Editor: Bertrand Meyer, Interactive Software Engineering, 270 Storke Road, Suite 7, Goleta, CA 93117; phone (805) 685-1006; fax (805) 685-6869; Internet ot-column@eiffel.com

The reusability challenge

Bertrand Meyer, ***Interactive Software Engineering***

Onward and forward

As I write this, the first installment of the Object Technology department is barely out, so it is too early to include any reader reaction. Suffice it to say that everyone I have talked to is excited about this new department. As a result, I will probably have more columns by guest contributors (and fewer by me) than originally planned: So many people have interesting things to say! To begin with, it occurred to me that the next column—still "Design by Contract," as announced—could just as well be written by Jim McKim of the Hartford Graduate Center, who is currently writing a book on that very topic.

The traditional "reuse or redo" dilemma is unacceptable; we need to reuse what is applicable to our current need and redo the rest. This is where object technology helps.

Another new contributor signed up for sometime in the next few months is Mary Loomis of Hewlett-Packard, whose topic will be object databases. Also scheduled are Brian Henderson-Sellers of the University of Technology, Sydney, and Ian Graham of Swissbank, who will tell about the effort they lead, in concert with many other OO methodologists, to come up with a broadly acceptable object method called Open.

As a reminder, you are welcome to post comments on this department in the Usenet object technology newsgroup, comp.object; letters to the editor can be e-mailed to ot-column@eiffel.com.

Reusability and object technology

Now for this month's technical topic. The argument we examine—that object technology promotes reusability—has been one of the central claims of OO proponents since the field's emergence in the mid-1980s.

What indeed makes OO ideas so special in the quest for reusable software?

The right level of abstraction. Earlier approaches to reusability, ever since McIlroy's seminal paper,[1] assumed that the routine would be the unit of reuse. With object technology, we have a coarser grain of decomposition: a *class*, the description of a set of objects characterized by applicable operations. For example, instead of reusing the routine to invert matrices of a certain kind, say "band matrices," we move one level higher: We put into our library the class *BAND_MATRIX*, complete with all applicable operations—inversion, but also transposition, determinant, column and row extraction, and so on.

This gives us components at the right level, describing whole abstractions rather than scattered pieces, and hence more coherent, robust, and easy to use.

Most of the other OO mechanisms, reviewed last month, help further. Let me mention two. Thanks to one, inheritance, we can organize the many variants that often exist for certain reusable concepts. The other is our ability to specify precisely, through assertions, the properties of reusable components. Be it in electronics or construction, no reuse is possible without abstract specification; software is no exception.

Programs with holes. One interesting aspect of classes is that their designers get to choose the exact level of refinement that suits the needs of the moment. A class can be fully abstract, or *deferred*; it can be fully implemented, or *effective*; but it can also be somewhere in between, the designer having chosen to spell out certain aspects and leave others to be filled in later.

Why is this important for reusability? The answer lies in the peculiar nature of software. If the common needs of our programs were fixed, the reusability problem would have been solved a long time ago; someone would have programmed all these useful patterns in the form of modules similar to the reusable VLSI chips of our hardware colleagues. But software's reuse requirements are more subtle. Only in some limited cases can we benefit from frozen implementations (as offered by subroutine libraries). Most of the time, even in application areas where we sense much repetitiveness, what recurs is a general scheme, with room for variable elements that each application may replace with its own variants.

In other words, *reusability* in software is inseparable from *adaptability*. The traditional "reuse or redo" dilemma is unacceptable; we need to reuse what is applicable to our current need and redo the rest. This is where object technology helps.

A typical example is an "iterator," as offered by the classes of several object-oriented libraries. An iterator applies a certain operation to all elements of a certain data structure (or file, or database) in a specified order. Much of the iteration is reusable; developers should not need to reprogram the control structure and the iteration scheme. Of course, the details of what you do at each stage vary with each application. In the list iterator classes of EiffelBase,[2] for example, you will find procedures such as

```
until_do is
    —Apply action to every item of target up to
    —but excluding first one satisfying test.
    —(Apply to full list if no item satisfies test.)
  do
    from
      start
    invariant
      invariant_value
    until
      exhausted or else test
    loop
      action
      forth
    end
  end
ensure
  finished: not exhausted implies not test
end
```

Thanks to inheritance, an application reusing this mechanism will provide its own *test* and *action* routines and optionally an *invariant_value*; everything else is preprogrammed. (Note that *start* is the procedure that positions the cursor at the beginning of the structure, *forth* advances it by one position, and *exhausted* tells whether we have examined all items.)

This example addresses a general data structure scheme, but it is not difficult to think of many application-specific variants. Business applications, for example, frequently follow a common model—process all the day's invoices, validate a payment request, enter new customers, and so on—with individual components that may vary.

OO techniques offer an ideal vehicle for such reuse-redo combinations. Particularly interesting is the ability to write deferred classes (such as *LIST_ITERATOR*) where some of the routines, such as *until_do* above, are effective (implemented) but call deferred routines such as *test* and *action*. Descendant classes can then effect these routines (provide effective versions) according to the application's needs.

This idea is not new with object technology. The general "don't call us, we'll call you" approach, where predefined elements call user-supplied mechanisms, is also common in graphics systems, often under the name of *callback*. At various stages in its execution the system calls certain functions, for which application developers "plant" their own variants at specified places. Even IBM's venerable database system, IMS, used a similar scheme. What the OO method offers is systematic, safe support for this technique, through classes, inheritance, type checking, deferred classes and features, as well as assertions that let the developer specify what properties the variable replacements must satisfy.

These observations help put into perspective the "Lego block" image often used to discuss reusability. In a Lego set, the components are fixed; the child's creativity goes toward arranging them into an interesting structure. This exists in software, but it seems more like the traditional idea of subroutines. Often, software development needs to do exactly the reverse: keep the structure, but change the components. In fact, the components may not yet be there; in their place, you may find placeholders (deferred features), useful only when you plug in your own variants. In the analogy with children's games, we should go back in age and summon the image of those playboards where toddlers have to match shapes of blocks with shapes of holes.

One can also think of a partially deferred class (or set of classes, called a "library" or a "framework") as having a few electrical outlets—the deferred features—into which the application developer will plug compatible devices.

The expression "programs with holes" comes to mind to describe such components, reusable but open.

From patterns to reusable components. Considerable attention has been devoted in the past two years to a neighboring notion, design patterns. An excellent book on the topic[3] has identified common programming schemes, each associated with a certain problem, and described them in detail.

The book, *Design Patterns: Elements of Reusable Object-Oriented Software*, is an impressive effort. But it is only a

OBJECTS AROUND THE WORLD

A number of important OO conferences will take place in the coming months. Here are some of them in chronological order, each with an e-mail address and Web page for further information. (Speaking of Web pages, the next column will include a list of some of the best on-line OO resources.)

- TOOLS Europe (Technology of Object-Oriented Languages and Systems), Paris, February 26-29, tools@tools.com, http://www.tools.com/tools.
- ISOTAS 96 (Second International Symposium on Object Technology for Advanced Software), Kanazawa (near Tokyo), March 11-15, isotas96@jaist.ac.jp, http://www.jaist.ac.jp/%7Etakuo/ISOTAS96.
- Object Technology 96, Oxford, March 25-27, emsi@login.ieunet.ie, http://www.sis.port.ac.uk/bcs-oops/otmenu.html.
- ECOOP 96 (European Conference on Object-Oriented Programming), Linz, Austria, July 8-12, ecoop96@ifs.uni-linz.ac.at, http://www.ifs.uni-linz.ac.at/ecoop96.
- TOOLS USA, Santa Barbara, Calif., July 29-August 3, tools@tools.com, http://www.tools.com/tools.
- OOPSLA 96 (Object-Oriented Programming Systems, Languages, and Applications), the mother of OO conferences, will be held this year on October 6-10 in San Jose, Calif. The paper submission deadline is February 15. Addresses for information are oopsla96@acm.org and http://www.acm.org.

Also noteworthy is the launching by SIGS Publications, which publishes the *Journal of Object-Oriented Programming* and other OO titles, of a free on-line journal at http://www.sigs.com/objectcurrents, edited by Bob Hathaway (the maintainer of the Frequently Asked Question list on comp.object). The first issue is out. It contains articles by Watts Humphrey ("The Changing World of Software"), David Shang ("Is a Cow an Animal?"), and François Bancilhon ("Object Databases"), as well as an interview of Grady Booch and my own contribution, a discussion of static typing.

first step. *Describing* patterns is not enough; we must take each pattern to its conclusion and provide a number of *directly reusable classes* that encapsulate the pattern. Without such software components, patterns serve a pedagogical need (like the fundamental algorithms and data structures taught in computing science courses), but they do not provide reusability. True software reusability—that is, reusability that can provide the needed breakthrough from current practices—is not reusability of personnel, of ideas, or even of design: It is reuse of actual software components, that is, of code.

What is the main problem? Underlying the previous discussion—which, of course, covers only part of the connection between reusability and object technology—is a view of reusability, developed in detail elsewhere,[4] as primarily a *technical* problem and primarily on the *producer* side. But this view is not universal. In particular:

- A number of organizations, including some well-known computer vendors, have established reuse programs that focus on *consumers* and encourage reuse by application developers. This is, I think, the wrong priority. As the software field evolves, competent developers will reuse components if they are good, but the difficult issues of reusability today are on the producer side: building the thousands of quality reusable components that the industry needs.
- A number of authors view reusability as primarily a *management* problem. The argument is eloquently made, for example, in a recent book[5] by Will Tracz (well known to readers of *Computer*'s Open Channel), which is recommended reading for everyone interested in the topic. Both management and technical issues are important, of course, but we do need to know where to put our strongest efforts.

Much as I appreciate the difficulty of inoculating developers and managers with the Culture of Reusability, I can only reaffirm, after almost 20 years of promoting reuse and building reusable (and reused) components, that the major difficulties lie in technical areas.

Here is a deal: Build the libraries for me, and I will take on your management problems any time.

References

1. Doug McIlroy, "Mass-Produced Software Components," in *Software Engineering Concepts and Techniques (Proc. 1968 NATO Conf. on Software Engineering)*, John Buxton, Peter Naur, and Brian Randell, eds., Van Nostrand Reinhold, 1976, pp. 88-98.
2. Bertrand Meyer, *Reusable Software*, Prentice Hall, Englewood Cliffs, N.J., 1994.
3. Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Menlo Park, Calif., 1995.
4. Bertrand Meyer, *Object Success: A Manager's Guide to Object Technology*, Prentice Hall, Englewood Cliffs, N.J., 1995.
5. Will Tracz, *Software Reuse*, Addison-Wesley, Menlo Park, Calif., 1995.