Object Oriented Application Frameworks

Ted Lewis Larry Rosenstein Wolfgang Pree
Andre Weinand Erich Gamma Paul Calder
Glenn Andert John Vlissides Kurt Schmucker

OBJECT-ORIENTED APPLICATION FRAMEWORKS

TED LEWIS *and*

GLENN ANDERT

PAUL CALDER

ERICH GAMMA

WOLFGANG PREE

LARRY ROSENSTEIN

KURT SCHMUCKER

ANDRÉ WEINAND

JOHN M. VLISSIDES

MANNING

Greenwich
(74° w. long.)

The publisher offers discounts on this book when ordered in quantity. For more information please contact:

Special Sales Department
Manning Publications Co.
3 Lewis Street
Greenwich, CT 06830
or
73150.1431@compuserve.com
Fax: (203) 661-9018

Design and Typesetting: Stephen Adams
Copy editor: Margaret Marynowski

Library of Congress Cataloging–in–Publication Data

Object-oriented application frameworks / Ted Lewis, editor.
p. cm.
Includes bibliographical references and index.
ISBN 1-884777-06-6
1. Object-oriented programming (Computer Science) I. Lewis, T. G. (Theodore Gyle), 1941–
QA76.64.0243 1995
005.1'1—dc20 94-48023
CIP

95 96 97 98 BB 10 9 8 7 6 5 4 3 2

Printed in the United States of America

Preface

One of the next major steps in object-oriented design and programming is framework design and programming. Frameworks are being commercialized by NeXT and Taligent, and to a lesser extent, by IBM, Microsoft, SunSoft, Borland, and Hewlett Packard. I think that the introduction of the Taligent products in 1995 will stimulate widespread interest in frameworks: what they are, how they work, and how they compare with one another.

This paperback tutorial/survey is designed to address the anticipated surge of interest in what has previously been a little-understood technology. I hope to do so in three parts: an introduction to the underlying principles of object-oriented design, a comparative survey of frameworks for personal computers and UNIX workstations, and an illustration of the uses of frameworks. Part I should make the book appealing to the beginner, and Parts II and III should appeal to the software project leader, MIS manager, or advanced programmer. This is a professional book, not a textbook.

What exactly is an object-oriented framework? It is an object-oriented class hierarchy plus a built-in model of interaction which defines how the objects derived from the class hierarchy interact with one another.

This rather simple definition belies the power of frameworks. In practical terms, the framework approach leverages capital-intensive software investment through reuse, and provides a much higher-level application programming interface, so that applications can be developed ten times faster. It is the next giant step in the progression toward more powerful desktop computers. Steve Jobs recognized the significance of object-oriented frameworks when he called his new company *NeXT* and his framework-based operating system *NeXTStep*.

Frameworks are not new-fangled research exotica, but rather the essential core of what is happening in software these days. The two most glaring examples of the framework approach come from Taligent Inc. and NeXT Inc. Both operating systems are based on the MACH kernel (or other multithreading systems, such as OS/2 or Apple's System 8.0), and both are layered with object-oriented software services in the form of object-oriented frameworks. The Taligent operating system makes more extensive use of frameworks than any other system, except perhaps the Apple Newton.

The purpose of the Taligent operating system is to win the battle of the desktop in the next round of operating systems wars. Of course, the players want to shift the balance of power to their own advantage as the PC industry converts from a 16-bit to a 32/64-bit architecture based on RISC processors such as PowerPC, DEC Alpha, and SuperSPARC. These better-performing processors with radically different architecture make existing PC operating systems obsolete. But what will replace them? This is the 100-billion-dollar question.

I believe whatever the outcome of these wars, the object-oriented framework concept will be at the core of the technology of the twenty-first century. I hope this book helps you get started on your journey toward understanding the subtleties of this new technology.

TED G. LEWIS, PHD

Contents

PART I

A Guide to Object-Oriented Design

Every book must start somewhere, so we will start at the beginning, with an overview of the history and origins of object-oriented programming and frameworks. What are they, how did they come about, and why are they so important? These, and other mysteries of the Universe, are covered in four adventure-packed episodes.

The advanced reader might want to skip Part I, especially if he or she already knows the ins and outs of objects, polymorphism, inheritance, class libraries, and Norwegian science. The rest of us can use these introductory chapters to brush up on some important ideas. For example, Chapter 1 quickly takes us up to the contemporary software scene, where programmers are making significant advances in productivity, pushing rapid application development to the limit, and making more money than ever. Chapter 1 is also an excellent place to find basic definition of terms that will linger throughout the rest of the book. While technically lightweight, this first foray into the object-oriented jungle will reward the reader with an orthodontically correct view of the subject.

Chapter 2 gets to the heart of the matter by defining what a framework is, and what it can do. This chapter's stark simplicity will be much appreciated by even the most dedicated nontechnical reader. In addition to introducing the very important model of interaction called MVC (Model-View-Controller), its clear writing, advanced thinking, and global perspective will place you on the edge of your sofa.

Things get tougher in Chapters 3 and 4, because here is where we start to get specific. First, Chapter 3 drags you through the design of a small application framework. The Table framework is a reusable component that creates and manages spreadsheet-like data. In addition to displaying data in a two-dimensional format like Excel or Lotus 1-2-3, the Table framework works as part of a larger framework (MacApp). This illustrates the way programmers of the future will work: borrowing instead of reinventing the software wheel.

Chapter 4 pushes these ideas a bit further, by overlaying a visual or diagram-like programming system on top of a framework for the Macintosh called Objex. This clever bit of programming magic encapsulates both design and code for storage, GUI, I/O, graphics, and MVC applications under the glossy cover of a point-and-click diagramming language based on the mathematically elegant Petri Network. This chapter also contains the first evidence to support the wild claims about programmer productivity that we make in the first chapter.

CHAPTER 1

Origins of the Species

Preview

The object-oriented design and programming revolution is based on the fundamental ideas of objects, classes, methods, messages, encapsulation, inheritance, polymorphism, and dynamic binding. Some of these are new words for old ideas, while others are names for new concepts in software design. How did we get here from the old-fashioned procedural programming heritage, and what do all of these new words mean? The following is an illustrated guide to the new way of thinking about, designing, and writing software.

The two great programming paradigms of the past three decades have been procedure-orientation and object-orientation. The procedural paradigm has served us well, but now it is time for something more powerful, because machines and consumer expectations have grown in power and sophistication. So, how do we deliver powerful software while sidestepping the software crisis? The answer goes all the way back to 1967, to Norway—not Japan, not Germany, and certainly not the USA.

In this chapter you will learn that object-orientation is a better way of writing modular programs. The modules of an object-oriented program encapsulate function, in the form of procedures, and state, in the form of storage variables. This, combined with the generality and flexibility of a programming trick called *polymorphism*, defines the object-oriented paradigm. Oh, and then there are *inheritance* and *dynamic binding*. But then, if we tell you what these are in the first page, you won't buy the book!

1.1 After 25 Years, Why Now?

First, a warning to the reader. The word *OOP* is not a faux pas on the part of the author, but rather an abbreviation of *object-oriented programming*. Another frequently-used phrase in this book is *object-oriented design*, abbreviated *OOD*.

Second warning. This chapter introduces OOP and OOD at the most elementary level. If you are beyond the novice level, skip to the next chapter, or if you are an expert OOP programmer and simply want to read about frameworks, skip to the next part.

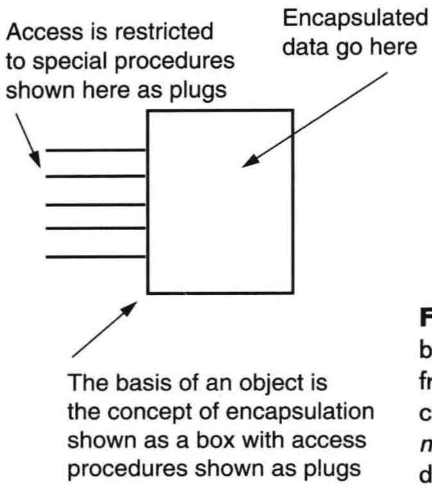
1.1.1 In the Beginning

OOP and OOD are at least two decades old, and the ideas have been around for perhaps even longer. The fundamental idea of composing software from a collection of modules called *objects* goes way back to the dawning era of software design, even before personal computers. Even before video games! In 1967, a new programming language called *Simula67* was invented for the purpose of writing simulation software. But, rather than raise interest in simulations, it went unnoticed by just about everyone except for a few leading-edge software academics. The academics liked Simula67 because it showed how to cluster data around procedures, thus simplifying software design, coding, and maintenance.

Data and procedure clustering prevented unauthorized access to data in a Simula67 program. Only procedures that belonged to the same cluster were allowed direct access to the cluster data. In a sense, Simula67 substituted function calls for data access everywhere it occurred in the program.

Clusters prevented indiscriminate data access. They set up software fences, and access procedures were software gates for controlling access. Thus was born the idea of *encapsulation*—one of the central ideas of OOP, so important in fact, that we have used an enclosed box to represent it, in Figure 1.1. Encapsulation is defined as any mechanism which separates a program variable from code which changes its state. Furthermore, encapsulation implies some form of access protection, a topic of great interest to programming language designers in the 1970s. But, we are getting ahead of our story.

Maybe because Simula67 was aimed at simulation programmers instead of general application development programmers, or maybe because it was invented in Scandinavia, it remained somewhat obscure until the same ideas were rediscovered by the computer scientists at Xerox Corporation's think tank, Xerox PARC (PARC is an abbreviation of Palo Alto Research Center). In the 1960s, 1970s, and early 1980s, Douglas Englebart and others at PARC had fun inventing things like menus and windows, the mouse, and computer-to-computer networks. One of their



Access is restricted to special procedures shown here as plugs
Encapsulated data go here
The basis of an object is the concept of encapsulation shown as a box with access procedures shown as plugs

Figure 1.1 *Encapsulation* is the basis of OOP: All data are closed off from the outside world, and only special procedures called *methods* or *member functions* are allowed to directly access the encapsulated data

goals was to make computers so simple that even a child could program them. To do this, they had to eliminate much of the complexity of traditional procedural programming, e.g., FORTRAN and Cobol had to go.

So, with the help of Allan Kay, Adele Goldberg, Larry Tesler, and others, the ideas of OOP were collected together to create yet another language, this time called *Smalltalk-80*, which incorporated the fundamentals of OOP. And, once again, the world pretty much ignored it!

1.1.2 The Software Crisis Scare

Two things happened in the 1980s which reversed the poor fortunes of OOP. First, computer hardware got faster and cheaper, and second, software got bigger and much more expensive. In fact, software costs soared to the point of creating a *software crisis*. The combination of these two trends is dramatized in Figure 1.2, which shows that the cost per hardware unit was plunging while the cost per software unit was soaring. (The cost per statement of a large program is higher than the cost per statement of a smaller program. So, Figure 1.2 is truly a comparison of the per-unit costs of software versus hardware. On the other hand, the cost of a single transistor decreases by a factor of about 4 every three years!).

Rapid advances in building faster and cheaper hardware were quickly absorbed by software that made intensive use of graphical user interfaces (GUIs). GUI software made using computers so simple that a new breed of applications was quickly invented. Drawing programs that processed

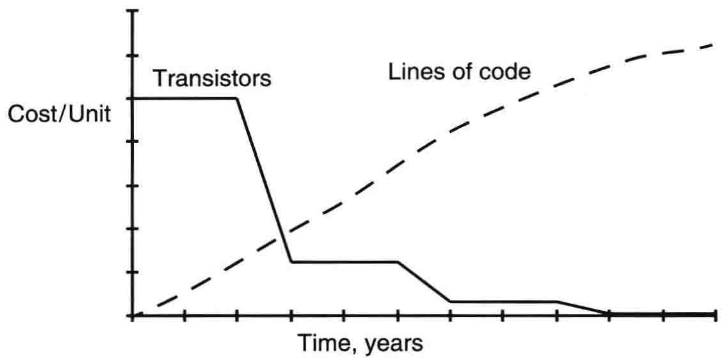


Figure 1.2 Cost trends in the 1970s and 1980s. Unless something is done about software, these trends will continue into the year 2000

pictures instead of text could run on inexpensive personal computers for the first time. Ordinary people could build financial models by merely pointing and clicking. These new toys looked suspiciously like the Xerox PARC computers designed for the children of silicon valley families living next to PARC. And guess what—the drawing programs were object-oriented! (This idea just kept showing up in all the right places).

One person's fool's gold is another person's treasure. The Xerox executives failed to appreciate the profit potential of cute little computers with object-oriented GUIs that everyone could operate. Profit potential was not over the head of Steve Jobs of Apple Computer, who saw clearly that OOP held the long-sought-after answer to the big questions of computer life. Apple bought the idea of OOP, along with most of the people who worked for PARC! (Larry Tesler became head of Advanced Technologies, and Allan Kay became what he always wanted to be—an icon). They made a small computer at a selling price comparable to a car instead of a house, and the Macintosh was born. The Macintosh changed the world in more ways than one, but to the OOP world, it was a godsend.

1.1.3 The Plot Thickens

The plot thickens even more, because the second problem of the 1980s persisted: the soaring cost of software was put into orbit by GUI programming. As it turns out, while Apple was busy making money from Xerox PARC's ideas, Apple was also contributing to the software crisis. GUI programming, networking, multiple-processor systems, and rising customer expectations were all driving programmers to the brink as software became bigger, more complex, and more elaborate. What was a programmer to do?

The software engineers of the 1980s had a solution, but they did not know how to implement it. *Reuse* is the so-called *mega programming* idea, whereby software components are scavenged from old programs then reused to construct new programs. Could the software crisis be held at bay through reuse of software components? It seemed like a good idea, but how? The reuse craze swept the land at about the same time rap music overwhelmed MTV. Oh, and OOP was reborn because the OOP paradigm is perfect for reuse. Not only did OOP properly encapsulate, but it also recycled old code into new.

As they say, the rest is being rewritten as history. OOP solved a major computing problem, making it possible to program faster and cheaper hardware much faster and cheaper than using the old-fashioned procedural style exemplified by FORTRAN, Cobol, Pascal, and C. The faster hardware made programmers stop worrying about the overhead of OOP, so they had time to worry about bigger and costlier application software. But, the side benefits of OOP reuse put a stop to the growth in complexity of the new GUI-intensive applications.

Apple Computer tried to promote OOP with MacApp, which was the first widely used reusable OOP tool. But the idea was still slightly ahead of its time. Then Apple Computer stumbled in its loyalty to founder Jobs, so he took his OOP ideas to NeXT Inc., and NeXTStep was born out of the OOP stew. Later, IBM thought that OOP was such a good idea, it bought the languishing technology from Apple and together they formed Taligent Inc., and another contender in the "OOP is for reuse" contest joined the frenzy.

1.1.4 Back at the (AT&T) Ranch

Meanwhile, the procedural programming gang was not giving up without a fight. Cute little computers were not going to rule the world, at least not while there were UNIX hackers, C language diehards, and AT&T. Smalltalk might be pure and perfect, Object Pascal elegant, and MacApp simply too advanced, but C was the *lingua franca* of real programmers. If it was not an extension of C, how could it be more than a toy?

Bjorne Stroustrup of AT&T Bell Labs offered a solution just in time to rally the UNIX programmers who were as out of date in the new OOP decade as disco. Stroustrup added object-oriented features to C, and gave the concoction a clever name, C++. Actually, C++ source code was originally converted into ordinary C by a program called *CFRONT* which AT&T gave away. The converted program was run through a C compiler to get executable binary code. Therefore, C++ was not really pure and perfect. Even today, C++ is euphemistically called a *hybrid OOP,* because a clever programmer can use it to write either a procedural or object-oriented program.

The invention of C++, Object Pascal, and other hybrids has weakened the foundation of OOP, because encapsulation is not strictly enforced in such upstart languages. Yet, these hybrids have been largely responsible for the slow conversion of procedural programmers to the OOP style of programming. In the 1990s the story is largely one of transition from pure procedural to pure object-oriented programming.

So, by the late 1990s, OOP will have replaced top-down structured programming as the software fad of the decade. Programmers will discard their worn-out ideas based on procedural languages and adopt an OOP language. If it were not so true, we would not have to say it, but OOP is to the 1990s what structured programming was to the 1970s. And, OOP is truly a paradigm shift because most of what you know about procedural programming is harmful to understanding the OOP paradigm.

1.1.5 Giving Life to Meaning: OOP As a Paradigm

What exactly is the OOP paradigm? Table 1.1 summarizes the terminology of both paradigms. In the new paradigm, *procedures* are no longer the fundamental software building blocks, rather, *objects* are. Now embedded in objects, procedures are activated only when a *message* is sent from one object to another. Objects and messages are the stuff of software design.

The procedural paradigm must be discarded to clear the way for an entirely different view of data, data types, and access to data. We will begin to tell that story in the next section. In the remainder of this book, we will illustrate these ideas and discuss how they are used to reverse the software crisis, stave off spiraling development and maintenance costs, and save the computer world. Well, maybe not, but at least you will become an expert