The new culture of software development

by Bertrand Meyer

Object-oriented design is an old idea and a new idea. The basic concepts have been around for almost twenty-five years, time for more than a few generations when measured against the rate of evolution of the computer industry. Only recently, however, have object-oriented techniques been exposed to enough people and applied to enough projects to yield a concrete idea of the practical power, benefits, and requirements of the method.

This column, adapted from an earlier article,[1] describes some of the issues that arise when the object-oriented approach is implemented on a significant scale. It argues that object-oriented techniques, as represented by Eiffel, imply a new culture of software development, and studies how this new culture can, for the time being, coexist with the old.

The basis for this discussion is the observation of many applications, developed in quite diverse contexts — some by people working with me, others without any direct involvement on our part.

THE PROJECT CULTURE

Object-orientedness is not just a programming style, but the implementation of a certain view of what software should be. Taken seriously, this view implies a profound rethinking of the software process.

1 *The New Culture of Software Development: Reflections on the Practice of Object-Oriented Design*, TOOLS 1, PROCEEDINGS OF TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS, Paris, June 1989, SOL/ANGKOR, Paris, 1990.

How profound is best understood by contrasting the mode of development implied by object-oriented techniques with the most common culture of software engineering.

That traditional culture is project-based. This means that the subject of discourse is the individual project, tailored to one specific set of requirements, and having as its goal the delivery of a program with the supporting documents.

A typical example of this view is Barry Boehm's well-known (and useful) book *Software Engineering Economics* [1], a 767-page discussion entirely predicated on the assumption of a linear lifecycle, meant at solving one particular problem. The mental frame of reference in that case is the project. This project starts at day one with, as its input, some large user's specific need. It ends some months or years later with a solution to that need — or, as the case may go, with no usable result at all, the book's purpose being to reinforce the likelihood of the former alternative.

Outcome	Results
Economics	Profit
Unit	Department
Time	Short-term
Goal	Program
Bricks	Program elements
Strategy	Top-down
Method	Functional, structured analysis/design, entity-relation, dataflow, Merise ...
Language	some PDL, C, Pascal ...

Figure 1. The project culture.

Typical of the assumptions is the way Boehm's introductory note (p. xxiii) attempts to awaken the student reader's awareness of the book's relevance:

> *There is a good chance that, within a few years, you will find yourself together in a room with a group of people who will be deciding how much time and money you should get to do a significant new software job.*

Note the use of the word *new*, which appears again later (p. 29) in the description of the lifecycle as starting *"from the earliest exploratory phases in which the feasibility of a new software product is addressed."*

"Product" here is always used in the singular; one senses that it is really there to mean "program," the more general term being used mainly to encompass other artifacts resulting from a project, such as end usage procedures.

Such assumptions are particularly surprising in a discipline where authors have fought and continue to fight so hard to obtain the recognition conferred by the title "software engineering." Surely, engineering in other fields — electronics, construction, automatic control — is characterized by a search for common building blocks and an effort to diminish, as much as possible, the "new" part in every development.

Boehm's classic treatise is not at issue here; it is merely representative of the general project view, pervasive through the software engineering literature. Some of the implications of this view, taken to the extreme, are summarized in Figure 1.

The outcome is results, produced by a program in response to user's requirements. The economics is one of profits, as produced by the results.

The organizational *unit* impacted is usually the department directly affected

by the project. The *time* frame is as short as it will take to produce the required solution. The *goal* is a program, or a few programs. The *bricks* of which this program is made are program elements: modules built for the occasion.

The *strategy*, as recommended in most textbooks and procurement policies, is top-down: start from the specific problem requirements and decompose. The *method* that follows naturally is based on analysis of the functions and data flow. The *languages* used for analysis, design, and implementation are any of the classical languages.

A corollary of the project culture is a highly sequential lifecycle model. The waterfall model, although somewhat of a caricature, still serves as a theoretical reference for many organizations. One of its many variants is shown in Figure 2.

THE COMPONENT CULTURE

The culture implicit in object-oriented design is quite different. It may be called the component culture: the subject of discourse is reusable components rather than individual projects. Some of the implications of this view, taken to the extreme, are summarized in Figure 3.

The outcome is reusable software elements, meant to be useful to a large number of applications. The economics is one of investment — which, of course, is intended as deferred profit.

The unit is, beyond an individual project, a department, a company, and sometimes an entire industry. The time frame is long-term. More than a program, the goal is to build systems. The bricks are *software components*, which distinguish themselves from mere program elements by having a value of their own, independently of the context for which they were initially designed. More will be said below about *generalization*, the task of transforming program elements into software components.

The strategy for obtaining quality reusable components embodies a considerable *bottom-up* aspect: working by extension, improvement, specialization, and combination of previously obtained components. This is exactly what the object-oriented method supports, thanks to multiple inheritance, genericity, assertions, deferred classes, and encapsulation.

The language used at the analysis, design, and implementation stages should reflect this method. The corresponding entry in Figure 3 has been left for the reader to fill, as a quiz to test how well you understood the previous installments of this column.

COHABITATION

The above characterizations are somewhat extreme. No industrial software development environment totally neglects tools; few can afford to neglect results. But the contrast between project and component cultures shows some of the problems associated with promoting object-oriented techniques on a broad scale.

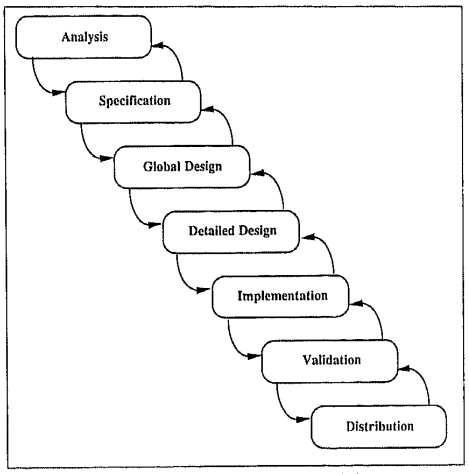

Figure 2. The waterfall model.

Outcome	Tools, libraries
Economics	Investment
Unit	Industry
Time	Long-term
Goal	System
Bricks	Software components
Strategy	Bottom-up
Method	Object-oriented
Language	

Figure 3. The component culture.

Without question, the dominant culture is project-based and will remain so for a long time. Customers, users, management, and shareholders all want results, and preferably fast. Posterity will come later.

The immediate issue then is not so much how to *replace* the project culture by a component culture, an impossible goal at least initially, but how to *instill* significant doses of component-oriented concerns into a context that is largely driven by project preoccupations.

One of the all-time favorite strategies of subversives — penetrating reactionary institutions rather than destroying them outright — indeed seems to work here.

Assume that, being an advance soldier of the object-oriented army, you are assigned the job of MIS director in some large, traditional computing organization. You can hardly decide, on your first day on the job, that all requests for specific developments will be turned down for two years, time for your department to build the right base of reusable components. You have users and customers, and must be ready to respond to their specific requests.

Catering to the short term does not mean, however, that you give up on tools and reusability. You will fulfill your customers' specific requests, but you will do more than these requests, seeing the eventual software components beyond the immediate program elements.

The effort involved in transforming program elements into software components may be called generalization and will be studied in more detail below. It involves abstracting from the original program elements so as to make them independent from their original context, more robust, better documented. Giving generalization a systematic role in the software development process is the key step in the progressive transition from project to component culture.

By starting from specific requests but going further, you can quietly start accumulating a repertoire of ready-made components that, little by little, will play an increasing role in your subsequent developments. With such a strategy you can, after a while, introduce a new attitude towards your users — more active and less reactive. You can respond to a new request, with its specific and sometimes baroque set of technical requirements, with a counterproposal, offering to do a somewhat different or perhaps simplified job much faster thanks to the use of preexisting components. Then you can give your customers a choice: ei-

ther tailor-made development, using traditional techniques, in *n* person-months, or "mix-and-match" development using object-oriented techniques in, say, 0.3 *n*. Some offers are hard to refuse.

GENERALIZATION

What does it take to transform a program element into a software component? Some aspects of this generalization process are obvious, and not specific to the object-oriented approach:

- Writing more complete documentation — perhaps unnecessary for an element that is only used as part of a given program, but required for its independent use as a component.
- Removing functional limitations — which may be tolerable when you have full control over a component's use, but not in a more general context.

Others, however,, are less straightforward: assertions; abstraction through inheritance; factoring out commonalities. The next few sections address them.

GENERALIZATION: ASSERTIONS

One of the fundamental generalization tasks is to add the proper assertions to the components. An assertion is an element of formal specification that characterizes the implementation-independent properties of a software unit — routine or class — in object-oriented programming. Assertions include particular preconditions, postconditions, and invariants (see [Meyer88]).

A routine precondition expresses under what condition the routine may correctly be called. For example, an insertion routine for a table of bounded capacity might have the precondition

```
require
  count < capacity
```

A routine postcondition expresses the abstract properties of the state resulting from a correct call to the routine. The postcondition for a routine inserting x with key k might be written as

```
ensure
  count = old count + 1;
  item (k) = x
```

where **old** serves to refer to a "snapshot" of a value (here *count*, the number of elements inserted) taken before the call, and a function *item* is assumed on tables, yielding the value associated with a certain key.

A class invariant expresses global consistency properties associated with all instances of a certain class, for example

```
count <= 0;
count <= capacity
```

For a mere program element, programmers are sometimes lazy about including the proper assertions. For a software component, this would be unacceptable: without assertions, it is not possible to produce truly industrial software components. They would be like electronic components without a precise specification of their accepted inputs, guaranteed outputs, and general conditions of use — the hardware equivalents of preconditions, postconditions, and invariants.

Adding assertions is thus an important part of the generalization process. Invariants, in particular, are not always understood right away; it takes some research into a class and often some practical use to obtain all the right invariant clauses. The result is always worth the effort, as the process of deriving the invariant yields considerable insights into the deeper semantics of the class.

The presence of assertions as integral parts of the language permits applications such as automatic documentation (producing class interfaces from the class text, as with the "short" tool of the Eiffel environment) and debugging (as with the Eiffel compilation options that make it possible to check assertions at run-time).

ABSTRACTIONS AND FACTORING

Two other interesting aspects of generalization have to do with how we obtain good inheritance structures. They may be called *class abstraction* and *extraction of commonalities*. (For further discussion see [Meyer90], which presents these techniques as applied to the evolution of the Eiffel libraries, [Casai90], and [Johns88].)

In both cases, when you have not been able to obtain inheritance structures in the orthodox way (from more abstract to more concrete) as recommended by theoretical presentations of object-oriented concepts, you may need to "switch into reverse" and produce a deferred version of an Eiffel class only after more concrete versions have been obtained, used, and analyzed.

The first case, abstraction, occurs when developers realize that a certain class *C*, which was meant to represent a certain concept, in fact describes only one implementation of that concept. Reestablishing the normal inheritance hierarchy, by adding a deferred class *B* as an ancestor of *C*, will preserve the future consistency of the class structure. It would have been better, of course, to obtain Figure 4 right from the start; but if this was not the case, better late than never. To continue with clichés, if to err is human, to persevere in not recognizing your parents would be diabolical.

Factoring of commonalities (Fig. 5) is similar. This occurs when you realize too late that two variants of a certain notion have given rise to two separate classes, with no common ancestor, but probably many redundancies, simply because no one recognized early enough that two closely related developments were going on. Again, for the sake of your class library's future evolution, you should cut your losses and accept the need to reorganize the hierarchy *a posteriori*.

ORGANIZATIONAL ASPECTS

Object-oriented development, the emphasis on reuse, and, more generally, a trend toward the component culture inevitably have consequences on the organizational and managerial aspects of software development, a few of which will be considered here.

The newest aspect, as discussed above, is the generalization step. This will cost money; not necessarily fortunes — depending on one's ambitions, the overhead on standard development costs may be anywhere between 10 and 50% — but hardly invisible.

This means, among other consequences, that serious object- oriented development cannot be done "on the side." Without management support, you can perform a few harmless experiments, but not implement true object-oriented design and programming with their immediate consequence: the development of investment-oriented tools and components.

The budgeting problem should not be overlooked. In most corporate environments, budgets reflect the surrounding project culture and are allocated on a project basis; "general" funds, not earmarked for a particular project, are usually much more limited. Yet, the generalization activity does not profit the current project so much as the next few projects — which, adding insult to injury, may well be under the responsibility of the project leader's peers and rivals in the career role! Mechanisms must be found to obtain funding for such undertakings — project-foolish, component-wise.

Another practical caveat concerns productivity. Standard productivity measurements, based on lines per person-months, may be deceptive. Assume a project that enthusiastically adopts object-oriented techniques. At the end of an initial development, a first measurement is made:

```
PROD1 = LINES1/EFF1
```

where *PROD1* is the productivity, measured as the ratio of the number of produced lines, *LINES1*, to the effort *EFF1*, measured in person-months or using some better criterion if there is one.

No doubt that if object-oriented techniques have been applied well and with good tools, *PROD1* will be a pleasant surprise to management as compared to the usual results. Assume now, however, that the project leader decides to go on and apply the generalization step. After a while, a new measurement is made:

```
PROD2 = LINES2/EFF2
```

Obviously, *EFF2* is greater than *EFF1*. But it may very well be the case that *LINES2* is less than *LINES1:* after all, much of the generalization work consists in removing duplicate elements (in particular as a result of "extraction of commonalities") and other dead wood. Unless properly briefed, management (and software engineers) will not like these figures.

If anything, this hypothetical story highlights the danger of simplistic approaches to assessing productivity improvements (see also [Gindr89] and [Jones86]). It also serves to remind us of the need to involve and educate management, always eager for figures showing immediate productivity gains. Although the productivity gains are undeniably there, we should not forget that in switching to object-oriented software engineering the really big prize is to be won over the long term, thanks to reuse.

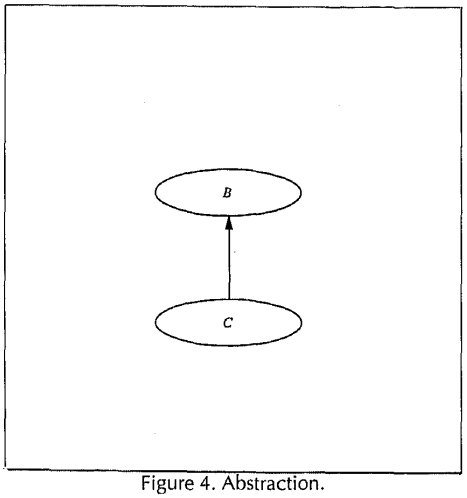

Figure 4. Abstraction.

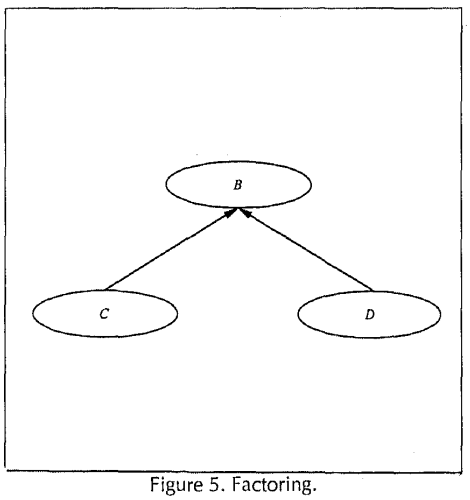

Figure 5. Factoring.

ABSTRACTION POLICE

Another important management issue is the question of who should be responsible for the generalization activities mentioned above: class abstraction and factoring out commonalities.

In theory, it could be the developers themselves, and if management has clearly emphasized the need for reuse and the long-term commitment to building a base of reusable software, developers may be expected to play their part in the collective search for generality. But this will not suffice if the goal is to establish a serious, organization-wide reuse base. Developers are inevitably prisoners, at least in part, of the project mood. They have immediate goals to fulfill; they have immediate goals to fulfill and deadlines to meet.

It appears necessary to designate a specific group of people whose mission is officially component-oriented and project-independent. The charter for this group (which typically will remain small) is to detect potential reusable components, perform the generalization steps as outlined above to remove their unjustified ties to specific projects or circumstances, and catalog them appropriately for easy retrieval.

We might call that group the "reusability brigade" or the "abstraction police," although most of the companies I know will probably choose a more sedate title such as "reuse administrator." In fact, such a name would be, not inappropriately, reminiscent of the traditional position of "database administrator." A database administrator's charter is to develop the organization's data investment and maintain its consistency; the reuse administrator does the same for the company's *software* investment. The job also includes other aspects that resemble the task of quality assurance engineers.

CLUSTER MODEL OF THE SOFTWARE LIFECYCLE

We will conclude with a brief discussion of the lifecycle model that seems most appropriate for the Eiffel-based component culture. (This section draws heavily on a previous article [Meyer89] and on a report by Eiffel users from Thomson [Gindr89].)

Despite the frequent criticism of the waterfall model, no satisfactory replacement has gained widespread acceptance. Variants such as the incremental model [Boehm82] or the spiral model [Boehm88]

make the process more flexible, but deviate from the fundamental tenet of the project culture, the single-product hypothesis. What kind of lifecycle is appropriate to the component culture and to Eiffel design?

Here are some of the main ingredients of a possible answer (see also [Hende90] for further developments):

- The merging of the design and implementation activities, traditionally considered to be different phases of the lifecycle.
- The general bottom-up approach, which deemphasizes the immediate requirements of the current project in favor of a long-term view of software production, and suggests that general-purpose utility modules should be built first, specific ones last.
- The new lifecycle phase described above: generalization.

One more concept is needed to complete the picture: the cluster concept. In Eiffel, a cluster is a group of classes that relate to a common aim; for example, a system could contain a basic cluster (the Basic Eiffel Library), a graphics cluster (the Eiffel Graphics Library or another set of graphics classes), a simulation cluster, a synchronization cluster, etc.

With this notion in mind, we can take a fresh look at the waterfall model. The continued success of this model in the software engineering literature, in spite of its known deficiencies, should perhaps be credited to two of its properties, already noted by Boehm ([Boehm82], pp. 38–41):

- The steps of the waterfall — analysis, specification, design, implementation, validation, and distribution — reflect meaningful and necessary activities of software construction, although, as we have seen, it may be appropriate to merge some adjacent pairs.
- It is hard to imagine a theoretically more satisfying order than the one given: although some readers will probably be able to draw counterexamples from their project experience, who would seriously advocate that developers must start the analysis after they have implemented or distributed the system?

We may realize, however, that nothing really forces us to apply this sequence of steps *to the system as a whole.* This would be keeping the negative legacy of top-down design: the all-or-nothing approach that considers system a monolithic entity fulfilling a frozen specification. The notion of cluster provides the appropriate unit to which each sublifecycle should be applied. As shown in Figure 6, these sublifecycles may overlap in time, and I believe they should.

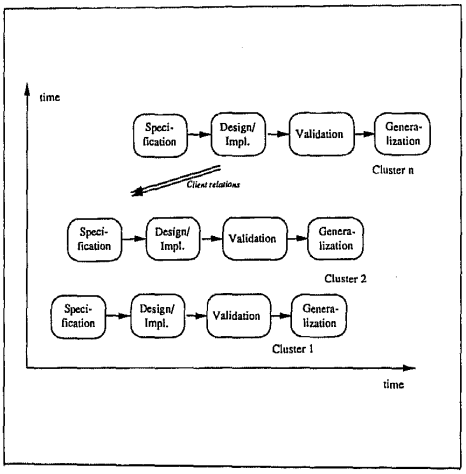

Figure 6. The cluster model.

The other ideas developed so far help further define this new lifecycle model, the cluster model of software development:

- The best order for starting cluster development is bottom-up: from the most general clusters, providing utility functions, to the most application-specific ones. Of course, some of the lower-level clusters will be available from the start as part of the standard delivery (in Eiffel, the Data Structure and Graphical Libraries). As the method is applied to repeated projects within an organization, other reusable clusters will become readily available.
- As opposed to the all too frequent advice of getting the interface right first (what may be called the "Potemkin approach," where the facade must be right at all costs, even if there is nothing behind), this strategy suggests that the key functions should be designed and implemented first, and one or (usually) more interfaces should then be built to satisfy needs. These may be program interfaces, command-line-oriented interfaces, full-screen interfaces, graphical ones, and so on.
- A possible sequence to apply to each sublifecycle includes the following three steps: specification; design, and implementation; validation; and generalization. (Gindre and Sada suggest in [Gindr89] that the last two may be merged.)
- Each cluster may be a client of lower-level ones. The client relation enables the design/implementation of the classes in a cluster to rely on the specification of classes in another. In contrast with hierarchical abstract machine methods, we should not require that each cluster only be a client of the immediately lower one; we may restrict, however, cycles of the client relation to occur within clusters only.
- As long as we do not start a more specific cluster before a more general one, we have many degrees of flexibility. At one extreme, we might work on just one cluster at a time, beginning with the most general ones. At the other extreme, we might work on all clusters in parallel, which would essentially take us back to the waterfall model. In between, many variants are possible, and we should choose according to how well we understand each part and what resources we have.

Although these ideas need more work to yield a full-fledged process model, I have found them, at their current stage of evolution, to yield a software development process that is smoother and more effective than traditional approaches because it integrates at its very basis the concern for change and the concern for reuse. In other words, it helps in the key transition that is required for the turning of software development into a real industry: the transition from a project culture to a component culture.

Next Column: With static you won't get the message, or why we need dynamic binding.

REFERENCES

[Boehm82] B.W. Boehm. SOFTWARE ENGINEERING ECONOMICS, Prentice-Hall, Englewood Cliffs, NJ, 1982

[Boehm88] B.W. Boehm. A Spiral Model of Software Development and Enhancement, IEEE COMPUTER, 21(5), 61–72, 1988.

[Casai90] E. Casais. Managing Class Evolution in Object- Oriented Systems, in OBJECT MANAGEMENT/GESTION D'OBJETS, D. Tsichritzis, ed., Centre Universitaire d'Informatique, Université de Genève, July 1990, pp.133–196.

[Gindr89] C. Gindre and F. Sada. A Development in Eiffel: Design and Implementation of a Network Simulator, JOURNAL OF OBJECT- ORIENTED PROGRAMMING, 2(2), 27–33, 1989.

[Hende89] B. Henderson-Sellers and J.M. Edwards. Object-Oriented Systems Lifecycle, COMMUNICATIONS OF THE ACM, 33(9), 143–159, 1990.

[Johns88] R.E. Johnson and Brian Foote. Designing Reusable Classes, JOURNAL OF OBJECT-ORIENTED PROGRAMMING, 1(2), 2235, 1988.

[Jones86] T.C. Jones. PROGRAMMER PRODUCTIVITY, McGraw-Hill, New York, 1986.

[Meyer88] B. Meyer. OBJECT-ORIENTED SOFTWARE CONSTRUCTION, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Meyer89] B. Meyer. From Structured Programming to Object-Oriented Design: The Road to Eiffel, STRUCTURED PROGRAMMING, 10(1), 19–39, 1989.

[Meyer90] B. Meyer. Tools for the New Culture: Lessons from the Design of the Eiffel Libraries. COMMUNICATIONS OF THE ACM, 33(9), 69–88, 1988.

Bertrand Meyer is President of Interactive Software Engineering, based in Santa Barbara, CA. He just published INTRODUCTION TO THE THEORY OF PROGRAMMING LANGUAGES *(Prentice-Hall, 1990), which focuses on denotational and axiomatic semantics; by the time this column appears, his next book,* EIFFEL: THE LANGUAGE, *a full presentation of Eiffel syntax and semantics, should be available. He can be reached at Interactive Software Engineering Inc., 270 Storke Road, Goleta, CA 93117, by telephone 805-685-1006, Fax 805-685-6869, or on email bertrand@eiffel.com.*