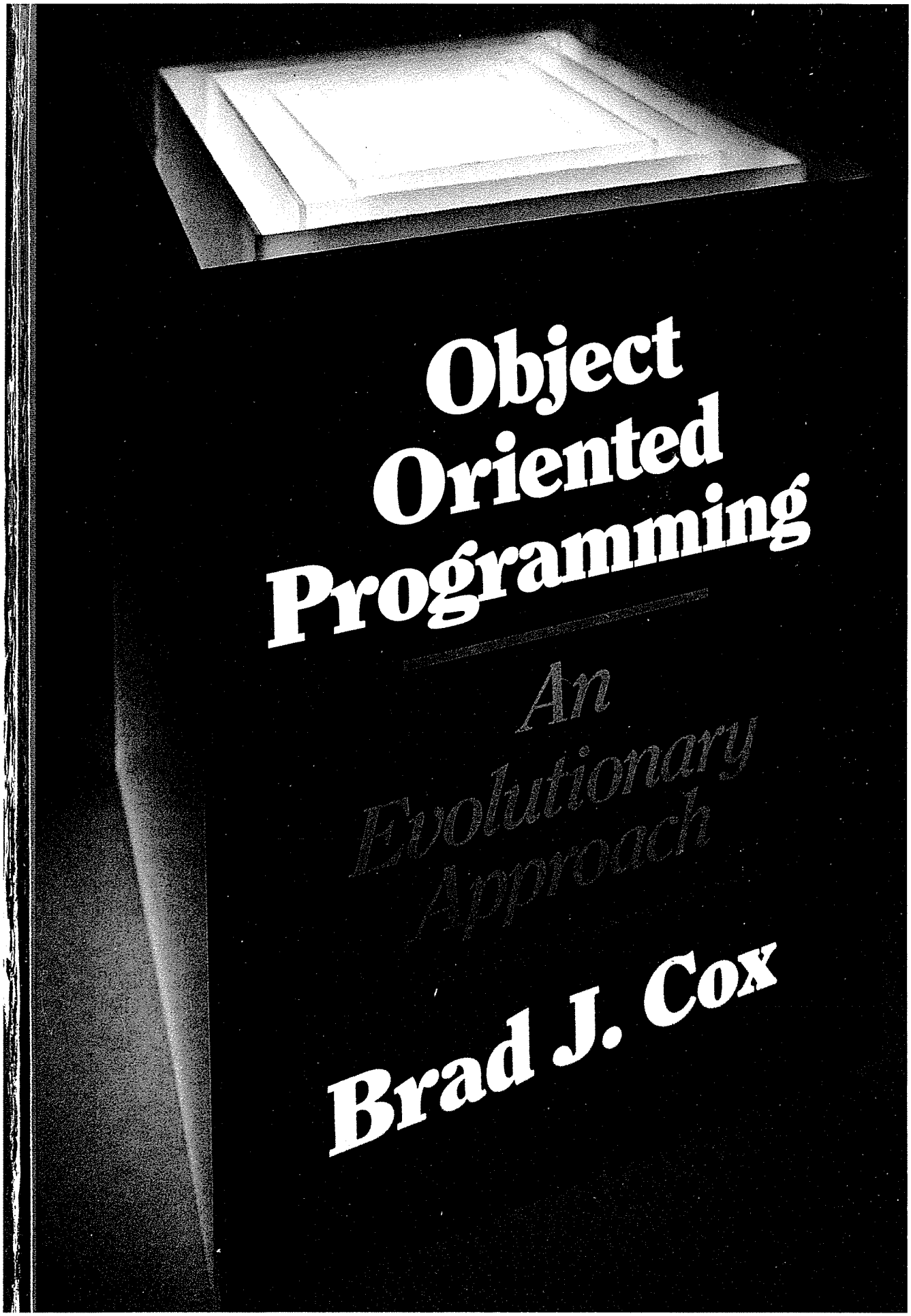
EXHIBIT 27


Object
Oriented
Programming
An
Evolutionary
Approach
Brad J. Cox

Object-Oriented Programming

An Evolutionary Approach

Brad J. Cox, Ph.D.
Productivity Products International

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts • Menlo Park, California
Don Mills, Ontario • Wokingham, England • Amsterdam
Sydney • Singapore • Tokyo • Madrid
Bogotá • Santiago • San Juan

Library of Congress Cataloging-in-Publication Data

Cox, Brad J., 1944
Object-oriented programming.

Includes index.
1. System design. 2. Computer software.
I. Title.
QA76.9.S88C69 1986 003 85-22921
ISBN 0-201-10393-1

Reprinted with corrections April, 1987

IJ-HA-89

Two

Why Object-oriented Programming?

Object-oriented programming is not so much a coding technique as it is a code packaging technique, a way for code suppliers to encapsulate functionality for delivery to consumers. It is this increased emphasis on the relationship between consumers and suppliers of code that separates object-oriented and conventional programming (Figure 2.1).

Binding is the process of integrating functionality from different suppliers into a consumer's code (Figure 2.2). Binding is more than what most programmers call linking; determining which binary modules must be combined to produce an executable image, assigning a memory address to each one, and patching external references with the correct memory addresses. Binding is the process whereby operators and operands of potentially many different types are published by suppliers and used by consumers.

There are several schools of thought about when and how binding should be done, each with distinct strengths and weaknesses. The early binding approach is the most widely known because it is the only approach provided in most conventional languages. With early binding, binding occurs while the consumer's code is being compiled so that the consumer and his tools (the compiler) bear responsibility for binding. Delayed binding (also known as late binding or dynamic binding) means that binding is done later than compile-time, generally while the program is running. Delayed binding moves responsibility for binding away from the consumer and onto the operands, effectively onto the supplier who defined this type of operand.

These approaches are not equivalent. They are different tools for different jobs. The difference is in the degree to which the design of a collection's

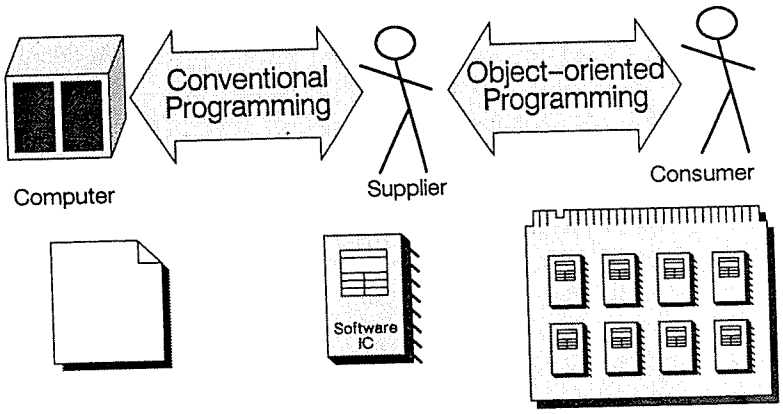


Figure 2.1. Program-building versus system-building. Conventional programming tools emphasize the relationship between a programmer and his code, while object-oriented programming emphasizes the relationship between suppliers and consumers of code.

contents can be allowed to affect the design of the collection; the degree to which the consumer's code is coupled to that of the suppliers' (Figure 2.3). Languages that provide early binding are ideal for building tightly coupled collections where all uncertainty about the data type provided by each supplier can be removed at or before the time the consumer's code is compiled. Dynamic binding is needed in loosely coupled collections where the consumer's code cannot predict the type of data to be operated on until the code is being run.

This chapter shows the effect of this coupling by contrasting several binding approaches—in a small example—a program for counting the number of unique words in a file.

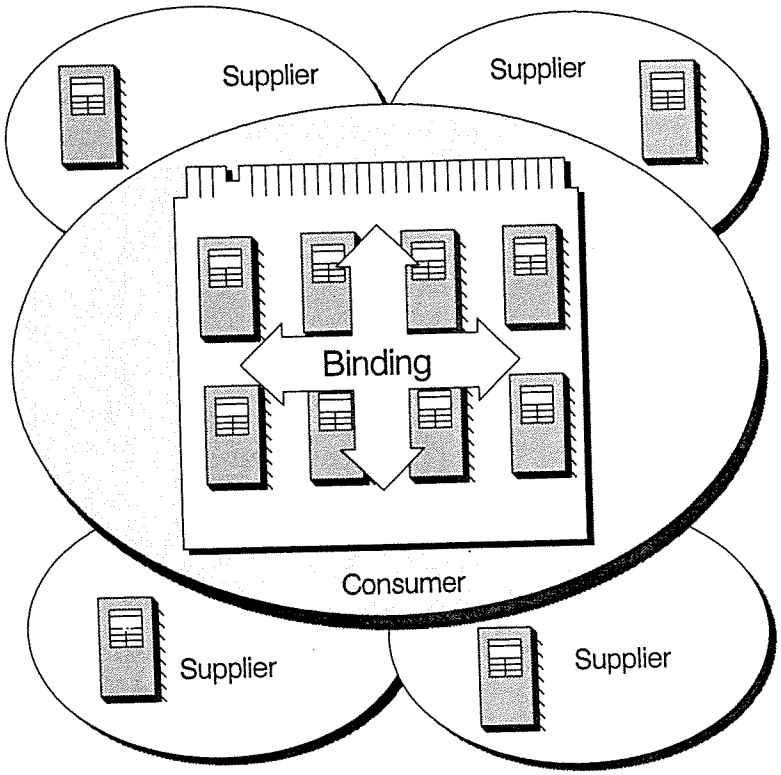


Figure 2.2. Binding. Binding is the process of assembling components from different suppliers into a larger component belonging to the consumer.

a programmer might like to compute the vector to its bottom right corner by writing

```
vector bottomRight = origin + extent;
```

This is not possible in C, nor in other conventional languages like Pascal, FORTRAN, and COBOL.[3] These languages do not allow the programmer to change the meaning of built-in operators like +, so that these operators cannot be applied to newly added data types like vector. These languages require the programmer to decompose each operation to primitive types manually and write:

```
bottomRight.x = origin.x + extent.x;
bottomRight.y = origin.y + extent.y;
```

The supplier of a new data type like vector cannot hide the implementation of this new type from consumers. They must know the implementation to use the type, because they must do all binding between vector and operations on vectors manually when they write their code.

The recent trend is toward languages that do provide this flexibility, as long as all types are known at compile-time. For example, languages like Ada allow vectors to be defined as a new data type with +, a legal operation on that type. When compiling the expression

```
bottomRight = origin + extent
```

the compiler notices that both origin and extent are vectors, and determines that a special meaning for the + operator is to be used—the one defined by the supplier of the vector type. These languages provide early binding in a more powerful way than C because the meaning of predefined operators like + can be changed for newly defined data types.

This approach can address some of the difficulties encountered when applying subroutine libraries to the UniqueWords problem. The problem was to build a new data type, Set, without prematurely committing to the kind of elements to be stored in the Set. Ada's solution to this class of problem is called a generic package, which is a package of code with a compile-time parameter that specifies which type is to be managed by the set. The UniqueWords application would be solved by writing a statement in the consumer's code that directs the compiler to compile a Set that expects to manage ByteArrays.

Dynamic Binding and Loosely Coupled Collections

The statically bound languages are perfect for building tightly coupled collections like vectors and rectangles. They can even be pushed to handle

[3]But it is possible in C++. See Chapter 3.

some harder cases like the UniqueWords example. Of course recompiling a supplier's code every time a consumer provides a new data type can be a problem, and not only because of the time to compile these custom sets and the memory to store them in. A more fundamental concern is that the supplier must trust the compiler to protect any proprietary interest he might hold in those sources. It is hard to see how a commercial marketplace in generic packages could develop as long as suppliers must trust the compiler to protect their proprietary interests in source code.

It is less widely recognized that statically bound languages are extremely poor for building loosely coupled collections—problems more like the automobile's trunk than its engine. Loosely coupled collections abound in applications like office automation, in information-oriented ensembles like DeskTop, Envelope, FileFolder, PaperClip, Mailbox, and FileCabinet. A desktop or a mailbox is a loosely coupled collection because it is neither possible nor desirable to state, at any point earlier than when the desktop is in use, what kind of items it contains.

When every data type is known when the code is compiled, static binding works. Otherwise binding must be done dynamically, period. There is no choice between static and dynamic binding, since dynamic binding is intrinsic to the very essense of a loosely coupled collection. However, there is a choice between one method of implementing dynamic binding and another, since dynamic binding can be done either manually, by conditional statements written by the programmer, or automatically, by the programming language and the run-time environment.

The choice is shown in Figure 2.4. This figure shows the kinds of objects in an office automation project as seven folders, each representing the work of a different programmer. The mailbox folder is open to show the mailbox developer's two choices for how to implement dynamic binding.

The left fragment shows the manual approach. A switch statement determines which kind of object is on hand, passing control dynamically to the operation (a C function) that properly manipulates that kind of object. The kind of object is represented in a field inside each object, represented as a C structure:

```
struct OfficeMemo { int type; ... } ;
struct WhileYouWereOutNotice { int type; ... } ;
struct FileFolder { int type; ... } ;
```

The fragment on the right shows the other approach in which dynamic binding is provided automatically by the programming language. The mailbox developer, as consumer of the other six data types in this system, specifies what each item is to do by writing the message expression [item doThis], and it is up to the object to decide how the command, doThis, should be accomplished for that kind of object.

The difference between these two approaches is crucial to the separation of responsibility between supplier and consumer. The mailbox developer is the

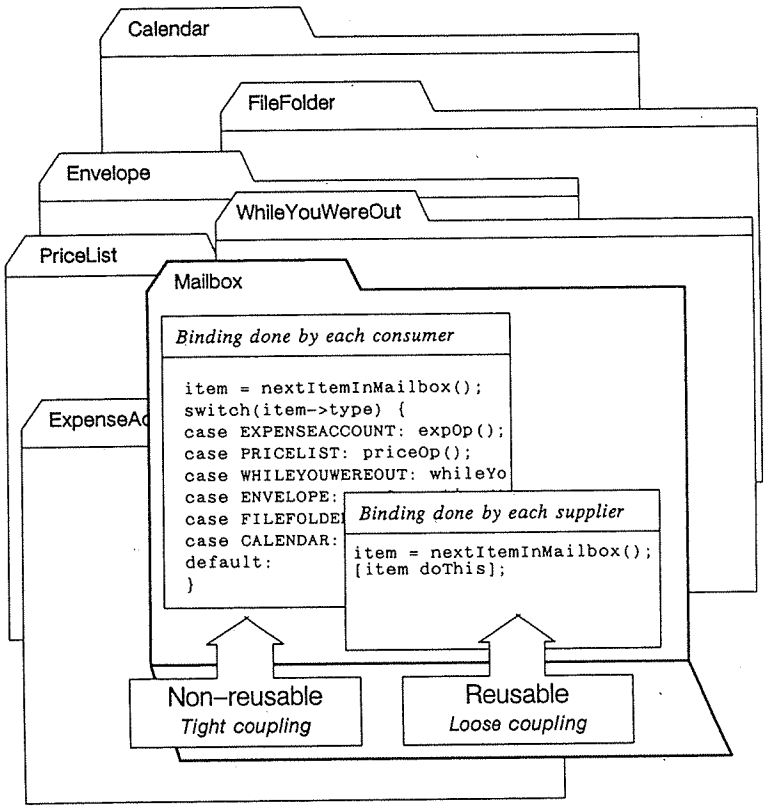


Figure 2.4. Binding in a loosely coupled collection. Dynamic binding can be implemented either by the consumer or by the supplier. The former leads to nonreusable code, because the consumer's code enumerates the set of data types that were known when the consumer's code was written. The latter can produce reusable code, because data types are mentioned in only one place—in the files that define each data type.

consumer of functionality provided elsewhere in his project. The first approach requires that he, the consumer, bear the responsibility for determining which supplier's data is at hand in the mailbox and for selecting which one of the supplier's subroutines is proper for that kind of data. This is called consumer-side binding. By contrast, the other approach puts that responsibility right where it belongs, on the supplier's side.

The problem with consumer-side binding can be seen in the case labels in the switch statement. These labels explicitly enumerate the data types this mailbox is prepared to handle, so each time a new type is added anywhere in this system, the mailbox code must be changed. Something foul has leaked across the boundary that should have isolated suppliers from consumers. This mailbox is useless in any other application, because the case labels explicitly state that it will only work correctly for those six types.

One effect of consumer-side binding is lack of reusability. It forces the mailbox to be written in a way that prevents it from ever being reused in other applications. Consumer-side binding also leads to lack of malleability. Each time a new data type is added to the system, the source for the mailbox must be modified. It cannot withstand the kind of evolutionary changes discussed in Chapter 1. Supplier-side binding, by contrast, increases the chance that the mailbox can be reused in other applications because it no longer mentions types that might change in some new application. And the system becomes malleable because new data types can be added over time without impacting working code.