the chapter unfolds, we draw examples from the model railroad design and payroll applications introduced in Chapter 4. It

is not possible to completely build each program in the span of a book such as this; each is enough for a book in its own right. However, we will use examples from those two applications and try to give a flavor of how the methodology works in a real project.

▶ Overview of the Analysis Phase

The analysis phase is the single most critical phase in a development project. It is common for this phase to consume half of the total development time—gathering, sifting, and integrating information into the solution-based model. Later phases are primarily concerned with refinements and implementations of information already gathered. The analysis phase is also concerned with organizing and presenting the information gathered in a way that is suitable for decision making by management, customers, product marketing, and others who must buy into the concept, budget, and schedule for the computer system.

▶ Objectives

The analysis phase ends when the following conditions are met.

- The concepts and scope of the software have been agreed to.
- The impact of the new software on the business has been clearly defined.
- The user interface and conceptual design have been agreed to.
- Enough is known about the technical aspects of implementation to allow reliable estimates of resources and schedules to be established.

Each condition ultimately requires someone in authority to certify that the analysis phase is complete and that the estimates are reasonable. Both judgments are based on that person's level of confidence in the information available. It is not possible to construct precise formulas that predict the length of a project or whether the requirements are solid enough to warrant commiting to a development budget, but these issues are not entirely subjective. The job of the development team during the analysis phase is to gather the right kinds of information and to organize and present that information in the right way to the right people. As we will see, the structure of a solution-based model and the discipline used in building it facilitate all of these objectives.

▶ Business Modeling

Much of the analysis phase is concerned with studying the "whole system," by which we mean both the new program and the business environment into which it is placed. This roughly corresponds to the classical activities of requirements definition and systems analysis from traditional software methodologies. However, we attempt to capture the information in a much more rigorous, usable form. We seek to understand the business as it currently operates and as it will operate with the future system.

The chief benefit of this approach to those developing for in-house use is that the overall impact on the business can be managed, not just the development of the software. For those developing software for sale, business analysis is even more critical because it links decisions on features and the structure of the program to product definition, positioning in the marketplace, and even pricing.

▶ Conceptual Design

An equally valuable part of the analysis phase is the conceptual design of the software. This includes both the user interface (how the program can be used) and a conceptual model of the information content of the system and the processes it can support (what the program can do). For the user interface, we create storyboards or software prototypes to demonstrate how the program will look and feel and how it is used to accomplish tasks. For both user interface and content, we create a conceptual description in terms of objects and categories that completely explains the inner workings of the program. Conceptual design is covered in Chapter 9.

▶ Design and Programming During Analysis

Though this is the "analysis" phase of the project, the work performed is not limited to analysis alone. At specific junctures, it is necessary to work in the Execution Plane and Program Plane by designing portions of the software architecture and writing prototypical code. It is important to remember that a phase may consist of multiple activities. Phases are organizational units that allow management to commit to budgets and schedules and track the progress of the project against those commitments. Transitions from one phase to the next represent movement from one box to another on a project plan or a shift in emphasis on one activity over another, not a fundamental change in the nature of the work. Our phases are based on achieving certain objectives, not on completion of specific

activities. Forays into the Execution and Program Planes during the analysis phase are generally considered to be prototyping, although we use the term "prototype" in a broad sense to include much more than just user interface mockups. This activity is discussed in Chapter 9.

▶ Activities

As discussed in Chapter 7, scenario formation and calibration are the two basic activities in a Solution-Based Modeling project. Both are put to immediate use in the analysis phase. Scenario formation is the principal tool for obtaining new information and calibration is the technique used to integrate new information into the model, ensure consistency in the model, and keep track of what remains to be done. Scenario formation is based on the Center-Periphery-Calibrate (CPC) process described in Chapter 7.

Two forms of calibration predominate in the Business and Technology Planes. Synthesis combines scenarios to maintain one consistent model and correlation ensures that the various planes are consistent. For example, each responsibility of the computer as a whole should be supported by some specific user interface features. Both forms of calibration are covered in great detail in this chapter. The third form of calibration, synchronization, has minimal use in the Business and Technology Planes.

Calibration is a sequence of detailed verifications, not all of which take place as new information is added. For example, we might add a scenario to the Solution Model but defer correlating it to the Reference Model until later. When calibration is deferred, *dangling threads* are produced. Dangling threads are elements of the model that must be revisited later. By allowing ourselves to leave dangling threads, we can blast ahead, staying as productive as possible for as long as possible, confident that we can retrace our steps and examine the dangling threads later.

The Technology Plane provides the first opportunity to apply specific techniques to achieve some of the Four Itys: modularity, maintainability, extensibility, and reusability. The trick is in choosing the objects to use in creating a conceptual model of the program and in assigning them responsibilities. Limits must be placed on each object's scope of responsibilities and its knowledge of data, other objects, and how objects are implemented.

▶ The Business Plane

The Business Plane contains the Reference and Solution Models. The Reference Model is a description of the people, documents, machines and other "things" of the real world that make up the business environment

into which the program will be dropped. For each real-world object, we define its responsibilities and collaborations. For example, a payroll clerk has the responsibility to calculate compensation and the employees have a responsibility to report their hours worked. The Solution Model is best thought of as tomorrow's Reference Model. We project ahead to a time when the new software is up and running and describe that environment in the same terms as in the Reference Model. Rounding out the Business Plane is the Impact Analysis. Conceptually, the Impact Analysis is arrived at by subtracting the Solution Model from the Reference Model. It accounts for all changes in the system, where "system" refers to the entire environment and not just the computer and software.

Building the Business Plane requires an understanding of the business and the nature of the problem we are trying to solve. Talking to domain experts, the individuals who are knowledgeable in the part of the business being automated is very helpful, but making them part of the team is even better. This includes, but is certainly not limited to, management, knowledgeable end users, marketing, accounting or finance personnel, and others. Domain participants, individuals who are not expert in their part of the business but are nevertheless part of it, should also either be on the team or be interviewed.

▶ Reference Model

The Reference Model is the starting point for the project. The authors have noted that many clients are surprised that we start with a description of the way things are today before plunging into a hypothetical future. Yet, in the absence of such a model, there is no coherent way to explain what problem you are trying to solve, what the economic value of the solution will be, and what overall changes to the business will result from the use of the new software. For in-house development, this is the place where management can begin by identifying a problem and determining the value of a solution. For commercial software products, this is where product marketing begins by understanding the environment into which you will sell the product and what the product is worth. By developing Reference Models for competitive products, you can begin to develop a positioning strategy by comparing the effectiveness of competing products in the customer's business rather than by comparing one product to another. When your own product is described in the Solution Model, you can compare through an Impact Analysis your product with the others in terms your customers will understand—how their businesses will run differently with the various products. The ability to say, not just "Here's our product,"but "Here's the impact our product will have," sharpens everyone's focus.

On a more basic level, understanding the present is the key to predicting the future. It is relatively easy to analyze the ways things are today; you are limited only by your perceptions and skills at organizing information. Once we leave the present everything becomes guesswork to some degree. Rather than creating a Solution Model and, ultimately, a piece of software in a vacuum, the odds of success improve dramatically if you can trace back, point by point, to the present environment. In this way, the good aspects of the current system can be retained and the aspects that are not so good can be accounted for in new and better ways.

▶ Overview

The Reference Model's frame defines the overall function of the business unit and the nature of the problem to be solved through software. The Reference Model describes the real "things" that exist in the business: people, documents, machines, and equipment. We are concerned with who or what does what, to, for, and in conjunction with whom. In other words, we model the objects of the world in terms of their actions. Our descriptions are in terms of natural world objects, grouped into categories as appropriate, together with their responsibilities for accomplishing objectives. Expressed in terms of VDL notation, the Reference Model is illustrated in Figure 8-1.

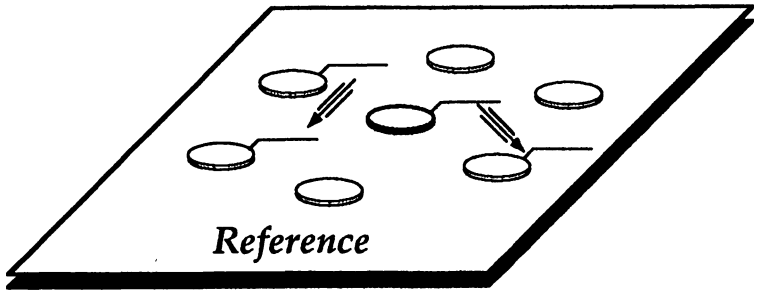


Figure 8-1. Building the Reference Model

▶ Frame

The Reference Model frame defines and focuses the scope of what we are trying to model. The Reference Model frame captures what is central to the business.

Behavior Set

One of the most important parts of the Reference Model's frame is a capsule summary of the major purposes and activities of the business unit as a whole. It is a tenet of management consulting that you need to do a better job of defining a unit's mission if you cannot articulate a small, succinct set of primary functions or objectives for a business unit. In payroll, we can start with two central functions for the department: pay employees and file appropriate government documents, as shown in Figure 8-2.

Figure 8-2. Reference Model frame for payroll department

To design model railroad layouts, one creates a design, then orders parts using a bill of materials, as shown in Figure 8-3.

This level of description provides gestalts that solidify everyone's agreement about the nature of the business. The set of activities, outputs, or objectives of the business system is the *behavior set* of the system. Behavior sets are generally easy to diagram using the VDL conventions discussed in Chapter 6. The business unit forms one natural world object. There may be others that interact with the primary unit as well, as shown in the expanded model railroad frame of Figure 8-4.

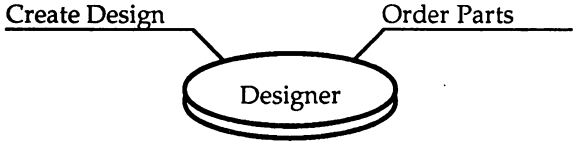


Figure 8-3. Reference Model frame for model railroad design

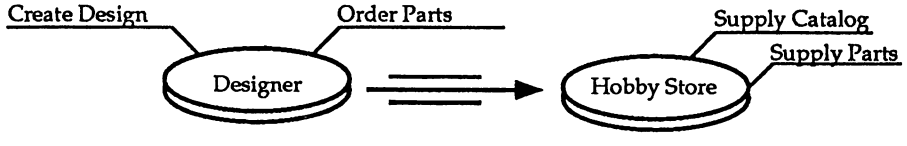


Figure 8-4. Expanded Reference Model frame for model railroad design

Defining the Problem

The second part of the Reference Model frame is a clear and succinct statement of the problem to be solved. This is difficult to diagram and is best described in text. For the payroll example, any of the following might apply.

- "Business expansion has exceeded (or will exceed) the capacity of the department."
- "Costs must be reduced."
- "Accuracy (or service or auditability) must be improved."

For the model railroad design example, any of the following might apply.

- "It is difficult to change designs in progress, and this suppresses creativity."
- "Catalogs are too big to be useful in design."
- "It is very labor-intensive to translate a design into a bill of materials."
- "Mistakes are often not apparent until the design is actually built."

The simplicity of these descriptions is deliberate. In practice, one provides more textual background material, but those elaborations should be held in the background until the simple version has been absorbed.

Building the Frame

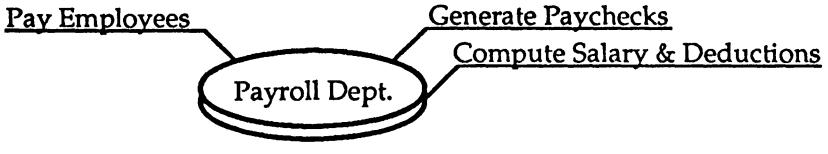
In building the Reference Model frame, one must consider not so much the facts as the mission and strategies of the business. This involves interviewing both domain experts and domain participants. Management often has a clear vision of where the business should be but can be out of touch with the way things really are today. It is only by bringing together these two perspectives that the frame can be properly defined.

We use CPC to build the frame incrementally. For example, Figure 8-2 showed a first attempt at the frame for the payroll example. These two functions, paying and filing forms, come immediately to mind as the central objectives or activities of the payroll department. Closer examination yields some peripheral but vital functions, shown in Figure 8-5.

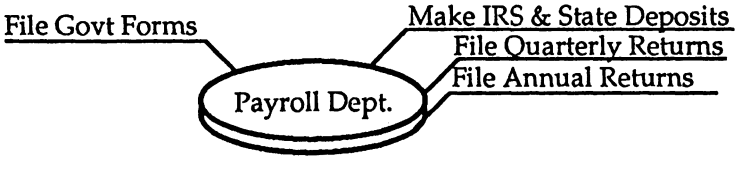
Notice that Figure 8-5 shows one central diagram for the frame, together with scenarios that cover subsidiary aspects of the department's operation.

It is not necessary in the beginning to "complete" the frame; in fact, it is probably not even possible for all but the simplest systems to be completed

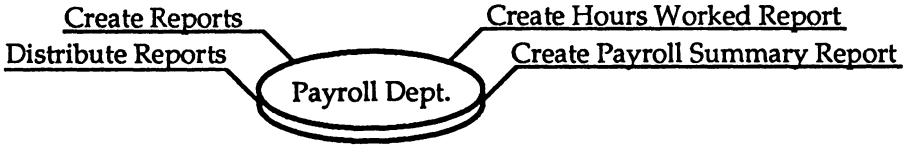
(a) Payroll Department



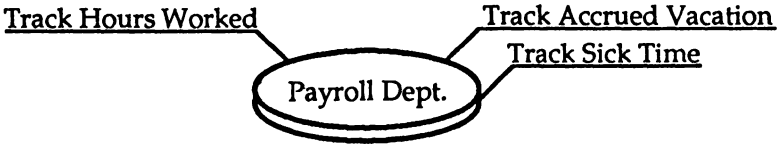
(b) Pay Employees



(c) File Govt Forms



(d) Create Reports



(e) Track Hours Worked

Figure 8-5. Expanded payroll Reference Model frame

in one pass. Instead, start with the center of the frame, as you and others involved perceive it, and proceed on to the model. You can also develop the frame and model in parallel rather than putting the frame first. As you will see, subsequent steps expand and refine the frame smoothly as more information becomes available. In any complex project, it is common to have a feeling of being stalled or hung up from time to time. When this happens, focus in on the center of the sticking point and leave the rest for later. If necessary, focus on the center of the center. That is the essence of CPC: Avoid stalling by quickly refocusing on the center or the center of the center whenever things bog down. The periphery arrives in due course and the methodology ensures that everything remains consistent while keeping track of loose ends.

▶ Model

If the frame defines what the system does, the rest of the model defines how it does it. In the model, we identify the main players, such as people, equipment, business records, and their functions within the overall system.

Double Descriptions and Correlation

The relationship of the Reference Model to its frame is the first of many examples we encounter of *double description:* two models or descriptions of something taken from different perspectives. In the case of the Reference Model frame and model, the double description can be described as the exterior and interior of the business unit. The unit as a whole cannot accomplish anything not accomplished in sum by its parts. Yet, looking at the parts does not give a complete description either; the sense of the whole system as a complete functional unit is lost. Thus, the frame and model complement one another in building our understanding of the business.

We will see many other examples of double descriptions as we proceed through the various parts of a Solution-Based Model. This is deliberate. Any time we add information to one part of the SBM, we synthesize the new information into the appropriate part of the model, then correlate that information against other parts of the SBM that doubly describe the same information. By operating in this way, omissions and mistakes are discovered (half the battle) and appropriate corrections can be made at the right points in time (the other half). Since this correlation occurs whenever new information is added, the corrections come at the earliest possible time when the errors might cause problems. Here, we correlate the Reference Model frame to the Reference Model and vice versa to ensure that both descriptions are consistent with one another. This is the simplest form of

correlation—making sure a model is consistent with its frame. Unlike top-down strategies, this use of correlation and double description does not penalize early mistakes and encourages you to explore the problem in the most natural sequence.

Elements and Relationships

The elements of the Reference Model are objects that really exist in the natural world and categories, or groupings, of those objects. For each object or category of objects, we describe their responsibilities and collaborators. Responsibilities are actions or objectives associated in the natural world with specific objects. For example, a payroll clerk might have the responsibilities to compute compensation and deductions, then issue check requests to a typist. The typist is a collaborator in an implied responsibility of the clerk (ensuring that checks are typed). Figure 8-6 shows this in the form of a scenario.

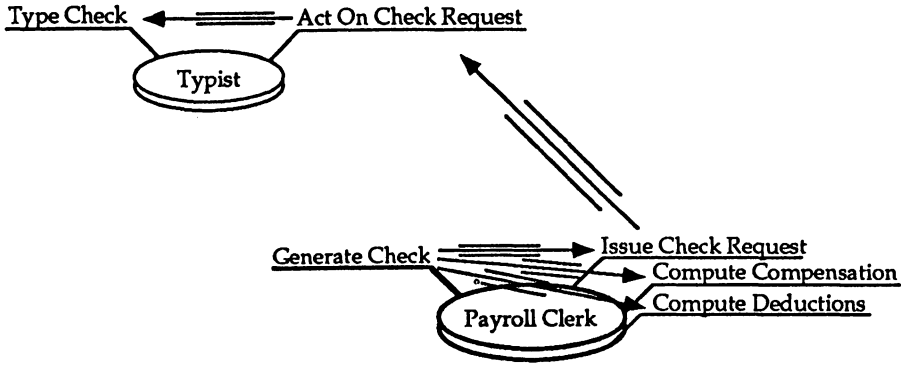


Figure 8-6. Preparing and typing paychecks

Certain responsibilities of objects in the model correspond directly to the behavior set of the overall business unit, as defined in the frame. These are *essential* responsibilities because they directly implement the objectives of the business. Incidental responsibilities exist only to support the essential ones, such as issuing a check request. We continue to use this distinction within each region of the SBM.

Objects and categories in the model can represent a wide variety of "things" as shown in the following examples:

- People: employees, customers, vendors
- Machines: computers, tools, manufacturing equipment, paper handling equipment
- Documents and files: reports, forms, index cards, computerized records, filing cabinets

Everything described in the Reference Model should be true of the real world, but not everything known of the real world should be in the model and its frame. The ideal model is one whose description covers the scope of the frame but uses no unnecessary elements or relationships.

Building the Reference Model

The process of building the Reference Model starts with the selection of one or a few central objects or categories, based on what is central to the frame. We then take the behavior set of the frame—the essential responsibilities for the model—and assign them to the objects. As we proceed toward the periphery of the model, we add incidental responsibilities and other non-central elements. Each time new information is added to either the frame or the model, we correlate the two.

The calibration process works in two directions: Information from the frame is pushed into the model and expansions of the model can alter or expand the frame. Essential responsibilities are key to calibrating with the frame. Every behavior of the frame should be accounted for by essential responsibilities in the model. Any frame behavior not accounted for is a dangling thread that must be picked up before the model can be considered complete. Expansions of the frame are pushed down to the model, although not necessarily right away. Similarly, it is very common to identify a responsibility in the model and realize that it is essential (that is, part of the behavior set of the business unit as a whole), even though it does not correspond to any behavior yet identified in the frame. This eventually results in an expansion of the frame. Again, this correlation to the frame can take place right away or be deferred by identifying the dangling threads.

Initially, we build the model by choosing central topics of the frame or central players in the business unit. The model and frame are then expanded in tandem until those central topics are adequately covered. When the central topics have stabilized in the model and we are ready to look for more to do, we pick up dangling threads in the frame and push

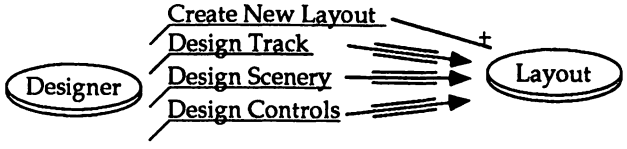
them into the model. Going the other way, any time we add an essential responsibility to the model, we either revisit the frame right away or mark the new responsibility as a dangling thread. Again, we eventually sweep along picking up dangling threads of the model and correlate them to the frame.

Where do the elements of the model come from? Identifying objects, categories, and responsibilities is not always a one-way street from frame to model. Often, the most effective way to build the model and refine the frame is to go searching for objects and categories directly. Here are some approaches to try.

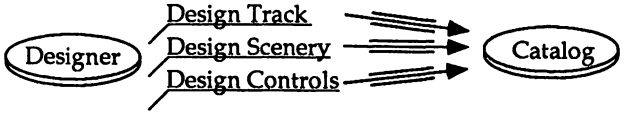
1. *Reflect.* Sit and think about the problem and the Reference Model.
2. *Interview.* Conduct interviews with users or domain experts.
3. *Read.* Consult textual descriptions of the system being studied. Remember that nouns often represent relevant objects or categories and verbs sometimes represent responsibilities.
4. *Analyze forms.* Review and catalog documents and forms used in the organization.
5. *Synthesize and decompose.* Identified elements can be a rich source of new elements. If an object in your model has parts, examine the parts; if an object is part of something, look at the whole; if you have a category, examine the members. For any element, consider categories that naturally describe how it groups with other elements.
6. *Follow responsibilities and collaborations.* Look at each member of a category for responsibilities and collaborations that are in addition to or perhaps in conflict with those of the category.
7. *Generalize.* Similarly, consider categories to which objects in your model belong.

The underlying objective of these tactics is to nudge your thinking and perceptions and those of the others on the team, then capture the information that bubbles to the surface in the form of scenarios. We are really looking for a good description of the objects, but categories can help to define them.

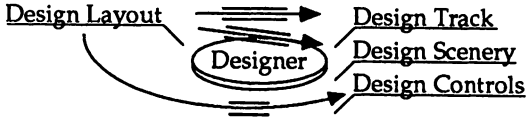
Without saying so, we have already implied many different sources of scenarios—central topics, dangling threads, correlation of specific elements, and part/whole assemblies and other structural relationships. As you gather information, do it in the form of scenarios and follow the process above to integrate the results into the overall solution-based model.



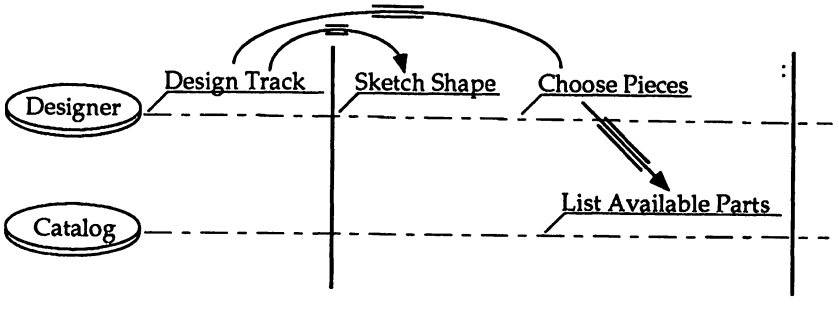
(1) Creating a layout



(2) Using the catalog to aid design



(3) A higher level of abstraction for 'Design'

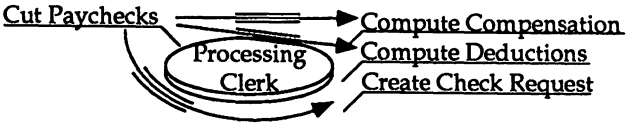


(4) Iterative nature of designing track

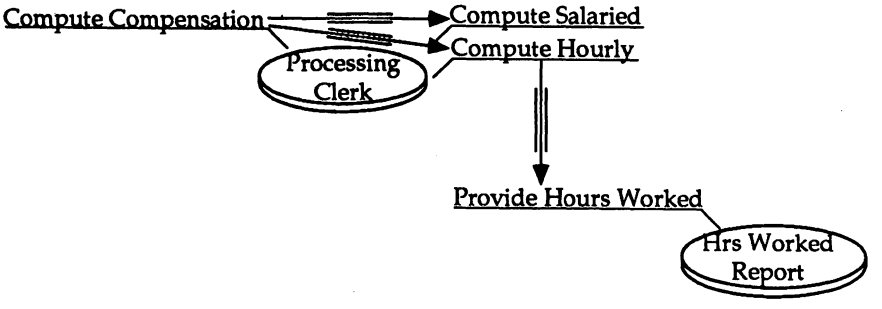
Figure 8-7. Reference Model for model railroad design

Let's look at how this process works in our two case studies. Figure 8-7 shows a typical evolutionary sequence for the Reference Model in the model railroad design problem. Notice that the work quickly centers on designing track.

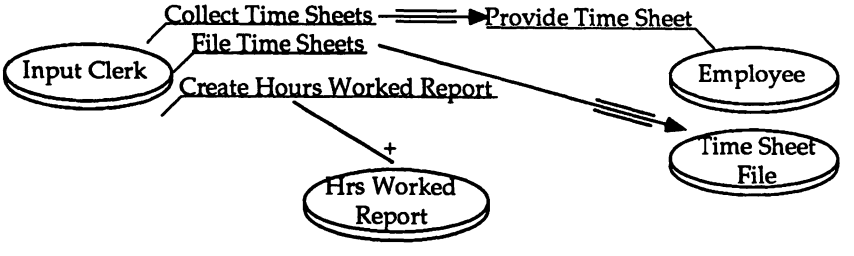
Figure 8-8 shows a typical evolutionary sequence for the Reference Model in the payroll example. In neither of these examples is work conducted top-down, yet the progression seems quite natural.



(1) Creating check requests



(2) Computing compensation



(3) Creating the Hours Worked Report

Figure 8-8. Reference Model for payroll

▶ Calibration Part I: Synthesis

From this point on, we will create numerous scenarios and we will need continually to integrate them with each other and with the solution-based model as a whole. Let's assume that you are starting with an empty model and have a set of scenarios stacked in front of you. The process goes like this.

1. Pick a scenario and make it the initial model.
2. Pick another scenario.
3. Synthesize the new scenario with the model. This may change the new scenario as inconsistencies are discovered. Alternatively, the model may need to be changed. If so, trace back the changes to the scenarios that have already been synthesized with the model.
4. Repeat steps 2 and 3 until all scenarios have been synthesized.

The outputs of this process are a unified model plus a set of scenarios derived from the originals but possibly corrected to ensure consistency. The scenarios are not thrown away at this point. They remain the definitive statements of their respective topics.

This process allows us to concentrate on one scenario plus the model at any point in the process. The precise details of synthesizing a single scenario with a model are summarized below.

1. Pick an object or category from the scenario that also exists in the model. Consider the possibility of two different names for the same thing (synonyms). Also consider the problem of one name for two different things (homonyms).
2. If the scenario element is an object and the model element is a category or vice versa, resolve the discrepancy.
3. If the scenario element and the model element are both categories, make sure they have the same members and that the semantics (meanings) of the categories are not contradictory.
4. Verify all structural relationships of the scenario and model elements: whole/part, membership, and containment. Resolve any discrepancies. In order to do this, you may need to temporarily set aside the scenario element until other, related elements have been synthesized.
5. For each responsibility of the scenario element, look for a matching responsibility in the model element. Again, watch for synonyms and homonyms. If no matching responsibility exists, add the responsibility to the model element. If a match exists, make sure that the semantics and collaborators of the scenario element's responsibility are not contradicted by those of the model element. If there is a contradiction, resolve it.
6. If the model element in the scenario being added belongs to a category, apply step 4 to the responsibilities and collaborators of the category. Also, verify that any responsibility or collaboration of the category also applies to the scenario element. If there is any discrepancy, resolve it.

7. Verify all remaining behavioral and calibration relationships of the element and its responsibilities—creation, destruction, implementation, and replacement.
8. Repeat steps 1–7 until there are no more elements in common.
9. Add each remaining element, along with its responsibilities and relationships, to the model.

The appendix illustrates a simple manual database that greatly facilitates this process and also helps with correlation and synchronization. When comparing a scenario element with the "overall model," you will refer back to scenarios that were previously synthesized and that contain that element. These are the gestalts that are easily digestible; the model as a whole is nothing more than the sum of its scenarios. We speak of "the overall model" as a concept, but in reality it is only the set of synthesized *scenarios* that counts.

As you can see from this algorithm, there are five kinds of discrepancies that can arise:

1. Synonyms and homonyms.
2. One scenario says something is an object and another says it is a category.
3. Semantic differences between categories.
4. Relationships may differ, particularly collaborations.
5. There may be a contradiction between responsibilities and/or collaborators of a category and a member of that category, that is, a counterexample for the category. As in the "metonymic" schema in Chapter 5, all members of a category are often assumed at an early stage to have the characteristics of one or more central members. As the other members are synthesized, it is common to discover counterexamples that do not have the shared characteristics.

The remedy for these problems is common sense. For example, when a counterexample is found for a category—say, a member that does not have a responsibility attributed to the category—the characteristics are usually reassigned to the members that do have those responsibilities.

When resolving discrepancies, we seek only to avoid contradictions and not to make the treatments identical in all scenarios. Consider a scenario that says that all pieces of track (a category) have two connectors, as in Figure 8-9. Another scenario states that straight track has two connectors but does not assert that this is true of all other types of track, as in Figure 8-10. This does not necessarily mean that one of these scenarios

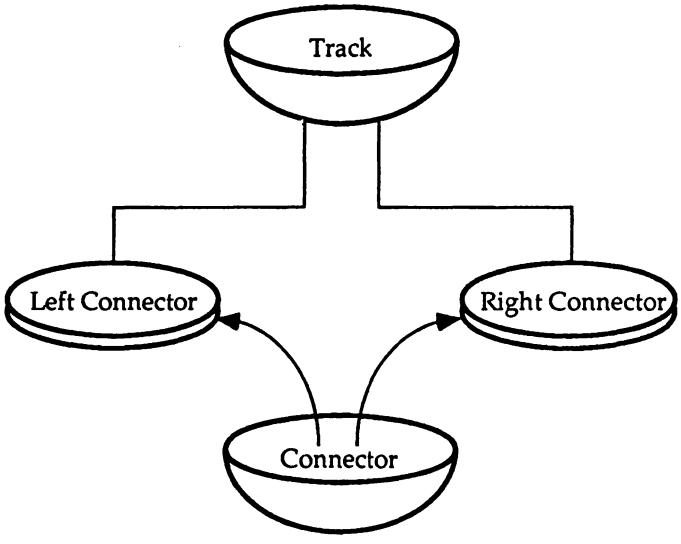


Figure 8-9. Connectors on all types of track

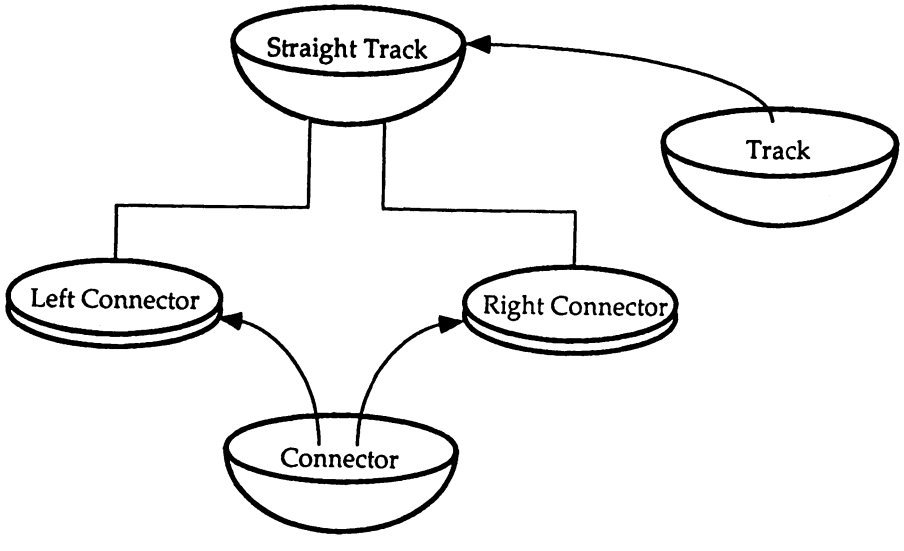


Figure 8-10. Connectors on straight track

is wrong and in need of revision. If, as in Figure 8-9, all track has two connectors, then Figure 8-10 is also true and neither scenario needs to be changed. If, on the other hand, the author of Figure 8-10 knew of a counterexample, such as a Y-shaped piece of track with three connectors, then Figure 8-9 is wrong and needs to be changed. As you can see from this example, it is not possible to make these decisions without knowledge of the problem domain. If both scenarios are correct, there is still only one treatment that is correct for the overall model, and the correct treatment is the one that preserves the most information. Figure 8-9 contains more information than Figure 8-10, because it carries information about more than just straight track. As in Figure 8-10, each scenario may contain less information, but the model must contain the sum of the information in all of its scenarios.

A final note on synthesis. You usually have a stack of scenarios surrounding a single topic, all of which need to be synthesized with the model. In such a case, proceed by first synthesizing the scenarios with each other, forming an intermediate model, then synthesizing the intermediate model with the full model. Because of the overlap between them, differences can be resolved much faster among the new scenarios before combining them with the overall model.

▶ Solution Model

The Solution Model directs our attention away from the way things are today to the way they will be in the future. The best way to think of the Solution Model is as tomorrow's Reference Model. The two models have the same structure, types of elements, and relationships. The only significant difference is the focus: Central to the Solution Model is the Macintosh running the program being developed. Slightly off to the side are users of the program and any devices or networks connected to the computer and relevant to the program. Figure 8-11 illustrates the relationship between the Reference and Solution Models.

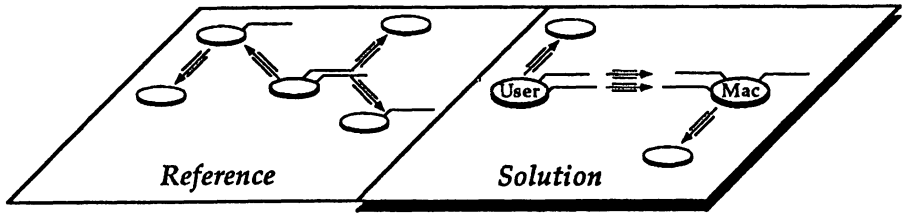


Figure 8-11. Building the Solution Model

▶ Overview

The Solution Model defines the overall function of the business unit in the automated environment and the nature of the solution to the problem(s) identified in the Reference Model frame. The solution generally involves a combination of many changes: shifting duties of personnel, redirecting funds or priorities, training, adoption of new techniques or policies, and, of course, the strategic and tactical uses of the computer. The computer is one part of the solution, not by itself the entire solution. The best way to think of this is to make the computer running the new program an element in the model in its own right.

We build the Solution Model from two directions. Central topics of the Reference Model are pushed down into the Solution Model and modified to suit the intended changes to the business. Equally important is direct exploration of the Solution Model. You generally go into a software project with some notion of what the new environment should be like, probably without having yet fully defined what impact that has on current operations. Thus, you should feel free to directly develop the Solution Model in parallel with the Reference Model, as long as the two are calibrated properly.

As information is added to the Solution Model, correlation is used to propagate new insights into the business today back to the Reference Model. In most projects, the process works both ways and information both filters down from the Reference Model and directly enters the

Solution Model from outside. In either case, we continue to use the tools of scenario formation and CPC to start at the center and expand out toward the periphery.

As with the Reference Model, we are concerned with real world objects and categories of them. Again, the emphasis is on the "system" in the broad sense, incorporating not just the computer and program but the organization surrounding it as well. The Solution Model cannot be said to derive from the Reference Model because we are not really constrained by the way things are today. However, we carefully correlate the two models, clearly identifying all differences and collecting them into an Impact Analysis. In the case of a commercial product, there are as many Reference Models and Impact Analyses to consider as there are competitive positions to define. Figure 8-12 shows the notation used to correlate the Reference and Solution Models.

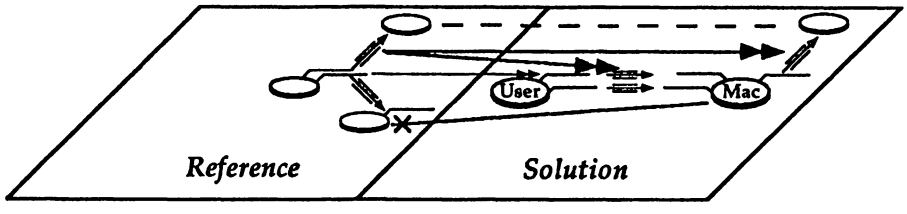


Figure 8-12. Correlating the Reference and Solution Models

The double-headed arrow means "implements" and the line with an "X" means "replaced by." The dashed line means "is the same as" or "unchanged." This system accounts for everything in the Reference Model in terms of the Solution Model and vice versa. This includes objects, categories, responsibilities, and relationships. This detailed work is the basis for the impact analysis, which is critical to the business. Correlation of these models also helps develop each one, by providing two-way feedback.

▶ Frame

The Solution Model frame describes the portion of the Reference Model we seek to change—what is kept, what is extended, and what is simply new. It is here that overall objectives for the development project are set. It is also

in the frame that we describe the overall role of the new computer system within the business system. The Solution Model frame contains a concise statement of the intended solution to the problem(s) identified in the Reference Model frame, the behavior set of the business unit, and constraints on the solution.

Defining the Solution

The general form of the solution must be laid out as early as possible. This is generally difficult to diagram and is best described using text. For the payroll example, suppose that we had made the following determination in the Reference Model frame.

- "Business expansion has exceeded the capacity of present staffing in the department."

In the Solution Model frame, we might identify the following combined solution.

- "Automate the computations for and production of paychecks."
- "Automate the generation of government forms and reports."

We might, however, have identified the problem in the following way.

- "Mistakes are made in processing payroll, resulting in unhappy employees and wasted labor when corrections are necessary."

We might then substitute the following solution.

- "Make sure that information is captured only once, then reused as needed by the computer."
- "Maintain automated audit trails for all activity."
- "Integrate the recording of time cards, personnel records, and payroll processing."

These two solutions overlap, but there are subtle differences that correspond to the different priorities of the Reference Model frame. We might, for example, be faced at some point in the project with a tradeoff of manpower to operate the program versus double-checking input. The frames help guide those decisions (in this case, perhaps in different directions).

Behavior Set

The behavior set in the Solution Model frame describes the new business environment. Everything said about the Reference Model frame's behavior set applies here, but with an emphasis on the changes we wish to bring about. The Reference Model frame is compared detail by detail with the Solution Model frame in the Impact Analysis to identify changes in the mission, scope, or capabilities of the business unit as the result of automation. It is therefore critical that the behavior set of the two frames be correlated completely and carefully. Figure 8-13 shows a Solution Model frame for the payroll example. Compare this with the Reference Model frame shown earlier in Figure 8-5.

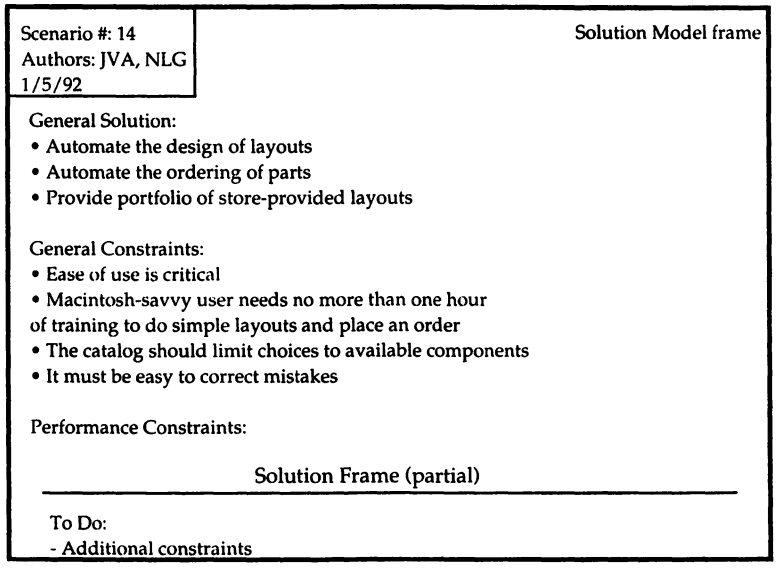


Figure 8-13. Solution Model frame for payroll automation

Constraints

When solving business problems, we are always operating under constraints on the solution. These constraints may be those of the available technology, limited resources, time, or other factors. We capture these as part of the Solution Model frame. Constraints are usually difficult or

impossible to diagram, so we rely on simple text. Some types of constraints to consider are listed below.

- Managerial: ability to measure success, auditability, accountability, operations, maintenance requirements, on-going review, and assessment.
- Technology: what the technology is capable of, price/performance ratios, technological risk, expected advances in technology, and price performance in the near future.
- Financial: absolute constraints on cost and comparisons of costs to benefits.
- Resources: time, people, use of equipment or space.
- Performance: speed, security, accuracy, and accessibility.
- Qualitative intangibles: corporate culture and environmental constraints, corporate policy, image, and history.
- Existing systems: interfaces, dependence on technology and people, impact on operations, replacement policies or objectives, and auditability of and accountability for the combined system.
- Strategic goals: relationship of mission critical applications to corporate strategy.

Although this is not an exhaustive list, most large projects take these constraints into account. Note that very little of this list can be traced to the definition of the problem or its intended solution. Instead, constraints tend to be imposed from outside the project and apply across all similar projects. Thus, the time spent in this area should in most cases be highly reusable on other projects.

It is often arguable whether something is a constraint or a behavior or part of the solution definition. It doesn't really matter; as long as the fact is recorded somewhere in the frame, the placement is not that important. For example, we might say that part of our solution is to reduce labor, but we can also express as a constraint the fact that labor costs must be reduced. It might also be implicit in the way the behavior set is defined. In Solution-Based Modeling, most divisions of information—frames, planes, regions, and so forth—exist principally to stimulate thought and perception and provide a framework for organizing known facts, not to generate arguments about where things belong. Since calibration is applied throughout the model, placement is not critical; leaving something out altogether, however, is a serious matter.

A final note on constraints. It is not always possible to calibrate constraints in the Solution Model frame right away. Performance constraints, for example, almost always require calibration against the architecture of the Execution Plane or against specific algorithms of the Program Plane. Constraints that are not satisfied within the Business or Technology Planes simply continue as dangling threads until we can resolve them.

Building the Frame

As with the Reference Model frame, we interview domain experts and participants in addition to our own snooping and thinking. There is, however, a fundamental difference in the interviewing process. In the Reference Model, we seek to describe the reality of today. In the Solution Model, we deal with some facts, but also with suppositions and projections. This places a greater burden on the team in analyzing feedback. Here are some of the "noise factors" to take into account.

1. Management may have a clear vision of the new mission but be unaware of many constraints on the solution, particularly technical ones.
2. Domain experts frequently are wedded to the way things are done today. They may simply have difficulty conceiving of new ways of doing things or may even feel that their expertise is threatened by change.
3. Domain participants have an even greater tendency to resist change.
4. Software and systems analysis professionals often don't know the current system well enough to understand the relative merits in business terms of various proposals for change. They may also be insensitive to issues of management or culture, concentrating primarily on technology.

As is often the case with software development, much of this noise can be traced to poor communications or a lack of cooperation. Dealing with gestalts in the frame helps quiet some of this noise by providing a common forum for communication. Another useful technique is to classify proposed solutions as "mechanization" or "automation." Mechanization basically means doing the same things faster or better as the result of using the computer, projecting manual operations onto the computer with little change. No one is threatened by mechanization, since job functions change little and the power structure of the organization remains intact. Little new training is needed beyond the mechanics of

using the computer. However, mechanization rarely yields strategic benefits to the organization; there may be a slight cost savings or improvement in quality, but that is the most that can be hoped for. It is automation that yields breakthroughs in new markets, new product lines, or new ways of doing things, redirection of the business, or repositioning in the marketplace. Automation requires going back to basic questions: What is the mission of the organization? What do we want to achieve? What are the real constraints, as opposed to historical assumptions?

Automation applies not only in the stratosphere of corporate strategy but in almost every decision made on the definition and design of a computer system. As shown in Figure 8-14, the rectangle drawing tool of various Macintosh drawing applications is a good example of automation.

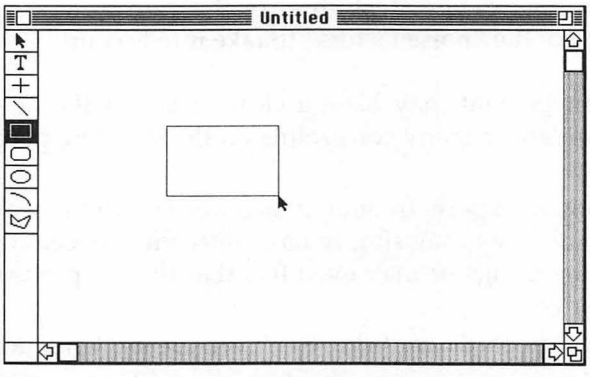


Figure 8-14. A typical Macintosh rectangle tool

Mechanization involves drawing four separate lines; automation conceives of the task at a higher level—construction of a rectangle, rather than drawing four lines—and results in the dragging behavior so familiar to Macintosh users. With the equivalent of one stroke on the page, we draw a complete rectangle, perhaps even filled with a pattern or using a non-standard line width. Now consider the other tools of a modern drawing program such as bezier curves, dashed lines, compound (grouped) objects that move and resize together, stretching, skewing, or rotating of shapes and compare the results to straightforward mechanization.

Tradeoffs of mechanization vs. automation begin in the earliest stages of a software project, usually without anyone consciously aware of the implications. In SBM, we take the time to develop a Solution Model frame that deals in objectives and priorities. This then becomes the basis for

discussion of later, more detailed decisions. If our frame says, "Speed the drawing of lines" we will fail to see the larger opportunities. If instead we say, "Reduce the turnaround time and increase the quality of our design documents," we have a yardstick that measures automations larger than mechanizations.

Of course, we continue to use CPC to develop and refine the frame. Central topics come from the Reference Model or by direct examination. As we expand the frame, we correlate it to the Reference Model frame and Solution Model to account for all differences between the models. Every part of the Reference Model frame must be accounted for through correlation in the Solution Model frame, and every element and relationship of the Reference Model must be accounted for through correlation in the Solution Model frame and Solution Model combined. Anything defined in the Reference Model and not accounted for is a dangling thread that must be picked up later. Similarly, we cannot add to the Solution Model frame without correlating back to the Reference Model and frequently expanding the Reference Model during correlation. Dangling threads in the Solution Model frame can occur when this correlation is postponed.

▶ Model

Because the Solution Model is tomorrow's Reference Model, it is structurally similar. We still deal with objects of the real world. In almost all cases, the computer and program together become one central element of the new model and principal users become other central elements. If the computer is attached to other devices or networks, those, too, become elements of the model. For most projects, the primary focus of the Solution Model is on how the computer and its users interact (that is, what the users are responsible for, what the computer is responsible for, and how they collaborate to carry out those responsibilities).

Elements and Relationships

The Solution Model contains as elements real world objects, categories of real world objects, and responsibilities of the objects. In most cases, the Macintosh, running the application in question, becomes a single element of the model. If the applications to be developed cooperate with each other, simultaneously running applications, each application is an element in its own right. If there are other computer systems or specialized devices collaborating with the target application, they, too, become elements. Each user of the computer becomes an element as well. In addition, we retain other people, machines, equipment, and records as elements, as in the Reference Model. Relationships are the same as those of the Reference

Model. We are primarily concerned with collaborations, but the other types of relationships discussed in Chapter 6, especially structural relationships, may apply as well.

Building the Solution Model

We build the Solution Model in much the same way as the Reference Model by starting with central features of the frame, then expanding and correlating. It is also useful to bring across elements and relationships of the Reference Model to the Solution Model, although these should be carefully considered. Do they retain the same meaning? Are we unconsciously mechanizing rather than automating? The objects and categories themselves rarely change much from the Reference Model to the Solution Model, but their responsibilities and relationships may change significantly.

We can also add to the Solution Model through direct examination. In the Reference Model, this is done by looking around at the world as it is today. In the Solution Model, you must place yourself forward in time. Whenever we add an element or relationship to the Solution Model, we must correlate to the frame and to the Reference Model. Keep in mind the two basic calibration relationships of "implements" and "replaces." If some element or relationship of the Reference Model is made obsolete as the result of something in the frame or model of the Solution Model, we use the "replaces" relationship. If an element just added to the Solution Model is truly new, it should be marked as such and not treated as a dangling thread. More often, it is a variation on some previously discovered part of the Reference Model or a replacement for one, in which case the calibration relationships should be drawn. Figure 8-12 showed an example of the use of calibration relationships between the Reference Model and Solution Models.

CPC applies in the same way as for the Reference Model, except that there tend to be more interviews, more scenarios, and more conflicts. People naturally expand their categories when they project into the future, and not everyone does it in the same way. When categories present problems, fall back on the objects themselves, about which there tends to be less disagreement. (Objects of the world tend to be either basic level or close to it, while most categories are far enough from the basic level to generate disagreements.) As with the Reference Model and the frames of both models, we both push information from other parts of the model into the Solution Model and directly expand the model from outside sources of information.

Let's consider an example from the model railroad application. In the Reference Model, we had the scenario shown in Figure 8-15.

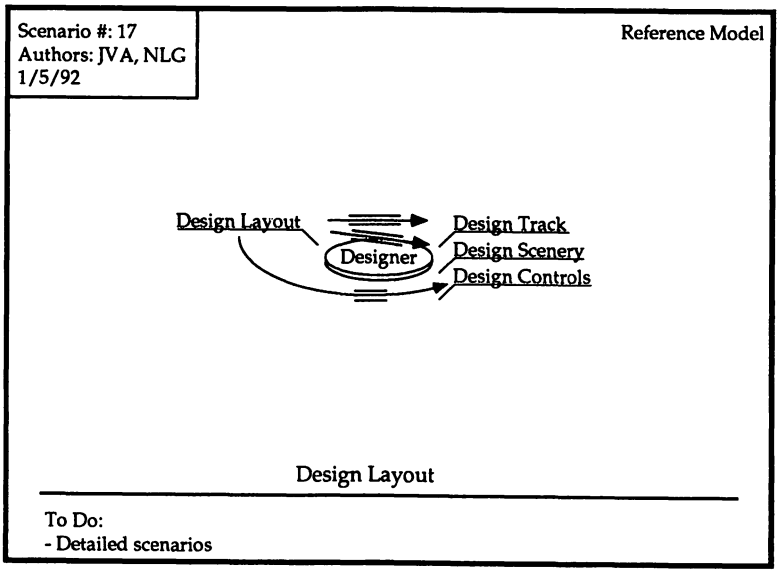


Figure 8-15. Layout design scenario

There is a single responsibility called "Design Scenery." In the Solution Model, we wish to automate that process. In order to do so, we need to know more about how the designer works now. As part of defining the solution, we can expand the Reference Model to the scenario shown in Figure 8-16. Adding this finer level of granularity to the Reference Model permits us to develop the Solution Model further, as in Figure 8-17.

Notice the emphasis on the user–computer interaction. Solution Model scenarios collectively define what the computer can be told to do, what the user needs to do, and how the two mesh.

As with the Reference Model, we draw a distinction between essential and incidental responsibilities. Essential responsibilities implement the behavior set of the frame. Wherever meaningful, we also attempt to draw implementation relationships between constraints and parts of the model. For example, a reduction in labor can be tied to specific responsibilities of the computer in the model. Not all constraints can be treated in this way, however. A constraint that "all operations must complete within five seconds" must wait for the lower planes of the model, particularly the Execution and Program Planes. Other constraints are best implemented in

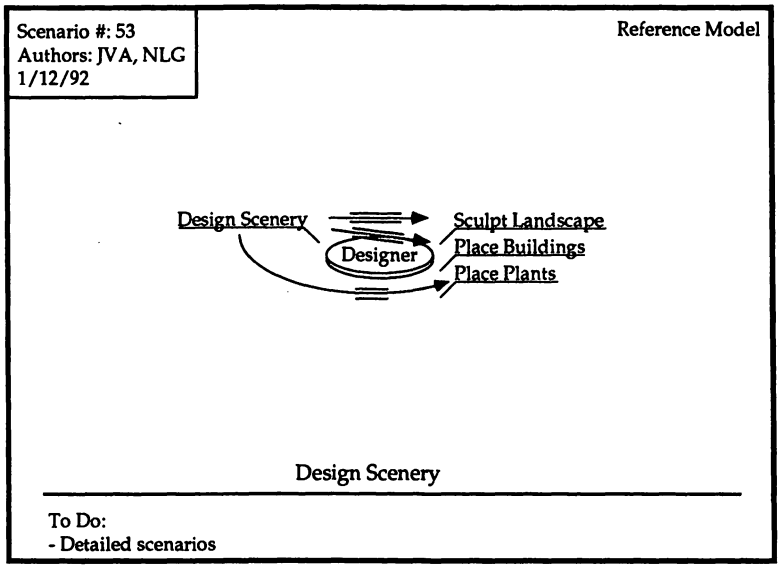


Figure 8-16. Expanded layout design scenarios

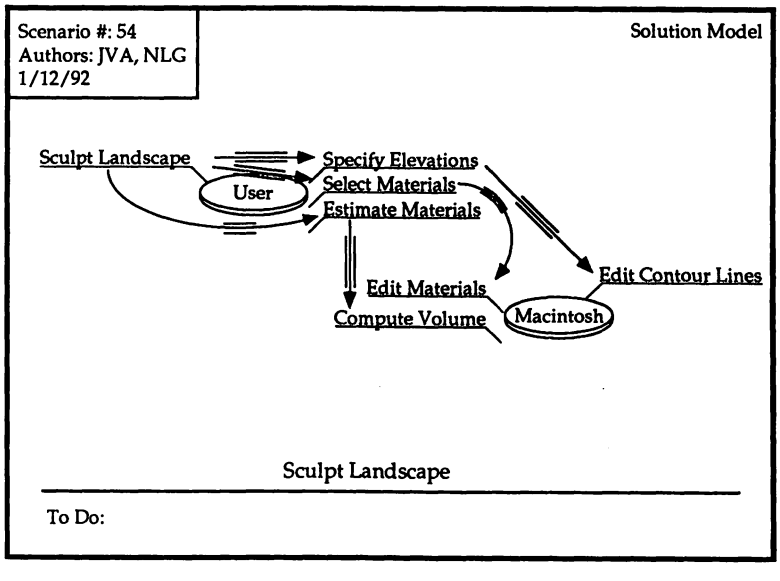


Figure 8-17. Solution Model scenario for layout design

terms of the correlation relationships between elements of the Reference and Solution Models. These relationships indicate the changes being made in the business. Improvement in accuracy might be associated with the shift in responsibilities from people to the computer; that shift is illustrated through correlation relationships, not purely in terms of the Solution Model.

▶ Impact Analysis

Figure 8-18 shows that the Impact Analysis comes from the correlation relationships between the Reference and Solution Models. The Impact Analysis contains all of the correlation relationships between the Reference and Solution Models, along with the elements at each end of the relationships and analysis of what the changes mean to the business. When we talk about reassigning staff or shuffling lines of authority, it is not enough to draw lines on a page. Textual backup is also needed to translate these changes into a plan for managing the transition. Figure 8-19 shows correlation relationships between parts of the Reference and Solution Models for payroll.

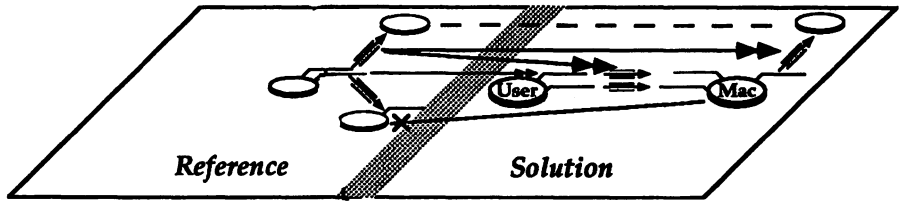


Figure 8-18. Impact Analysis

The difference between these two models is more than the computer. We have abolished broad areas of responsibility from some people and shifted others to different people. Figure 8-20 annotates this to account for the impact on the business.

The Impact Analysis is one of the most important business tools available within SBM. For in-house development, it allows management to understand the changes in business, not just technical, terms. Provision for training, accountability, the transition to automation, and a host of other decisions can be made at the same time the computer system is designed rather than later. The Impact Analysis also is key to analyzing costs versus benefits since it directly compares the before and after images of the business and provides keys to the cost of the transition. For commercial

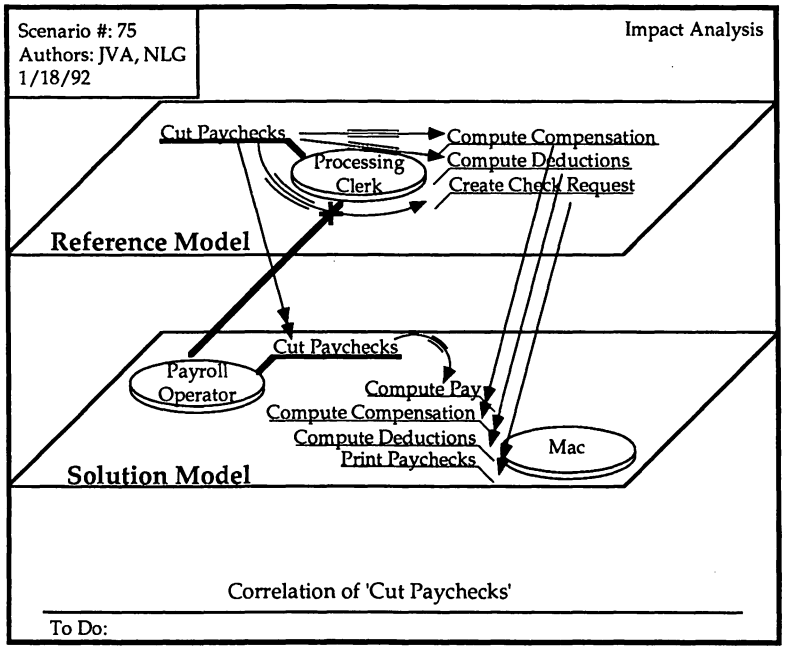


Figure 8-19. Correlation of Reference and Solution Models in payroll

developers, this is the acid test. The Impact Analysis, used against Reference Models for competition and the "null" case (no product currently in use), clarifies product position and can be the cornerstone of pricing by identifying all costs and benefits to the customer. It can isolate potential buying objections by identifying what the customer must change in order to use the product. Finally, it can form the backbone of sales by expressing the product in terms the customer will understand: "What does this change in my business?"

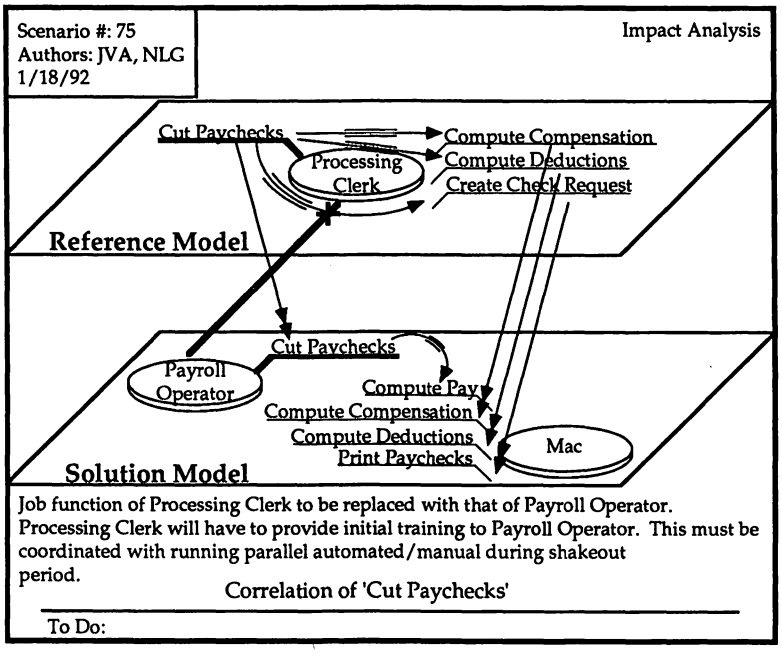


Figure 8-20. Annotated Impact Analysis for payroll

▶ Existing Computer Systems

So far, we have rather naively assumed that no computer is in use in the Reference Model, and that we have introduced our (one) new system in the Solution Model. Such assumptions are not realistic. In reality, any of the following statements might be true.

- Other systems are being replaced by the Macintosh.
- Other systems are being used with the Macintosh.
- Other applications are in use on the Macintosh already and have some relevance to the problem at hand.
- The project is broken down into multiple, cooperating but separate applications co-resident on the Macintosh.
- We are upgrading an existing application on the Macintosh.

In all cases, we need to develop Reference Models that incorporate the current state of affairs, computers and all. We also need to draw a distinction here between systems being replaced or upgraded versus systems that will collaborate with the new program. Replaced systems will appear in the Reference Model but not in the Solution Model; collaborative systems will appear in both. Collaborative systems are treated as users of the new program of a sort. In the Technology Plane, we provide an Environment Model to describe these collaborations in much the same way we describe user interactions in the User Interface Model.

▶ Summary

The objectives of the analysis phase are to establish the scope of the work and develop a plan for completing the project. To accomplish these objectives, we combine three basic activities: business analysis, conceptual design, and prototyping. Business analysis centers in the Business Plane, which has a Reference Model representing the way the business runs today and a Solution Model, which represents the way the business will run in the automated future.

- The frame of the Reference Model contains a definition of the problem to be addressed by the new program and a description of the major purposes of the business unit as a whole. Behaviors of the business unit become the essential responsibilities of the model, as opposed to the incidental responsibilities that support the essential ones through collaboration.
- The frame of the Reference Model and the model itself form a double description. The collective elements of the model and their responsibilities must correlate to the behavior set defined in the frame. We use the CPC process to build both the frame and the model, pushing central issues of the frame into the model and correlating back to the frame all additions to the model. As we add scenarios, we use synthesis to maintain the integrity of the model. We resolve several types of possible problems including synonyms and homonyms, object/category conflicts, semantic differences, relationship conflicts, and category/member conflicts.

The Solution Model is tomorrow's Reference Model. It focuses on the interaction of the computer with the users and attached devices. The Solution Model is built both by pushing central issues of the Reference Model to the Solution Model and by direct expansion of the Solution Model. As information is added to either model, correlation is used to keep

track of all changes to the business that will result from the project. These correlation relationships ultimately become the Impact Analysis.

If we are upgrading or replacing an existing computer system or program, the process is the same. The only real difference is the presence in the Reference Model of the existing system. Collaborative systems interact with the program under development in the Solution Model in much the same way that users do.

9 ▶ Analysis Part II: The Technology Plane and Beyond

▶ What This Chapter Is About

This chapter continues our discussion of the analysis phase. We now turn our attention to the Technology Plane and conceptual design of the program. The Technology Plane has three regions: The Content Model holds a description in objects of the inner workings of the program, shorn of its user interface and interfaces to other devices. This describes the underlying capabililities of the software. The User Interface Model allows us to describe interactions between users and the computer at a very fine level of detail, again using objects and categories. The user interface portion of the design cannot do much by itself; instead, it maps the capabilities of the Content Model onto specific buttons, windows, and other features of the user interface. The Environment Model describes how our system controls or is connected to other computers, devices, or networks. The Environment Model functions for those devices much as the User Interface Model functions for the user by isolating specific interactions from the underlying content of the program.

This chapter also discusses in more depth correlation, the second of the three forms of calibration. Correlation ensures consistency across planes. During the analysis phase, it is used principally between the Solution Model and the three regions of the Technology Plane. This chapter also introduces the first of our strategies for achieving the Four Itys: maintainability, extensibility, modularity, and reusability. In building the Content Model, we create objects to suit our purposes. The choice of objects and the methods used to assign responsibilities to them largely determines how successful we will be in achieving the Four Itys. The third major skill

covered in this chapter is prototyping. This includes mockups of user interfaces as well as uses of the Execution and Program Planes to support the analysis process.

The chapter closes with a discussion of how one knows when the analysis phase is complete. The quick answer is, "When someone in authority says it is done." However, there is more to that statement than meets the eye.

▶ Content Model

The Content Model is a description of the entire capability of the program, independent of its user interface. The Content Model is an idealized model of the objects that make up the interior of the program. "Idealized" means that it is a simplified description rather than a literal blueprint. The Content Model stores the data held by the application. The collective responsibilities of the Content Model objects implement all responsibilities of the computer or program element of the Solution Model. Figure 9-1 shows the relationship between the Solution and Content Models.

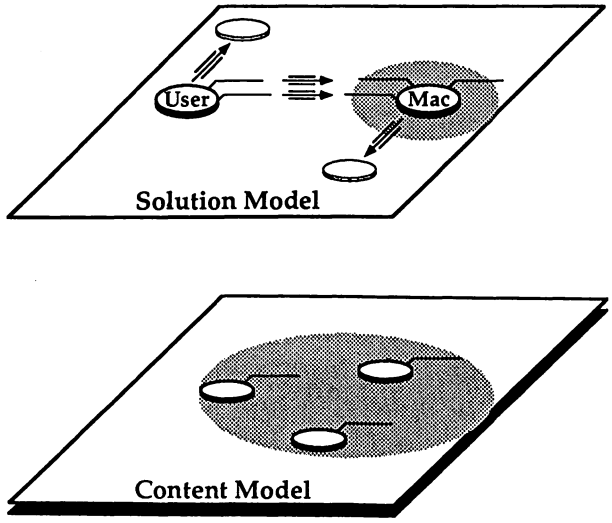


Figure 9-1. Building the Content Model

▶ Overview

The Content Model is composed of conceptual objects and categories, their responsibilities, and relationships. The word "conceptual" is very important here. Content objects are seldom real-world objects in a literal sense. More commonly, they are metaphors for the objects of the real world; some, in fact, may have no relationship to the real world at all. We create the objects in the Content Model to suit our program; they are creatures of mind, not fact. Yet, at the same time we are still prepared to use all of the cognitive machinery discussed in Chapter 5—basic level types and cognitive categories—through metaphors for and simulation of real-world objects.

We build the Content Model by first developing an *object candidate list*, a set of names we can potentially turn into conceptual objects, then assign responsibilities to them. Many of these responsibilities come from those of the computer element of the Solution Model; these form the center of the Content Model. Others are the result of expansion of the Content Model toward the periphery.

▶ Content Frame

The Content Model has as its frame the set of responsibilities assigned to the computer in the Solution Model. Each responsibility of the computer is transferred to one or more specific elements of the Content Model. The Content Model is thus an explosion of the computer element of the Solution Model into constituent parts, that is, objects and categories. The Content Model is framed by and also frames the User Interface and Environment Models.

▶ Elements and Relationships

The elements of the Content Model are objects, categories, and their responsibilities. Both structural and behavioral relationships apply within the model. Correlation relationships are drawn to the Solution Model. Behavioral relationships exist between objects of the three regions of the Technology Plane. Responsibilities of the computer element of the Solution Model become the essential responsibilities of the Content Model; all others are incidental and should serve to directly or indirectly support the essential ones.

In the Content Model we see the beginnings of the "expert" ideas explored in Chapter 4 such as track that lays itself, layouts that validate themselves, employee objects that pay themselves. We use anthropomor-

phism and other more targeted techniques in deciding what objects should be created and what responsibilities they should have. Although we are no longer dealing with the real world, we still want to keep our concepts and designs accessible to non-programmers by applying the cognitive principles of Chapter 5.

▶ Building the Content Model

In order to build the Content Model we must be able to do two central things: define objects and categories and assign responsibilities to them. Defining the objects and categories is itself no trivial task. Simply simulating real world objects and categories in software is seldom the best solution. We need guidelines on how to decide what is a good object or category and what isn't. We must also have some mechanism for assigning the right responsibilities to the right elements. Remember the expert treatment of the problems in Chapter 4? These are not, despite all appearances to the contrary, plucked out of thin air, nor are they simply the use of anthropomorphism. We will talk a great deal about how to choose the objects and categories and how to assign them responsibilities.

The Content Model is initially built by constructing an object candidate list then assigning the essential responsibilities from the frame to those objects and categories. The object candidate list is just that: a set of potential objects and categories from which we can choose those that make the most sense for our program. Once this initial set of elements is in place, the Content Model is expanded and refined from five different directions.

1. The Content Model can be directly expanded by fleshing out responsibilities and elements already in the model.
2. As the User Interface Model is expanded, it is correlated to the Solution Model which, in turn, correlates to and expands the Content Model.
3. Changes to the Solution Model propagate through correlation with the Content Model.
4. As the Environment Model expands, correlation to the Solution Model indirectly expands the Content Model.
5. When the Content Architecture of the Execution Plane is expanded, correlations with the Content Model expand both.

This places the Content Model at the center of a whirlwind of activity for much of the project.

Building the Object Candidate List

One of the benefits of structuring our programs and models around objects is the comfort it brings that results from the sense of recognition by both programmers and non-programmers. However, objects alone are not enough. We must be careful to choose names for them that bring a similar sense of comfort and familiarity. A program that uses only names like "X" and "A001" may, in theory, function identically to one that uses "Employee" and "Paycheck," but the latter program is more likely to turn out correct. The correctness of a program relies on the degree to which people can examine it, understand its structure, and relate it to the world they perceive. Even though we seldom use objects in the same way as they occur in the world, there is great value in selecting names that encourage people to use anthropomorphism and metaphor as aids to understanding and exploration. The most effective names, therefore, are objects and categories that are familiar to people involved in the business. The least effective names are "computerese" names like "Sorted Doubly Linked List" or "X.25 Packet."

Kinds of Objects. We can look for six basic kinds of objects and categories to build the object candidate list. This list is used to create the Content Model, but we also refer back to it when building other regions of the model.

1. *Directly manipulated.* These are the "things" from the world that are directly visualized on the screen and that the user can manipulate with the mouse. Railroad track in our railroad design example is directly manipulated: The user can click on it and drag it around. These tend to be perceived as basic types translated directly into software, but the actual responsibilities and collaborators in the computer model can add to or modify those of the real world.
2. *Manufactured.* These are objects the system must produce in order to carry out its responsibilities, such as reports, checks, and other outputs. They frequently do not exist in the unautomated environment or are substantially different from their real-world counterparts. On-screen manufactured objects, such as those of a data entry screen, may have no real-world counterpart at all. However, we still attempt to cast them as analogous to real-world documents. Hard-copy outputs are a curious mixture. They really exist in the world—you can pick them up, turn them over, and put them away—but they are created by your program. These are truly manufactured objects.

3. *Reconstructed.* These are metaphors for familiar real-world objects and categories. The name is the same as that of something familiar, but the responsibilities and behaviors may be mostly or completely different. This is best illustrated by example. In an object-oriented program, payroll records kept manually in the real world become employee objects that store their own data and perform computations on it, such as calculating deductions. It is easier to anthropomorphise a person object than a document object. Thus, "The employee pays herself" is easier than "The payroll record pays itself." More generally, the closer the object is to the basic level, the better are the prospects for use in reconstruction. When you use reconstructed objects and categories, responsibilities of the Macintosh and program as a whole get redistributed from their owners in the real world to their "natural" owner in the program. As we will see later, this often requires assigning behaviors that use data to the objects that contain or own the data. It is this type of object that prints itself, draws itself, does computations on its own data, or sends itself messages.
4. *Temporal.* These are objects that bridge time and space, such as transactions. In an automated teller machine, we can accept data from the customer, assemble it into a transaction until all of it has been received, then send the transaction object to a host mainframe. Another use might be to keep an audit trail of all transactions.
5. *Automation.* These objects are created solely to represent concepts of the program. They often occur when part of a business is being automated in such a way that new concepts and responsibilities are being introduced. They also may exist when previously distributed responsibilities can be combined for efficiency on the computer. For example, a date object might be able to add and subtract days from itself. Automation objects also exist to account for things that cannot be done in the natural world. We might conceive of an "animation object" in a program that moves things around in a window.
6. *Auxiliary.* These are service providers required by implementation on a computer. Pure container classes like lists and sets are good examples. Other examples arise in interfacing to the platform—Macintosh, Toolbox, and so on—in the Execution and Program Planes.

It is not important into which of these six categories a candidate falls; many are good candidates in several of these categories. Instead, think of the previous list as a set of filters through which to view the world in search of candidate objects.

Finding Objects. Where should you look? Two great places to start are with the elements of the Reference and Solution Models. These make good targets for reconstruction since they are already familiar in the business. Outputs of the program such as reports, display screens, and so forth are obvious candidates for use as manufactured objects. The general concepts behind the user interface often yield direct manipulation and manufactured objects. Anything the user drags around on the screen, such as pieces of track on the layout, is an obvious candidate for use as a direct manipulation object or category. If you are dealing with changes to a database or other store of data, or otherwise have activity that spans time or space, temporal objects can be spotted wherever there is an action that the user or the program can initiate. Automation objects and categories generally come into play after the first round or two of development of the Content Model when it becomes clear that there are no good candidates for certain responsibilities. Auxiliary objects are the least likely to have any relation to the real world and apply chiefly to the Execution and Program Planes.

The best candidates are those that are close to the basic level, immediately relevant to the problem in the real world, and easy to anthropomorphise. Categories of people are good candidates as are things that people interact with, such as track. Forms, documents, and records are common in the real world, but as a rule they tend to make poor object candidates. Often they are artifacts of an existing manual process that get in the way of automation. They are not at the basic level and so make bad targets for metaphor or anthropomorphism. Remember our example of choosing an "employee" object over a "payroll record" object. Exceptions are documents that come from outside sources, such as catalogs, phone directories, and the like. Documents generated internally purely as an incidental part of the business's operations are the worst candidates.

An object candidate list emerges from all of these views of the world. Not all candidates are used and, once the list is drafted, we don't care about which of the six types a given candidate represents. However, we now have a rich source of names to use in making concepts concrete.

Let's see how these principles apply in our two running examples. Figure 9-2 shows part of the object candidate list that the authors constructed for the model railroad design application. For each item on the list, the type or types of the object from the list of six types is indicated.

Directly Manipulated:
Track
Scenery
Car
Layout

Manufactured:
Bill of Materials
Layout
Portfolio

Reconstructed:
Bill of Materials
Catalog
Layout
Portfolio

Figure 9-2. Object candidate list for model railroad design

Figure 9-3 shows part of the object candidate list for payroll. Almost all of the items on this list were already in some way present in the Business Plane, particularly in the Reference Model. This includes people and documents already in use in the manual environment.

Directly Manipulated:

Manufactured:
Check
Hours Worked Report (and other reports)
Time Sheet

Reconstructed:
Employee
Check
Time Sheet

Temporal:
Check
Transaction

Automation:
Database

Auxiliary:
Employee List

Figure 9-3. Object candidate list for payroll

Mapping Responsibilities onto Objects and Categories

Once the object candidate list is in place, the next step is to take the essential responsibilities—the responsibilities of the computer element of the Solution Model—and assign them to names on the list. In the process, it is common to break one responsibility into several collaborating responsibilities, or even to determine that there are several distinct responsibilities that should be in the frame. This sounds simple, but it is not always obvious what the best distribution is. Should a check compute an employee's compensation, should an employee object, or should there be some sort of "compensation computation object?" Should a layout know how to instruct all of its parts to simulate the operation of the layout, or should there be a "simulation object" that contains parameters of the simulation such as speed? These are not easy questions to answer in general, and they mark the sharpest break between the productivity of experts and mere mortals.

Before exploring this issue of assigning responsibilities in greater depth, let's close this section with a discussion of incidental responsibilities. Many incidental responsibilities are added to the model as the result of decomposing other responsibilities. Others, however, are the result of direct examination. It makes sense, given an employee object, to give it incidental responsibilities to supply its name, address, and social security number upon request. We can add these to the model even at a time when it is not clear what client objects and responsibilities, if any, might be interested in that information or those actions. In the Business Plane, we asserted that incidental responsibilities should not be included unless they directly or indirectly support essential responsibilities. In the Content Model, that is not always true. Incidental responsibilities can be added at an early stage in anticipation of a need for them later, or as a way of building in future expansion capability at an early stage of design. They can also be used as a way of expanding the model by saying, "It makes sense that the scenery object should know its own cost; what other objects should be interested in that information?"

▶ Object-Oriented Software Engineering, Part I

In the Content Model, we assign responsibilities according to our metrics for "good" design, not based on the real behavior of real objects in the real world. We rely on metaphor and anthropomorphism to keep the designs accessible. One of the most basic objectives in assigning responsibilities, which also influences which relationships we set up, is achieving

independence in the model. We want the model and, ultimately, the program to allow evolution of one or a few objects and methods at a time without interacting detrimentally with other parts of the whole. Our goal is to be able to say, with a straight face, "Don't bother to test the whole thing, I only changed one line of code." This is not as easy as it sounds.

▶ Achieving Independence: An Overview

There are five objectives that can be used to achieve independence in an object-oriented design, and they apply to the Content Model as well.

1. *Limit responsibilities.* Form objects that have a narrowly defined purpose and a small set of responsibilities dedicated to that purpose.
2. *Limit data knowledge.* Minimize the amount of information that is passed around from one object to another, and avoid the duplication of information in multiple objects.
3. *Limit implementation knowledge.* Care only about results, not the steps used to obtain them.
4. *Limit relationships.* Limit the set of objects of which any given object has knowledge.
5. *Limit type knowledge.* This applies only when we impose classes and inheritance on the model, which does not occur until the Program Plane. For this reason, limiting type knowledge is deferred to Chapter 10.

We can label two general approaches as the "black box" and "client/server" architectures. In a black box design, one or more objects are hidden through an interface provided by another object. Take the example of a whole/part assembly such as a model railroad layout. Rather than having a client call each individual component of the layout to tell them to draw themselves on the screen, we tell the layout object to draw itself and it, in turn, tells its components to draw themselves. In other words, we make the entire layout a black box, with the entire interface funneled through the whole, and hide the parts from prying eyes. This is *whole/part encapsulation.* A client/server relationship exists whenever one object (the client) has knowledge of another (the server), but not the other way around. Since the relationship knowledge is one-way, the client can evolve without causing side effects for the server. These two strategies can work in concert, as in an assembly in which the parts are ignorant of the whole. A discussion follows regarding general strategies for each of the types of limits we seek to achieve.

▶ Limiting Responsibilities

We start by defining objects that are highly specialized and single purpose. We want an object to be absolutely brilliant about what it can do, and totally ignorant about everything else except other objects it needs as collaborators. In addition, we want to take our objects and decompose them into parts and wholes. Often this is a strategy used in the Execution or Program Planes, but it is wise to keep in mind even early in the process. If an object must do too much, you should either split it into multiple objects or break it down into a whole with component parts or even a container of separate objects. Individual responsibilities should be highly specialized. Ultimately, most methods in the program are between one and ten lines of code. Although we are not concerned with code in the Technology Plane, that ultimate objective helps set the tone for when to split responsibilities.

▶ Limiting Data Knowledge

In analysis and design we are concerned only with identifying who is responsible for providing information on demand, not how that is done or the details of information storage. For example, an employee object can be given a responsibility to provide its name and hourly wage; we need not specify where or how that information is stored. All information is accessed functionally through responsibilities. Nevertheless, certain information can be described as having an "owner," which is the object ultimately responsible for providing the information directly. Continuing our example, the employee name and wage might be passed along from object to object (though we hope not!) but the employee object is the ultimate source if it owns that information. The concept of information ownership is extremely important in deciding where to place responsibilities and is used in the following fundamental rules of information hiding in object-oriented programs.

1. *Owners are responsible for their information.* Responsibilities that use information should generally reside with the owner of that information.
2. *Minimize information transmission.* Where information is passed from object to object, minimize the overall transmission of information. A good way to think of this is to visualize the information sent from one object to another over expensive phone lines.
3. *Minimize redundancy.* Minimize redundancy in the information; that is, for a given datum have a single owner and ask the owner for the information when it is needed. A corollary of this rule is to recompute

derived information when it is needed rather than storing it redundantly with the source information used to compute it, unless there is a serious performance problem with the computation.

4. *Use accessors.* Within the owner, information should be accessed through a single responsibility and changed through a single responsibility. These *accessors* are probably incidental responsibilities, used only to insulate the information from the rest of the design; that is, they exist for software engineering reasons, not because they are part of the conceptual outline.

The combination of these rules encapsulates the information nicely. This is why, for example, we shift the responsibility to compute compensation from a clerk to the employee herself. The employee is the natural owner of the underlying information used in the computation and, therefore, should have the responsibilities that make use of that information. An object—in this case, the clerk—that does nothing more than ask some other object for information, perform a computation on it, and pass the results along to someone else may not even need to exist in a program once its responsibilities have been distributed to the owners of the information. This is one of the expert tricks of object-oriented design, since on the surface it appears to have no grounding in the real world. Real-world objects do, in fact, contain lots of information in the form of their attributes; take the employee object, for example. They just don't do the kinds of operations on their own information that we would like to see in a modular software architecture.

If a computation requires information from two different objects, the following should be considered:

1. Computations should stay with the most stable information of the two; that is, the information that changes least as the program runs.
2. Consider merging the two objects to achieve better encapsulation of information. Balance this against other objectives, since taken to its extreme it would mean combining all information and all responsibilities into a single object!
3. Consider who really should own the information. It is possible that one or both sets of information belong with a different owner.

▶ Limiting Implementation Knowledge

Data encapsulation—hiding data behind a functional interface—is usually what comes to mind when people first learn about encapsulation in object-oriented programs. Just as important is the encapsulation of imple-

mentation. Objects as black boxes allow us to encapsulate algorithms. As with information, there should be a single path to a critical algorithm or procedure, rather than bundling it with other functions. In fact, often one spins off an algorithm to a separate object simply to isolate it further from clients. Many examples arise in interacting with the host operating system, where we can use some objects to isolate other objects from lower-level operating system functions and even the computer itself. Another common strategy is to decompose a whole into parts to isolate key implementation knowledge in a single, small object, hiding that knowledge from the whole and its other parts.

▶ Limiting Relationships

Finally, we want all of our objects to be as ignorant as possible of the existence and characteristics of other objects. For example, a car object might have as components a steering wheel, a brake pedal, and a gas pedal. A driver object probably has knowledge of the whole and its three parts: the driver turns the wheel, presses on the gas pedal, and presses on the brake pedal, as shown in Figure 9-4. Or does he? If we recast the concepts

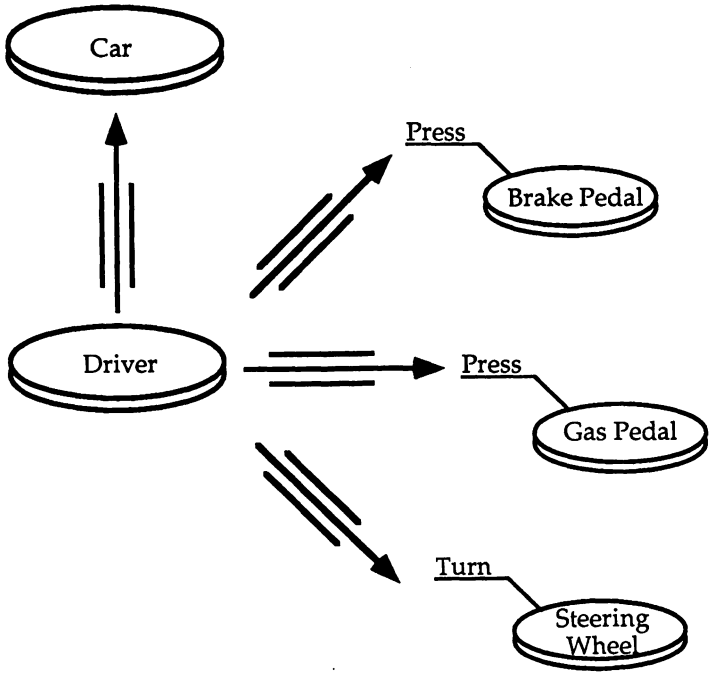


Figure 9-4. Driving a car

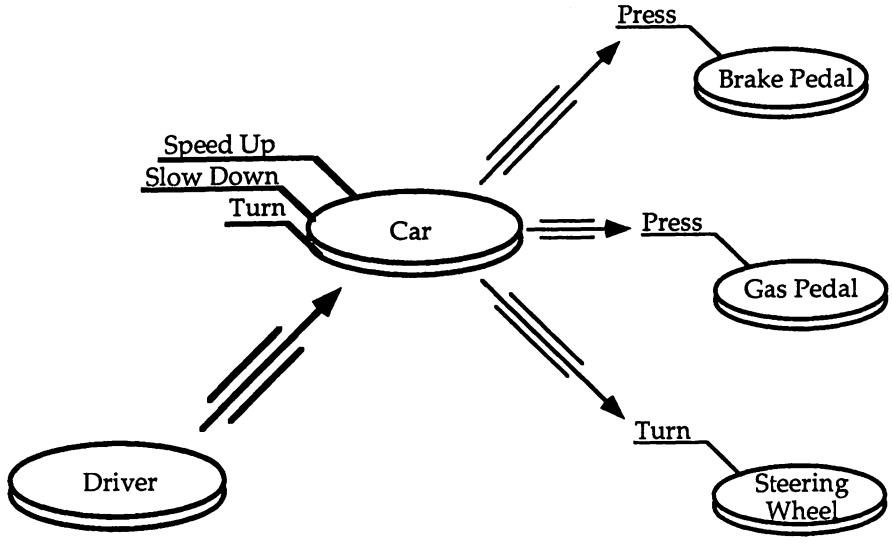


Figure 9-5. Driving an object-oriented car

a little, it turns out that the driver need know nothing about the components, as shown in Figure 9-5. In the latter scenario, the components of the car are encapsulated within the car itself. The driver issues general instructions like "speed up" or "turn" and the car object knows how to accomplish the feat using its components. We have significantly limited the relationships in the model. This is a good example of part/whole encapsulation, which limits relationship knowledge at the expense of a few extra methods and more relationship knowledge in the whole. The driver must interact with more methods of the car (the whole) but now need have knowledge of only one object rather than four.

▶ Conflicts among the Limits

We just said that the whole should be decomposed in order to limit implementation knowledge, but that parts should be encapsulated into their wholes in order to limit relationships. In this example, these statements are not really in conflict. In the process of encapsulating the car's components, we recast the responsibilities in such a way that implementation knowledge was also limited. We could reasonably construct an alternative car without a steering wheel or pedals that would still respond properly to the responsibilities to turn, speed up, and slow down as requested by the driver object. However, it is not unusual for conflicts to

exist among the limiting objectives. One sacrifice made here, for example, is that the car object is no longer as narrowly defined as it would be if clients had to directly operate its parts. Building an object-oriented car is likely to be expensive! More generally, it seems that there is always a dynamic tension between independence of data, responsibilities, and implementation on the one hand and the need to limit relationships and design limited-purpose objects on the other.

Some dependencies such as those of wholes and their parts or between collaborators are unavoidable. This points the way to a partial solution. Where a relationship has to exist for structural or other reasons, you should prefer to eliminate other relationships. This is what we did with the car: The car had to have knowledge of its parts, but the driver did not. After taking this factor into account, one can roughly prioritize from the most important to least the remaining objectives in this order.

1. Limit data knowledge.
2. Limit implementation knowledge.
3. Limit relationships.
4. Limit responsibilities.

In fact, we often *add* responsibilities in the interest of achieving the other objectives. It is only when there is no conflict with the first three objectives that limiting responsibilities becomes an important goal.

Before leaving this discussion we should define what we mean by "dependence" and "independence." Most programmers are used to having a control structure that "runs" the program. Object-oriented software is very different. The flow of control and, therefore, dependencies, are defined by the passing of messages from object to object, not as a formal control structure. An object-oriented program is a system of independent entities in equilibrium, with control implicit in the system of messages. For all of these reasons, we should not expect traditional top-down methods of analysis or design to result in good modularity. It is only through analysis of often overlapping scenarios that we can reveal the best strategies for achieving the types of limits we seek.

▶ Calibration, Part II: Correlation

Thus far, we have dealt with correlation quite informally. However, we have now reached a part of the project where correlation becomes a more complicated affair of correlating the Content Model with the Solution Model. Previously, we dealt with correlation of the two business models, both of which contained real-world objects and both of which shared a

common structure. The Content Model is different structurally and in purpose from the Solution Model and requires more careful treatment. The methods we are about to cover will then guide us through the rest of the Solution-Based Model.

The basis of correlation is the concept of double description in which two parts of the SBM describe the same thing. We have used this to make sure that the Reference and Solution Models correspond and collected all differences in the Impact Analysis. That is the only example we will see where correlation is used between models of the same plane. Throughout the rest of Solution-Based Modeling, correlation is used between a region of one plane and a corresponding region of the plane immediately above or below. Specifically, correlation is used with the pairs of regions shown in Figure 9-6.

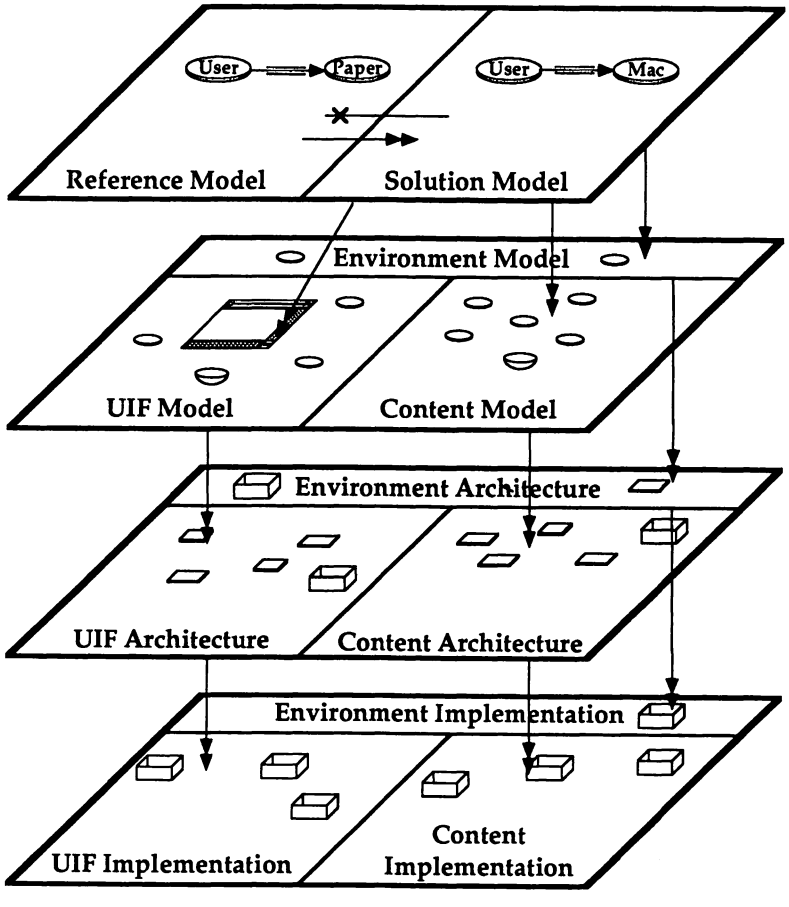


Figure 9-6. Correlation in a Solution-Based Model

The process and the basic principles are the same for correlating all of these pairs. In each case, we can refer to an "upper" and a "lower" region, the upper region being the one on the higher plane. No element or relationship of a lower region should be unaccounted for in some way with respect to the upper region. Given an element (object, category/class, responsibility/behavior/method, or attribute) or relationship in the lower region, we must use one of the following justifications:

1. The element or relationship implements—or is part of the implementation of—at least one element or relationship of the upper plane.
2. The element or relationship replaces, in whole or in part, at least one element or relationship of the upper plane.
3. The element or relationship is new and does not correspond to anything in the upper plane. This should never be a default assumption, but should instead require someone to positively state that it is so. For example, the computer element of the Solution Model may be completely new if in the Reference Model no computer is in use for the intended purpose.

Any element or relationship of the lower region that has not been correlated in one or more of these ways is a dangling thread that must be accounted for at some point. Similarly, we do not allow elements or relationships of the upper region to go unaccounted for in the lower region. Generally, every element must be implemented or replaced by something or it is a dangling thread. The sole exception is in correlating the Solution Model to the Content and User Interface Models, where we need only correlate the computer or program element to the Technology Plane. With this one exception, any element of the upper region that is not accounted for by implementation or replacement is a dangling thread. In the case of the Solution Model, the same principle applies to all responsibilities of the computer or program element.

Does this seem like a lot of work? It is. Does it seem mentally taxing to classify all correspondences as "implementation" or "replacement," even at times where this is not a straightforward one-to-one relationship between a pair of elements? This is also hard work. *However, not taking this step of correlation does not save time over the course of a project.* The process of correlation ensures that communications have not broken down and that distortion has not been introduced in proceeding from one aspect of the project to another. Put more simply, we want to make sure that we end up solving the same problem in the same way we initially intended. Recall from Chapter 1 that communications problems and distortion are major components of software costs, both in initial development and on-going

maintenance. Because that cost is chiefly labor, we can conclude that the time will be spent, one way or another. It is better to get the problems ironed out at the earliest possible time to minimize the side effects when mistakes are made. This minimizes the total labor required and makes sure that time is spent constructively, rather than in fixing mistakes or false assumptions.

Correlation is also a way of allowing freedom of movement. Because we know that dangling threads will not be forgotten, we need not attempt to complete one part of the model before exploring another. People naturally jump around a great deal in solving problems; we should encourage this process, not work against it. By providing a way to smoothly shift from business modeling to analysis to design to programming and back, we encourage people to spend their time on whatever makes them most productive. This also avoids "stuckness," the feeling that you are not making any progress. When stuck on any part of the model, we refocus either on another plane or on the center and move on. Correlation—indeed, all three forms of calibration—ensures that we eventually return to pick up the dangling threads left behind. In Robert Pirsig's *Zen and the Art of Motorcycle Maintenance*, a passage describes a student suffering from writer's block in composing an essay on the United States of America. Her teacher breaks the block by suggesting that she start with the upper left brick of the front wall of the Opera House in Bozeman, Montana. We have described this process before as central to CPC: If you get stuck or even slow down, focus on the center, or the center of the center, or shift to a slightly difference perspective, but in any case keep moving. Calibration makes this process manageable as part of a large project involving many people. Correlation allows focusing on different planes that are at a finer level of detail or take a different perspective; synthesis allows us to shift our gaze to different, overlapping snapshots of the problem; synchronization, which we will describe more fully soon, is within a single plane what correlation is across planes.

Finally, correlation is a tool of managing a complex project. At any point in time, you know what correlation remains to be done: tying up the dangling threads. This does not necessarily help with expansion into new topics, but it does give a very accurate picture of how close to completion the current scope is. This, in turn, can be used as a tool of scheduling, budgeting, and progress assessment.

The one thing we have not discussed is when to correlate immediately and when to leave dangling threads for later. This is based on marginal progress. As long as you are quickly and productively expanding one region, continue to do so and defer correlation to its upper and lower counterpart regions. As soon as progress slows, even slightly, correlate to

the region above, then either push the center down to the region below, refocus on the center in the same region, or expand toward the periphery.

▶ User Interface Model

Remember that this is not a book on user interface design, but on overall development methodology. There are a number of excellent books on user interface design available, starting with Apple's *Human Interface Guidelines*. We deal with user interface design issues only in the limited sense of how to integrate your user interface concepts into the Solution-Based Modeling methodology.

▶ Overview

The User Interface Model is based on the computer element of the Solution Model, along with elements representing users of the computer, as shown in Figure 9-7. The Solution Model identifies what the users are responsible for doing and what responsibilities the computer must have in order to support their efforts. For each responsibility of the user, we provide a sequence of events in the user interface that allows the user to carry out that responsibility.

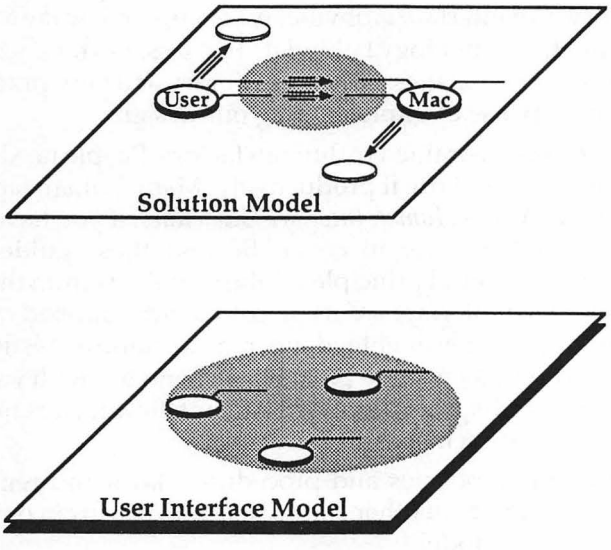


Figure 9-7. Building the User Interface Model

The scenarios we form in the User Interface Model use the normal VDL conventions, but add snapshots of the screen and output reports. Static and time-sequence scenarios are used to completely describe both the look and feel and the function of the interface.

The User Interface Model is built by starting with the collaborations of the computer and user as identified in the Solution Model, mapping those onto specific user interface features such as buttons, windows, and scrollable lists, then expanding the User Interface Model and Solution Model in lockstep toward the periphery. Calibrations are also needed with the Content Model throughout the process.

▶ User Interface Frame

The User Interface Model is framed in several ways.

1. The Solution Model tells us the set of responsibilities the user relies on to use the computer. We seek to provide concrete mechanisms for invoking them through the User Interface Model.
2. The Content Model defines the functional capabilities of the program, independent of the interface. The user interface can determine the sequence in which these functions are carried out but cannot expand them.
3. The user interface is obviously constrained by the Macintosh platform and the technology behind it. To a lesser extent, we take into account the characteristics of the class library as a very practical constraint on the expense of implementing our designs.
4. We are constrained by human factors. People need to understand the program and use it productively. Many human factors are discussed in the Apple *Human Interface Guidelines*; if you have not already done so, read it cover to cover. Beyond those guidelines, you cannot separate sound principles of the visual arts from the design of Macintosh user interfaces. Certain colors carry implied meanings—red, for example, often denotes danger in our culture. Aesthetics are a factor—do you *really* want to put a black menu bar with yellow letters on the screen? Chapter 5 discussed some of these factors in light of categories and image schemas.
5. Company policies and procedures, laws and regulations, security, and a variety of other external factors constrain or influence the User Interface Model.

Most of these factors are implicit parts of the frame, including the other regions of the SBM, the technology used, and, if one includes the Execution and Program Planes, the class library. "Human factors" is too broad a subject to completely describe in the User Interface Model frame, but we can and should capture any central issues. For example, one could lay down rules such as "Fully conform to the Macintosh human interface guidelines," or "If it's visible, it's clickable." (That is, anything you can see on the screen will respond to a mouse click.)

The same human factors tend to apply to all projects, but the set of factors that are central to the success of any one project vary. In an accounting system, it is not as critical to worry about the use of color as it is in a circuit layout design program that must represent multi-layer PC boards using color. It is strictly a judgment call when human factors are an analysis issue versus when they apply only to the details of design and programming.

▶ Elements and Relationships

The User Interface Model contains objects that represent the visible exterior of the program: buttons, icons, windows, reports, and so on. These objects normally are not described as holding data of their own. Rather, they both obtain and update data by interacting with objects in the Content Model. In other words, the User Interface Model is a content-free description of the program. Returning to an earlier analogy, the Content Model contains the fuel tank, fuel pump, and engine, while the User Interface Model contains the gas pedal, fuel gauge, and the cap and hole through which you pump gas.

Manufactured Objects

Objects in the User Interface Model are a curious cross between the real-world objects of the Business Plane and the conceptual objects of the Content Model. It is true that the objects of the user interface are not quite like the people, computers, machinery, and so forth of the Business Plane. You can't pick them up or touch them. Yet, neither are they purely conceptual; they are visible to the eye and you can manipulate them by using the mouse and keyboard. When the program is running, *they really do exist, but only on the surface of the computer and on the printed page.* They are best thought of as real-world objects that are added to the world as the result of creating the program. In the terms we discussed for object candidate lists, most user interface objects are manufactured.

Command Objects

There may be additional objects that are implied but not visible. The user must understand the existence of such objects in order to fully comprehend what the program is doing and how to use it. One example is a *command* object. When the user selects a menu item, presumably the program takes some action as a result. The action itself is a *temporal* object in the sense described earlier for the object candidate list. When the user chooses the menu item, the command object is created and becomes part of the program. It generally persists until it can no longer be undone, but in some cases command objects might stay around longer to maintain an audit trail.

This is more flexible than associating the action with the menu item "object" itself, since it allows a single command to be represented in several ways in the interface. The same command can be invoked through a menu item, clicking in a palette, through some action in a window, or even remotely when an event is received from another program. Command objects can also implement the sometimes complex sequences required to support Undo and Redo commands. These types of objects are common features of class libraries.

There are different types of command objects, but most share the following characteristics.

1. They change the state of the program.
2. They are capable of undoing that change in response to a choice of "Undo" from the Edit menu.
3. They can almost always be invoked from a menu. In addition, they can be invoked through other means as well, such as palettes or Apple Events messages from other programs.
4. They represent a single unit of work *as perceived by the user*. This is a critical point and is the reason we take command objects into account in the analysis phase as part of the user interface design. Commands represent concepts we want our users to be able to understand, actions they know they can take in using the program. They are not "mere implementation details," but a critical part of the user's overall understanding of the program.

Some command objects act immediately, while others may introduce modes that prohibit other actions by the user until the command is completed. For example, a command object may pose a modal dialog, then take action when the user clicks on the "OK" button. As a final note,

command objects are sometimes confused with *trackers*, which are objects that track the mouse and provide "rubber band" or other visual feedback. Trackers usually do not change the state of the program; they merely provide feedback to the user.

Document Objects

Another, more subtle, implied object is the *document*. The concept of a document in a Macintosh program is often misunderstood and, even when understood, can be difficult to apply. Let's carefully define what a document is, then explore how to use them in the User Interface Model.

Many menu items are polymorphic; that is, a single menu item may invoke any of a number of commands, depending on the current context of the program. "Undo" comes immediately to mind: undo what? The action taken depends on the state of the program at the time that the user chooses "Undo." One way to inform the user of the context is to change the title of the menu item as the context shifts. Thus, instead of simply "Undo," we might display "Undo Paste" or "Undo Copy" to clarify what happens when the item is chosen. The most typical way to determine the context of a Macintosh program, however, is by associating context with the frontmost window. This is especially true of the File menu items Close, Save, Save As, Revert, and Print. Each window has its own interpretation of these commands. When the command is invoked, the program looks at the topmost window and takes action accordingly. In object-oriented terms, the event is sent through a message to the window object for the topmost window.

This works fine for simple paint or draw programs that have a one-to-one correspondence between a window and a file on the disk. In fact, in such cases there really is no need for a document object. The File commands are passed by the window directly through to the associated file object (in the Content Model). There are, however, a number of other less straightforward situations that may arise. For example, several windows may share all or part of their data. In a spreadsheet program, you can have one window display the grid of numbers, while another shows the same information in the form of a pie chart. Also, the data may not be in a file, but in a database. The data may even be shared by other users and other programs. In such cases, it quickly becomes clumsy to force windows to handle the File commands by themselves.

Let's try the first variation: multiple windows sharing data. In this case, choosing Save in any one of the windows should save the data in all the windows. Yet, we really don't want to force each window to know about all the other windows; that would be a gross violation of modularity.

Enter the document object. The document knows about the underlying data and how to save it, revert it, and so on. Each window knows how to find its document. When a window gets a Save command, it passes that command through to its document. Viewed in this light, a document is nothing more than an abstraction, the common actions of its windows. This is a restrictive view, however, since the whole idea of the File menu is to set up a concept in the user's mind, a metaphorical piece of paper containing information. Windows can show that information in different ways, but the information itself comes from the document. In other words, a document is more than an abstraction: it is something meaningful to the user.

What restrictions are there on the metaphor of a document? The most obvious is the one-to-many relationship with windows. A document may be associated with any number of windows, but in general a single window can have only one document. If this were not the case—for example, if a window had two or more documents—the user could not understand the context of the program by looking at the topmost window and would wonder which document for this window she was saving or reverting. The second restriction is that documents usually represent partitions of the data available to the user. A partition of a set, in mathematics, is a division into smaller subsets in such a way that every element of the set is assigned to precisely one subset. In other words, the subsets do not overlap but together cover the entire set. Applying this to documents, we say that no two documents are allowed to share data. The reason is subtle and is related to the Revert and Save commands. Suppose you have two windows that share some piece of data, as shown in Figure 9-8. Assume that everything in both windows is editable.

Make a change in the topmost window to shared data and another change to non-shared data. Now switch to the other window and make a change in the shared data—a change that overrides at least part of what you did in the first window—and also make a change to non-shared data. Now switch back to the first window and choose Revert. What happens? The only consistent interpretation is to undo all changes made in both windows, both in overlapping and non-overlapping data. But this implies that the Revert command is a property of the two windows combined; in other words, since they share data, they must both use the same document. If we use separate documents for the two windows, the documents must share data, meaning that the Revert command cannot be applied to just the document of the topmost window. The same line of reasoning applies to the Save command. This problem arises whenever windows are allowed to share data and overlapping updates are permitted.

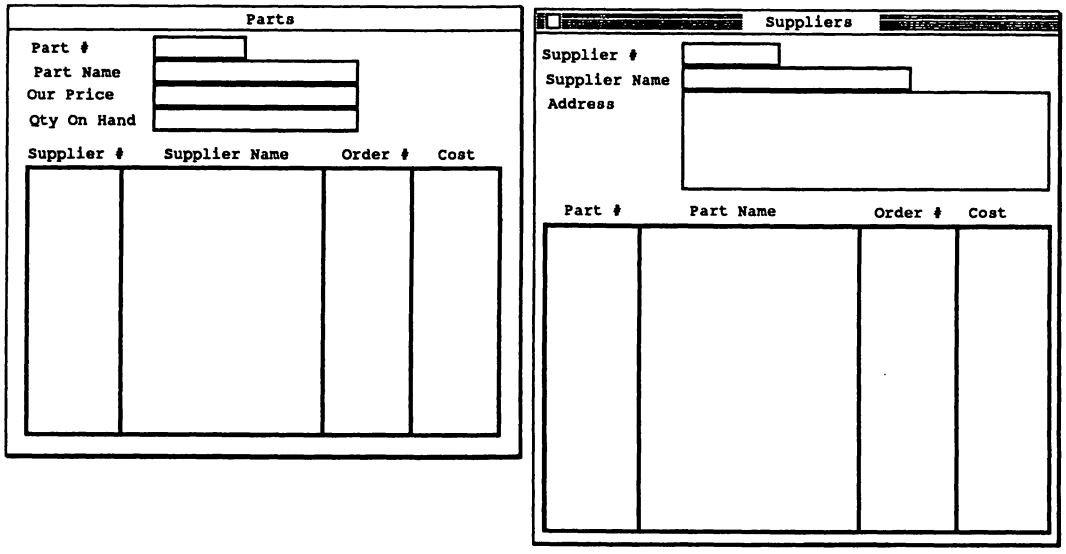


Figure 9-8. Windows that share data

This is a severe restriction when designing programs that use databases, particularly when ad hoc queries are allowed. There is no way of predicting in advance when queries will overlap. The only practical solutions in such cases are to use a single document for the entire database, which trivializes the role of Save and so forth, or not allow Revert, Save, and possibly other commands of the File menu. A notable example of the latter approach is Apple's HyperCard, which does not allow Revert in the interest of handling data shared across windows and which automatically saves changes as they are made.

It is important to draw a distinction here between the document paradigm in the user interface and the content objects in the Content Model. Documents first and foremost are user interface objects that determine the context of certain commands, such as Revert. They do not contain data, but they can cause change to occur in the data held in the Content Model. A document object does not determine the *actual* organization of the data inside the program; it determines the *apparent* organization of data as perceived by the user. For example, a draw or paint program could be rewritten to store its data in a DB2 database residing halfway across the continent in a mainframe and a user would be none the wiser (other than possible performance problems). The User Interface Model would not change in switching from simple files to the host database architecture, even though the Content Model might need a big expansion. It is perfectly

possible to design document objects, then behind the scenes flip between files, local databases, and even remote sources of data in the Content Model to provide the actual data. To the user, there isn't any difference between these sources, since the document provides all the context that is needed.

As with command objects, the document, though not necessarily visualized, is an important concept to the user. It explains why the commands in the File menu behave the way they do. Documents should not be designed based on the characteristics of the data and specifically not the way the data is stored, but on the user's *cognitive categories* of data.

Common Responsibilities in the User Interface

Although the exact set of responsibilities varies with the program, the user interface is constrained in ways that force certain kinds of responsibilities on most elements: responding to mouse clicks, keypresses, and commands to draw in a window. Class libraries commonly provide abstract classes to provide a common interface to such responsibilities.

Relationships

Relationships may be any of the structural or behavioral kinds discussed in Chapter 6.

1. *Membership* relationships apply in the usual way: We form categories as a shorthand way to describe features of the interface.
2. *Part/Whole* relationships are very common. Perhaps the best example is a cluster of radio buttons. The cluster is the whole, each button a part.
3. *Containment* relationships are also common. Menus contain menu items; dialog boxes contain buttons, editable text, static text, and so on; palettes contain icons, each of which represents a command or state; windows contain visible things, perhaps within a scrollable view. Many containers in a user interface are, in fact, implementations of cognitive categories, since the items contained may have few or no common properties.
4. *Collaboration* relationships exist throughout the user interface. For example, when a button is pushed, it may need to collaborate with other objects to dim or highlight other parts of the interface. It may also collaborate with one or more content objects to change the state of the program.
5. *Creation* and *destruction* relationships occur whenever windows are opened or closed and at many other times in a user interface.

Calibration relationships also exist between objects of the User Interface Model and specific features of the visible user interface, responsibilities of the computer in the Solution Model, and elements of the User Interface Architecture in the Execution Plane.

▶ Building the User Interface Model

The User Interface Model is intricately woven in with the Solution Model and Content Model and as one expands, so must the other two. However, there is a natural sequence to the development of the User Interface Model during analysis.

1. Start with the Business Plane as already described.
2. Push central aspects of the Solution Model to the Content Model and iterate until the central aspects are stable and well covered.
3. Create user interface snapshots for central responsibilities of the computer in the Solution Model. These suggest additional responsibilities for the computer which must be calibrated with the Content and Solution Models. Iterate until the central responsibilities stabilize.
4. Through scenarios, define the objects—windows, controls, and so forth—in the snapshots. Identify their responsibilities and relationships. Form scenarios that illustrate how a user carries out his or her responsibilities, as shown in the Solution Model, using the objects in the User Interface Model. Iterate until stable.
5. Expand the User Interface, Content, and Solution Models toward the periphery. There is no order to the three models at this point. Work tends to jump around among the three models. Use calibration to keep the three models in sync. Keep track of all dangling threads, tying them up whenever it is convenient to do so.

The snapshots you use in the User Interface Model may be simple mock-ups done using a paint or draw program, or they may be more sophisticated prototypes. We will talk more about prototypes in a few moments. Remember, though, that we shouldn't "complete" the User

Interface Model before descending to the User Interface Architecture and User Interface Implementation. Just as we iterate between the Solution Model and User Interface Model, so do we iterate between the User Interface Model and the lower planes.

▶ The Environment Model

The Environment Model rounds out the Technology Plane. There are certain objects that do not naturally fall into either the User Interface or Content Model. Consider, for example, objects needed to communicate with an external database. These are not really content, since they themselves hold no information. Neither are they user interface, since they are invisible to the user. However, they are not "mere implementation details," since the use of a given database may be a major feature of the end product. It is appropriate, at times essential, to include them as part of the analysis phase. Figure 9-9 shows the relationship between the Solution Model and the Environment Model.

As you can see, we focus on interactions between the program and other objects in the environment in which it is used.

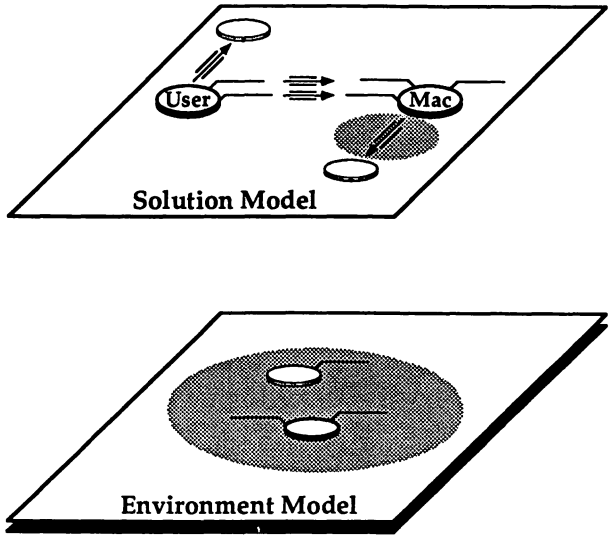


Figure 9-9. Building the Environment Model

▶ Elements and Relationships

The objects in the Environment Model hold no data of their own, are not directly visible through the user interface, and communicate with one or more external entities. These objects are not the external entities, but encapsulations of them. Let's assume that your program needs to communicate with a remote host computer over a network. You might create a single object or network of objects in your program that presents the entire interface to that host to the rest of the program. Examples of entities for which we might create objects in the Environment Model are databases, networks, and specialized devices that either communicate with or are controlled by the computer. Such objects are not properly part of the Content Model, since their purpose is to communicate, not hold knowledge. They are not part of the User Interface Model, since they are not directly visible to the user. They are to the external objects what the User Interface Model objects are to the user.

As with the User Interface Model, we may have temporal (command) or automation (document) objects to consider in the Environment Model.

▶ Building the Environment Model

The objects in the Environment Model vary greatly, depending on the complexity of the interactions between the program and its environment. In many cases, an external device is completely encapsulated by a single object of the Environment Model. In other cases, more complex models may apply. In all cases, however, we can use the Solution Model as a starting point. If a specialized device is attached to the computer, it should exist as an object in the Solution Model with the essential responsibilities and collaborations laid out. These can then be pushed down to the Environment Model as the central starting points. Of course, the usual CPC process also applies: direct examination of the Environment Model causes expansions, which are correlated to the Solution Model and the other regions of the Technology Plane. As those other regions are expanded, synchronization with the Environment Model takes place.

▶ The Execution and Program Planes During Analysis

Most activity during the analysis phase of a project is on the Business and Technology Planes. However, there are times when it is appropriate to proceed down to the Execution or even the Program Plane, even though the project is still formally in an analysis phase. There are at least four basic

reasons for doing so: to develop prototypes, to aid in estimation or scheduling, to refine the concepts in the Technology Plane to take into account the class library to be used, and to keep people busy while other reviews or approvals are pending. The first three reasons are under the broad umbrella of "prototyping" in Solution-Based Modeling.

▶ Prototyping

A prototype can be anything from sketches on index cards spread over a table to working software that has a good chunk of the functionality of the finished product. In fact, we have already used prototyping without identifying it as such in the example of the screen snapshots included in the User Interface Model scenarios. Before discussing how the different kinds of prototypes can be used in Solution-Based Modeling, we should first set some objectives we hope our prototypes can help us achieve.

Objectives of Prototyping

Prototyping can serve several purposes, from simply communicating progress to demonstrating that certain problems do, in fact, have solutions in actual code. It is important to keep in mind that prototyping can at various times serve the objectives of different people: engineering, management, users, systems analysts, and so on. Although other purposes can no doubt be served by prototypes, the following objectives are found in most Macintosh software projects in varying degrees.

Heisenberg Prototyping. In Chapter 1, we talked about the Heisenberg Uncertainty Principle of physics, which states that in order to observe something you must somehow affect it. A good prototype should first and foremost stimulate our perceptions—that is, it should add to our observations—but in the process we should expect earlier assumptions to be invalidated or refined. Prototypes are not really additive. You will not start with a stable body of knowledge and expand it by developing a prototype. Instead, as you prototype you constantly fine-tune and occasionally overhaul the work that has gone before. The authors call this Heisenberg Prototyping—deliberately jiggling the problem and your assumptions about it in the hope that anything that is not fastened securely comes loose. When choosing topics for this kind of prototyping, you should seek out controversy, not shy away from it. It is precisely in the less well-understood parts of the model that prototyping yields the greatest insights. In fact, Heisenberg prototyping can be a very effective tool in smashing through the conceptual brick walls that frequently arise

in a software project. If a problem seems sticky in the Technology Plane, choose some central aspect of the problem and work on a prototype of that center in the Execution or Program Planes. If that still bogs down, deal with the center of the center and so forth until something jiggles loose. This is technological brainstorming, so dare to be creative and occasionally outrageous, as long as you remember that backtracking to more solid ground is also part of the process.

Validating Abstractions. When you are working on the Content Model, things often seem a little vague. A second role for prototyping is to add crispness to the concepts of the Technology Plane in general and the Content Model in particular. In the Content Model, we form abstractions such as, "These objects are all examples of reports, and all reports print themselves." Is this really true? Do we mean the same thing when we say "print a paycheck" that we mean when we say "print an hours worked report?" These questions are best answered in the Execution Plane which, as we will see in the next chapter, can add a great deal of clarity to the responsibilities and relationships of objects. This use of prototyping can be thought of as validating the abstractions and definitions of the higher planes of the SBM.

Estimation, Scheduling, and Engineering Feasibility. A third role of prototyping is to aid in project management by reducing the risk involved in project scheduling. In every software project, certain parts of the project cause sleepless nights for both engineering and general management. These are either the cornerstone algorithms or structures without which the project will fail or performance constraints on key aspects of the application. It may be unknown whether they are even possible, but more likely it simply isn't known how long it takes to implement them or what resources are necessary. These are central issues to the design and implementation, even if they aren't central to the user's perception of the program. Prototyping can be a great help in developing estimates and schedules or in establishing the engineering feasibility of the solution.

Minimizing Costs. In the Technology Plane, there are often many different approaches to a problem, particularly in the user interface. If one approach clearly stands out as best, use it. If it is unclear which approach is best, considerations of which approach best utilitizes the class library often tip the balance. For example, in MacApp, one can nest views within a window to an arbitrary level, but the standard print handling classes cannot handle subviews. A decision in the User Interface Model to print exactly what is on the screen at any time can prove prohibitively expensive

since the standard print handling classes give no support for nested views. It may be more economical and almost as effective to create a separate print image for each report, rather than attempt to unify the print image with the editable, displayed image. It is difficult to identify these kinds of tradeoffs without resorting to prototypes in the Execution Plane, where user interface objects are mapped onto the classes of the class library. This is true in inverse relation to your experience with the class library: The less you know about the class library, the more valuable this objective becomes.

Demonstration and Confidence Building. Finally, prototypes can be used to demonstrate progress and build confidence in the process being followed. These prototypes have entirely different objectives and audiences than the ones we have talked about so far. They should shy from controversy, not stimulate it. They should target central features as perceived by the target audience, whether end users, management, product marketing, or members of the target market, not central features as perceived by the project team. Demonstration prototypes are important for their own reasons, but they should not be confused with prototypes whose purpose is to further develop the model. They communicate work in progress, but are not necessarily part of that work.

Kinds of Prototypes

Now let's look at the different approaches you can take to implement prototypes and relate them to the objectives just discussed.

Storyboards. The simplest form of prototype is the storyboard. This can be constructed by assembling a series of screen shots associated with the scenarios of the User Interface Model. They can be prepared using pencil and paper, paint programs, HyperCard, or white boards and markers. Storyboards need not be electronically linked through on-screen buttons; simply posting them on a large wall or spreading them over a table is quite adequate. Storyboards are especially good for demonstration and Heisenberg prototypes. They can be put together in a hurry, which allows them to be up-to-the-minute. They can be changed equally quickly, especially if you use some sort of cut-and-paste technique for drafting them. They don't take a huge chunk out of your budget. Storyboards are also useful in clarifying the User Interface Model; in fact, it is difficult to proceed without them. Storyboards are not as useful for achieving the other objectives of prototyping.

Simulations. Simulation prototypes show the screen snapshots and provide them on-line with simulation of much of the functionality of the finished program. Providing on-line simulation can sometimes be a help, but you must seriously question the time spent on that linkage compared to simple storyboards. The ability to see lots of snapshots spread over a table at once is a distinct advantage of the manual methods. Simulations are typically constructed using HyperCard, prototyping tools like AppMaker or Prototyper, or by writing code that implements the prototype. All but the last of these—writing code—are extensions of the storyboard approach. They do not yield much information about feasibility, cost, or other information you typically want from a prototype. Creating actual code is a good way to establish engineering feasibility, develop estimates, and explore the relationship between your program's interface and content and the features of your class library.

Scenarios. Simulation prototypes are not very effective at validating abstractions because they require too much work in return for the information they provide. Simply forming scenarios in the Execution Plane is far more valuable and productive in adding crispness. This, in fact, is the most effective form of "prototyping" for achieving many of the objectives we have laid out: validating abstractions of the Technology Plane, estimating, demonstrating engineering feasibility, and minimizing costs by making the most effective use of the class library. The scenarios of the Execution Plane are close to the level of code, but without classes and inheritance. We can achieve much of the benefit of actual code without the extra baggage of compilation and linking, user interface, and the other hassles which accompany programming. It is also an easy way to proceed deeper into the design without the need for coding a user interface.

▶ Advance Scouting

In addition to prototyping, the Execution and Program Planes can be explored during the analysis phase simply to allow more parallel activity to take place. It is common for the User Interface Model to require much more review and with more people than the Content Model, resulting in idle time for some members of the team. In such cases, it is often productive to allow the team to proceed on to the Content Architecture or other parts of the Execution Plane while the rest of the Technology Plane fills out. This is actually design work, but remember that we have defined our analysis phase separately from analysis activities. Until we reach the milestones for completion of the analysis phase, all work that takes place is part of this phase, regardless of on which plane and for what purpose the work takes

place. This parallel activity can only strengthen the results of the analysis, at the risk of some wasted effort. For this reason, it is important when doing advance scouting to choose stable topics and stay within the scope already established during analysis.

▶ Completing the Analysis Phase

We have now covered all of the activities of the analysis phase. We now need to talk only about the transition from analysis to design.

▶ How Do You Know When You Are Done?

The objectives of the analysis phase are to establish scope and to develop schedules and estimates for the remaining work. Since the Business and Technology Planes together define the scope as it is understood at any one point in time, we should complete these planes before moving on. However, as we know, these planes are never really complete. The best we can hope to accomplish is to have someone in authority certify in writing to the following.

1. The Business and Technology Planes, while subject to refinement as work proceeds, represent the intended scope of the project. We know of no expansions of these planes required to make the product complete; all expansions from this point through delivery will be the result of calibrations from the lower planes.
2. The project schedule, resource allocations, and estimates are credible and backed by sufficient detail.
3. The Solution Model is fully correlated to the Reference Model and Technology Plane.

This person relies on the opinions of others and his or her own impressions of the credibility of the project team. Once again, one reaps the benefits here of a team approach to the development. Solution-Based Modeling is designed to keep everyone in touch throughout the process. VDL, the use of scenarios, performing analysis in terms of cognitive concepts, rather than computer technology—all are intended to maintain communications throughout the analysis phase. By the end, there should be few questions left to answer in deciding whether the phase is over, since management has been kept in the loop. (If you haven't done this, go back and reread Chapter 7!)

There is a necessary, mechanical condition that must be met before one can declare analysis complete. Dangling threads in the Business and Technology Planes *must* be tied up, with the exception of threads that can be resolved only in the Execution Plane. In other words, the planes must be self-consistent and, within the scope they cover, complete. Note that "complete" here is a relative term, meaning only that dangling threads have been tied up. Absolute completeness can only be determined by someone exercising individual judgment.

▶ Estimating, Scheduling, and Planning

Part of wrapping up the analysis phase is planning what happens next. In fact, the development of a sound project plan is a major objective of the analysis phase. The authors hope that you haven't skipped straight to this section expecting to find a magic wand you can wave to produce good estimates. This is still an area that requires judgment and experience, both with software development in general and with the unique circumstances of your organization. In other words, estimating is still more art than science. That said, we can provide a few pointers on how to proceed and structure the estimates.

Each element of the Technology Plane—objects, categories, responsibilities—and all relationships must be designed and implemented. In fact, these constitute the vast majority of the effort through completion. This provides a ready-made basis for estimation. However, this is an overwhelming amount of detail, certainly too much to build into a schedule. (It is, however a checklist for *completion*. It is just too fine a grain of detail for use in estimating.) It is better to base estimates for design and programming on scenarios: how long will it take with what resources to implement this and this and this scenario? The key here is to choose scenarios that together cover the entirety of the Technology Plane. This means not just scenarios that define all the elements and relationships, but time-sequence scenarios that show their dynamic interactions as well. It has been the authors' experience that the time per scenario is relatively constant across a project. In the early going, scenarios take longer due to the foundation work taking place; later scenarios can start from an established base. However, since calibration increases as the scenarios pile up, the net result is a relatively steady pace. The average time per scenario depends on the nature of the application and the organization and team doing the work.

Schedules should also be based on completion of scenarios. Completion of a single object means nothing; only when it is successfully placed in all

relevant contexts can one call an object "complete." Dependencies of one task on another can be traced to shared elements of the scenarios.

▶ Summary

The Technology Plane has three regions: the Content, User Interface, and Environment Models. The Content Model holds the data content of the program and represents the capabilities of the program shorn of its user interface. The User Interface Model contains the objects that make up the visible part of the program. The Environment Model contains objects that encapsulate external programs or devices, such as host computers, networks, and specialized equipment attached to the computer.

- The Content Model contains conceptual objects that do not exist in the real world. We attempt to create metaphors for real-world objects to make the analysis and design easy to grasp. Creative construction and review of object candidate lists can help us identify these conceptual objects.

  Responsibilities are assigned in the Content Model according to four principles: limit responsibilities, limit data knowledge, limit implementation knowledge, and limit relationships. A fifth consideration, limiting type knowledge, applies to the Program Plane. Two general strategies help achieve these objectives: the black box approach and client/server architectures. The inevitable dynamic tension among these five objectives requires judgment and experience to resolve.

  Correlation takes place across planes of the SBM ensuring that each element is implemented or replaced by elements in the planes below it.
- The User Interface is framed by the Solution Model. It is also constrained by the other regions of the Technology Plane, the Macintosh platform, and underlying technology, the class library chosen, human factors, and company policies and procedures. Most elements of the User Interface Model are manufactured objects: They really exist, but as artifacts of the computer. There are also objects that are added to represent concepts important to the user such as documents and commands.
- The Environment Model encapsulates in objects all interactions between the Content Model and the external, but non-user interface, environment.

  While in the analysis phase of a project, one might work in the Execution or Program Planes for two basic reasons: prototyping or making progress while awaiting reviews or approvals. Prototyping

can serve several different objectives addressing different audiences: enhancing observations of the problem or solution, assisting estimation or scheduling, establishing engineering feasibility, or allowing demonstrations.

The analysis phase ends when someone in authority says it does. Ideally, it should not end until no further expansions of the Business and Technology Planes are needed to establish scope and the schedules, estimates, and resource allocations are acceptable. To ensure this, there should be no dangling threads in the Business Plane and none in the Technology Plane except those which can only be resolved in the Execution Plane.

10 ▶ Design

▶ What This Chapter Is About

This chapter begins where Chapter 9 ended: at the conclusion of the analysis phase. Now we turn our attention to design and the construction of the Execution Plane.

This chapter discusses a number of concepts and skills used for the first time in this plane. The Execution Plane contains run-time objects that do not rely on inheritance or polymorphism. The distinction between run-time objects and their language-based implementation can be difficult to grasp, so we'll spend some time clarifying just what is relevant in describing the Execution Plane. We will distinguish between an "abstraction," a shorthand for two or more run-time objects that share properties, and a class, which is something used to write a program. Also new to the Execution Plane is the last of the three methods of calibration, synchronization, which is used to ensure consistency within the plane. The Environment Architecture contains new objects that represent application-level concepts, including event dispatching and event handling.

The discussion of the User Interface Architecture leads naturally to more general topics of object-oriented software architecture. The user interface is broken down into three types of objects: renderings, display containers, and managers. Also addressed is the general subject of dependency management, which holds that one object may need to be notified whenever another changes its state.

Following these topics, each of the four regions, content, user interface, and environment, is discussed separately. This leads into a general discussion of a number of important architectural concepts in object-oriented

systems. The chapter concludes with a discussion of project management issues in the design phase.

▶ Overview

The analysis phase centers on defining a business solution and a conceptual design that implements the Macintosh portion of that solution. These are captured in the Business and Technology Planes. During the design phase, you express those concepts in software by describing the objects that will exist in the running program, stripped of classes, inheritance, polymorphism, and other object-oriented language tricks. The Execution Plane differs from the Technology Plane in four ways:

- It uses program objects instead of conceptual objects.
- It is more detailed.
- It divides objects according to principles of software architecture.
- It uses very rigorous validation procedures, especially the third and final form of calibration, synchronization.

The design phase begins where the analysis phase leaves off and ends when the Execution Plane covers the entire scope of the Technology Plane and is fully validated. Where a great deal of judgment was required to find the end of the analysis phase, there is a much more precise set of tests to detect the end of the design phase.

▶ Using CPC During Design

The basic processes introduced in the previous chapters apply equally to the design phase.

- Use CPC to determine the order in which the plane is constructed.
- Build and synthesize scenarios to gather and assimilate new information.
- Correlate to make sure that all work is consistent across planes.

In addition, you will now use synchronization as a way of performing a very meticulous audit of the design within the Execution Plane.

New information in the design phase consists mostly of expansions and refinements of the Technology Plane, rather than the new concepts and original solutions to problems that are developed during the analysis phase. You will undoubtedly have already done some work on the Execution Plane as part of the analysis phase. This usually provides a good

starting point for expansion. In addition, drill down central topics of the Technology Plane, then expand them within the Execution Plane, correlating to make sure the two planes continue to describe the same conceptual solution.

▶ Program Objects vs. Conceptual Objects

In the Technology Plane, you formed conceptual objects to describe design ideas. In the Execution Plane, these objects must conform to three rigorous requirements of object-oriented programming: (1) objects are completely described by their responsibilities and attributes, (2) responsibilities have precisely defined formal typed call parameters and return values, and (3) attributes have specific data types. In the Execution Plane, you will impose these restrictions as the first step toward code. For this reason, it is time to switch from the conceptual model and notation to the programmatic model and notation, as shown in Figure 10-1.

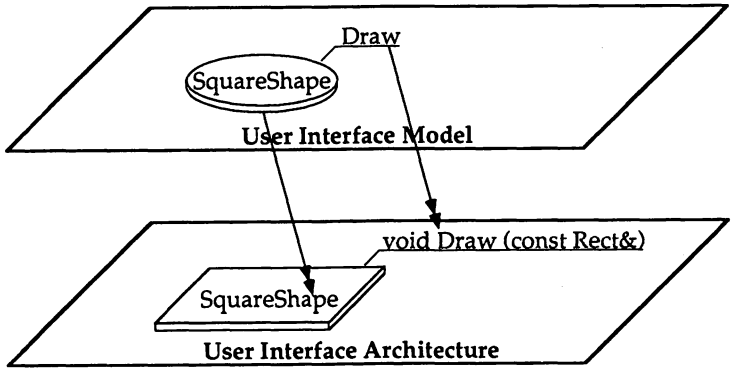


Figure 10-1. Program objects vs. conceptual objects

▶ Adding Detail

The Execution Plane contains a much finer level of detail than the Technology Plane. There are several specific sources of this added detail.

- As noted, in the Execution Plane, you completely specify all data sent to and received from a collaborating responsibility. In the Technology Plane, it is not necessary to specify arguments or return values when invoking responsibilities.

- The Technology Plane makes no attempt to connect the Content Model to the User Interface and Environment Models; the Execution Plane connects objects across regions as needed.
- In the Technology Plane, objects are generally assumed to exist when needed. In the Execution Plane, the creation and destruction of every object must be precisely specified and coordinated with all collaborations of the object. This and other detailed issues are audited by the process of synchronization.
- In the User Interface Architecture, objects are refined and divided such that each fits neatly into one of three general categories: *renderings, display containers,* and *managers.* These can be very roughly described as "things that draw," "places for things to draw," and "things that decide what to do when the user does something," respectively. In the Technology Plane, these concepts are frequently jumbled together in the interest of a simple conceptual model of the program.

▶ Adding New Objects

Certain kinds of objects make their first appearance in the Execution Plane.

- Application class libraries—The user interface objects of the run-time program must be mapped onto classes of the application class library (for example, MacApp). You will not decide at this time how and when to inherit from library classes, but you will identify which features of the library will be used to implement the design. This is also done for the other regions of the plane, but the task is dominated by the User Interface Architecture.
- Automation and auxiliary objects—The Environment Architecture is augmented with automation and auxiliary objects to account for the application itself, event handling, and interactions with the Macintosh platform, Toolbox, and operating system.

▶ Run-Time Objects

One of the most important principles underlying the Execution Plane is that the designer must deal with two very different object-oriented environments: run-time and compile-time. In Solution-Based Modeling, these environments correspond to the Execution Plane and Program Plane, respectively. It is central to Solution-Based Modeling that the two concepts be kept separate, so it is appropriate to pause and discuss the differences in some detail.

▶ How Are Objects Implemented?

To demonstrate what run-time objects are and how they differ from compile-time objects, we will make up a language called "C−", which uses a brute-force implementation of run-time objects from compile-time classes. Once we have introduced C−, we will return to discuss a couple of real languages. This is not a technical treatise on how to write object-oriented programming languages, just an attempt to drive home the important differences between a run-time object and a compile-time object class.

Objects in "C−"

C− uses the syntax of C++, but isn't quite as smart in the way it implements the objects. Nevertheless, it provides a perfect contrast between run-time instances and compile-time objects. Start with the following class definition.

```
class foo {
 private:
  int a;
 protected:
  void do_something (void); // A method that does something useful
  public:
  // Does something else useful
  virtual void do_something_else (void);
  };
```

This is a signal to the compiler to create a structure something like this.

```
struct foo_class {
  void (*f)()[];  // A pointer to an array of method addresses
  int a;
  };
```

In other words, the class gets translated into a rather conventional data structure, the first field of which is a pointer to an array of method addresses. Let's call this array the "mtable." Following the mtable are fields corresponding to the various data members. When the running program needs to create an object of class `foo` it allocates space in memory for a `struct foo_class`, initializes the mtable to point to the global array of methods for that class, then calls the constructor method to initialize the data members and take any other action you have defined. Put another

way, each object is simply a data structure, part of which points to its methods. Now let's throw in a subclass and see what happens.

```
class bar : public foo {
  private:
    long b;
  public:
     //Overrides superclass version
     virtual void do_something_else (void);
     //A new method for this subclass
     void do_nothing_and_pretend (void);
  };
```

Now the compiler must produce a structure that includes the features of both the subclass and its superclass.

```
struct bar_class {
  void (*f)()[]; // The mtable - methods of the superclass "foo"
           // followed by methods of "bar"
  int a;   // From the class "foo"
  long b;  // From the class "bar"
  };
```

The data members were combined to produce a new, big structure. Because we were clever about the order in which things were defined in the structure, the first two entries look just like a `struct foo_class`. This is how one can address a `bar` as if it were its superclass `foo` and still get the right results—the beginning of a `bar` is in fact a `foo`. We handle the mtable in a similar way: The beginning of the table contains methods of `foo`, except that wherever `bar` overrides one of those methods—in this case, `do_something_else`—we substitute the address of the subclass version in the same slot of the array. This is followed by methods of `bar` that are not overrides of inherited methods, such as `do_nothing_and_pretend`. If someone, thinking this is really a `foo` when it is, in fact, a `bar`, should call an overridden method, everything works fine, because the address of the overridden version is at the same offset in the mtable as the superclass version in the mtable of the superclass. The implementation of that slot in the mtable is different, but the interface, calling sequence, and offset into the mtable are all the same, regardless of which subclass has grabbed control of that slot.

The next step is to change the calling sequence to methods. If the program contains the call `x->do_something (list_of_args)`, we

follow a two-step process: (1) look up the correct address of the implementation in the mtable of `x`, then (2) translate the call into the general form `class_of_x_do_something (x, list_of_args)`, where `class_of_x_do_something` is the implementation of `do_something` pointed to by the mtable of `x`. In other words, pass the address of the target object (`x`) as an implied parameter to the method. This allows the method to access both the fields of the structure and its other methods.

Objects In C++ and Object Pascal

C++ actually uses a scheme fairly similar to C−, but the C++ compiler is smart enough to optimize special cases. C++ has a vtable where we had an mtable. If a method is not virtual, it need not go into the vtable at all. It is simply assigned a global name and called as if it were a global function with the object's address as the first argument. In fact, C++ won't even create a vtable if there are no virtual methods for a given class. Also, the C++ compiler will not use friendly names in the generated code. In C−, the method name was a combination of the class name and the method name (for example, `class_of_x_do_something()`). In C++, the real name is a mishmash of the method name, the class name, and the data types of its arguments, all encoded in a way that would do the CIA proud. Despite these differences, the essential character of C− remains: to call a virtual method, look up the method in a table for the class.

Object Pascal achieves the same results, but the order of method lookup is the opposite of C++. Rather than put a vtable into each object, it puts a 16-bit integer class identifier at the beginning of each object record. Instead of a table for each class, there is a table for each overridden method. In each table is a set of (class ID, method address) pairs. When OP has to find the right virtual method to use, it looks in this table of pairs, searching within that entry for the class identifier matching that of the object. This is the opposite of the strategy used by C++, but the effect is the same. Each run-time object has a specific set of methods associated with it. The only differences are related to optimizations provided by the compilers.

Figure 10-2 shows how one may visualize method lookup as a two-dimensional array, with classes for rows and method names for columns. In C++, you first find the row, then index to the correct column. In Object Pascal, you first find the column, then search for the right row.

In both cases, the compiler takes care to hide the messy details of the lookup table from the programmer.

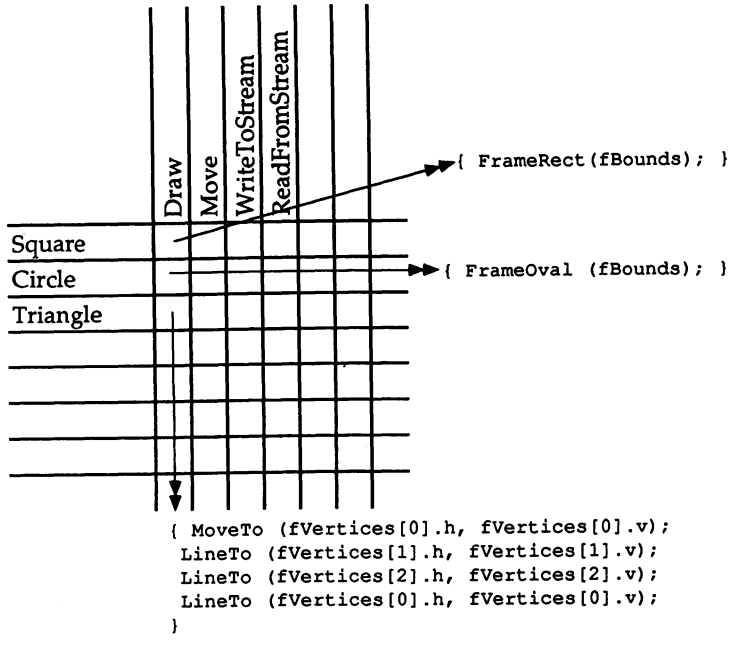


Figure 10-2. The two-dimensional structure of method dispatching

Where Did the Inheritance Go?

Look at `struct bar_class` again. Do you see inheritance there? No, because there isn't any. *That was all taken into account when the mtable was set up: It was frozen at compile time.* The compiler chose a very specific set of attributes and methods to assign to instances of this class at run time. In fact, it is impossible to look solely at the definition of `struct bar_class` and deduce anything at all about its ancestors! You cannot tell what superclasses it has—in fact, you can't even tell from what *concrete* class it came! You also cannot tell from the mtable which methods were overridden and where. Inheritance and polymorphism are purely compile-time concepts in C++, as in most object-oriented languages. Object Pascal carries around a little extra baggage called *metainformation* that allows you to deduce more about an object's class and ancestry at run-time, but the structure of a class and its methods are fixed at compile-time. The metainformation only provides more information about what is already frozen at compile time. In both cases, there is a sharp distinction between the run-time object (a data structure accompanied by a method lookup table) and the compile-time class, which must be combined with its ancestors in order to make any sense of the class.

In fairness, we must point out that certain object-oriented languages actually allow classes and objects to be dynamically changed at run-time. These include Smalltalk and Macintosh Common Lisp with its CLOS class library. However, these features are not commonly used outside development environments. It is still the objects themselves that do all the work; except in unusual cases, the same principle of separating the run-time and compile-time worlds still applies. Even when these dynamic features are used, the concept of the Execution Plane remains valid, but some of the clear distinctions between the Execution and Program Planes diminish. Specifically, classes themselves become objects with responsibilities in the Execution Plane in order to account for these specialized behaviors.

Still not convinced? Try using Neon, one of the first object-oriented languages. There are no classes at all, only run-time objects. You start with an empty object and add attributes and methods one at a time. In Neon, the closest thing to instantiating a class is cloning an object to make an identical copy. This is true object-oriented programming, but completely without classes or inheritance, something we alluded to in Chapter 2.

▶ Classes vs. Abstractions

Chapter 5 documented why classes are not all that "meaningful" in complex projects, even though objects are. Classes may be useful for software architecture, but not for semantics; they simply don't correspond to the way we think. By sticking to objects, you use the most general and powerful tools for describing the running program, keeping the design accessible, and deferring most language dependencies to implementation.

Even within the realm of shared properties, a class is still a special case of a much more general way to describe sets of run-time objects using *abstractions.* In many programs, there are relatively few run-time objects and they can all be enumerated in the Execution Plane. However, if your program consists of hundreds or thousands of objects, many of which appear and disappear in response to user actions, it is not practical to describe each and every object individually in the Execution Plane. Instead, you need a way to describe sets of instances. If two objects share two attributes and three methods, we can describe "the set of objects that have these two attributes and those three methods" as an abstraction in the Execution Plane. More generally, any set of instances can be described as an abstraction that contains the shared methods and attributes of the members of the set. *That abstraction may or may not ultimately become a class in the program.* It is useful as a description of the run-time architecture, regardless of whether it becomes a class or not.

Abstractions are used to describe sets of instances in the Execution Plane for two primary reasons. The first reason is that abstractions can be allowed

to overlap (intersect) one another in arbitrary ways. Classes can overlap only according to the models of inheritance supported by the particular language you choose. For example, Object Pascal does not support multiple inheritance. Even in C++, multiple inheritance can get murky in a hurry. What if you inherit from the same superclass more than once? In abstractions of the Execution Plane, there is no need for such tomfoolery, since the abstractions are only descriptions of or assertions about the run-time objects. Multiple, overlapping abstractions provide an extremely useful way to describe a run-time architecture that you should be able to use long before worrying about the particular features of the language.

The second reason is that there are many different ways to implement an abstraction in code, should one wish to do so. Some don't even directly involve the compiler and language! We will talk about these techniques in Chapter 11. The point here is that, faced with a variety of ways of implementing a single concept, you should use the concept itself as the basis of the architecture and leave the rest to implementation and optimization.

For these reasons, you can describe a set of instances in the Execution Plane using abstractions of their attributes and responsibilities. These abstractions are created for convenience: They can overlap, form hierarchies or not, and use any subsets of the attributes and responsibilities of the instances that best describe the run-time objects.

For example, in our payroll application there are many objects that need to be printed, including paychecks, W-2 forms, and so on. It may be useful to define an abstraction for printable objects called REPORTS, which may become one or several classes in the Program Plane, or it may turn out to not be relevant to the implementation at all.

▶ Categories vs. Abstractions

You might be wondering about the relationship between a category of the Technology Plane and an abstraction in the Execution Plane. After all, categories also provided a means of describing sets of either conceptual or real-world objects. You assign responsibilities and attributes to categories as a shorthand way of saying, "All members of category C have these properties." Sounds an awful lot like an abstraction, doesn't it? Without ever explicitly saying so, we have insisted, through the process of synthesis, that our categories either have no properties or that all members share all properties assigned to the category; that is, we allowed, but did not force, categories to also be abstractions. If no shared properties apply, you can form categories without any properties. These function as conceptual groupings of objects or other categories, but little else.

Categories without properties must be modeled using containers, not inheritance and abstraction, since there is nothing to inherit or abstract. There is always a trivial level of sharing due to the simple fact of membership in the category, but this is really a degenerate use of abstraction. This might be the case among the tools in a tool palette in the user interface, for example. Beyond a trivial level (for example, containment in the palette and handling a mouse click), these are likely to share no attributes or methods. Each tool has its own agenda and probably shares little else, beyond membership in the palette, with the other tools. On the other hand, where shared properties exist and are considered helpful in describing the model, you can superimpose abstractions on top of cognitive categories. Such categories will often, but not always, turn out to be useful abstractions for the Execution Plane. There is no rule that says categories need to become abstractions in the Execution Plane or classes in the Program Plane.

▶ Building the Execution Plane

For the most part, the Execution Plane is a straightforward expansion of the Technology Plane. Before plunging into a detailed discussion of each region, let's summarize that which is common across the entire plane.

▶ All Regions

We listed the differences between the Technology Plane and Execution Plane in the overview. Now let's expand on them.

Add Calling Sequences

Full calling sequences are now used for responsibilities, including return data type and arguments. We recommend that you use the syntax of your language of choice, but pseudocode is also acceptable. Adding calling sequences frequently exposes problems with the Content Model, especially synonyms (for example, two different names for the same responsibility). Figure 10-3 shows an example of a scenario from the payroll Execution Plane concerning the category REPORTS.

As you can see, all REPORTS have been lumped into one category and abstraction, with the same collaborators. However, consider what happens when we fill in the calling sequence in the Execution Plane, as in Figure 10-4. Paychecks, previously thought to be "the same" as other members of the category REPORTS, can now be seen to require slight differences in protocol. This exposes a synonym: the responsibility "Generate Print Image" from the Technology Plane is seen to actually represent two different responsibilities, one of which applies to paychecks and the

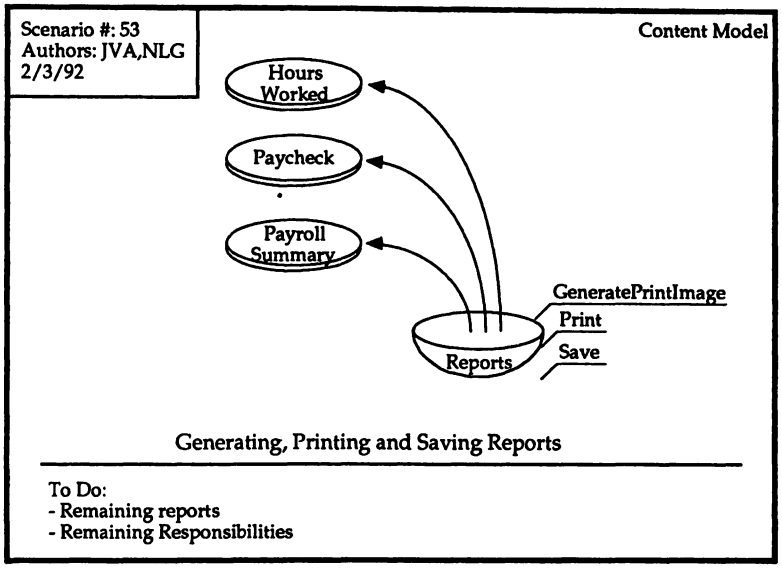


Figure 10-3. Payroll reports in the Execution Plane

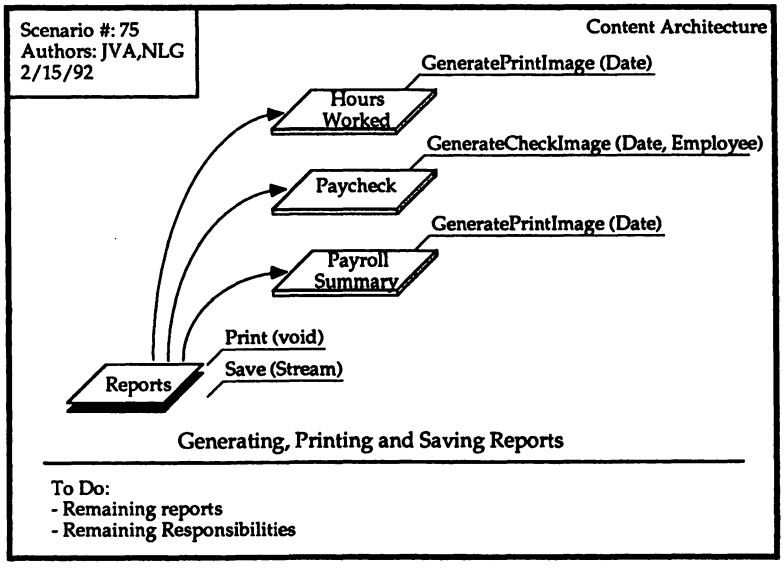


Figure 10-4. Revised payroll reports in the Execution Plane

other to all other members of the category REPORTS. We say that paychecks are a *counterexample* within the category REPORTS, since it violates the abstraction of the category. In this case, we must also change the properties of the category REPORTS or remove the erring member from that category. In Figure 10-5, the latter approach was taken to resolve the differences.

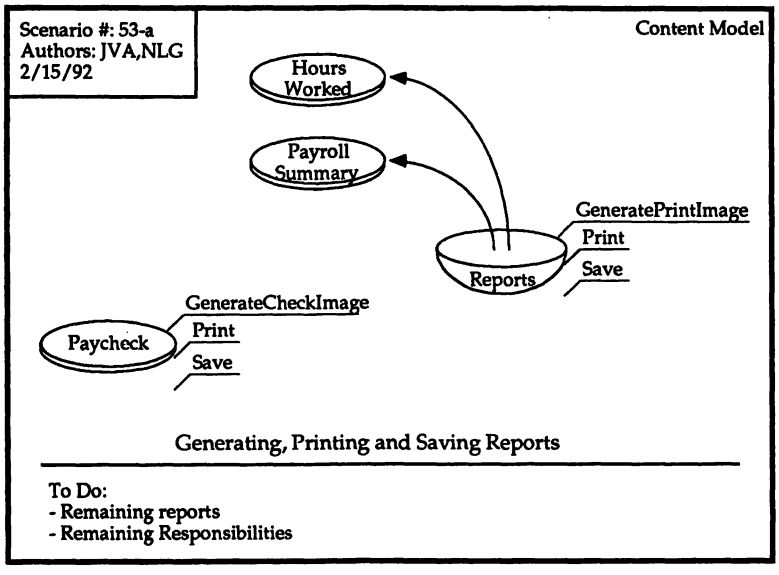


Figure 10-5. Resolving a category/member conflict

Synchronize

Another activity new to the Execution Plane is the use of synchronization. Synchronization takes the conceptual design of the Technology Plane and puts it through a thorough audit. Synchronization, for example, ensures that objects have been created before they are needed as collaborators and that objects have the addresses—or the means to get them—of their collaborators. As a result of synchronization, it is common to spend a lot of time jumping between the Technology Plane and the Execution Plane, ironing out the wrinkles exposed by synchronization. Much of this synchronization must be done across the boundaries of regions of the Execution Plane. For example, key objects of the Content Architecture may be created or destroyed by objects of the Environment or User Interface Architectures. We will describe the details of the process of synchronization later in this chapter. For now, note that it is a critical process which spans the entire plane.

Decompose Responsibilities

Responsibilities should be broken into as fine a grain of detail as possible. Figure 10-6 shows an example of this.

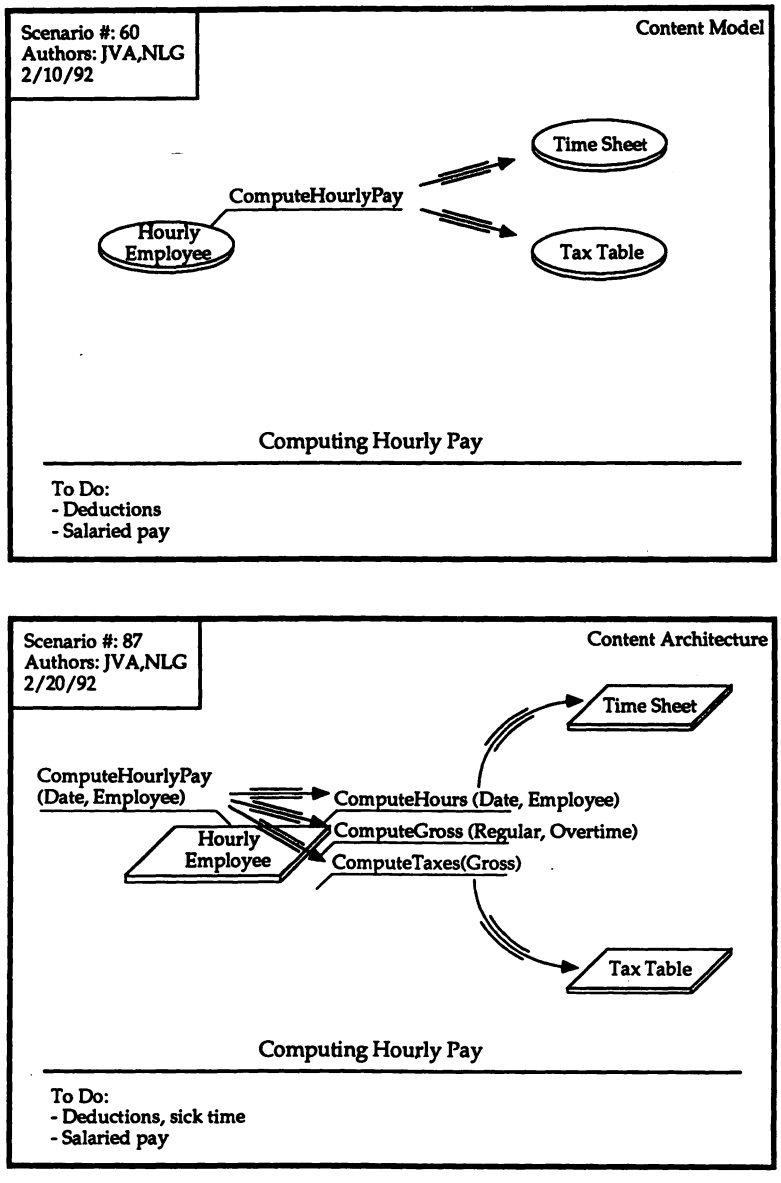


Figure 10-6. Detail in the Execution Plane

In the Technology Plane, we used a single responsibility to call several collaborators. In the Execution Plane, this is broken out into a series of responsibilities.

Use Accessors

Accessors come into play in the Execution Plane. Generally speaking, if an object stores some given attribute, provide the corresponding "Return" and "Accept" responsibilities to access it and change it, respectively. This is in addition to whatever higher-level responsibilities may have already been defined for the use of the attribute. In most cases, you should have a single responsibility for storing the attribute and a single responsibility for returning it. There are some trivial exceptions, such as the one shown in Figure 10-7.

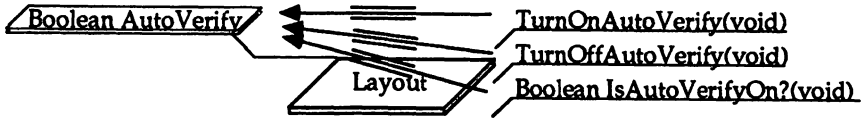


Figure 10-7. Accessors for a Boolean-valued attribute

Here there are two different "Accept" responsibilities: one to set the attribute to true and one to false, rather than a single responsibility with a Boolean argument. These arise whenever a general data type, such as Boolean, is being used to represent a more specific idea in the program.

It is important to maintain a sense of perspective on the subject of accessors. There are those who advocate that there be a single "Return" and a single "Accept" responsibility for each attribute. As with much of object-oriented software development, this is a guideline and not a commandment.

Implement Categories as Containers and Abstractions

You must also decide what to do with the categories of the Content Model. Some of these represent useful abstractions that translate in a straightforward fashion into the Content Architecture. Others, especially those with no properties, may have no role to play in the architecture. They may be valuable in describing the conceptual design of the Technology Plane, but need not have anything to do with the design or implementation of the program beyond that. Some categories can actually be seen, on close inspection, to represent groupings of run-time objects. These invariably turn into containers in the architecture. This is almost always

the case when the only properties shared by all members of the category have to do with the mechanics of membership in the category.

For example, in the model railroad application, it may be useful in the Technology Plane to create a category for the combination of track objects and certain connecting scenery objects such as bridges and switches. However, this category may not be relevant to the design or implementation of the program subsequently. Or it may be useful to create an abstraction for all of the objects that make up a single layout. This may turn out to be an important container object in the final implementation.

Map Objects onto Classes of the Class Library

Although we are still not prepared to decide exactly how to use inheritance, it is foolish not to consider the impact of the class library on the architecture. If, for example, you choose a library that does not support nested views or panes within a window, you should think long and hard about whether to design in such features. On the other hand, if you know that some sophisticated feature, such as floating windows, is available off the shelf at little or no cost and that it might result in a better product, it is foolish not to consider taking advantage of what the class library has to offer. You can indicate how the class library will be used without resorting to inheritance by describing the relationship between a run-time object and a library class as a collaboration. Figure 10-8 shows an example of this, drawn from the MacApp class TCheckBox.

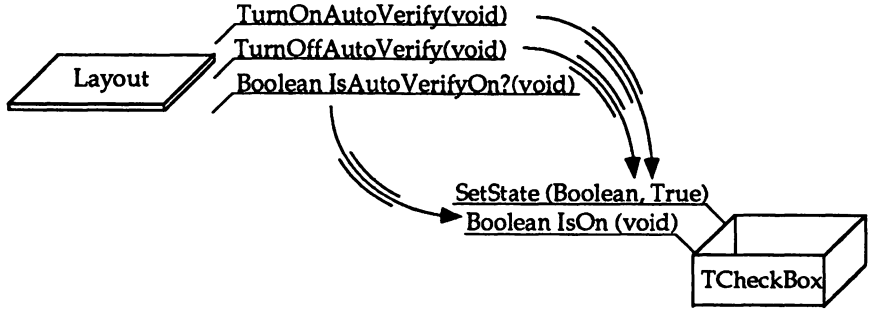


Figure 10-8. Use of TCheckBox in the architecture

Figure 10-8 shows clearly what responsibilities and attributes will be used from the library class—at least as far as this scenario is concerned. It does not yet impose a decision about how to implement this collaboration. In practice, inheritance is used most of the time, but as we will see in Chapter 11, this is not our only option and often not the best one.

The class library is most likely to have a big impact on the User Interface Architecture. Container classes such as the MacApp class TList, abstract data types (ADTs) and other similar classes may be useful in other regions, and the application-level classes that do event dispatching and interface to the Macintosh operating system and Toolbox affect the Environment Architecture. However, in most cases it is only in the User Interface Architecture where the class library has a big impact on costs and technical risk.

▶ Content Architecture

The Content Architecture is a very straightforward expansion and refinement of the Content Model. There is very little to say about it beyond restating general principles that apply to the entire plane. To build the Content Architecture, start with central topics of the Content Model and drill them down to the Content Architecture. Adding calling sequences and refining responsibilities results in correlation to the Content Model, and a cycle develops until both regions stabilize. You can then expand the Content Architecture either by drilling down more of the Content Model or by direct expansion. For example, in the payroll application, it may be necessary to drill down the REPORT object into PAYROLL REPORT, TAX REPORT, PAYCHECK, and W-2 objects before refining the responsiblities of each. On the other hand, it may be possible to directly expand the responsibilities of the CASH ACCOUNT object that is periodically credited to cover the payroll.

It is a good idea to start synchronizing the Content Architecture early. Much of synchronizing the Content Architecture depends on your having made substantial progress in the User Interface and Environment Architectures, but that is no reason not to start synchronizing within the Content Architecture as soon as enough scenarios have accumulated to make it meaningful. Also, keep in mind one of the principal design objectives for the Content Architecture: No object of the region should be aware of the existence of objects outside the region. This is the only region of the plane for which this is true, but it is one of the most important disciplines you can apply to your design toward achieving the Four Itys.

As the design phase proceeds, the Content Architecture tends to expand as the result of synchronization with the other regions of the plane. The Content Architecture absorbs some of what used to be treated as user interface or environment objects as their data content is brought to the surface. Very little goes back: that which starts in the Content Architecture tends to stay there. For example, in the payroll application, drawing from the Reference Model we may have thought early on that a form was needed to support the issuance of each check. Later, as we decide how to automate

the process and calibrate with the Reference Model, we realize that this form was an artifact of the manual system. The data it contains is relegated to the Content Architecture, where it remains through the implementation.

The final consideration in the Content Architecture is the use of off-the-shelf object classes provided by the class library. Most class libraries for the Macintosh tend to revolve around the user interface. However, there are a few types of classes that are useful in the Content Architecture. A type of class often called a *collection* is useful in implementing your container objects. A collection is a set of objects and they come in all flavors and sizes: unordered, sorted, hash tables, bags (uniqueness is not guaranteed), association lists, linked lists, queues, stacks—you name it, and someone, somewhere has implemented a generic version. In SBM, we use the term container rather than collection since our containers may be more than simple sets of things; they may have responsibilities and possibly attributes that have little to do with the objects contained. Our model railroad program, for example, treated a layout object as a container, but its responsibilities are certainly not limited to storing and retrieving its contained objects! Another generally useful category of classes is that of *abstract data types* (ADTs). This is most useful for C++, which is specifically designed to support them. Some examples of ADTs are classes for complex numbers or numbers of unlimited precision, or perhaps specialized string classes. A popular class library of ADTs has been developed by the National Institutes of Health.

As already noted, the proper way to show the use of a library class in the Execution Plane is to show a collaboration between your run-time object(s) and the class. Figure 10-9 shows an example of this in the use of a MacApp TList class as part of the design of the layout object in the model railroad example.

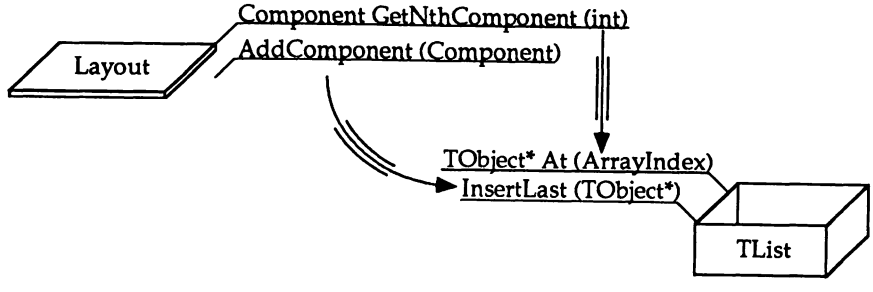


Figure 10-9. Use of TList in the model railroad program

▶ User Interface Architecture

The User Interface Architecture is developed in much the same way as the Content Architecture, by drilling down from the User Interface Model. One principal difference is the importance of mapping to the class library. In the User Interface Architecture, most classes that draw on the screen in some way use library classes. This is not imposing a class hierarchy, but simply acknowledging the dominant role the class library plays in determining the costs and risks of implementation. Figure 10-10 shows an example from the payroll program, where an editable text box on the screen collaborates with the MacApp class TEditText.

In addition, there are some special architectural issues in designing the objects of the User Interface Architecture. Specifically, you must decompose objects into highly specialized units, each of which falls into exactly one of the three types listed below.

Renderings

Renderings are the "things" of the user interface. They draw themselves, have distinct boundaries, and know about their corresponding content objects (if any). Examples of rendering objects for the model railroad program are TRACK and SCENERY, both of which are basically drawings of objects within some area of the screen. Renderings must almost always have knowledge of one or more content objects, but it is still important to limit these relationships to those that are strictly necessary. In most cases, a rendering has no knowledge of its manager(s). Renderings ideally should know nothing about their display container. When a rendering is asked to draw itself, it should be handed a "drawing environment" object (for example, one that encapsulates QuickDraw) as an argument. However, in commercial class libraries like MacApp, it is often not feasible to design this way because the user interface classes are structured otherwise.

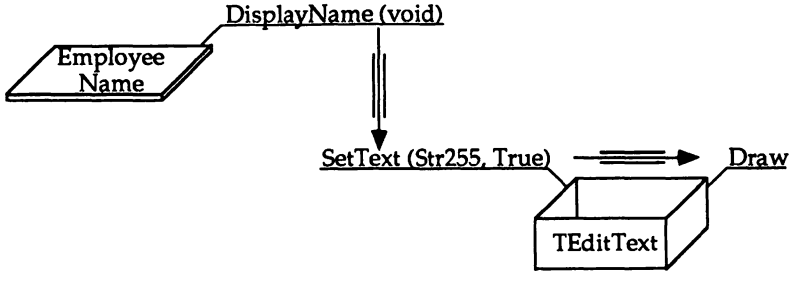


Figure 10-10. Use of TEditText in the payroll program

Renderings must ask their enclosing display container for the GraphPort in which to draw and other related information. This is unfortunate, but in the real world, class libraries seldom are structured the way you would prefer.

Display Containers

Display containers provide drawing environments for the renderings. Examples are windows and nested portions of windows. The terminology for these nested, usually bordered areas within a window gets a little confusing, since each class library seems to call it something different: "view," "pane," even "window," used recursively. To add to the confusion, class library versions of these concepts usually add functionality beyond providing a drawing environment for a set of rendering objects. We use the term display container to mean precisely that: a container of renderings that provides them a drawing environment and nothing else. Display containers may draw a border or background, but nothing else. Display containers have within their borders renderings and other, nested display containers. Other than borders and backgrounds, ultimately it is the renderings that do the actual drawing. Display containers seldom need to know about content objects, other than trivial knowledge connected with drawing borders and backgrounds. Display containers usually have no knowledge of their manager(s). A display container should generally have as little knowledge as possible of the renderings and nested display containers contained within its borders.

Managers

Managers translate events, chiefly from the user, into actions. For example, a manager object might be responsible for receiving a mouse down event, determining what the event means (for example, help click vs. drag vs. double click), determining which rendering was hit by the mouse, and carrying out changes as a result. Managers typically have a good deal of knowledge about all of the objects they control: content, display containers, renderings, and even environment objects.

Comparison to Model-View-Controller

A side note for those with some knowledge of Smalltalk. You may recognize similarities between this structure and the Smalltalk concept of "model-view-controller" (MVC). The idea in MVC is to separate objects into "model" objects that contain the pure encapsulation of data content, "view objects" that draw, and "controller" objects that act on events. Our

model is embodied in the content regions. The V part of MVC is refined through the distinction between renderings and display containers. The C in MVC corresponds to our manager objects. The reason for the different terminology is to avoid confusion with the "pure" Smalltalk usage of the term model-view-controller, since our concepts are slightly different for two of the three. The authors really don't want to introduce another acronym, but if you absolutely must have one we suggest MDRC, for "Manager-Display Container-Rendering-Content."

▶ Environment Architecture

Everything in the Environment Model is carried forward to the Environment Architecture in the same way the Content and User Interface Models are carried forward. However, there are some new objects to consider in the Environment Architecture, starting with the *application object.*

Conceptually, an object-oriented program for the Macintosh is a dormant beast that waits for someone to poke something into its cage. That "something" is an event of some sort: mouse down, keyboard, network activity, AppleEvents message, menu selection, and so on. One of the biggest challenges in designing an object-oriented application for the Macintosh is translating those events into actions. There are two components to this: *event dispatching*, which makes sure that the event is sent to the right object and *event handling*, which involves translating the event into an action or sequence of actions.

We have already discussed manager objects, which do the event handling. Event dispatching is principally the job of the application object. The mechanism used in MacApp was discussed briefly in Chapter 9: offer the event first to the "active" view of the frontmost window, then work up to its enclosing view, and so on, until it is offered in succession to the window, the window's document, and the application object. In most cases, you won't have to change this mechanism, but if your application has special needs in this area, you need to show the collaborations with the library's application class.

The application object has other responsibilities that are often customized to suit your application. When the user chooses "New" from the File menu, who handles the event? This is typically done by the application object, along with the "Open" menu item. Both of these can, on occasion, be sent to a window or document, but more commonly they apply throughout the application. The "About. . ." dialog is another example of an application-level responsibility.

The application object also interacts with the operating system in a number of ways. On startup, the application is given a list of Finder

documents to open, usually because the user double-clicked on one or more document icons. The application object must sort through this list, deciding which ones can be opened, and create documents for those that can. Many applications must be sensitive to what happens when the user switches to another application. The operating system tells an application when it is about to lose control to another application and when it has regained control. Again, the application object must make sense of these events and guarantee that the right objects are notified. Parceling out idle-time processing to objects is another application-level concept, along with processing commands. It is possible to have a single application object cover all these areas and more, as in MacApp version 2.0, or the duties of the application can be divided into multiple, specialized objects as in MacApp 3.0.

This is not a tutorial on a specific application class library (MacApp), but rather a review of the general areas one must consider in laying out the Environment Architecture. Whether a single application object takes care of all such housekeeping details or multiple objects break down the problem into bite-sized chunks, none of these responsibilities naturally falls into either the content or user interface regions, nor do they derive from the Solution Model. They deal with bookkeeping details of life in the Macintosh environment. In a sense, you can think of the Macintosh operating system and Toolbox as being much like the attached devices and networks of the Environment Model, but relevant only when one is prepared to address a lot of architectural detail.

These considerations can be easy or hard, depending on the chosen class library and the nature of the application. In most cases, you will simply use the facilities provided by the library and add a couple of hundred lines of code to fill in the blanks. You should selectively reference the library classes in your scenarios wherever they clearly affect other parts of the design. This is particularly true where the application creates or destroys your objects or initiates events such as switching in or out that otherwise aren't accounted for in the Technology Plane. However, keep in mind that your job is to document your program, not the class library itself.

▶ Dependency Management

Consider the following example. A spreadsheet program has one window containing grid-like data, another that shows a pie chart representation of that data, and a third that shows that data embedded inside a report that also contains text and data derived from elsewhere. Now change the data in the grid. The user has a right to expect that the change will be propagated

to the pie chart and report automatically. That is, all three renderings of the same underlying data are *dependent* on that data. In terms of our design principles, one or more content objects contain the data and several user interface objects depend on those content objects. Put another way, the dependent objects must receive *notification* whenever the underlying content objects change their state. In object-oriented design, this general problem of dependency and change notification is part of *dependency management*. Dependency management has been studied since the early days of Smalltalk and several object-oriented class libraries provide generic facilities in this area, including Smalltalk and MacApp version 3.0.

▶ Basic Principles

The basic idea in dependency management is to provide two facilities.

1. An object can register itself as being dependent on another object. We use the terms *dependent object* for the object that receives subsequent notifications and *notifying object* for the object on which it depends. In almost all cases, the notifying object is a content object, while the dependent object may be of any type.
2. Whenever a notifying object changes its state, it can send a change notice to all dependents or cause such a notification to be sent indirectly.

The exact mechanisms for implementing these concepts vary from one environment to the next. The brute force implementation keeps the entire implementation within the notifying and dependent objects, in ways that we will discuss in a moment. The more elegant treatment is to set up a *dependency manager*, which is a single, globally accessible object that handles all aspects of registration and notification. A dependency manager maintains a dependency graph, a data structure that encapsulates all information about who is dependent on whom.

For architectural purposes, it really doesn't matter which implementation you use. Either allows you to escape from a Hobson's choice; either violate the rule that content objects know nothing about non-content objects or adopt an ugly, inflexible architecture in which manager objects have to know a great deal about one another. In the spreadsheet example, suppose you did not use dependency management. There would only be two ways to implement the notification required.

1. Have the content objects send messages directly to the user interface objects, thereby giving them knowledge of those interface objects.

This is a gross violation of our rule for content objects that states they should have no knowledge of the objects in other parts of the architecture.

2. Have the manager that causes the content objects to update also send along notification to the managers of the other renderings. If the update comes through the grid presentation of the data, the manager of that rendering needs to know about the other renderings (pie chart and report) in order to send notification to their managers. This means the manager objects are not independent of one another and, therefore, are likely to be difficult to maintain and reuse.

Providing an abstract means of registering dependence and sending change notifications avoids both of these problems. Dependency management is such a powerful technique and so easy to implement if it is not provided for you, that the authors have trouble justifying ever *not* using it. Because of this, a special variation of the collaboration symbol from VDL is used to indicate a change notification that indirectly results from one object changing state, as shown in Figure 10-11.

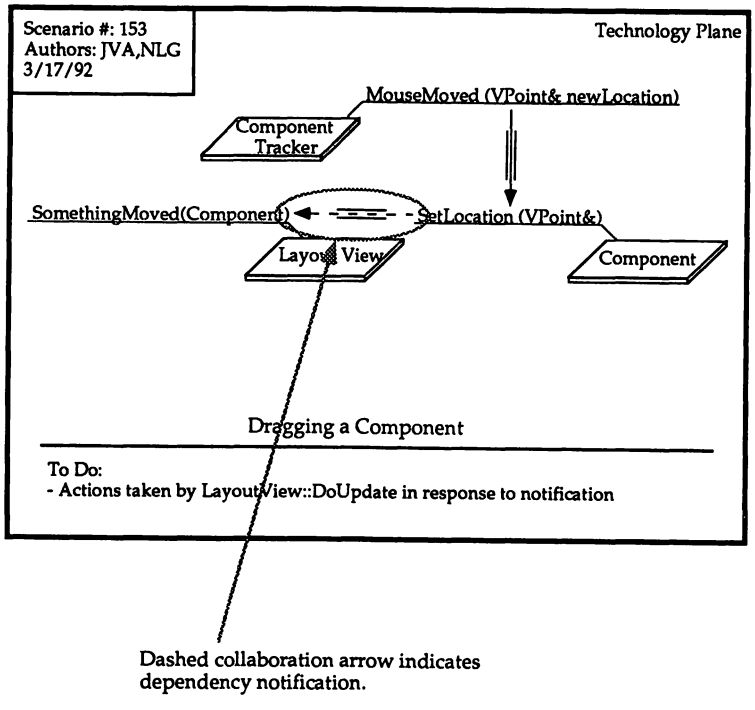


Figure 10-11. VDL convention for change notification

This shows the sequence of events when the data in the grid is updated. As you can see, the renderings indirectly receive change notification and, therefore, know to redraw themselves. This notation avoids having to constantly include the dependency manager object in scenarios.

▶ A Generic Scenario for Dependency Management

Figure 10-12 shows a generic scenario for dependency management. There are three objects pictured: the dependency manager, a dependent object and a supporting object.

As you can see, there isn't much involved. The dependency manager has three responsibilities: register the dependency of one object on another, remove a dependency, and send a change notification. (The "remove" responsibility actually takes three forms: remove a specific dependent/notifying pair, remove all pairs for a given dependent, and remove all pairs for a given supporting object. We have omitted the detail here.) The notifying object has two responsibilities: tell the dependency manager when it changes state and send a termination notice to the dependency manager immediately before it is destroyed so that all references to it can be removed from the dependency graph. The dependent object has three responsibilities: register its dependencies, accept change notifications from the dependency manager, and notify the dependency manager immediately before the dependent object is destroyed so that all references to it can be removed from the dependency graph. The arguments to the change notification are the object that has changed, the type of change, and the locus of change. The latter two data types vary according to the implementation and the needs of the application, but in the spreadsheet example they might be, respectively, "update" (vs. deletion or creation) and a range of cells affected.

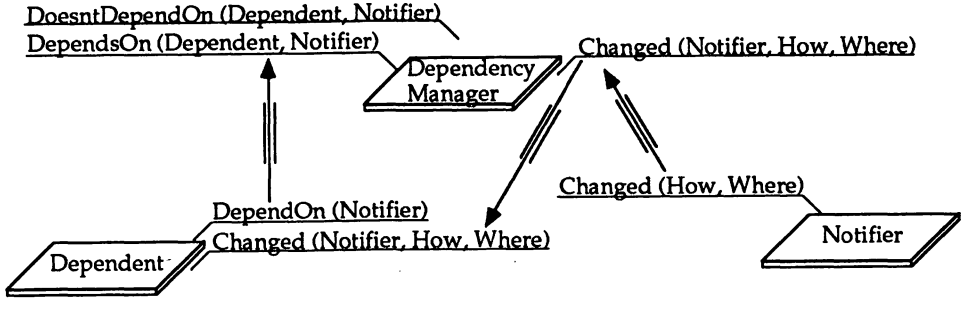
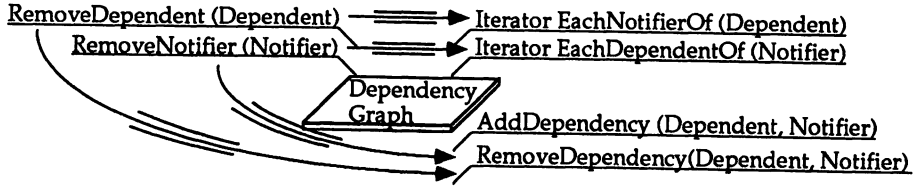


Figure 10-12. Generic dependency management

This might not correspond exactly to the implementation available to you, but the basic structure is common to almost all dependency managers.

▶ Implementing Dependency Management

If you don't have dependency management available, do not despair. It is always possible to implement a simple version of dependency management in a few days. The guts of the implementation are in the dependency graph, which has the responsibilities shown in Figure 10-13. Inside this object, the actual graph can be treated as a set of dependent/notifying object address pairs, implemented using your favorite data structure: sparse array, linked list, hash table, binary tree, or anything else you can steal from standard texts on sorting and searching. An alternate approach is to have dependency management inherited from "dependent" and "notifying" abstract classes (in single inheritance, combine these into a master class such as MacApp's TObject). This approach is shown in Figure 10-14. This approach has its drawbacks, but it is quick, simple, and reliable to implement.



Note: An Iterator is an object that supports First-Next iteration over a set of objects, in this case Dependents and Notifiers. The Each methods return iterators for use by the caller. The EachDependentOf method is used by the Dependency Manager to send notices to all dependents of a notifier.

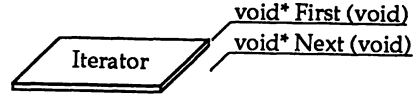


Figure 10-13. Dependency graph object

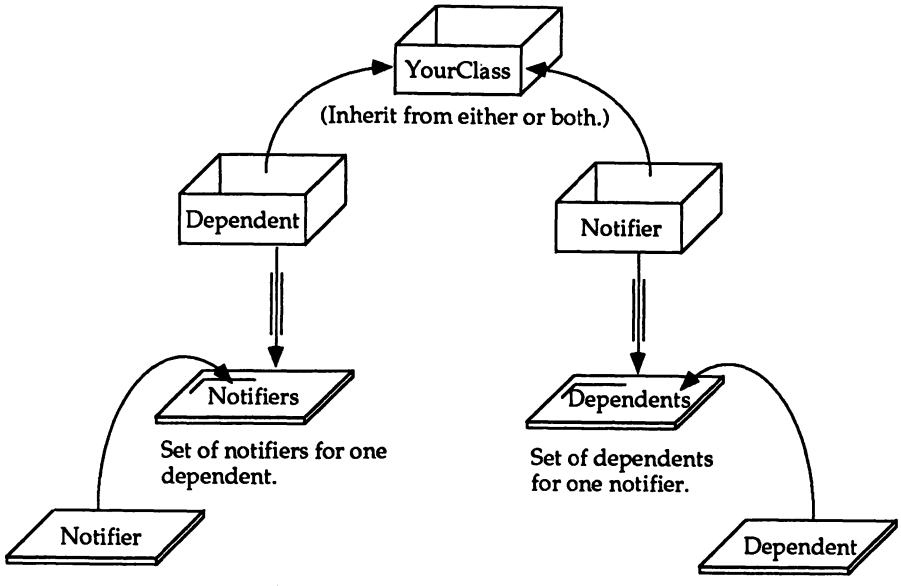


Figure 10-14. Using a master class to implement dependency management

▶ Calibration, Part III: Synchronization

So far, we have taken in-depth looks at two of the three forms of calibration: correlation and synthesis. Correlation is the way you make sure that double descriptions are consistent with one another. This applies between the Reference and Solution Models and otherwise applies only between regions of different planes. Synthesis allows you to integrate scenarios with one another and with the overall model. Synchronization rounds out the picture by providing a way to ensure consistency within a plane.

There are five basic types of synchronization, all of which derive from common sense principles. However, even though the principles are simple, it is a challenge to apply them rigorously. The types of synchronization are

1. Knowledge of other objects and data
2. Creation and initialization
3. Destruction
4. Message protocol
5. Connectedness

▶ Knowledge of Other Objects and Data

Ultimately, collaborations are messages sent from one object to another. In order to send a message to a receiver, the sender must have its address. How did the sender acquire that address? This may sound like a trivial question, but it is one of the most important design issues you will face. On the one hand, greater knowledge allows more flexible collaborations. On the other hand, the more widely known an object is, the more difficult it is to change it without side effects.

Let's look at an example from the Technology Plane of the payroll application. Suppose we have a window that contains an editable text field. In that field, we display the name of an employee. The name itself resides in the Content Model, while the editable text field is part of the User Interface Model. Clearly, the two must collaborate. Figure 10-15 shows one possible treatment of this collaboration in which the editable text object asks the content object for the information and tells the content object when to change the information.

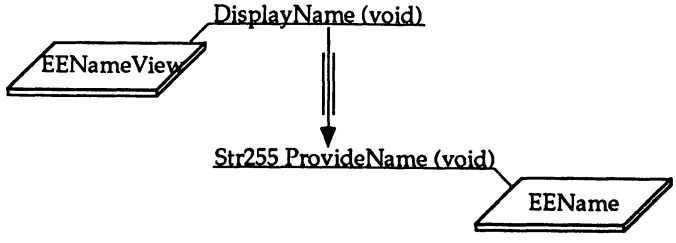


Figure 10-15. Collaboration between user interface and content objects

This is all well and good, but how did the editable text field find out about the content object? Does it have the address of the content object as an attribute, as in Figure 10-16? If so, how did that attribute get set in the first place? Did the user interface object create the content object, or was it handed the address of the already created object at some time in the past?

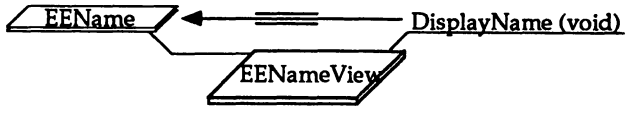


Figure 10-16. Object address as an attribute

Alternatively, does the user interface object ask some other object for the address as needed, as in Figure 10-17? If so, how does it have knowledge of that third object, and how does that third object have knowledge of the content object?

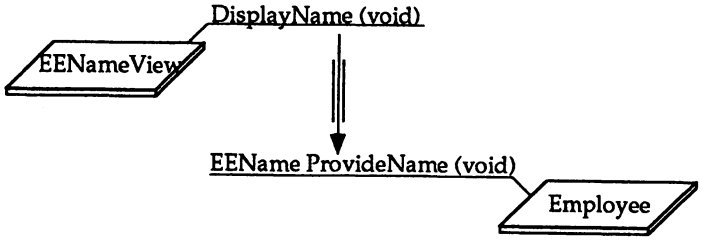


Figure 10-17. Obtaining an object's address on request

In general, there are two choices: store the address as an attribute or ask somebody for it. It is often necessary to recursively trace the knowledge through several other objects. For the moment, we are not concerned with which approach is correct, only that the decision be made and the knowledge accounted for. *Any collaboration for which we have not explained the object knowledge is a dangling thread.*

Continuing on the same theme, if one object sends data to another, how did it acquire the data to send? Again, there are two legitimate answers: the data is stored within the object, or the sender has to ask someone else for it as required. This can be an evolving issue, as data stored within an object is distributed to subassemblies or other objects. However, as with knowledge of other objects, knowledge of data must be accounted for or be treated as a dangling thread.

▶ Creation and Initialization

If one object is to collaborate with another, both must exist at the time. This, too, sounds trivial at first, but actually making sure that objects are created before they are needed is not so easy. For any given collaboration, we seek to account for when and how the collaborator came into being and to verify that the creation occurred in time. This becomes entwined with the first type of synchronization, knowledge of other objects, since it is at the time an object is created that its address comes to be known. This form of synchronization should also be extended to initialization of non-object data.

▶ Destruction

The flip side of synchronizing creation is synchronizing destruction of collaborators and non-object data. In order to be a collaborator, an object must not only have been previously created; you must also assure that it is still around when needed! This requires tracing, for each collaboration, how the collaborator will ultimately be destroyed and verifying that it is not destroyed prematurely. The result is often a scenario like the one shown in Figure 10-18.

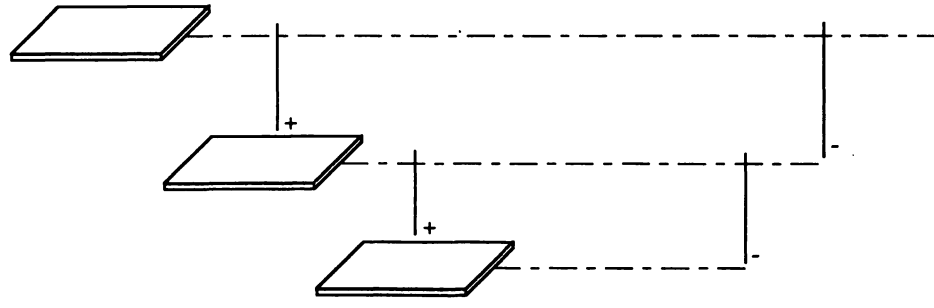


Figure 10-18. Creation and destruction of objects

Synchronization of destruction also requires demonstrating that each object is properly destroyed when it has outlived its purpose. Recall that in Chapter 9 we discussed the concept of "information ownership," in which the object that ultimately must return information is described as the "owner" of that information. It is important not to confuse *information ownership* with *object ownership*, which describes which objects destroy which others. Let's assume for the moment that you are using a language like C++ or Object Pascal, which does not have automatic garbage collection like Smalltalk and Macintosh Common Lisp. If object A is the only object with the right (and responsibility!) to destroy object B, then A is said to be the *owner* of B. If an object has any owner, it can have only one owner.

It is also possible for an object to have no owner and to be a *self-owned* object that destroys itself when it is no longer needed. Consider the following example from the payroll application. Employee information is stored in a file on a disk and retrieved in the form of employee records. Each employee record is represented by an object in memory, but obviously we only want some of the records in memory at any one time. When a client object, such as a paycheck, needs an employee record, it asks the file object to find it. The file object looks at its cache of in-memory record

objects to see if it is already in memory. If not, it reads the information from disk and creates a new memory-resident employee object, returning its address and setting its initial "reference count" to 1. If the record is already present, its address is returned after incrementing its reference count. When a client is through with a record, it sends a message to the record object releasing it, which results in decrementing the reference count. Thus, at any point in time, the reference count is the number of clients that are referencing the record. If that number goes to zero, the employee record removes itself from the cache list, then destroys itself. This is a classic self-owned object. There are many other ways to implement the same basic idea of an object that is smart enough to know when it is no longer needed and destroys itself at that time.

If you are fortunate enough to be using a language with automatic garbage collection, ownership is not all that important a concept. Simply by removing references to objects, you allow objects that are no longer referenced to be cleared out automatically by the system. Unfortunately, automatic garbage collection is expensive, and few systems have the luxury of applying a single scheme uniformly.

▶ Protocol

This form of synchronization verifies that the sender and receiver of a message both expect the same protocol to be used. Specifically, the information passed as part of the message must be agreed to by both ends of the line. This is taken into account as part of the synthesis process described in Chapter 9.

▶ Connectedness

Except for deliberate provisions for future growth, everything in your run-time architecture should somehow contribute to the program's function. This means that every responsibility should under some circumstances be called during execution, every attribute should be accessed at some time, and every run-time object should be used. Object-oriented programs for the Macintosh, as we have previously observed wait for events to occur, then react to the events. Events include mouse and keyboard activity, Apple Events messages, operating system interrupts or other external inputs, or the initial launch of an application. As part of synchronization, you should make sure that every feature of your run-time architecture is ultimately connected to external events.

A responsibility called in direct response to some event is by definition connected. A responsibility that collaborates with a connected responsibil-

ity is also connected. The same definitions can be used to identify connected attributes, which are accessed by connected responsibilities. Testing for connectedness is a matter of recursively applying these rules. By implication, the connectedness test also identifies a set of unconnected responsibilities and attributes, which are not specifically marked as connected. An unconnected responsibility does not necessarily indicate a design flaw, but it should raise a red flag. For an unconnected responsibility, one of the following must apply.

1. The responsibility is truly not needed to implement the scope of the Solution Model and Technology Plane. The only justification for leaving it in is to simplify future expansion of the program or to use it as a "hook" for maintenance and testing. The remedy is to remove the method or classify it as "future expansion" or "maintenance." For example, in the payroll application we might define a responsiblity for ranking an employee based on annual earnings, but leave this as an unimplemented future feature.
2. It may be a synonym for some other method. This is common. Early in the design process, something is given one name and later, for whatever reason, a new name is used in scenarios. This is uncovered by the synthesis process described in Chapter 9 and, therefore, is not really subject to a separate test as part of synchronization. In the payroll application we may have defined two responsiblities: "Compute hourly compensation," and "Compute compensation" that may turn out to be the same when we further define these behaviors.
3. There may be some missing but necessary connection. This is also common. Often one will plunge into the middle of some area of the design, planning on later connecting it to other parts of the design. It is easy for connections to be missed. The remedy is to create scenarios that connect the unconnected methods. In the payroll application, we might discover that we neglected to use scenarios for employees to elect periodic charitable contributions. We would have to add scenarios that connect the scenarios that connect them to the methods to the employee object.

The same logic can obviously be applied to attributes as well. There are subtle implications for correlation of the Execution Plane to the Technology plane inherent in this test. If a responsibility is essential in the Technology Plane—that is, it is an implementation of some responsibility of the computer from the Solution Model but is unconnected in the Execution Plane, you have a problem. You have specified that the program

is capable of doing something in the Solution and Technology Planes, but have not provided any external events that can cause that activity to be carried out! Perhaps you left out some feature of the user interface or some interface to the operating system or external devices and networks, or perhaps the capability was not really needed in the upper planes in the first place. If you identify an unconnected responsibility, consider its correlation to the Technology Plane in deciding how to resolve the problem.

▶ Applying Synchronization

As noted, some synchronization tests are implicit in the way one performs synthesis. Specifically, synthesis, properly applied, validates protocol and synonyms for method names. Otherwise, synchronization is usually the laggard in the analysis and design race. The tendency is to blast ahead by creating lots of scenarios and synthesizing them, occasionally correlating to tie up dangling threads. We don't need to immediately synchronize those objects, as long as the synchronization is not put off for too long. Synchronization tends to bring out of the woodwork all sorts of very serious oversights in the conceptual or actual design. It is best to get these problems on the table early, before the design sprouts deep roots. The first time to synchronize is when an initial batch of scenarios has covered some central topic pushed down from the Technology Plane. From this point on, synchronization should occur whenever any sort of milestone of expansion is reached in either the Technology Plane. Expect synchronization to take some time and to cause you to expand dramatically both the User Interface and Content Models, particularly during the early stages of the project.

▶ Managing the Design Phase

The design phase uses the same basic techniques used for analysis: CPC, scenarios, synthesis, correlation, and prototyping. To this we add synchronization. The process is driven by the set of scenarios chosen at the conclusion of the analysis phase to form the basis of project scheduling and measurement, the *design set*. Each scenario in the set must be drilled down to the Execution Plane at some point during the design phase. Choosing an appropriate order in which to tackle these scenarios is not as important as making sure that all are dealt with and that the cumulative quantity correlated downward is consistent with the schedule at any given point in time. Prototyping will be freely used for all the same reasons cited in Chapter 9, resulting in a substantial body of code at the conclusion of the phase.

▶ Use of Scenarios

In Chapter 9, we talked about using scenarios, not just as a tool for model-building and exploration, but for project management as well. The schedule for the design phase should be based on a specific set of scenarios chosen from all scenarios available at the end of the analysis phase for the purpose of organizing the project. This design set should collectively cover all of the Technology Plane and any work performed on the Execution and Program Planes during the analysis phase of the project. No scenario in the set should be completely redundant with any other; when in doubt, include borderline scenarios in the set. It is perfectly acceptable to generate new scenarios from existing ones, specifically for inclusion in the design set.

The design set then becomes the fundamental unit of organization during the design phase. Schedules should be based on completion of scenarios or groups of scenarios. Team members should be made responsible for completing the design and synchronization of scenarios or groups of scenarios. This is a much better basis for scheduling, assigning tasks, and monitoring progress than other schemes based either on a top-down decomposition of the project or by object/class.

You should expect that it will take roughly the same amount of effort per scenario throughout the design process. In the early days, there will be little other material to draw on, but calibration will be easy. Later, there will be a good body of design scenarios as starting points, but calibration will take longer as scenarios accumulate. A good way to develop the design schedule is to pick a few representative scenarios from the design set and drill them down to the Execution Plane, keeping track of productivity as you go. This can then be extrapolated to the rest of the design set.

▶ Priorities

Although the design phase need not proceed in any particular sequence, the emphasis is usually in the following order of priority.

1. Content Architecture. As the ultimate source of the program's functionality, the Content Architecture is where most of the technological hurdles must be jumped. These hurdles tend to be not just difficult but also central to the design. Thus, it is common to spend much of the early effort on the Execution Plane dealing with the tougher content issues.
2. User Interface Architecture. As the look and feel of the program grows in importance during the project, so does the user interface. As

synchronization takes place, much of the consequent changes force you to bounce between the content and user interface portions of the design.

3. Environment Architecture. If your application deals with attached devices or networks, this may move up in importance. In most applications, however, there simply isn't much there. The class library handles most of the rote stuff for you; you need only customize the behavior of the library.

▶ Prototyping

You must drill down to the Program Plane in order to get the Execution Plane right. All of the reasons cited for prototyping in Chapter 9 are equally valid in the design phase. Prototypical code helps demonstrate engineering feasibility, particularly acceptable levels of performance. It serves the Heisenberg Prototyping objective of testing your designs in the fire of running code. It can provide runnable code to use as the means of obtaining critical feedback, both from the software engineers and others. Prototyping provides good measures of implementation productivity, which is useful in choosing from otherwise competing design alternatives. Finally, design proceeds somewhat unevenly in a typical project. Some areas proceed rapidly and smoothly, while others bog down. Everyone participates in some areas, while others are the province of specialists. Whenever someone is not busy designing, there is no good reason not to go ahead and make progress with the code.

A prototype should completely cover one or a small number of scenarios. Your objective is to validate that those scenarios are correct or feasible or desirable. This is best done by covering entire scenarios, not pieces of them. It also makes it easier to perform the calibrations that result from new insights gained through the prototype.

In general, it is best to use the most straightforward implementations of run-time objects during the design phase. That is, in most cases implement a single run-time object as a single concrete class that does not inherit any more than is necessary to complete the prototype. When choosing subjects for prototyping, focus on specific members of categories or abstractions. Staying away from inheritance allows you to make rapid progress, a key to successful prototyping. Don't worry about throw-away code: much of the code you write will ultimately be cut and repasted elsewhere in the final hierarchy, but little will be thrown away. We stress the value of examples because they tend to expose problems with generalizations. Counterexamples lead to very expensive problems and should be exposed as early as possible. If you have described a single

responsibility that turns out to require two or three distinct implementations, or if you have created an abstraction that turns out to be flawed, it will cost dearly if not discovered until a class hierarchy has been put in place. Remember: Inheritance is the optimal way to implement the run-time objects, but optimizations should come only after the basic behavior of the objects is well understood.

▶ When Is the Design Phase Complete?

The design phase ends when three conditions are met.

1. The Technology Plane has been completely correlated to the Execution Plane; that is, every feature of the Technology Plane is accounted for in the architecture.
2. The Execution Plane is completely synchronized, according to the five tests outlined in this chapter.
3. The Execution Plane covers all known sources of events, whether from the user, the Macintosh systems software, or attached devices and networks.
4. A credible project plan for completion of implementation is in place.

Unlike the rather fuzzy definition of when analysis ends, the first three of these conditions provide a very precise, verifiable test of completion. The last, preparation of a project plan, requires judgment regarding the technical risks still faced. Ideally, these risks have already been managed through prototyping before the design phase is declared complete.

Let's assume that your architecture has passed the above tests, meaning that the design phase is now complete. Congratulations! You now have an extremely detailed, well-validated design that is just short of code, along with a body of prototypical code developed to support analysis and design. Next comes the easy part: implementation.

▶ Managing the Transition to Implementation

As with the transition from analysis to design, the central problem in the transition from design to programming is identifying a set of scenarios that completely covers the scope of the Execution Plane. This *implementation set* is then used as the basis for project planning, estimation, and tracking. Scenarios of the implementation set should be handed to specific individuals in related groups, with the groups appearing on the project plan as distinct tasks. The implementation set should not contain

scenarios completely redundant with others in the set, but when in doubt, include too many rather than too few. If it is not possible to estimate the implementation effort for a scenario or group of scenarios, your design phase is not complete. You need to do some prototyping to nail down the implementation risks and estimates. Once the implementation set has been constructed and the project plan for implementation is in place, you are on the home stretch.

▶ Summary

This chapter focuses on the Execution Plane, which describes the detailed program objects we design to construct the system envisioned in the Solution Model.

- We develop the Execution Plane by applying CPC, paying a great deal of attention to calibration. The objects we design are program objects, not real-world objects, and we must construct them to conform to the rigorous requirements of object-oriented programming. Many objects exist to support the implementation rather than to model the external world. Some of these implementation-related objects are supplied to us, such as pre-existing application class libraries; some, such as automation and auxiliary objects, we have to design ourselves.
- In the Execution Plane we describe sets of objects using abstractions, which may or may not turn into classes in the final program. We add a lot of detail to our solution-based model as we drill down the Content, User Interface, and Environment Models. We define the specific calling sequences for responsibilities and break them down as far as possible. We define accessor functions for each attribute to encapsulate objects. A major concern is synchronization: An object must exist before it can send or receive a message, and the sender must know the address of the receiver. We must also correlate to ensure that all categories from the higher planes are mapped into Execution Plane objects or abstractions. In the Execution Plane we design how our application will exploit the objects associated with the Macintosh platform.
- Again, in the Execution Plane we check that each responsibility has a purpose that is ultimately used in response to some event. We use design sets of scenarios to manage our efforts. Our general priority order is to focus on the Content Architecture, followed by the User

Interface Architecture, and then the Environment Architecture. Prototyping is necessary during this effort if we want to design the Program Plane properly.

- We are done with the Execution Plane when the Technology Plane is fully correlated to it, when it is synchronized, and when it covers all known sources of events. The transition to the programming phase is managed by identifying an implementation set of scenarios that covers the Execution Plane, then using that set as the basis of project estimation, organization, and planning.

11 ▶ Programming

▶ What This Chapter Is About

This chapter describes the programming phase of Solution-Based Modeling. In the programming phase you will create a program that, when compiled and run, produces the run-time objects of the Execution Plane. There are two central activities during this phase: designing a class hierarchy that optimizes the implementation and implementing the methods and attributes. One of the primary considerations during the programming phase is how to properly implement the abstractions of the Execution Plane. As you will see, inheritance is only one of many techniques and it is not always obvious which is best. The decision is driven by technical objectives and the software engineering objectives embodied by the Four Itys: Maintainability, Reliability, Extensibility, and Reusability.

This chapter also discusses management of the programming phase. Finally, we will discuss how a program developed using Solution-Based Modeling evolves after its initial release.

▶ Overview

Figure 11-1 shows the relationship between the Execution and Program Planes.

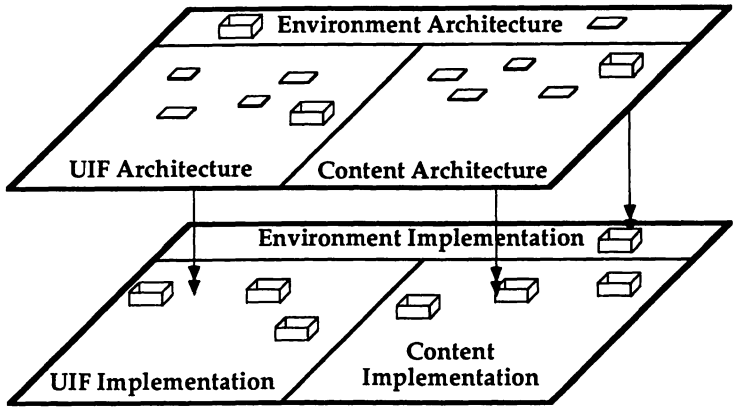


Figure 11-1. Implementing the Program Plane

The Program Plane is broken into the same regions as the Execution Plane, but they are called "implementations," rather than "architectures." The Program Plane is the implementation of the application, not just a model or design of it. Here we deal with all of the restrictions and features of the language and class library. Specifically, we take into account restrictions on inheritance; restrictions on names of classes, methods, and attributes; the need to create run-time objects from concrete classes; and any peculiarities of the class library that prevent it from aligning precisely with the desired architecture. Recall that in the Execution Plane you merely indicate collaborations between library classes and run-time objects. In the Program Plane, you must decide exactly what form those collaborations will take, through inheritance or otherwise.

In the Program Plane, we leave behind objects and deal with classes. For each run-time object in the Execution Plane, we must make sure there is a fully implemented concrete class in the Program Plane. That concrete class is instantiated at run time to produce the needed object. The concrete classes are arranged into a class hierarchy that provides an optimal implementation of the program, taking advantage of inheritance to share code between objects and polymorphism to simplify the logic of the program.

This is by far the easiest phase of a Solution-Based Modeling project. In the Execution Plane, you recorded enough information to completely specify the interfaces to all run-time objects. Their responsibilities were decomposed to the point that most will be implemented as methods made up of a few lines of code in the Program Plane. You have also probably implemented a good deal of prototype code at this point, much of which

can be reused in the final implementation. The design has been reviewed by interested parties external to the software team from several vantage points: how it supports business needs, how it uses technology, and, to a lesser extent, how understandable and robust the software architecture is. All that remains is to design the class hierarchy and provide code that implements the responsibilities as methods.

The first step, designing the class hierarchy, should be viewed as an exercise in software engineering. Classes are designed to achieve software engineering objectives, rather than because they have "meaning." The objectives are principally the Four Itys, Maintainability, Reliability, Extensibility, and Reusability. The last of these takes two very distinct forms. The first is reuse within the application. Where run-time objects have code or attributes in common, inheritance is often, though not always, the best way to avoid reimplementing the same code for each object's concrete class. An entirely different issue is reuse of code from one project in another. Reuse within this project is the main objective; reuse across projects is generally a consequence of doing a good job of design on this one.

Implementation of methods and attributes is a pretty mechanical process. Other than specialized algorithms for image processing, number crunching, or the like, most methods are between one and ten lines of code that directly reflect the architecture of the Execution Plane. By the time the Program Plane is reached, all collaborators of each responsibility are known. All that remains is to write specific code that calls the methods that implement collaborating responsibilities, implements if-then and iterative logic, declares and uses local variables, and so forth.

As you have probably already guessed, these two activities—designing class hierarchies and implementing methods—play off each other. It is not always possible to spot the best way to use inheritance until methods are written for concrete classes. On the other hand, decisions on the use of inheritance certainly affect the code. Neither comes before the other; instead, as with all of Solution-Based Modeling, programming is a highly iterative process.

As with the design phase, the programming phase of the project is organized around scenarios. This applies equally to estimation and scheduling, assigning tasks to team members, tracking progress, and testing.

▶ Designing Class Hierarchies

Let's start by getting one thing straight: Inheritance is only one of many ways of implementing the abstractions of the Execution Plane. It is wrong to assume that you can or should simply gather up all the abstractions left

lying around in the Execution Plane and turn each one into a class. It is common to do so, but there are often better alternatives. In fact, depending on the language you are using and other factors, it may not even be possible to turn some abstractions into classes. One of the hallmark differences between the expert and the novice is the ability to choose wisely from among these alternatives.

▶ (At Least) Six Ways to Implement Abstractions

Suppose we have two run-time objects in the Execution Plane that share some abstraction, as shown in Figure 11-2.

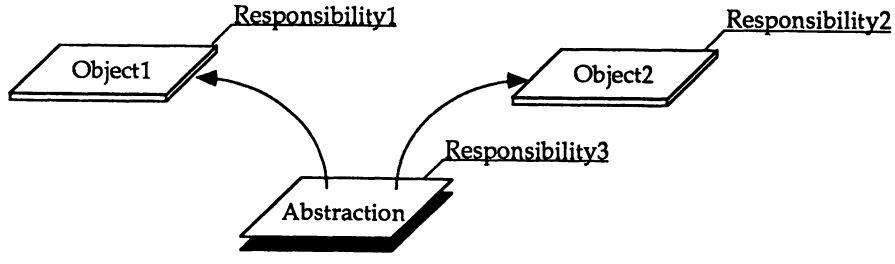


Figure 11-2. An abstraction

The most obvious way to implement these objects is to turn each into a concrete class, each of which inherits from an abstract class that mirrors the abstraction, as shown in Figure 11-3.

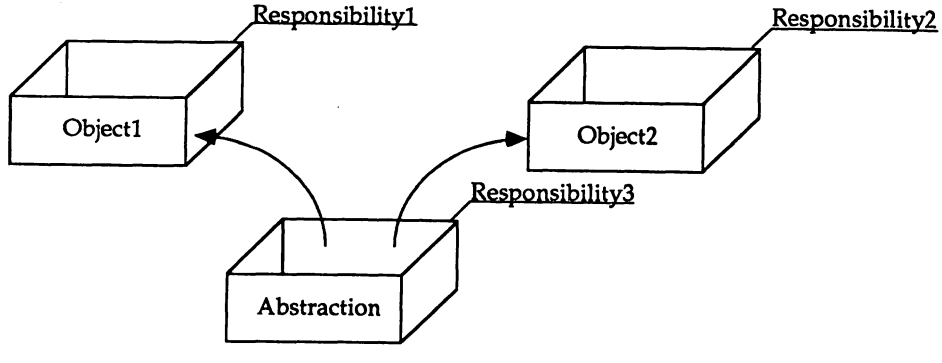


Figure 11-3. Implementing an abstraction using inheritance

However, there are at least five other approaches to implementing an abstraction. Let's take a look at each of these to establish that alternatives exist, and then return to a discussion of when each technique is appropriate.

Separate Implementations

The simplest alternative to Figure 11-3 is to provide a separate implementation of each run-time object. This means creating a concrete class for each, with no sharing via inheritance. An abstraction may, on rare occasions, be a useful device for describing the run-time objects but turn out not to be all that useful in the implementation. This is shown in Figure 11-4.

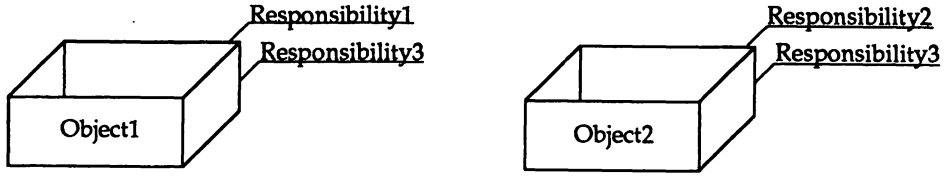


Figure 11-4. Separate implementations of an abstraction

Copy/Paste

A related strategy is to create one implementation, then duplicate it to create the implementation of another class. This often requires nothing more than the sequence of steps shown in Figure 11-5.

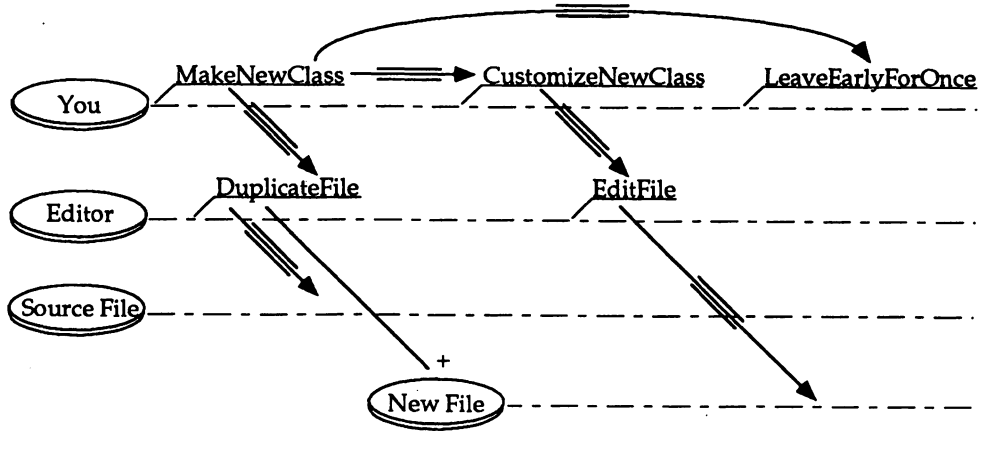


Figure 11-5. Using copy/paste to implement abstractions

That is, you first implement one version, then copy it and customize the copy to produce the second version. Often this requires nothing more than a global search and replace of the class name plus changes to a few lines of code. A variation on this theme is to create MPW scripts or even simple programs to do the copying and renaming automatically.

Helper Objects

Figure 11-6 shows an alternative implementation that uses helper objects, first discussed in Chapter 2.

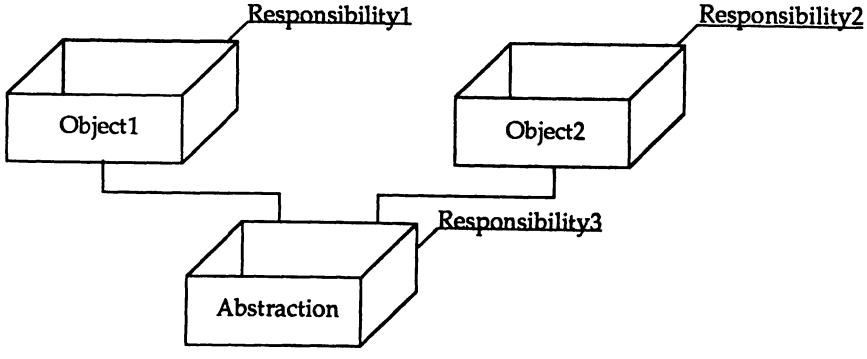


Figure 11-6. Implementing an abstraction using helper objects

Here, we create a class specifically to implement the abstraction's methods and attributes, then attach an instance of that class to each of the run-time objects we specified in the Execution Plane. A variation on this theme uses inheritance, but not in the same way as in Figure 11-3. In Figure 11-7, the abstraction is turned into an abstract class. For each run-time object, we attach a helper object that comes from a subclass of the abstract class.

Combinations of these strategies are also possible. One of the run-time objects might descend from the abstraction's class, while the other uses a helper. The helper may be a direct instance of the abstraction's class, or it may be an instance of a descendant of that class.

Combine Abstractions

Figure 11-8 shows an expanded version of Figure 11-2, in which there are now two abstractions that overlap.

There are many different strategies that present themselves here. The ones we have already discussed certainly apply. In addition, we might consider several ways to accommodate the overlap. Figure 11-9 shows one technique: Form an abstraction of the two abstractions and assign it the shared properties.

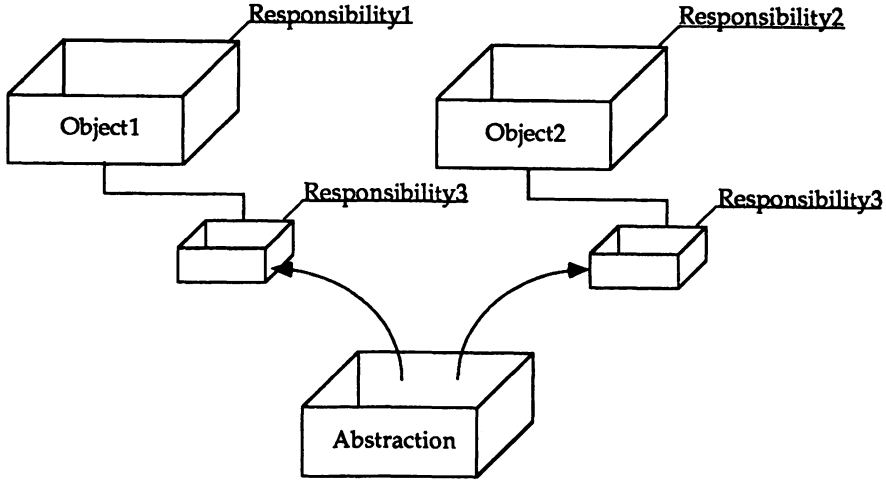


Figure 11-7. Another implementation using helper objects

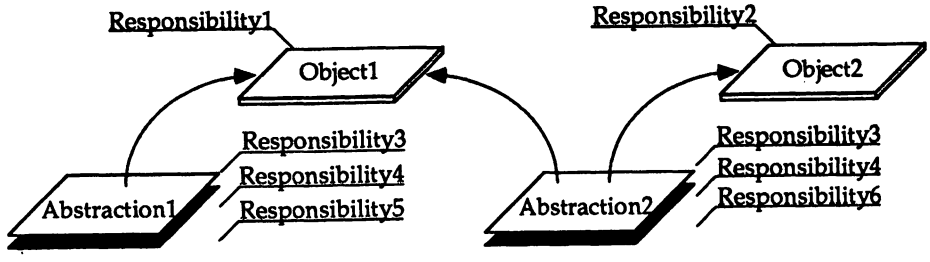


Figure 11-8. Overlapping abstractions

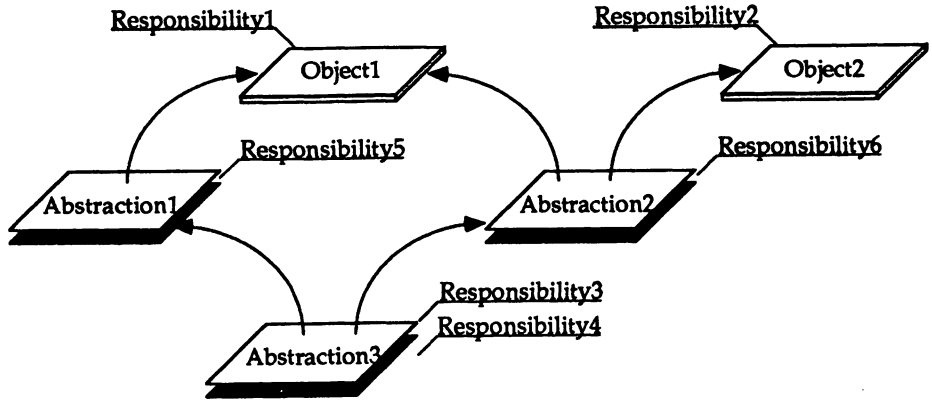


Figure 11-9. An abstraction of abstractions

This translates into a class hierarchy, diagrammed in Figure 11-10.

Another approach is to combine the two abstractions into a single abstract parent for both concrete classes. This is done by combining all of the properties of both abstractions to form a single class, then implementing each concrete class in such a way that it uses only those properties that come from the original abstraction. This is shown in Figure 11-11.

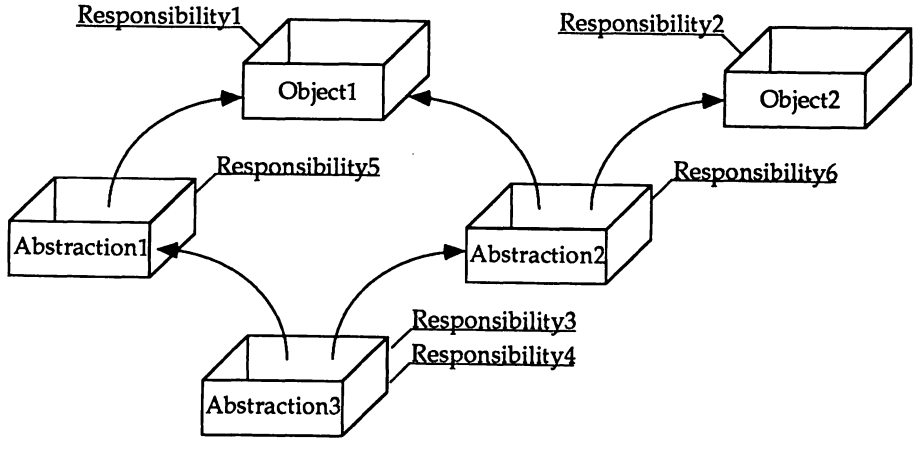


Figure 11-10. Abstraction hierarchy implemented as a class hierarchy

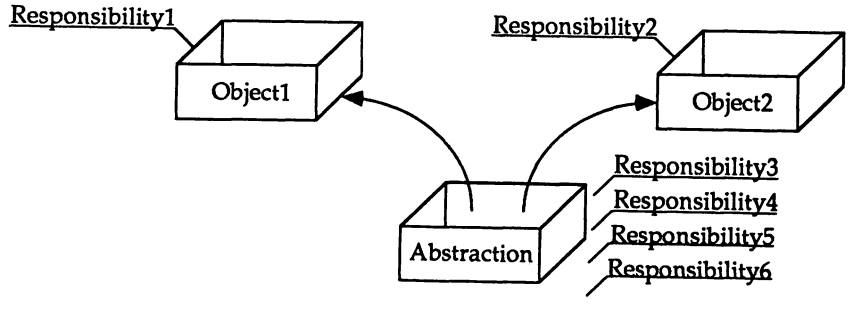


Figure 11-11. Combining abstractions

A variation on this theme is called *single-threaded inheritance*, in which you form an artificial class hierarchy that does not mimic the abstraction hierarchy. Each of the two abstractions becomes a class, but now one of them inherits from the other, as in Figure 11-12.

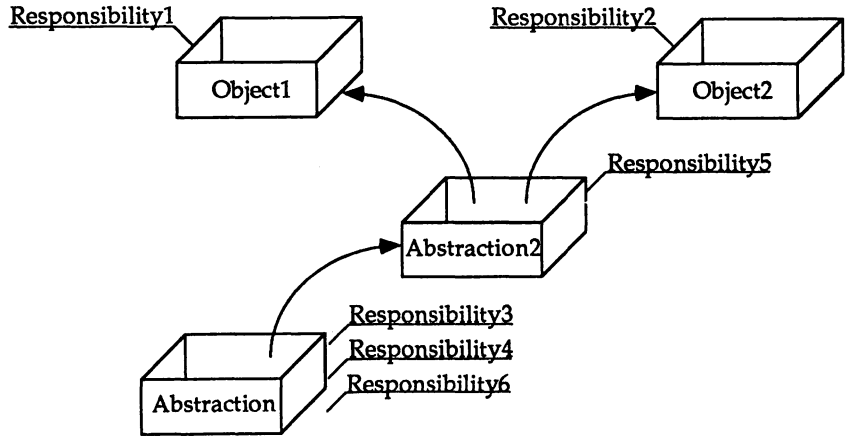


Figure 11-12. Single-threaded inheritance

As you can see, this has the same net result: a concrete class that holds all the required properties without requiring either multiple inheritance or helper objects. The shared properties are usually assigned to the superclass, as in Figure 11-12.

Split Abstractions

The opposite strategy is also possible: Form one abstraction with the shared properties and two more to represent the non-shared properties, then implement each as an abstract class, as shown in Figure 11-13.

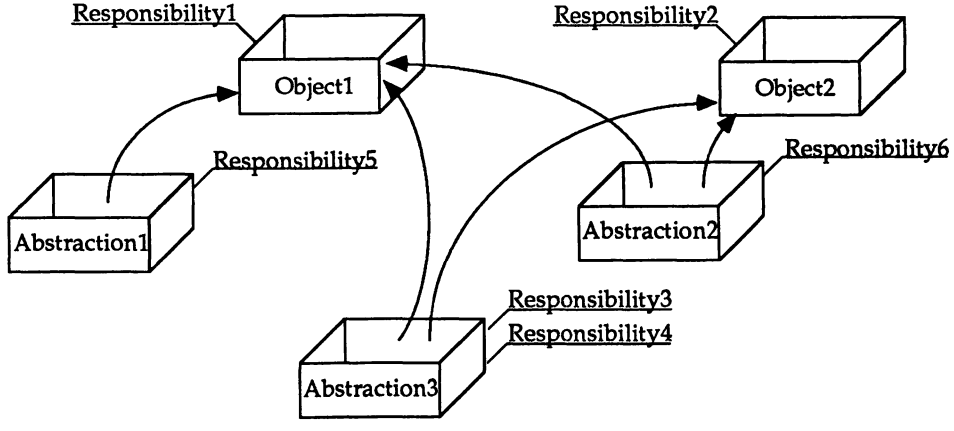


Figure 11-13. Splitting abstractions

Combination Strategies

Finally, there are myriad ways to combine these strategies. For example, one can inherit from the common properties and use helpers for the non-shared properties, as shown in Figure 11-14.

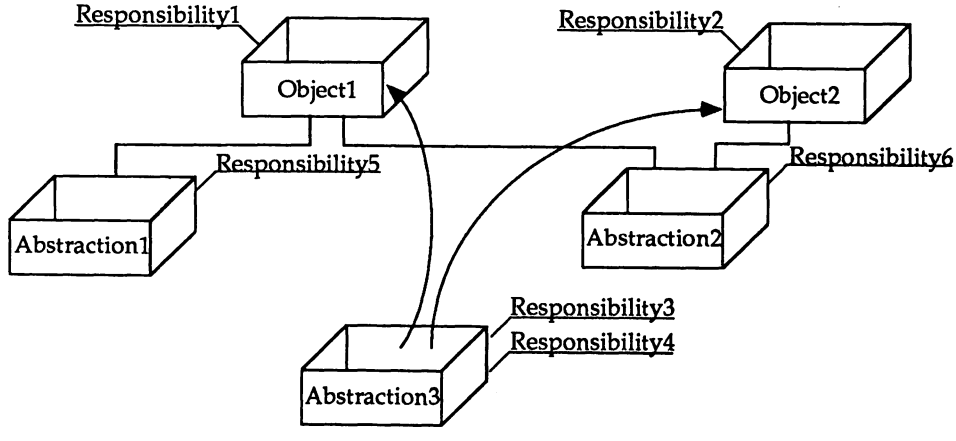


Figure 11-14. A combination approach to abstractions

▶ Choosing the Best Strategy

Some of these strategies may seem a little contrived, especially the copy/paste technique and its variants. The authors have had occasion to use every one of these, including every variation listed. Each has its place and choosing the best technique for a given situation is much of what sets the expert apart from the beginner. The choice is dictated by many considerations, some of which follow.

- Are the abstractions static, or do they change over the running life of the program?
- Does the language support multiple inheritance? Object Pascal does not. Even if you are using C++, if you are using it with MacApp you will have to stick to single inheritance with any class that descends from a MacApp class.
- Does the language support dynamic inheritance or creation and modification of classes at run time? Object Pascal and C++ do not, but Smalltalk, Macintosh Common Lisp with CLOS, and a few other languages do.

- What will yield the greatest potential code reuse, both within this project and across projects?
- What is the phase of the project: analysis, design, or programming?
- Are you creating a prototype, developing the program, or polishing the code in a final pass?
- How do classes of the class library collaborate with the run-time objects and abstractions?

As you can see, this list spans issues of architecture, language, class library, software engineering, and project management. Let's revisit each of the five basic techniques, this time to formulate rules for when to use each technique.

Copy/Paste

This is appropriate at almost any time, but particularly during prototyping. When you write a prototype, it probably deals with one or two specific types of run-time objects. These are implemented without regard to other similar objects in the interest of making rapid progress. As the prototype is expanded, more run-time objects are thrown into the pot, but one or two at a time. It is still more productive to clone the existing prototype code and customize it for the new objects, rather than spend a lot of time on optimizing a class hierarchy. For all of the various reasons to prototype, constructing a class hierarchy is not a high priority.

Copy/paste is also a good way to work around language restrictions. Take, for example, the following linked list class.

```
class List {
  private:
    List *next_in_list;
  public:
    List (void) {next_in_list = NULL;}
    ~List (void)
      {if (next_in_list != NULL)
        delete next_in_list;
      }
    List *GetNext (void) {return next_in_list;}
    void SetNext (List *next) {next_in_list = next;}
  };
```

This is a straightforward class that does nothing more than maintain a linked list of instances of class `List`. Now look at the following subclass:

```
class FunnyString : private List {
  private:
    char c;
  public:
    void PrintString (void)
      {
        putchar (c);
        if (GetNext() != NULL)
          ((FunnyString *)GetNext())->PrintString();
        else
          putchar ('\n');
      }
  };
```

The details have been spared here in the interest of brevity. In the real class, one also makes provision for setting the character values and so forth. Notice the line

```
((FunnyString *)GetNext())->PrintString();
```

Doesn't this seem a little tortured? One would like instead to simply say

```
GetNext()->PrintString;
```

but that does not work: `GetNext()` has been defined to return a `List`, not a `FunnyString`. Now envision a class that has many methods, not just `FunnyString`, that need to walk through the list. Each time you access `GetNext()`, you must coerce it to point to a `FunnyString`. If client objects need to walk through the list, this only gets worse since they also need to do the type coercion. Even if you are willing to put up with the clumsy syntax, you will have lost the strong type checking otherwise applied by the compiler; that is, the compiler cannot warn you if you stick a `Foo` into the list instead of a `FunnyString` and then attempt to print the list. When you write `((FunnyString *)x)->DoSomething()`, you are saying to the compiler, "Trust me, I know what I'm doing." It is far too easy to make a mistake in these situations; when you do, you will end up in a low-level debugger trying to figure out where that bus error came from.

This is a case where you might prefer to simply copy the text of the `List` class and change the name `List` to `FunnyString`, adding new methods and attributes like `PrintString` and `c`.

```
class FunnyString {
  private:
    FunnyString *next_in_list;
      char c;
  public:
    FunnyString (void) { next_in_list = NULL; }
    ~FunnyString (void)
      { if (next_in_list != NULL)
        delete next_in_list;
      }
    FunnyString *GetNext (void) { return next_in_list; }
    void SetNext (FunnyString *next) { next_in_list = next; }
    void PrintString (void)
      {
        putchar (c);
        if (GetNext() != NULL)
          GetNext()->PrintString;
        else
          putchar ('\n');
        }
};
```

This is much better. There is no type coercion, which means the compiler performs full type-checking. Also, it is easy to customize the code without worrying about what happens to other sibling subclasses of a `List` class. In C++, one can sometimes improve further on this by using `#define` macros, at the expense of making the code more obscure. (This does not work for Object Pascal, since there are no `#define` macros in Pascal.)

These situations fall under the specific heading of *genericity*. Some languages provide language-level support for generic classes (for example, classes whose instance variables and method arguments can hold different data types). Unfortunately, the current versions of Object Pascal and C++ do not provide this support. Copy/paste or macros are often the cleanest and most reliable way to share code across classes.

Other reasons to use copy/paste might include the following.

- There might be competing superclasses in a single-inheritance language. There are, of course, other work-arounds to make up for the lack of multiple inheritance, but copy/paste is a viable strategy.

- You are borrowing code from another project or library, but using inheritance to access the code is unworkable.
- The inheritance hierarchy is set and you need to make "just this last change," usually with the boss standing, arms crossed, foot tapping, eye glued on her watch while blocking the door out of your office. Using inheritance in the optimal way often takes more, not less, time than brute force techniques. Your Four Itys will be happier if you polish the inheritance, but your boss may not be if it means slipping a critical ship date.

Helper Objects

We already discussed one of the principal reasons to use a helper object in Chapter 2: to implement multiple inheritance through a side door when using a language like Object Pascal that only supports single inheritance. Note that all MacApp classes and their descendants are restricted to single inheritance, even if you use C++ with MacApp. This decision was made to ensure compatibility with both languages.

A second reason is that in most languages, inheritance is compiled into object code, although the behaviors may change dynamically as the program runs. Consider a simple rectangle-drawing program. This is like any typical Macintosh drawing program, except that it has only two tools in its palette: an arrow tool used to select and drag objects and a rectangle tool used to draw new ones. A portion of the run-time architecture from the Execution Plane is shown in Figure 11-15.

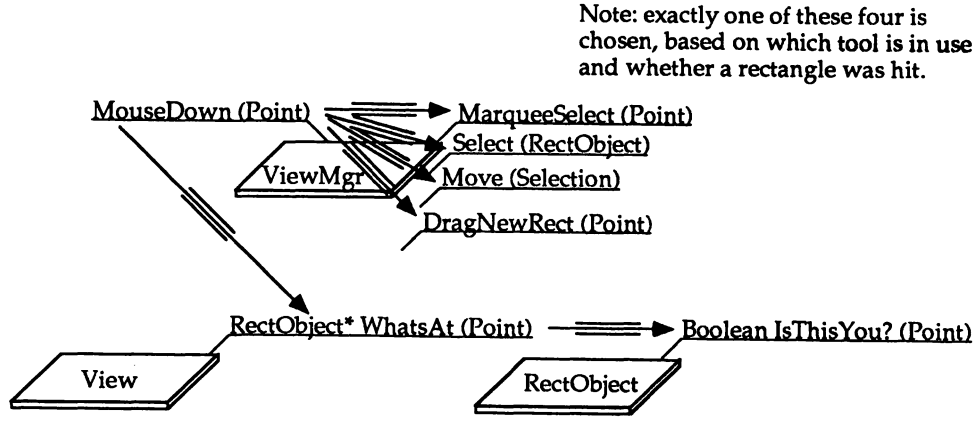


Figure 11-15. Rectangle-drawing program

Now consider what the drawing's manager must do when confronted with a mouse click in the drawing.

1. If the arrow tool is in use and the mouse did not hit a rectangle, the manager must launch a marquee-selection process to select all rectangles within the area dragged over.
2. If the arrow tool is in use and the mouse hits a rectangle, the rectangle must be selected and possibly dragged.
3. If the rectangle tool is in use, a new rectangle must be formed with corners at the locations of the mouse down and mouse up.

These are three distinct sets of behaviors. One way to account for these behaviors is to sprinkle the manager with if-then or switch-case logic. Another approach is illustrated by MacApp 3.0's TEvtHandler class. TEvtHandler maintains a linked list of "behavior" objects. When an event arrives, it is handed off to each behavior in the list until one grabs the event. If none grabs the event, the TEvtHandler itself does the work or hands it to the next TEvtHandler, which then uses the same logic. This allows the program to add and delete from the behavior list at run time rather than compiling in the inheritance of all capabilities that might be used. Figure 11-16 shows what happens when the user clicks on the arrow tool, and Figure 11-17 shows what happens when the rectangle tool is selected.

For the arrow tool, we install two behavior objects, the first of which grabs a mouse down when nothing has been hit and the second of which handles selection and dragging. For the rectangle tool, we install a single behavior for drawing new rectangles. This is a nice, modular architecture that replaces a very messy problem of inheritance. This is a good example of using helper objects to achieve dynamic changes in behavior.

Finally, a helper object may be reusable, but direct inheritance may not. A helper class that has no knowledge of the object being helped is a particularly good candidate for reuse.

Combining Abstractions

There are three variations of this technique, each with its own distinct uses and drawbacks. In the first variation, you form a hierarchy of abstractions that is then turned directly into a class hierarchy, as in Figure 11-10. This avoids implementing the shared properties separately for each abstraction. Often this second-tier abstraction has an intuitive meaning, in which case it provides a natural way to describe the concrete class. If the shared methods or attributes are heavily referenced by the non-shared methods, it can also produce clean inheritance by concrete classes. In

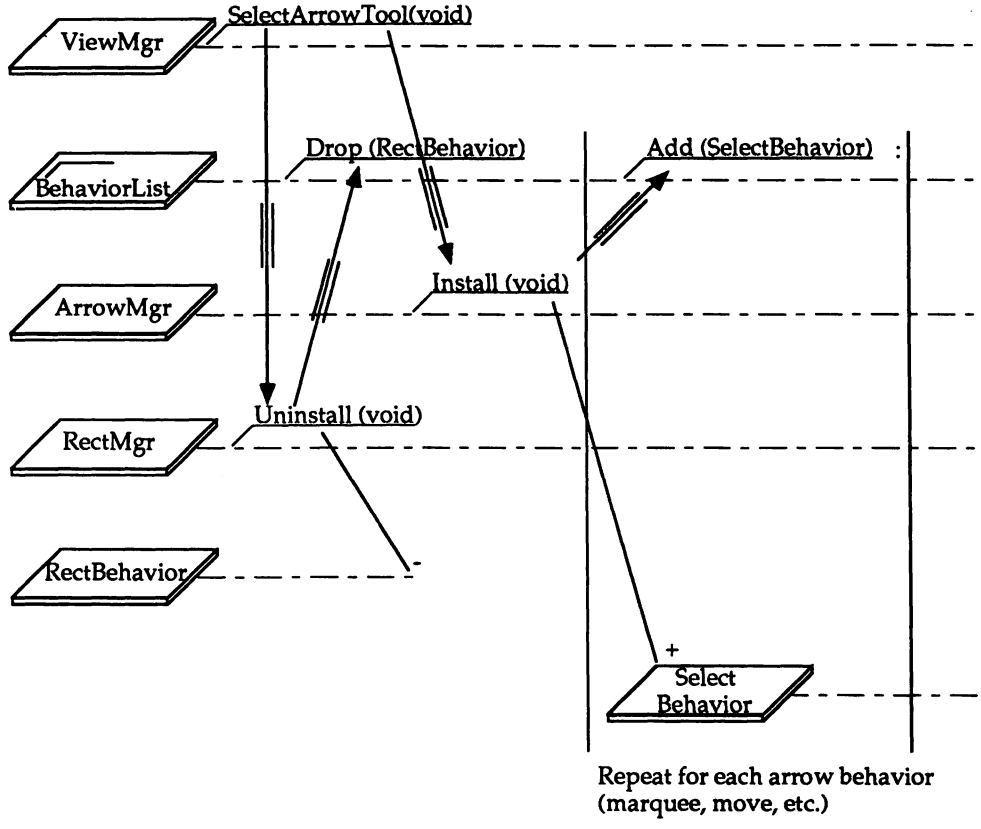


Figure 11-16. Selecting the arrow tool

general, however, this imposes an extra layer of inheritance which, as we will see, is to be avoided where no good justification exists.

The second technique involves implementing two or more abstractions in a single abstract class, as shown in Figure 11-11. This achieves exactly the same effect as the first variation, but does not require the use of multiple inheritance. In a single-inheritance language like Object Pascal, this strategy can sometimes be used to good effect as an alternative to helper objects, but it hinders reuse of code for other concrete classes that require only some of the combined properties.

The third variation, single-threaded inheritance, is really a compromise between the first two. The higher of the two abstract classes remains "pure," unpolluted by its subclass, but the subclass is, in effect, just the sort of combined class produced by the second variation. This is also a popular

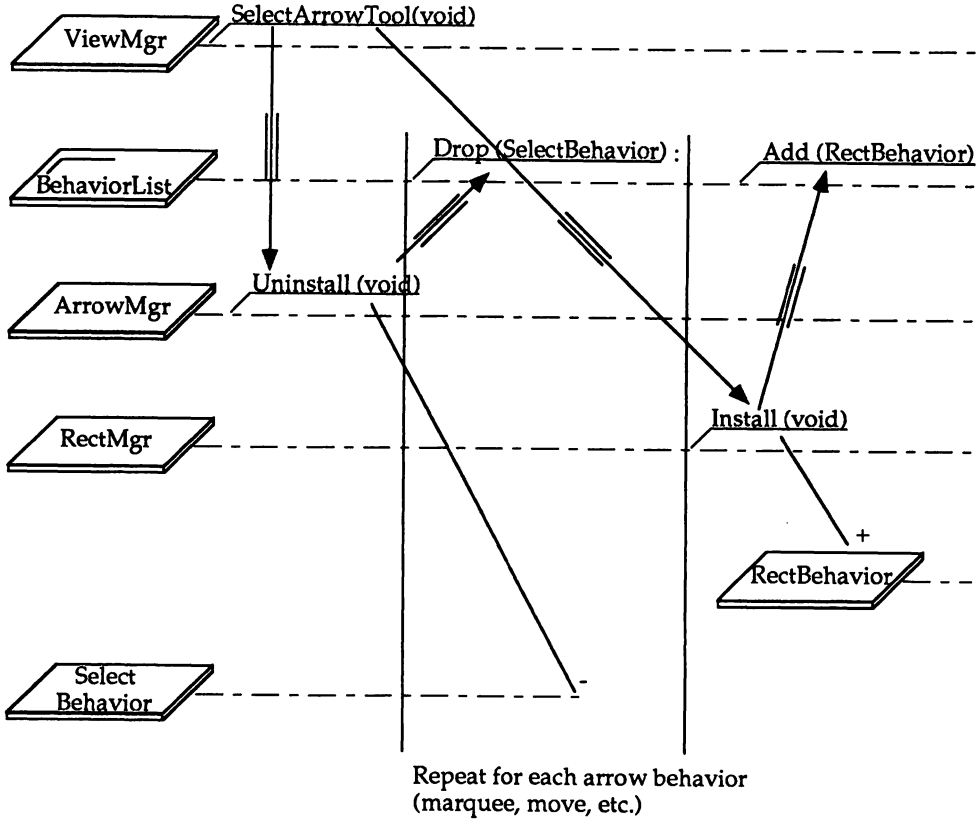


Figure 11-17. Selecting the rectangle tool

technique with single-inheritance languages, but carries the same baggage as the second variation in addition to imposing an extra level in the class hierarchy. Single-threaded inheritance is also a cheap way to change the behavior of a whole class hierarchy with minimal changes to off-the-shelf classes. For example, if you want all descendants of MacApp's TEvtHandler to record in a file all events received and their disposition, you can transform the hierarchy in Figure 11-18(a) into that of 11-18(b).

The only change to the original TEvtHandler is its name. A new subclass has been slipped in between the old TEvtHandler and the remaining classes in your program, with the new capability built in. You would never dream of designing such a monstrosity from scratch, but you are stuck with an off-the-shelf hierarchy and instructions from your boss to do minimal or no damage to the original MacApp source. Even if it were

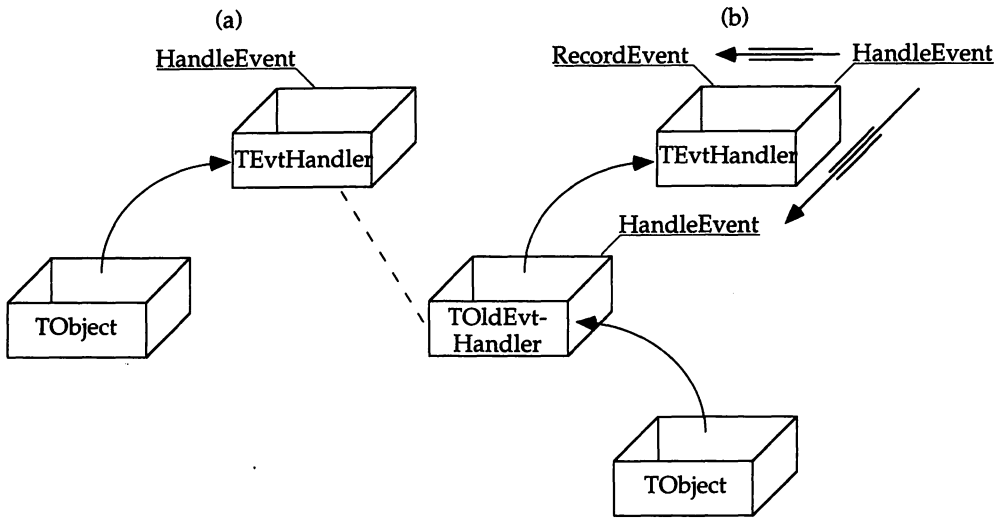


Figure 11-18. Redirecting inheritance in TEvtHandler

available, multiple inheritance would not help here. Hold your nose and use single-threaded inheritance.

Here are some specific rules for use of each technique.

1. First variation—Two-tiered hierarchy—Works with multiple inheritance if the shared properties are referenced by the non-shared methods. If they are independent of the non-shared methods, split abstractions should be used instead (next section). If only single inheritance is available, this technique is obviously not available.
2. Second variation—Mash the abstractions together into a single class. If multiple inheritance is available, the rule is simple: Don't do it. If you are tempted anyway, take a walk, sip some herbal tea, then come back and use some other technique. With single inheritance, make sure that the separate abstractions are not going to be reused anywhere else. In other words, look for other run-time objects that conform to one, but not both, abstractions. If you find any, don't use the mashing technique.
3. Single-threaded inheritance—Use with either single or multiple inheritance to minimize changes to an off-the-shelf class library. In all

other cases where multiple inheritance is available, this is a poor way to implement any architecture the authors can think of. With single inheritance, the technique can be used sparingly to cut down on the number of objects created by eliminating helper objects, knowing that reuse will suffer as a result. There are times when the number of objects at run time is a significant factor in performance, particularly the allocation and deallocation of memory for the objects. Eliminating helper objects can then become an important implementation goal.

Splitting Abstractions

Splitting abstractions requires implementing the shared properties in their own abstract class, then using multiple inheritance to combine the shared and non-shared properties in the concrete class, as shown in Figure 11-11. This achieves the same objective as creating a hierarchy of abstractions (Figure 11-10), but does not create an extra layer of inheritance. This can be very useful if the shared methods and attributes are not referenced by the non-shared methods: The abstract class representing the overlap may prove highly reusable. However, if the shared methods and attributes are referenced by the non-shared methods, the result can be something that is strongly discouraged in object-oriented programming: making inheritance from one class dependent on also inheriting from another class. Consider Figure 11-19, in which one of the original abstractions is now used to implement yet another concrete class. The new concrete class must specify inheritance from two superclasses. If any of the combined abstraction techniques had been used, inheritance from only one superclass would have been required, as in Figure 11-20.

It is generally a good idea to minimize the levels of inheritance, but not at the expense of creating complex networks of dependencies, in which inheriting from a single class no longer works without also inheriting from

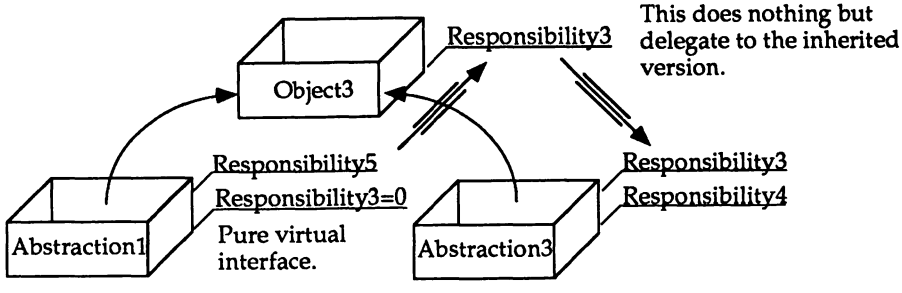


Figure 11-19. Would you buy a used car from this programmer?

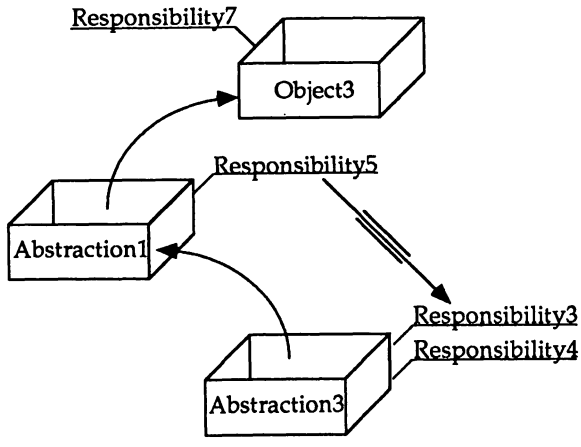


Figure 11-20. Use of a two-level abstraction

or using helper objects of other classes. This issue of dependencies in a class hierarchy is complex and will be addressed further in a moment.

Combination Strategies

Combinations of different ways to implement abstractions can be used to address special cases. For example, there may be overlapping abstractions in which only one of the abstractions references the shared properties from the non-shared properties, or in which one, but not both, of the abstractions will be reused elsewhere. Copy/paste strategies combine well with all of the other techniques, particularly during prototyping.

▶ Object-Oriented Software Engineering Using Inheritance

Interwoven with all of the practical rules we have just laid out are the principles of object-oriented software engineering discussed in Chapter 9.

- Limit responsibilities
- Limit data knowledge
- Limit implementation knowledge
- Limit relationships

To these four we add a fifth objective, which applies to the use of inheritance.

- Limit type knowledge; that is, make clients of an object unaware of whether they are dealing with a subclass or the superclass by which they know the object.

Of this list, three are important objectives in laying out the class hierarchy: limiting data knowledge, limiting relationships, and limiting type knowledge. Limiting responsibilities and implementation knowledge is more a function of decisions you make in the Technology and Execution Planes.

So far, we have discussed these limits in terms of objects, not classes, but you should get used to thinking of each class as a module, separate from its superclasses and subclasses. The separation is not as strong as with distinct run-time objects, but many of the same principles apply. For example, attributes and their use can be partitioned using a class hierarchy. Attributes declared in a class can be hidden from subclasses and superclasses by also placing the methods that use the attributes within the class. This limits data knowledge through inheritance, one of our three objectives. Relationships are the most important factor in code reuse, both within a project and across projects; the more relationships a class has, the less reusable it is. If you arrange your class hierarchy so that the upper-level classes have only attributes, interfaces, and implementations and few or no relationships, then implement relationships in lower-level subclasses, the higher-level classes will probably be reusable, even if the lower-level ones are not. Limiting type knowledge is a critical objective in designing a class hierarchy. We will explore this subject in some depth in a moment; for now, we simply observe that limiting type knowledge can be tricky when using inheritance and almost impossible without inheritance.

Although the use of inheritance is often perceived as one of the black arts of the expert, we can quickly clear up some of the mystery by placing inheritance in a clear theoretical framework. This involves revisiting just what it is that a class inherits from its superclasses and looking at two fundamentally different uses of inheritance, *normal* and *non-normal*. We will conclude with a final look at a somewhat controversial assertion about inheritance: Object-oriented programming can be a step *backward* in software technology when inheritance is not used properly.

▶ What Does a Class Inherit?

The traditional view of inheritance simply observes that a subclass inherits methods and attributes from its superclass(es). As we are about to demonstrate, this is an accurate but woefully incomplete perspective. Consider the following fragment of code.

```
class foo {
  ...
  };
class bar {
  private:
    foo *aFoo;
  ...
  };
class subbar : public bar {
  ...
  };
```

The traditional view states, "`subbar` inherits the attribute `aFoo`, which is a pointer to an object of class `foo`." In reality, this should state, "`subbar` inherits one or more *relationships* with an object of class `foo`." An attribute that holds the address of another object is not the same as an integer, string, or other piece of data. It represents structural and/or behavioral relationships between two objects. These relationships are inherited along with methods and other kinds of data. Now consider the following abstract class.

```
class drawthing {
  public:
    virtual void Draw (void) = 0;
  };
class square : public drawthing {
  public:
    virtual void Draw (void);     // Draws a square
  };
```

The seemingly trivial class `drawthing` may, in fact, form the top of an entire class hierarchy. When the class `square` inherits from `drawthing`, just what is it inheriting? There are no attributes, no relationships, and no methods, the sum total of what one normally thinks of in connection with inheritance. What is inherited is the interface to `drawthing`, which in this case means the interface to the method `Draw ()`. We need to be careful in distinguishing what we mean by inheritance of a method; it might mean the interface, the implementation, or both.

This alters our view of inheritance. Instead of simply inheriting methods and attributes, a subclass may actually inherit any of the four features shown in Figure 11-21: data, interface, implementation, and relationships. When designing inheritance, all four must be taken into account.

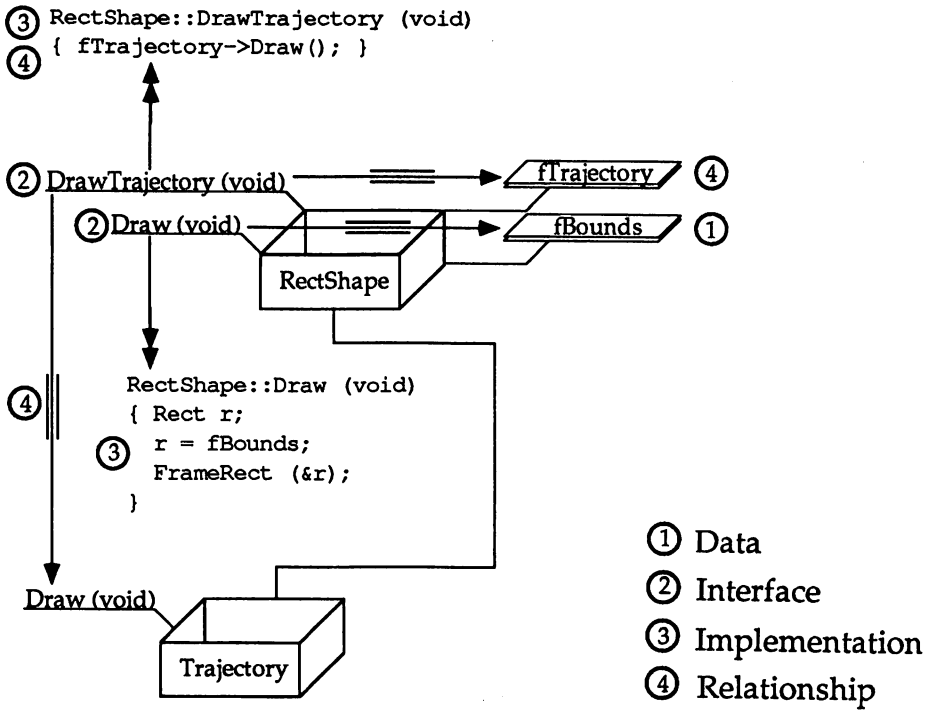


Figure 11-21. Four aspects of inheritance

One of the authors recently came across a good example of this principle. When designing a MacApp 3.0 program, he implemented a series of behavior objects, as described above for the rectangle drawing program. These objects had little in common except for their relationships to the display container (view, in MacApp parlance). He created an abstract class that had nothing but the address of the display container and some simple methods implementing the simple relationships. The subclasses did all of the real work. This is an example of using inheritance to reuse relationships and nothing else. The resulting program had the structure shown in Figure 11-22.

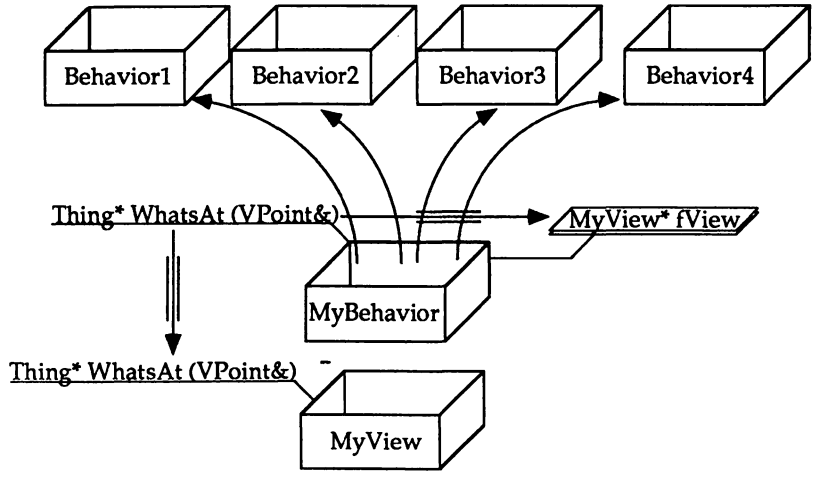


Figure 11-22. Example of using inheritance of relationships

▶ Normal Inheritance

Let's assume for a moment that we have a method in a superclass that has an implementation; that is, it is not a pure virtual method. There are two ways to override such a method in a subclass: inherit its interface only, providing a completely different implementation, or inherit the implementation as well as the interface by calling the superclass implementation from within the subclass's override. In C++, this means that somewhere in the implementation of the override, you place a call to `SuperClass::Method();`, in Pascal, `INHERITED Method;`, where `SuperClass` is the name of the superclass and `Method` is the name of the overridden method. When a method calls its overridden, inherited counterpart, we say that the inheritance is normal. If the implementation is completely overridden, the inheritance is non-normal. Since a pure virtual method has no implementation, all overrides of pure virtual methods are considered normal. We can extend this definition to inheritance of entire classes: If all overridden methods of a subclass inherit normally, the subclass as a whole inherits normally. This seemingly simple distinction between normal and non-normal inheritance is actually one of the most important factors in achieving the Four Itys.

Consider the inheritance shown in Figure 11-23.

In this scenario, the class YJunction inherits from the class Track. There is a display container class, LayoutView, which is a client of the superclass Track. When you subclass Track, you must not only make sure that the

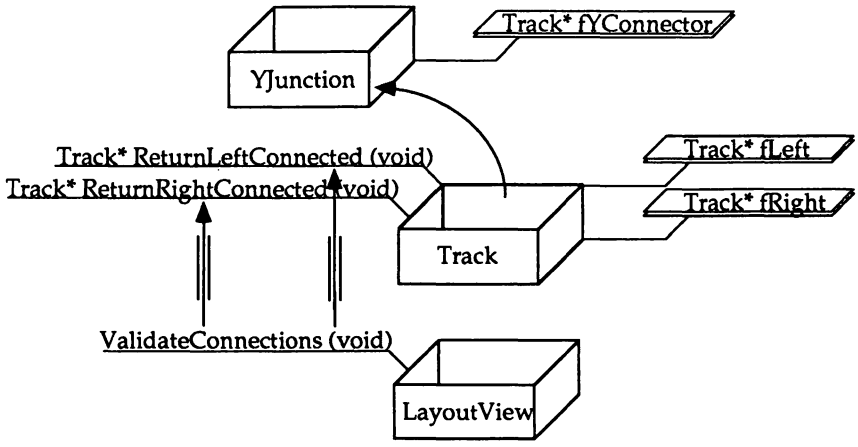
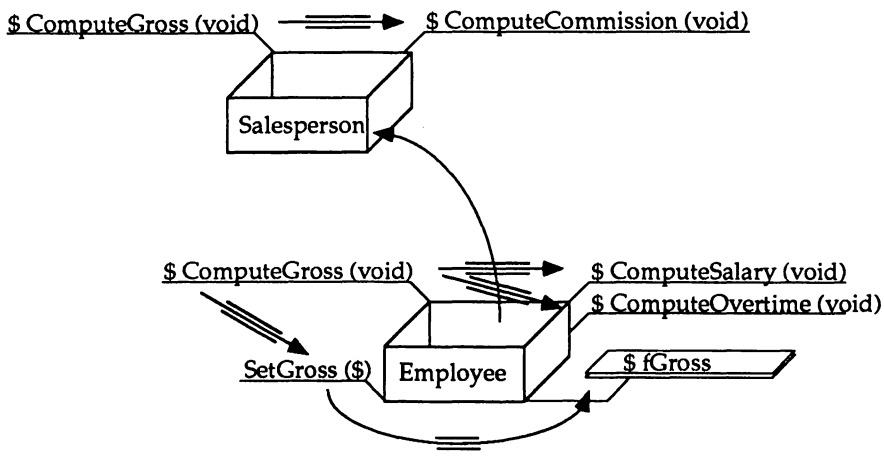


Figure 11-23. Simple inheritance—or is it?

class YJunction produces the right run-time objects; you must also make sure that any clients of Track, such as the one shown, do not misbehave when presented with a subclass like YJunction. In this example, suppose we asserted that Track had relationships to two other instances of Track, a "left-hand" instance and a "right-hand" instance, on the assumption that most members of Track are straight pieces with two connectors. Those relationships could, in theory, be completely overridden in the subclass YJunction, which has three connectors. This, in turn, might upset a client that relies on the two relationships of the superclass. LayoutView might, for example, try to walk through the track by proceeding left-then-right from any given piece of track. This may or may not be good design for that particular task, but that is beside the point. If you use non-normal inheritance, at best you will end up spending a lot of time scratching your head in designing the clients, convincing yourself that everything will work properly. At worst, clients will have to constantly ask themselves, "Just which subclass am I dealing with now?" This kind of *type knowledge* is to be avoided wherever possible. It leads to very poor modularity, little or no code reuse, and very confusing, unstable designs.

The problem gets worse if a method of the superclass that is overridden by the subclass has side effects. Figure 11-24 shows just such a situation. The problem is now not just maintaining relationships with other objects, but maintaining the very consistency of the object itself! Furthermore, anyone trying to understand what this program is doing cannot tell what is going on without bouncing up and down the inheritance tree. This problem has been called the "yo-yo" phenomenon of inheritance.



The superclass version of ComputeGross has a side effect - setting the value of fGross - not shared by the overridden version. Other parts of the program that rely on that side effect will not work with the subclass, requiring type knowledge to compensate. The override is suspect, but the real problem starts with the design of the original ComputeGross, which does not lend itself to normal inheritance.

Figure 11-24. Calling an overridden method from the superclass

These problems are reduced, though not always eliminated, by using normal inheritance. If each overridden method is required to call its superclass counterpart, there is a good degree of assurance that relationships and behaviors of the superclass expected by clients will be maintained. There may be additional behaviors in the subclass, but there are no big surprises for clients of the superclass. It is also unlikely that the superclass or its clients will need to use type knowledge in order to carry on their business; that is, they will not need to have code like the following.

```
if (x->GetType() == kFoo)
    do_this();
else if (x->GetType() == kBar)
    do_that();
```

and so on. This use of type knowledge takes many subtle forms: returning an integer type code of some sort, directly asking for the class identifier, or asking for other information from the object that controls the sort of

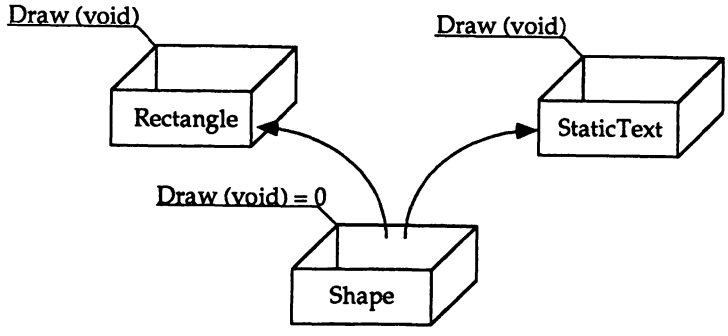


Figure 11-25. Using a pure virtual interface to achieve normal inheritance

`if-then` or `switch-case` logic seen above. One of the most important objectives in designing a class hierarchy is to avoid this kind of type knowledge in your program. Normal inheritance is a very important tool in achieving that objective.

One key strategy in achieving normal inheritance is to make liberal use of pure virtual methods in abstract superclasses. Suppose that you have two run-time objects that differ only in the way they draw; say, a rectangle and a piece of text. Figure 11-25 shows the correct way to handle this by creating an abstract class from which both can inherit the same interface. All methods other than Draw are either not overridden or are inherited normally. The Draw method is completely different for the two run-time objects, so the only abstraction that can be formed is the interface.

Figure 11-26 shows a common mistake in handling this situation. Here, the text class inherits from the rectangle class. The Draw method is completely overridden by the subclass, since they have nothing in common. This non-normal inheritance may seem benign at first glance, but it

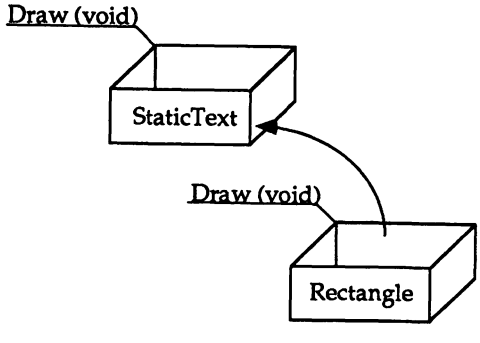


Figure 11-26. Non-normal inheritance for no good reason

isn't. Suppose the superclass version uses relationships with other objects to carry out its purpose, such as asking some other object for its size? Suppose we want to change the implementation of the superclass version of Draw so that side effects are introduced. Can we do so by looking only at that class, or must we consider all subclasses at the same time? What if Draw is called from other objects, expecting the standard behavior provided by the superclass? The Four Itys run for cover in these situations. It is usually easy, and well worth it, to find ways to turn non-normal inheritance into normal equivalents.

The most defensible reason to use non-normal inheritance is that we are stuck with an existing class which is unusable otherwise. This may be an indication of poor design, or it may mean that the program was designed for one set of requirements and must now be changed minimally to add additional capabilities. Though it is undesirable, non-normal inheritance may be unavoidable under such circumstances. Another reason might be the use of a single-inheritance language. Mashing two abstractions together into a single superclass is most likely to make sense when you are stuck with single inheritance. When that happens, it is inevitable that some inheritance will end up being non-normal. If you really must use non-normal inheritance, try to limit it to methods that do not use relationships with other objects, are not called directly from outside the superclass, and do not have side effects such as changing the values of attributes.

▶ Inheritance: The GOTO of the '90s?

The best way to view a class hierarchy is as a set of superimposed modules. Modularity in object-oriented software can be achieved through creating separate, collaborative objects, but it can also be achieved by dividing the data, relationships, implementations, and interfaces into superclasses. As with any technique of modularity, simply forming modules is not enough to guarantee good design. Modules must be independent of one another, simple, reusable, and maintainable in order to provide benefits. This means that classes should be as independent of one another as possible, not just across relationships, but up and down their inheritance structure as well. Subclasses will always depend on their superclasses, but it is surprising how tempting it can be to make superclasses dependent on their subclasses! Type knowledge and the yo-yo phenomenon lead to just such violations of modularity through inheritance.

Back in the language wars of the Dark Ages of Computing, circa 1970, the most telling argument against using GOTO was that it prevented you from looking at a line of code and telling exactly how your program got

there. No matter where you were, the program could suddenly swoop in out of nowhere. Poorly designed inheritance has the same effect. Looking at a method, it may be difficult or impossible to tell without examining the entire program how, when, and why the program might execute that method. For this reason, inheritance in object-oriented programs—particularly non-normal inheritance—might well come to be known as the GOTO of the 1990s.

▶ Programming

The implementation of methods and attributes is almost an anticlimax in Solution-Based Modeling. For each method, the semantics of the method and its collaborators have already been determined. Even the order and circumstances in which collaborators are called by a method have in many cases been defined through dynamic scenarios of the Execution Plane. Furthermore, it is likely that by the time you reach the Programming Phase a good deal of prototypical code has been implemented. What follows are some useful guidelines.

1. Write concrete classes for specific examples of objects first. This is true of prototyping and the same strategy continues into the programming phase. Worry about abstract classes and optimized class hierarchies only after you have working, concrete classes.
2. Dare to reorganize. At any point during the programming phase, you should be prepared to reorganize the class hierarchy and the accompanying implementations as new ideas and information are uncovered. At a recent conference, one company told of overhauling a MacApp application containing in excess of 150,000 lines of code. It took less than three days. This is very common in object-oriented software development and not to be feared. You lose very little code during a reorganization; most ends up copy/pasted somewhere in the revamped program.
3. Look for shrinkage in lines of code. In traditional programs, the number of lines of code grows as the project goes forward. In object-oriented software, the number of lines of code expands to a point, then starts shrinking as completion is approached. This is due to better reuse of code and elimination of type knowledge as the design is optimized for final release. If the number of lines stabilizes or starts to shrink, you are on the home stretch.
4. Keep up the paperwork. You will uncover new information during programming, some of which will undermine scenarios previously

drafted. Be sure to keep the scenarios up to date, despite the tendency to take a "damn the torpedos, full speed ahead" attitude as the project winds down. This will help prevent oversights that occur when everyone is in the heat of programming and assures that quality assurance and on-going maintenance will have a solid base of documentation.

5. Implement scenarios, not classes. We have discussed this several times before: Design and implementation should be organized around scenarios, not objects or classes. It is important to maintain control over the programming phase, and that is best done by working from scenarios that can be checked off against a project plan as they are completed.

One of the most common forms of reorganization of the code is illustrated by the following sequence. Start with the following code fragment (in pseudo-code).

```
class foo {
  public:
    void Draw (void);
  };
void foo::Draw (void) {
  // This may actually be several lines of code
  do_some_of_the_drawing;
  do_the_rest_of_the_drawing; // As may this
}
```

Now introduce a superclass and split the implementation as follows.

```
class bar {
  public:
    virtual void Draw (void);
  };
class foo : public bar {
  public:
    virtual void Draw (void);
  };

void bar::Draw (void) {
  do_some_of_the_drawing;
}
```

```
void foo::Draw (void) {
  bar::Draw();
  do_the_rest_of_the_drawing;
}
```

In Object Pascal, the initial form would be as follows.

```
foo = OBJECT
  PROCEDURE Draw;
END;
PROCEDURE foo.Draw;
BEGIN
  { This may actually be several lines of code }
  do_some_of_the_drawing;
  do_the_rest_of_the_drawing;  { As may this }
END;
```

The reorganized code then becomes

```
bar = OBJECT
  PROCEDURE Draw;
END;

foo = OBJECT (bar)
  PROCEDURE Draw; OVERRIDE;
END;

PROCEDURE bar.Draw;
BEGIN
  do_some_of_the_drawing;
END;

PROCEDURE foo.Draw;
BEGIN
  INHERITED Draw;
  do_the_rest_of_the_drawing;
END;
```

Notice that no code was lost. The implementation was simply split up, with one part assigned to the superclass and the rest left in the subclass. This allows the initial part, `do_some_of_the_drawing`, to be reused by other subclasses. An equivalent reorganization follows.

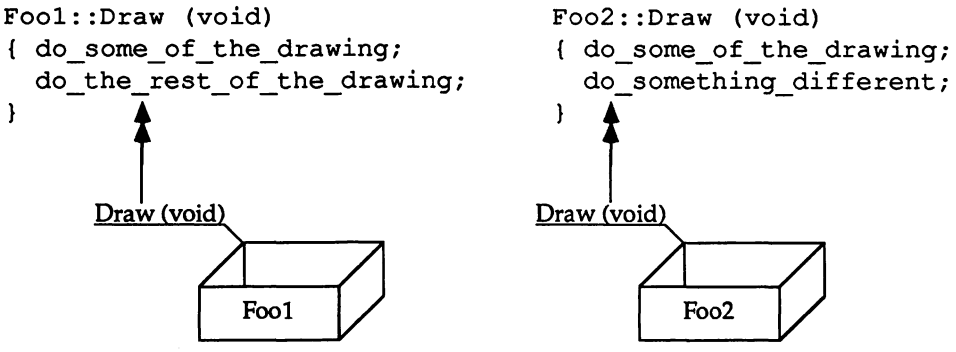
```
class bar {
  protected:
    void PartialDraw (void);
  public:
    virtual void Draw (void) = 0;
  };

class foo : public bar {
  public:
    void Draw (void);
  };

void bar::PartialDraw (void) {
  do_some_of_the_drawing;
}
void foo::Draw (void) {
  this — PartialDraw();
  do_the_rest_of_the_drawing;
}
```

Or, in Object Pascal,

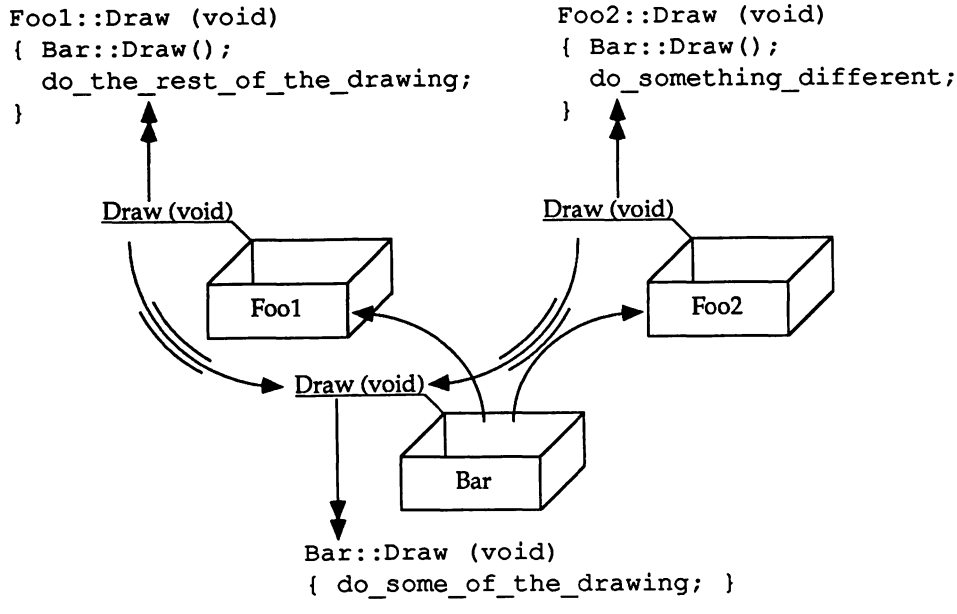
```
bar = OBJECT
  PROCEDURE PartialDraw;
  { A pure virtual method - implementation does nothing}
  PROCEDURE Draw;
END;

foo = OBJECT (bar)
  PROCEDURE Draw; OVERRIDE;
END;

PROCEDURE bar.Draw;
BEGIN
END;

PROCEDURE bar.PartialDraw;
BEGIN
  do_some_of_the_drawing;
END;
```

```
PROCEDURE foo.Draw;
BEGIN
  SELF.PartialDraw;
  do_the_rest_of_the_drawing;
END;
```

Figure 11-27
(a) Before optimization



(b) Putting common code into the overridden method



(c) Splitting out common code into a separate method

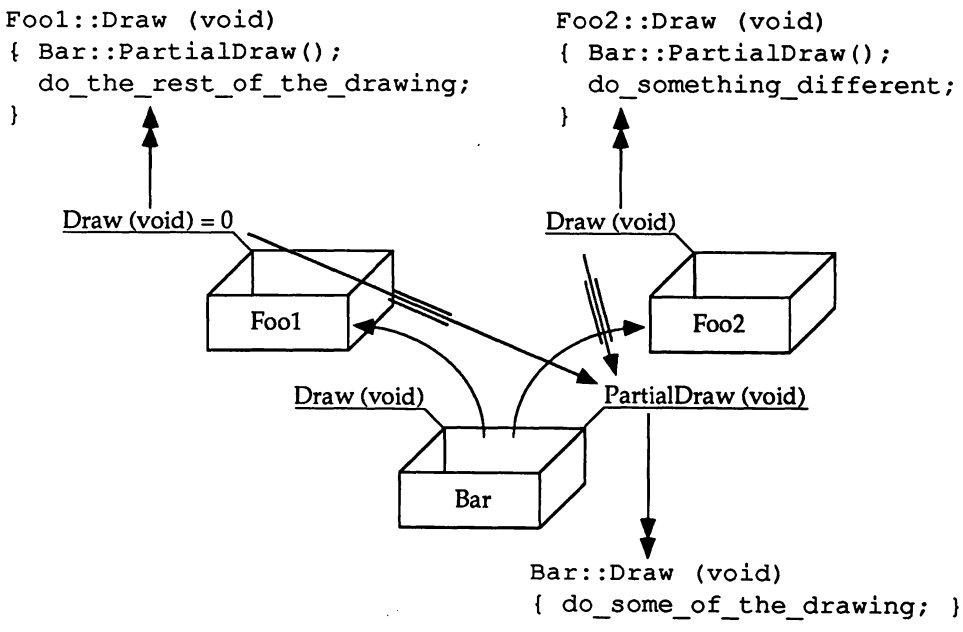


Figure 11-27. Reorganizing the implementation to share code

Figure 11-27 (a) shows the "before" scenario and (b) and (c) show the two alternative "after" scenarios for these code fragments extended to illustrate code reuse by two concrete classes.

Both approaches use normal inheritance. The second is a little cleaner because it separates the inheritance of an interface from the inheritance of a partial implementation, but the authors have no quibble with either approach. In both cases, if you measure the magnitude of the reorganization by lines of code affected, you must conclude that almost everything in the program has changed. On closer examination, it is clear that almost nothing in the implementation really changed; it was the class hierarchy that got shuffled. This is very common, particularly as the program nears completion and you start worrying more about optimization and less about whether the project will finish in time. This sequence is especially common if you heed the previous guidelines and work first on examples of objects, then later design the class hierarchy. The tendency under those circumstances is for code to migrate up the hierarchy, not down, as in this example.

We can illustrate the third guideline by continuing the example. It may well be that `do_some_of_the_drawing` was repeated at first in several classes and that the reason for creating the extra layer of inheritance was specifically to share that code. If so, the net effect is likely to be a reduction in the total lines of code. Or it may be that by introducing a pure virtual interface, you eliminate some `if-then` or `switch-case` logic, again reducing the overall size of your program. The combination of these two factors explains much of the reduction in code experienced by companies that have reengineered existing software products using object-oriented techniques. The resulting programs have typically been one-fourth to one-tenth the number of lines of code used by the original program.

▶ Managing the Programming Phase

The programming phase uses the same basic techniques used for analysis and design: CPC, scenarios, synthesis, correlation, and synchronization. The process is driven by the set of scenarios chosen at the conclusion of the design phase to form the basis of project scheduling and measurement, the implementation set. Each scenario in the set must be implemented at some point during the programming phase. Choosing an appropriate order in which to tackle these scenarios is not as important as making sure that all are dealt with and that the cumulative quantity completed is consistent with the schedule at any given point in time.

This chapter discusses techniques specific to object-oriented programming and SBM, but you will certainly use traditional tools of programming as well: source code control systems such as the MPW Projector tools, conventions for source code formatting and organization, debuggers and the accompanying standards for initial testing of source code, and other useful development tools and techniques. SBM does not replace any of these techniques, but they are beyond the bounds of this book.

▶ Use of Scenarios

In Chapters 9 and 10, we talked about using scenarios, not just as a tool for model building and exploration, but for project management as well. The schedule for the programming phase should be based on a specific set of scenarios chosen from all scenarios available at the end of the design phase. This *implementation set* of scenarios should collectively cover all of the Execution Plane. No scenario in the set should be completely redundant with any other; when in doubt, include borderline scenarios in the set. It is perfectly acceptable to generate new scenarios from existing ones, specifically for inclusion in the implementation set.

The implementation set is the fundamental unit of organization during the programming phase. Schedules should be based on completion of scenarios or groups of scenarios. Team members should be made responsible for completing the implementation of scenarios or groups of scenarios. This is a much better basis for scheduling, assigning tasks, and monitoring progress than other schemes based either on a top-down decomposition of the project or by object/class. A good way to develop the programming schedule is to pick a few representative scenarios from the implementation set and code them, keeping track of productivity as you go. This can then be extrapolated to the rest of the implementation set.

▶ Quality Assurance

The process you have used to get this far—scenarios, calibration, and an overall team approach—means that testing and quality assurance began with the start of the project. However, that is no substitute for a formal software testing process at the completion of the programming phase. Ideally, QA personnel have been part of the entire project, from analysis through programming. In the early stages, QA got up to speed on the requirements, saving time later; provided feedback on how testable the scenarios drafted for the Business and Technology Planes are; and began selecting scenarios of those planes to be the basis for the final check-out testing. If you did not bring QA in at earlier stages, get them involved now, as soon as the programming phase begins. They will have a good deal of preparation to do before the software is released for its final check-out.

▶ Use of Prototype Code

Prototype code developed in early phases should not be treated as sacred. Remember that 75 percent of your investment in developing that code was in analysis and design, not programming. Recreating hastily drafted, proof-of-concept code is much less expensive than was the initial drafting. That said, it is likely that much or most of the prototypical code accumulated so far will end up being used in the finished product. It may be shuffled around, as discussed previously, but there is usually a place for good code in the finished product.

▶ When Is the Programming Phase Complete?

The programming phase ends when the program implements the scenarios of the Execution Plane, has been accepted by QA, and has been released for alpha and beta testing. This requires that someone in authority

says that the phase is done; there is no rigorous, air-tight test for completion. However, if QA has selected scenarios of the Business and Technology Planes to use as the basis of testing, and if you have properly calibrated the Execution and Program Planes, there is a firm foundation on which to render that judgment.

▶ Beyond Programming

What happens after the project T-shirts have been issued and everyone is back from a well-needed vacation? The software, of course, must evolve. So far, we have behaved as if every SBM project starts from scratch or from software not developed using SBM. However, you are now ready to reap one of the biggest benefits SBM has to offer: carrying forward the body of scenarios and documents into the full life cycle of the software. When we said earlier that the Reference Model was simply one region of the Business Plane, we described only initial use of SBM, not changes to a prior SBM project. If you are maintaining and changing a program developed using SBM, the true Reference Model is actually the combination of all four planes—the Solution Model, Technology Plane, Execution Plane, and Program Plane—as of the last release of the software. As you work toward the next release, you will correlate this entire model to its counterparts in the new SBM in exactly the same way you correlated the Reference and Solution Models at the beginning of this project. However, this correlation will be at four levels, not one. The concept of an Impact Analysis also carries forward. Where before the Impact Analysis was a measure of change to the business, now it is a measure of change to the business, conceptual design, architecture, and program.

This explains some of the curious, glossed-over inconsistencies in the Business Plane. Why was it that correlation applied only across planes, except between the Reference and Solution Models? The answer is that the Reference Model was actually at the top of a four-tiered, separate model, the bottom three tiers of which were, at that time, empty. Why is it "Solution-Based Modeling," not "Reference-Based Modeling," since we encouraged you to start with the Reference Model? Again, the true solution-based model has only one region on its topmost plane, the Solution Model.

By recasting the overall model in this way, evolution is seen as a natural outgrowth of what you've been doing during the initial development. The same techniques and principles apply over the useful life of the software.

We can now also, in the closing pages of the book, close one other open issue. If you are developing several, cooperative programs at once, each gets it own complete solution-based model. These models share all or part

of a unified Solution Model, but underneath can be treated as separate projects. With this change, SBM can be easily extended to cover distributed systems and multitasking networks of programs. It is beyond the scope of this book to provide more details in this area, but the extensions should be quite natural to anyone who has come this far.

▶ Summary

This chapter focused on the programming phase of Solution-Based Modeling, the final step in constructing the initial version of the system envisioned in the Solution Model. Since we are developing systems for the real world, this initial system is actually the beginning of an evolving series of systems that will satisfy your business needs. This is the easiest phase of SBM, because we simply code the responsibilities defined in minute detail in the Execution Plane to obtain the Program Plane.

- During the programming phase we spend a great deal of time deciding how to properly implement the objects of the Execution Plane using inheritance and other techniques. We have many strategies to choose from, including copy/paste and other "low-tech" approaches. Our choice of how to implement the abstractions and run-time objects of the Execution Plane as class libraries and program modules will greatly influence how well our application system satisfies the Four Itys, Maintainability, Reliability, Extensibility, and Reusability. These are best achieved by thinking of inheritance as passing along relationships, interfaces, and implementations and not just attributes and methods.
- Implementation of methods and attributes is straightforward and uses the same techniques and technology as for any other programming project. Object-oriented programs are frequently reorganized, even up to the closing days of a project. Prototypical code is often, but not always, reused in the finished product.
- When our software has been tested, passed an independent quality assurance process, and survived an appropriate, prescribed period of user testing, we reap the reward of SBM: an application system designed to serve the business needs set out in the Solution Model. From that point, the entire Solution-Based Model becomes the new Reference Model for subsequent work, thereby closing the life cycle on the methodology.

Appendix
A Manual Database for Solution-Based Modeling

This appendix describes a simple database that both facilitates all three forms of calibration and functions as an index to the set of scenarios. It can be maintained manually on paper or index cards but is also easy to automate using HyperCard or database programs. To use this database, each scenario should be assigned a unique identifier, typically a document number. For each object, category, or class, we create a *summary card* and for each responsibility, a *relationship card*. A sample summary card is illustrated in Figure A-1.

Plane/Region: _____ Scenarios: _____

Element Name: _____ _____

Attribute Name	Data Type	Scenarios

Figure A-1. Summary card

The plane, region, and name together uniquely identify the object, category, or class. The upper right of the card lists all scenarios in which the object appears. The table lists all attributes, their data types, and the scenarios for each. Figure A-2 shows a sample relationship card.

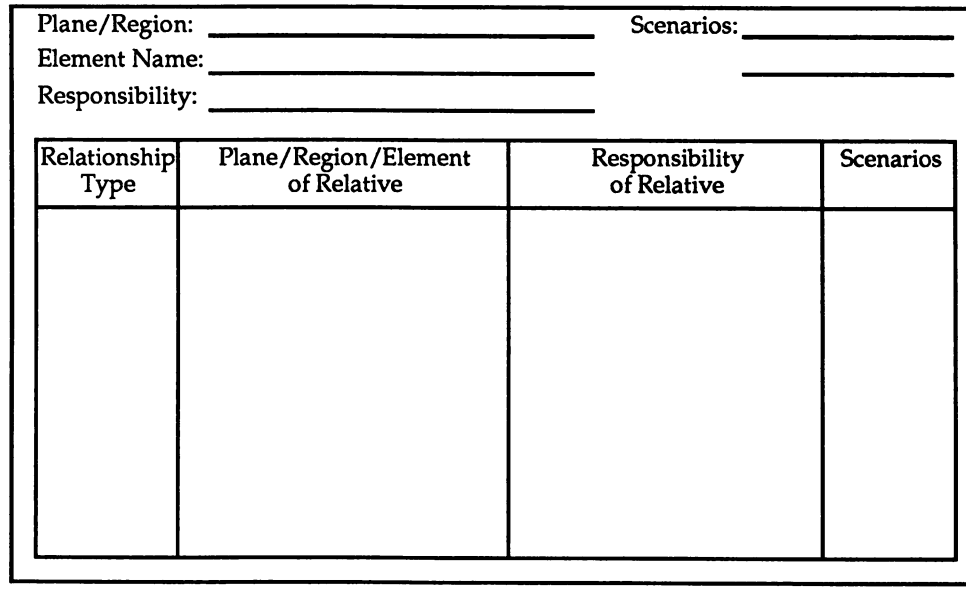


Figure A-2. Relationship card

The plane, region, and name are as in the summary card. Many relationships connect a responsibility of this object to another object. If that is the case, the name of the responsibility is listed in the heading. The table lists all relatives of this responsibility. The second column identifies the object. The third column identifies the responsibility of the relative, which may be blank. The relationship type is one of the VDL relationship symbols. ⟹ means "calls" while ⟸ means "called by" the relative. ⊥ means "creates" and ⊥ means "created by" the relative. The last column lists scenarios that contain the relationship. The scenarios listed in the upper right corner are those in which the responsibility occurs without relationships.

For a given element, there will be one relationship card with the responsibility left blank for relationships such as "created by" or "is part of," plus one card for each responsibility. The former is called the open relationship card and has special significance for synchronization. A relationship is

always listed in two places, corresponding to the element on each end. Responsibilities in the Execution and Program Planes are listed with complete calling sequences in addition to the name.

This is a simple and effective cross-reference to synthesized scenarios. Its use for synthesis is obvious. It is the "overall model" we spoke of in Chapter 9. For correlation, scan each relationship card for "implements," "implemented by," and "replaces" relationships. For synchronization, the open relationship cards quickly identify scenarios in which objects are created and destroyed. Dangling threads are indicated with an asterisk or stick-on, color-coded dot. Cards should be maintained either in pencil or on-line, as the information changes frequently.

All processes of Solution-Based Modeling are supported by this simple database. More sophisticated database programs can provide automated support for all three forms of calibration.

Bibliography

To help the reader, we list only books that we think are especially beneficial or that have been of special help in our own research. The bibliography is divided into three general subjects: software development, cognitive science, and graphic arts.

Software Development

Many books on object-oriented analysis and design are so at odds with the authors' views on the subject that it would be misleading to list them as part of this bibliography. The books that are listed are complementary, though not perfectly consistent in their views.

Object-Oriented Design

Booch, G. 1991. *Object-Oriented Design with Applications.* Redwood City, CA: Benjamin/Cummings Publishing Company. Packed with principles and insights, though it contains little methodology and falls into the objectivist trap. You don't need any other book on object-oriented design. You also don't need another bibliography on object-oriented software; Booch's is excellent.

▶ Object-Oriented Analysis

Rumbaugh, J., et al. 1991. *Object-Oriented Modeling and Design.* Englewood Cliffs: Prentice Hall. One of the few other good attempts at an overall methodology for analysis, design, and programming called Object Modeling Technique (OMT). Takes a data-driven, not procedural, approach.

Wirfs-Brock, R., et al. 1990. *Designing Object-Oriented Software.* Englewood Cliffs: Prentice Hall. Perhaps the best work available on the concept of responsibility-driven design. This book is about more than design. It discusses modeling and analysis as well, and only occasionally succumbs to objectivism.

▶ General Object-Oriented Software

Carr, R. and D. Shafer. 1991. *The Power of PenPoint.* Reading, MA: Addison-Wesley. OK, this has nothing to do with the Macintosh, but in many ways PenPoint, the operating system from GO Corporation, shows the potential for object-oriented software. Every well-rounded OOPer should at least peruse PenPoint, even if you never use it.

Kim, W. and F. H. Lochovsky. 1989. *Object-Oriented Concepts, Databases and Applications.* New York: ACM Press/Addison-Wesley. One of the best collections of papers on the fundamentals of object-oriented software.

Winblad, A. L., S. D. Edwards, and D. R. King. 1990. *Object-Oriented Software.* Reading, MA: Addison-Wesley. An outstanding who's who and what's what review of the object-oriented software industry.

Zdonik, S. and D. Maier. 1990. *Readings in Object-Oriented Database Systems.* Palo Alto, CA: Morgan Kaufmann. If you wade through this book and still believe that there is such a thing as a single definition of object-oriented programming, check the cover because you grabbed the wrong book. This huge volume contains so many academic papers so at odds with one another on such fundamental subjects that you have to wonder how object-oriented programming ever got past the starting gate.

▶ General Software

Martin, J. and C. McClure. 1988. *Structured Techniques: The Basis for CASE.* Englewood Cliffs: Prentice Hall. A remarkable book that attempts to synthesize all that is common to software methodologies, with good

success. (Note that we use the word "common," not "correct.") Must reading for any serious methodologist.

▶ Cognitive Science

We list many works here in the hope that you will read a few of them. Most technologists are not familiar with the tremendous body of work available in this area. We list works according to their relevance to this book as well as their readability for a general audience.

▶ Gregory Bateson

Gregory Bateson, anthropologist, philosopher, and cognitive scientist long before the term came into use, has had a major impact on many people, including the authors. His books are not easy reading, but well worth the effort. His contribution to twentieth-century thought cannot be overstated, particularly regarding the relationship of man to his own concepts and environment.

Bateson, G. 1972. *Steps to an Ecology of Mind.* New York: Chandler.

Bateson, G. 1980. *Mind and Nature: A Necessary Unity.* New York: Bantam Books.

Bateson, G. 1980. *Naven.* Stanford CA: Stanford University Press.

Bateson, G., and M. Bateson. 1987. *Angels Fear: Towards an Epistemology of the Sacred.* New York: Macmillan Publishing Company.

▶ Of Particular Interest

We have found the following books particularly useful in helping us understand how people perceive the world around them. They have made significant contributions (both positive and negative) to our approach to categorization, how people build models, the calibration process, and the interaction between language and perception.

Alexander, C. 1979. *The Timeless Way of Building.* New York: Oxford University Press. Although computer people often refer to themselves as architects, Christopher Alexander *is* one, and the issues he discusses, related to architecture, are remarkably similar to the ones we have raised in systems analysis and design. Well worth reading to get a different perspective on the same truth. Alexander's other works also make fascinating reading.

Hardison, O. B. 1989. *Disappearing Through the Skylight: Culture and Technology in the Twentieth Century.* New York: Penguin Books. A fascinating, disturbing but ultimately exhilirating account of the inability of tradition and traditional forms of expression to deal with twentieth century life and the remarkably similar responses of artists, architects, poets, musicians, and technologists to the problem.

Lakoff G. 1987. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind.* Chicago, IL: University of Chicago Press. If you read no other book on cognitive science, as an object-oriented sort of person you owe it to yourself to read this one. The bibliography is also a good source of additional reading.

Lakoff, G., and M. Johnson. 1980. *Metaphors We Live By.* Chicago, IL: University of Chicago Press. Nicely complements *Women, Fire, and Dangerous Things.* Excellent discussion of the role of metaphor in human perception and problem solving.

▶ Interesting Reading

Bronowski, J. 1978. *The Origins of Knowledge and Imagination.* New Haven: Yale University Press.

Brown, G. 1973. *Laws of Form.* New York: Bantam Books.

Bruner, J. 1986. *Actual Minds, Possible Worlds.* Cambridge MA: Harvard University Press.

Campbell, J. 1982. *Grammatical Man.* New York: Simon & Schuster, Inc.

Goodman, N. 1985. *Ways of Worldmaking.* Indianapolis IN: Hackett.

Gregory, B. 1990. *Inventing Reality: Physics as Language.* New York: John Wiley & Sons.

Miller, G. 1956. The Magic Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review.*

Nagel, E., and J. Newman. 1967. *Godel's Proof.* New York: New York University Press.

Sapir, E. 1958. *Culture, Language and Personality.* Berkeley and Los Angeles: University of California Press.

von Foerster, H. 1984. *Observing Systems.* Seaside, CA: Intersystems Publications.

Watzlawick, P. 1984. *The Invented Reality: How Do We Know What We Believe We Know?* New York: W. W. Norton & Company.

Whorf, B. 1976. *Language Thought & Reality.* Cambridge, MA: M.I.T. Press.

▶ Graphic Arts

You need look no further for graphic arts information than the following two books by Edward R. Tufte, both of which are already cult classics among those who work with graphical user interfaces for computers.

Tufte, E. R. 1983. *The Visual Display of Quantitative Information.* Cheshire, CT: Graphics Press. Time only makes this book more compelling. The book is every bit as sweeping as its title would have you believe.

Tufte, E. R. 1990. *Envisioning Information.* Cheshire CT: Graphics Press. A landmark book on the importance of three-dimensional visualization ("escaping flatland") and the appropriate use of color.

Also of interest, although not quite a book on graphic arts:

Laurel, B. ed. 1990. *The Art of Human-Computer Interface Design.* Reading, MA: Addison-Wesley. The best book yet published on the design of Macintosh-like interfaces.

INDEX

D

E

M

P

W

Z

Titles in the Macintosh Inside Out Series

- **Extending the Macintosh® Toolbox**
  Programming Menus, Windows, Dialogs, and More
  John C. May and Judy B. Whittle
  A complete guide to programming the Macintosh interface.
  352 pages, $24.95, paperback, order #57722

- **Programming QuickDraw™**
  Includes Color QuickDraw and 32-Bit QuickDraw
  David A. Surovell, Fred M. Hall, and Konstantin Othmer
  The first in-depth reference to the Macintosh graphics system.
  352 pages, $24.95, paperback, order #57019

- **Programming for System 7**
  Gary Little and Tim Swihart
  A complete programmer's handbook to the newest version of the Macintosh system software.
  400 pages, $26.95, paperback, order #56770

- **Programming with AppleTalk®**
  Michael Peirce
  An accessible guide to creating applications that run with AppleTalk.
  352 pages, $24.95, paperback, order #57780

- **The A/UX® 2.0 Handbook**
  Jan L. Harrington
  A complete and up-to-date introduction to UNIX on the Macintosh.
  448 pages, $26.95, paperback, order #56784

- **System 7 Revealed**
  Anthony Meadow
  A first look inside the important new Macintosh system software from Apple.
  368 pages, $22.95, paperback, order #55040

- **ResEdit™ Complete**
  Peter Alley and Carolyn Strange
  Contains the popular ResEdit software and complete information on how to use it.
  576 pages, $29.95, book/disk, order #55075

- **The Complete Book of HyperTalk® 2**
  Dan Shafer
  Practical guide to HyperTalk 2.0 commands, operators, and functions.
  480 pages, $24.95, paperback, order #57082

- **Programming the LaserWriter®**
  David A. Holzgang
  Now Macintosh programmers can unlock the full power of the LaserWriter.
  480 pages, $24.95, paperback, order #57068

- **Debugging Macintosh® Software with MacsBug**
  Includes MacsBug 6.2
  Konstantin Othmer and Jim Straus
  Everything a programmer needs to start debugging Macintosh software.
  576 pages, $34.95, book/disk, order #57049

Macintosh Inside Out

> $26.95 USA
> $34.95 CANADA

Developing Object-Oriented Software for the Macintosh®

NEAL GOLDSTEIN
JEFF ALGER

Based on Neal Goldstein's widely acclaimed Object-Oriented Design and C++ seminars at Apple Computer, Inc., **Developing Object-Oriented Software for the Macintosh®** takes Macintosh software developers step by step through the object-oriented software development process. This is the first book to deal with the complete process of developing object-oriented software, from analysis through design and programming. Programmers, systems analysts, managers, and anyone concerned with Macintosh software development will benefit from the concepts and methodology of this practical, hands-on guide.

The book first covers the basics of creating object-oriented software, focusing on the essential concepts and principles. It then presents the author's acclaimed Solution Based Modeling methodology and notation for analysis, design, and programming in object-oriented development. Special attention is paid throughout to problems inherent in large-scale Macintosh development, and code examples are provided in C++ and Object Pascal.

You will also learn how to:

- Design efficient, maintainable object-oriented programs
- Design for portablility to other computers
- Communicate object-oriented designs effectively to both programmers and non-programmers
- Control projects from initial requirements through the development process.

Developing Object-Oriented Software for the Macintosh is essential reading for all Macintosh software professionals.

Neal Goldstein is widely known for his Apple Developer University courses on C++ and object-oriented design.

Jeff Alger lectures at the Apple Developer University and is Chairman of the Board of Directors of the MacApp Developers Association, the leading organization for the object- oriented development for the Macintosh.

52695>
9 780201 570656
ISBN 0-201-57065-3
57065

Cover design by Ronn Campisi
Addison-Wesley Publishing Company, Inc.