Chapter 2
The New Culture of Software Development:
Reflections on the practice of object-oriented design

Bertrand Meyer
Interactive Software Engineering, Inc.

ABSTRACT: Object-oriented techniques, when applied seriously and on a broad scale, reflect a new culture of software engineering, which may be called the *component culture*. After contrasting this new culture with the more traditional *project culture*, this article examines some of the technical, economical and managerial implications of the new approach.
The discussion explores the fundamental object-oriented processes of *abstraction* and *extraction* (recognizing inheritance structures a posteriori), and introduces a new lifecycle model which seems to fit best with object-oriented development of reusable software: the Cluster Model.

2.1 OVERVIEW

Object-oriented design is an old idea and a new idea. Simula introduced the basic concepts more than twenty years ago, time for 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 approach.

This article describes some of the issues that arise when the object-oriented approach is implemented on a significant scale. It argues that object-oriented techniques 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 a large number of object-oriented applications over a period of many years, initially in Simula, and then in Eiffel. Some of these applications were developed under my leadership; others were built by fellow users (trained by us or by others) and by users of our tools. No attempt has been made to transpose this experience to environments other than Eiffel.

2.2 THE TWO CULTURES

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

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

Outcome	Results
Unit	Department
Economics	Profit
Time	Short-term
Goal	Program
Bricks	Program elements
Strategy	Top-down
Method	Functional
Language	C, Pascal, ...

Figure 2.1 The project culture.

The traditional culture – implicit in most of the software engineering literature, and in the usual software lifecycle models – is project-based. In other words, the subject of discourse is the project, which starts with a certain specification and ends with the delivery of a program with the supporting documents.

Some of the implications of this view, taken to the extreme, are summarized in Figure 2.1 above.

The *outcome is* results, produced by a program in response to user's requirements. The organizational *unit* impacted is usually the department directly affected by the project. The *economics is* one of profit, with a *time* frame as short as it will take to obtain this profit. 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* which follows naturally is based on analysis of the functions and data flow. The *language* used is one of the classical languages.

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 on Figure 2.2 below.

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

Figure 2.2 The component culture.

The outcome is reusable software elements, meant to be useful in a large number of applications.

The unit is, beyond an individual project, an organization: a department, an entire company, sometimes an entire industry. The economics is one of investment – which of course is intended as deferred profit. As a consequence, the time frame is more long-term. More than a program, the goal is to build systems – assemblies of modules which can be adapted to suit various specific needs. 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 the transition from program elements to 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 and encapsulation. The language used at the specification, design and implementation stages should reflect this method.

2.3 COHABITATION

The above characterizations are exaggerated. 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.

Without question, the dominant culture is project-based and will remain so for a long time. Customers, users, management, 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 which is largely driven by project preoccupations.

One of the favorite strategies of all-time subversives – penetrating institutions to

transform them from the inside, rather than toppling 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 fulfil 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. Its goal is to give the elements a value of their own, independently of the original developments for which they may initially have been conceived. In this process, the elements become more general, more robust, better documented.

In the pure project culture, there is no room for generalization: if a program element satisfies the requirements thrust on it by the program to which it belongs, there is no economic justification for an activity that will not have any immediate effect on the quality of this program, being aimed solely at improving the element's usefulness for *future* developments. But in the component culture generalization becomes a key element of the software development process.

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

2.4 GENERALIZATION

What does it take to transform a program element into a software component?

Some aspects of this generalization process are obvious, and are not specific to the object-oriented approach:

- Writing more complete documentation – perhaps not necessary for an element which is only used as part of a given program, but, rather, is 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.

Some less obvious aspects, such as assertions, abstraction and the factoring out of commonalities, deserve a specific discussion

2.4.1 Assertions

One of the important tasks of the generalization process is to add the proper assertions to the components. An assertion is an element of formal specification which characterizes the implementation-independent properties of a software unit – routine or class in object-oriented programming. Assertions include in particular preconditions, postconditions and invariants.

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 a, *l* 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.

Although assertions can in principle be added as comments in any language, their inclusion 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 which turn assertion monitoring on).

2.4.2 Class abstraction

Another important aspect of generalization is class abstraction. The need for this activity is a consequence of a universal characteristic of programmers, which they share with their fellow human beings: the yearning for the concrete.

Object-oriented design is a quest for abstraction. Using inheritance means that you write classes that are more general than what is immediately needed for the problem at hand. *Deferred* classes, which describe general mechanisms (scenarios, scripts, iterators...) without committing the details of each step, are particularly useful here: once you have captured a general pattern through a deferred class, you or others may implement specific variants by writing effective (non-deferred) classes which implement the parts of the pattern that had been left open in the deferred class. Object-oriented techniques ideally support this remarkably elegant process of working from the abstract to the concrete, from the general to the specific[1].

In practice, however, the scheme is not always as smooth and intellectually satisfying as the theory of object-oriented development would have it. Even if they are convinced of the benefits of generality, developers will tend to produce classes which initially are often too specific: particular implementations of a certain abstraction, rather than the abstraction itself. It is hard to blame them: programmers are problem solvers. Nobody will complain if they get the job done first.

If reusable components are part of the goal, however, the process cannot stop there. When you realize that a certain class is less general than it could have been, you should use this discovery as an opportunity to reorganize the inheritance hierarchy. As a simple example, this is what happened in the design of the Eiffel Data Structure Library when we realized that our initial *TREE* class was too specific, describing just one implementation of trees rather than the general concept. A deferred class was then written, of which the original became an heir.

---

[1] Because of the common graphical representation for inheritance diagrams, this process is sometimes mistakenly viewed as "top-down". It is in fact a typically bottom-up process of particularizing general-purpose tools.

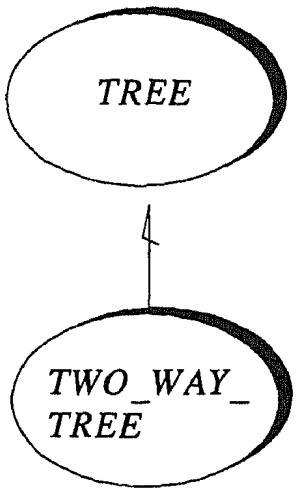

Figure 2.3 Partial inheritance structure for trees.

The resulting inheritance structure, shown below, could of course have been obtained right from the beginning; but it is better to produce it late than never.

Such an a posteriori change is typical of work that only makes sense in the component culture. From the project viewpoint, the original program element was satisfactory and there would have been no economic incentive to improve the enclosing inheritance hierarchy.

The process of class abstraction is aided in the Eiffel environment by a variant of the **short** class abstracter. The command

short -e class name

will produce a deferred version of the requested class, with all implementation details removed. This is usually a good basis for obtaining a more abstract class while keeping the interface.

2.4.3 Extraction of commonalities

A related activity arises from the a posteriori realization that duplication of efforts has led to similar classes being written by different people, or even by the same person at different times. Inheritance is the ideal mechanism for capturing commonalities between similar components. If the developers initially missed the commonalities, then it is always possible to reconstruct the inheritance structure a posteriori.

The need for "extraction of commonalities" arises when two developers, say John and Jill, produce classes B and C, not realizing early enough that they were in reality working on variants of the same concept. Had they recognized this from the start, they would have been working on the hierarchy shown below, but instead they wrote B and C as independent classes. Here again, it is better to reinstate the hierarchy *ex post facto* than not at all.

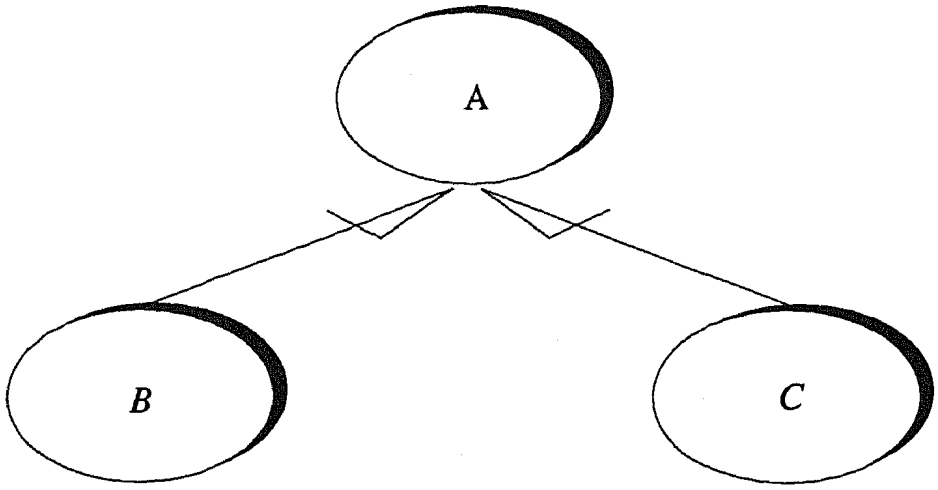

Figure 2.4 Inheritance structure for variants of a common concept.

As with the previous case, the result is to produce more abstract classes, often deferred, of which the original classes become descendants.

As an example, also from our own development, we initially developed the Winpack Library (covering non-graphical windowing) and the Eiffel Graphical Library as independent products. Only later did we realize that the notion of window, and many other central concepts, should be common to both libraries, with specialization for graphical or non-graphical platforms intervening quite low in the inheritance graph.

2.4.4 Switching to reverse

What is common to the previous two activities – abstraction, extraction of commonalities – is that they depart from the view of inheritance which is usually suggested in the object-oriented literature: the idea that the bright designer will somehow obtain the proper inheritance structure the first time around. It is always preferable, of course, to get the inheritance right initially. But it serves no useful purpose to pretend that this will always be the case. Better recognize that the process may involve trial and error, as a result of our yearning for the concrete, and of our frequent failure to detect commonalities early enough. Better be prepared for the inevitable changes of direction – switching to reverse, as it were – in building the inheritance structure. What counts is that in the end we should get the useful and elegant inheritance hierarchies that condition good object-oriented reuse of components.

An important aspect of both abstraction and extraction is that they normally do not affect the clients of the classes being restructured, since one does not change the interface of a class by rewriting it with a different ancestry. In Eiffel, clients will not even be recompiled, since the automatic (makefile-free) recompilation mechanism will recognize that an interface has remained untouched and that the clients are hence still valid as compiled before.

This observation highlights a fundamental, although often misunderstood, aspect of inheritance: inheritance is an implementation mechanism, not an interface mechanism. For the clients of a class, what the class inherits from is irrelevant. Such tools as the Eiffel flattener (command flat of the Eiffel environment) support this view by providing inheritance-free versions of a class when needed for the benefit of clients.

As a result of the abstraction and extraction activities, a general phenomenon may be observed in organizations that have consistently and seriously promoted reusability through object-oriented techniques. This phenomenon – apparent in our own developments, and reported by users of Eiffel – is a progressive elevation of the level of abstraction of the classes produced by a group or organization committed to object-oriented programming. As you start reusing your previous classes, cataloging them, archiving them into libraries, you realize the need for more general versions. It does not make sense to lament that these versions were not produced right from the start; what counts is the constant improvement in quality and generality that the process yields if properly implemented.

2.5 SOME ORGANIZATIONAL ASPECTS

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

The newest aspect, as discussed above, is the generalization step. This will cost money; not necessarily fortunes, 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 you cannot implement true object-oriented design and programming with their immediate consequence: the development of investment-oriented tools and components.

The funding 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 race for corporate leadership!). 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:

$$prod_1 = lines_1 / eff_1$$

where $prod_1$ is the productivity, measured as the ratio of the number of produced lines, $lines_1$, to the effort in person-months, eff_1[2].

No doubt, if object-oriented techniques have been applied well and with good tools, $prod_1$ 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:

$$prod_2 = lines_2 / eff_2$$

Obviously, there is a relation of the form

$$eff_2 > eff_1$$

But it may also very well be the case that

$$lines_2 < lines_1$$

since, after all, much of the generalization work consists of removing duplicate elements, in particular as a result of "extraction of commonalities" as studied above, and other dead wood. Unless properly briefed, management (and software engineers) will not like the resulting productivity figures.

If anything, this hypothetical story highlights the danger of simplistic approaches to assessing productivity improvements (see also [3]). It also serves to remind us of the need to involve and educate management, and to emphasize that, real as the short-term productivity gains are with a good object-oriented environment, the really big prize is to be won over the long term, thanks to reuse.

2.6 LIFECYCLE: THE CLUSTER MODEL

We will conclude with a brief discussion of the lifecycle model that seems most appropriate for the object-oriented product culture. (This section draws on a previous article [5] and on a very interesting report by Eiffel users from Thomson [6]. See also [2] for further developments.)

The well-known waterfall model, of which a form appears in Figure 2.5, has been repeatedly criticized. Yet no satisfactory replacement has gained widespread acceptance. It is fair to ask what kind of lifecycle is appropriate to object-oriented design.

Here are some of the main ingredients of a possible answer:

- The merging of the design and implementation activities, traditionally considered to be different phases of the lifecycle.

---

[2] Such simple productivity measures are of course subject to criticism; concepts such as "person-month" and "line" need to be defined precisely.

- The general bottom – up approach, which de-emphasizes 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, which seems to be profitably merged with the more usual phase of component validation.

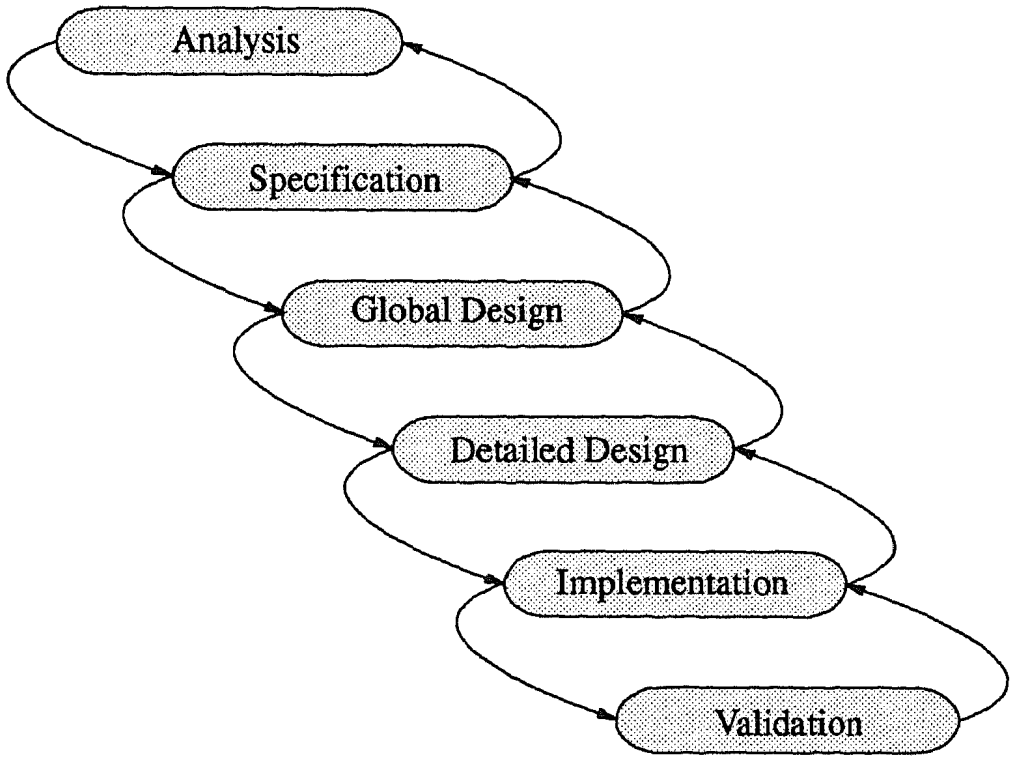

Figure 2.5 The waterfall model.

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

In Eiffel there is no need to define "cluster" as a language construct because the notion of directory, available on all modern operating systems, provides an appropriate basis. Eiffel classes are stored in files; the files containing a set of logically related classes will naturally be kept in the same directory.

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 [1]:

- The lifecycle steps (requirements, specification, design, implementation, validation) reflect meaningful and necessary activities of software construction,

  - although we have seen that it may be appropriate to merge some adjacent activities,

and encountered the need for a new activity, generalization, conspicuously absent from the traditional model.

- It is hard to imagine a theoretically more satisfying order than the one given: who would seriously advocate distributing before specifying?

If the activities are essentially right and the order is right, what then can be improved? An important observation is that nothing really forces us to apply the above 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 which considers a system as a monolithic entity fulfilling a frozen specification. The notion of cluster provides the appropriate unit to which each sub-lifecycle should be applied. As shown on Figure 2.6, these sub-lifecycles may overlap in time, and they probably should.

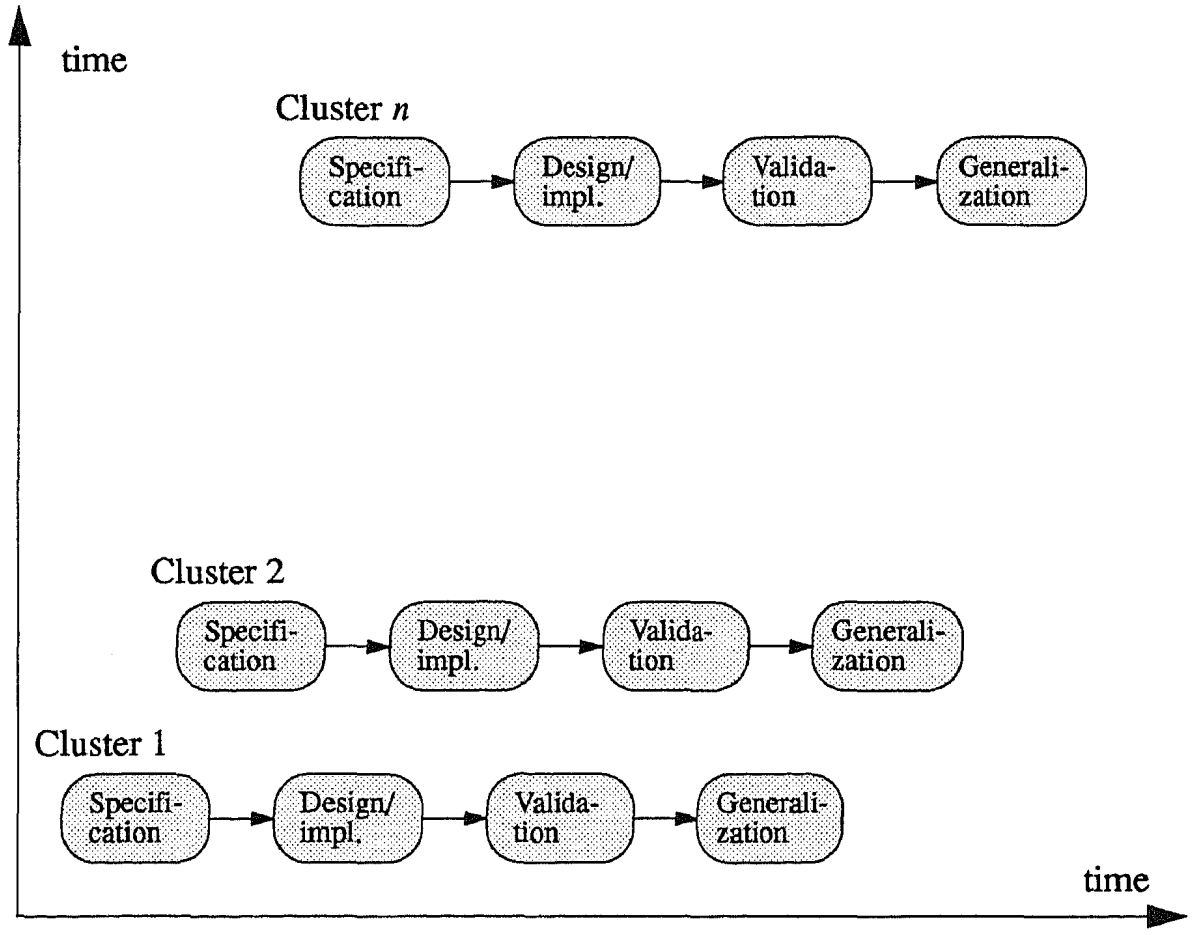

Figure 2.6 The cluster model of the software lifecycle.

The other concepts introduced so far help further define this new lifecycle model – the cluster model of software development. It is based on the following principles.

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): and 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 sub-lifecycle includes the following four steps:

- Specification of individual clusters and classes.
- Design and implementation.
- Validation.
- Generalization.

Each cluster may be a client of lower-level ones. The client relation enables the design-implementation step 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.

This approach appears to yield a software development process which is smoother and more effective than traditional approaches because it integrates at its very basis the concem for change and the concem for reuse; in other words, because 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.

REFERENCES

[1] Barry W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[2] Brian Henderson-Sellers and Julian M. Edwards, "The Object-Oriented Systems Life Cycle", in *Communications of the ACM*, vol.33, 9, pp.142–159, September 1990.

[3] T. Capers Jones, *Programmer Productivity,* McGraw-Hill, 1986.

[4] Bertrand Meyer, *Object-Oriented Software Construction,* Prentice Hall, 1988.

[5] Bertrand Meyer, "From Structured Programming to Object-Oriented De sign: The Road to Eiffel", in *Structured Programming,* vol.10, 1, pp. 19–39, January 1989.

[6] Cyrille Gindre and Frederique Sada, "A Development in Eiffel: Design and Implementation of a Network Simulator"; in *Advances in Object-Oriented Software Engineering,* eds. Dino Mandrioli and Bertrand Meyer (this volume).