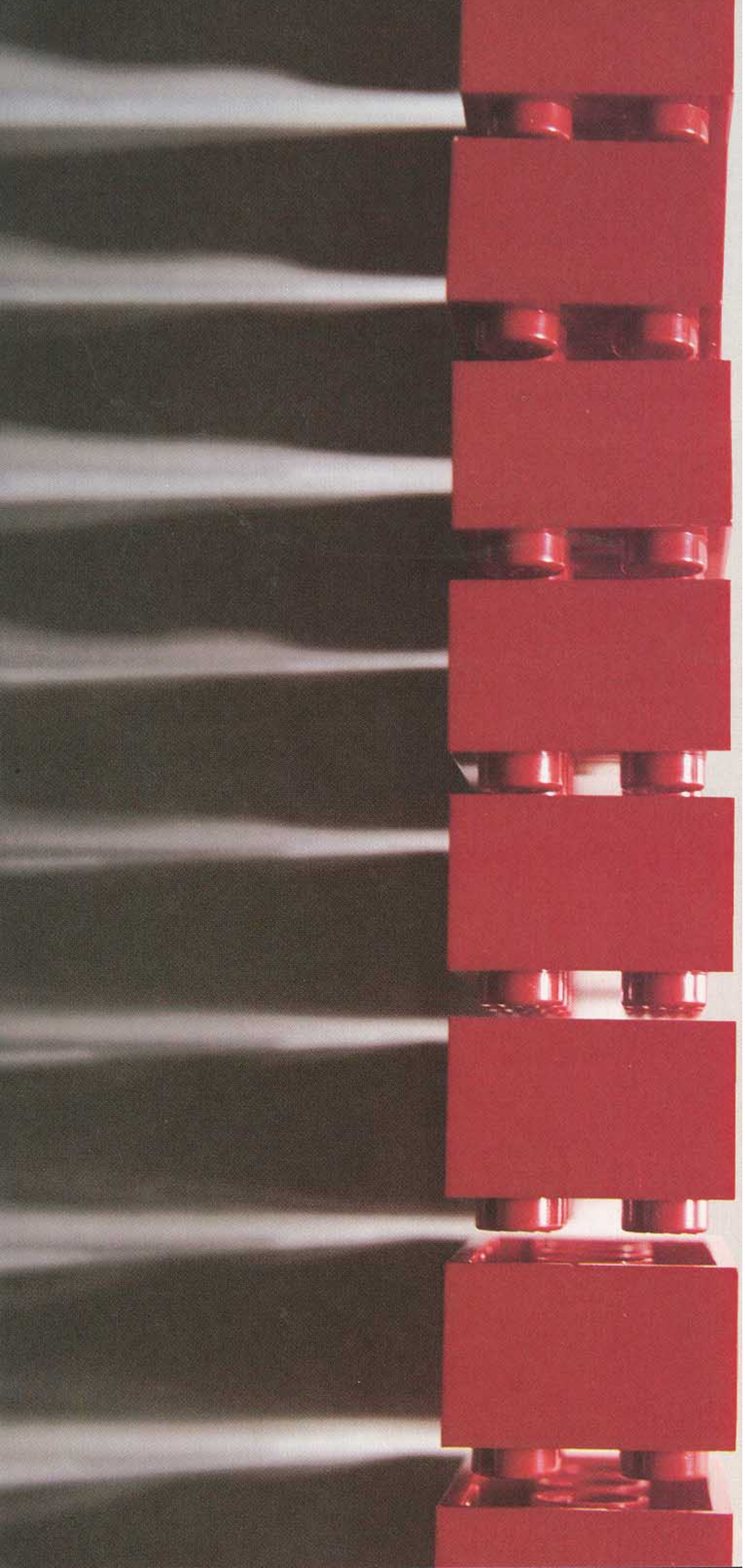


The nature of programming is changing. Most of the software engineering literature still takes for granted a world of individual *projects,* where the sole aim is to produce specific software systems in response to particular requirements, little attention being paid to each system's relationship to previous or subsequent efforts. This implicit model seems unlikely to allow drastic improvements in software quality and productivity.

Such order-of-magnitude advances will require a process of industrialization, not unlike what happened in those disciplines which have been successful at establishing a production process based on the reuse of quality-standardized components. This implies a shift to a "new culture" [14] whose emphasis is not on projects but instead on *components.*

The need for such a shift was cogently expressed more than 20 years ago by Doug McIlroy in his contribution, entitled *Mass-Produced Software Components* [10], to the now-famous first conference on software engineering:

Software production today appears in the scale of industrialization somewhere below the more backward construction industries. I think its proper place is considerably higher, and would like to investigate the prospects for mass-production techniques in software. [...]

My thesis is that the software industry is weakly founded [*in part because of*] *the absence of a software components subindustry* [...] *A components industry could be immensely successful.*

Although reuse has enjoyed modest successes since this statement was made, by all objective criteria McIlroy's prophecy has not been fulfilled yet; many technical and nontechnical issues had to be addressed before reuse could become a reality on the scale he foresaw. (See [1] and [20] for a survey of current work on reuse.) One important development was needed to make this possible: the coming age of object-oriented tech-

TOOLS FOR THE NEW CULTURE:

LESSONS FROM THE DESIGN OF THE EIFFEL LIBRARIES

Bertrand Meyer

nology, which provides the best known basis for reusable software construction. (That the founding document of object-oriented methods, the initial description of Simula 67, was roughly contemporary with McIlroy's paper tends to confirm a somewhat pessimistic version of Redwine and Riddle's contention [18] that *"it takes on the order of 15 to 20 years to mature a technology to the point that it can be popularized to the technical community at large."*) Much of the current excitement about object-oriented software construction derives from the growing realization that the shift is now technically possible.

This article presents the concerted efforts which have been made to advance the cause of component-based software development in the Eiffel environment [12, 17] through the construction of the Basic Eiffel Libraries.

After a brief overview of the libraries, this article reviews the major language techniques that have made them possible (with more background about Eiffel being provided by the sidebar entitled "Major Eiffel Techniques"); it then discusses design issues for libraries of reusable components, the use of inheritance hierarchies, the indexing problem, and planned developments.

The Eiffel Libraries

The standard Eiffel delivery currently includes seven libraries:

- The **Kernel Library** includes classes for basic system needs: handling arrays and strings; input and output; exception handling; universal features; access to command-line arguments; arithmetic conversions.
- The **Support Library** provides classes for "browsing" (accessing the properties and structure of Eiffel classes), persistent storage and retrieval of objects, debugging and interactive testing of classes, access to internal object structures, pattern-matching in strings, mathematical operations, memory management.
- The **Data Structure Library** contains implementations of many fundamental data structures and algorithms: lists, trees, stacks, queues, hash tables and numerous others.
- The **Lexical Library** makes it possible to produce lexical analyzers for regular languages.
- The **Parsing Library** offers facilities for developing parsers and compilers.
- The **Winpack Library** supports the development of window-based applications for character-oriented (non-graphical) terminals.
- The **Graphics Library** allows programmers to write applications using windows, menus, mouse input and geometric figures.

Together, these libraries currently include 300 classes totaling about 5000 visible operations (features). They are rapidly growing.

The libraries discussed here only cover the standard Eiffel libraries developed by Interactive Software Engineering. Third parties are developing other libraries, especially in fields such as user interface toolkits and databases.

Techniques

It suffices to consider the amount of time that has elapsed since the publication of McIlroy's article to realize that achieving industrial reuse takes more than wishful thinking. A solid set of methods and techniques is needed.

Providing the basis for a realization of McIlroy's dream was the major design goal for Eiffel. The various components of the language and environment resulted directly from our first attempts at building general-purpose libraries, and an analysis of what was needed beyond the mechanisms of Simula and Ada.

The sidebar "Major Eiffel Techniques" reviews the most important ideas and techniques. This section concentrates on some of the aspects that most directly affect the contents and style of the libraries.

Classes and Clusters

The unit of reuse is the class. This seems to provide the right level of granularity; in particular, an individual routine does not constitute a reusable module independently of the class to which it belongs.

It is useful in practice to gather classes into groups, which may be called **clusters** and play a key role in a suggested life-cycle model for object-oriented design (see [18] and below). For the present discussion, "cluster" is really a synonym for "library": for example we may talk about the Kernel cluster, the Data Structure cluster etc. The notion of cluster also applies, however, in the case of non-library classes.

In practice, the classes of a given cluster are usually kept in the same directory on a hierarchical file system such as Unix or MacOS. A further criterion for clusters is that cycles in the client relation between classes should usually be constrained to occurring within clusters rather than between different clusters.

Assertions

Perhaps the most immediately visible aspect of the library classes is the presence of elements of formal specification, called **assertions**, associated with each routine and class.

Three language constructs using assertions are particularly relevant to the construction and use of libraries: preconditions, postconditions and invariants. As an example of the first two, routine *put_left_sibling*, which replaces the sibling to the left of a tree node, has the following form:

```
put_left_sibling (other: TREE [T])
is
        --Make other the left sibling
        --of current node
    require
        not is_root;
        not left_sibling.Void
    do
        ...Routine implementation
    ensure
        left_sibling = other
    end--put_left_sibling
```

The **require** clause introduces the precondition, which is the condition under which a call is correct (here the node must not be a root, and must have a left sibling). The **ensure** clause introduces the postcondition, which characterizes the situation resulting from a successful execution of the call (here the new left sibling of the tree passed as argument to the routine).

The same class includes an example of the third major construct using assertions, an invariant clause:

```
invariant
        arity > = 0;
        is_leaf = (arity = 0);
        ...Many other invariant
        properties...
```

Here *arity* is the number of children of a node. The second property indicates that query *is_leaf* (an attribute or function) must return true if and only if the node's *arity* is 0. The invariant expresses properties which must be ensured on instance creation and maintained by every exported routine.

A hardware analogy is useful for understanding why assertions are essential to reusable libraries. Few

Major Eiffel Techniques

The Eiffel libraries make full use of the facilities provided by the language and tools. This sidebar reviews some of the basics of Eiffel programming. More details may be found in [2] and [7].

The various components of the method were designed as a whole: assertions condition documentation tools, genericity complements inheritance, dynamic binding is the natural associate of static typing, and so on. It would be difficult to remove any of these elements without impairing the consistency of the overall construction.

Classes

The object-oriented approach to reusability begins with the premise that practical reusable components should be organized around objects (data structures) rather than functions (action structures). This leads to the fundamental modular construct of object-oriented programming, introduced by Simula [2], which is the **class**, a module built around a data abstraction. A class is a model for a set of data structures (objects). Typical examples of classes extracted from the Basic Eiffel Libraries include:

- *TREE*, describing objects which are trees (or, equivalently, tree nodes), from the Data Structure Library.
- *MENU*, describing objects which are pull-down menus, from the Graphics Library.
- *CONSTRUCT*, describing objects which are components of structured texts, from the Parsing Library.

Such a class is a descriptive program text specifying the properties of run-time objects. An object conforming to a class specification is said to be an **instance** of the class. For example, an instance of class *MENU* is an individual pulldown menu, created during the execution of a software system. A system using a certain class may, during its execution, create an arbitrary number of instances of the class.

A class definition introduces a number of **features** representing operations applicable to instances of the class. Examples of features of class *MENU* are the following (whose non-obvious names are explained in the text):

- *count*, an integer, which for any instance of the class indicates the number of its menu entries.
- *item*, which for a suitable integer i indicates the i-th menu entry.
- *put*, which replaces a menu entry by another.

The first two examples are "query" features, which merely return information about an object; the third is a "command," which may change the object. A query with no arguments, such as *count*, may be implemented as either:

- An **attribute** of the class, in which case each class instance includes a field containing the corresponding value.
- A **function**, represented by an algorithm for computing the value for any given instance.

A query with arguments, such as *item*, must be represented by a function.

A command, such as *put*, is represented by a **procedure.** Procedures and functions constitute the **routines** of a class.

This technique of describing a set of run-time objects (such as tree nodes, menus, construct specimens) through the set of operations (the features) follows from the theory of abstract data types, which suggests that objects should be known through applicable operations (and the properties of these operations, described through assertions) rather than implementation aspects.

A class is indeed not just a module but also a type in the traditional programming language sense—obtained here as an implementation of an abstract data type. The identification of modules and types is central to the object-oriented form of software architecture. In Eiffel, this identification is complete: there is no other form of module than the class, and all types (including basic types such as *INTEGER* and the like) are defined by classes.

This identification goes further than the idea of "programming with abstract data types" which may be applied, for example, in Ada or Modula-2. A module in one of these languages may be based on an abstract data type, but the module and the type remain distinct notions. In Eiffel, the module—the class—is also the type. Apart from the conceptual simplicity that this fusion of concepts confers to the method, its main advantage is to open the road to inheritance, as discussed below.

As a result of this summary, it may be noted that "object-oriented" is a misnomer. Not that objects are unimportant; in fact, the entire execution of any system is devoted to creating objects and applying features to them. But the same is true, to a large extent, of a Pascal, Ada or C program, even if the objects in those cases are called records or structures. What is really different with Simula or Eiffel is the notion of class. Much confusion would be avoided if the field were known by the more accurate term *class-oriented* design and programming.

Information Hiding

Various classes may use each other's facilities. A class which relies on another is said to be its **client**; the other class is the **supplier.**

In the implementation of a class, some features will play a purely internal role and should not be visible to clients. The syntax of a class includes an **export** clause which lists those features which are available to the class's clients. Any other feature is secret.

Assertions

As explained in the text, a class and its routines may be characterized by assertions, stating their precise semantic properties.

The use of assertions is rooted in work on formal specification and verification, beginning with the original papers of Floyd and Hoare [3, 5], although the idea of supporting assertions in programming languages dates back to Algol W [6] and may be found in more recent designs such as the Ada-based language Anna [9].

The assertion sublanguage of Eiffel is not a full-fledged formal specification language but is limited to boolean expressions, with a few extensions. Purely applicative expressions are usually sufficient to cover the most important semantic properties of routines and classes; more advanced properties are captured by functions.

Inheritance

Inheritance in Eiffel serves both as a module inclusion facility and subtyping mechanism.

The relation on classes induced by inheritance may be characterized as "is-plus-but-except":

- "Is" since the instances of an heir class may also be used as instances of a parent class. For example, an instance of class *FIXED_TREE* (describing trees where each node has a fixed number of children) may be used wherever an instance of the parent class *TREE* (describing trees in general) is expected.
- "Plus" since an heir may (and usually does) add new features to those of its parent.
- "But" since an heir may change the implementation of any feature or its signature (within the constraints of the type system). This is the **redefinition** mechanism. The specification of a redefined feature must remain compatible with that of the original by including an equal or weaker precondition, and an equal or stronger postcondition. (This means that the redefined version is a subcontractor to the initial version; both client and subcontractor are subject to the terms of the original contract. The precondition rule means that no additional constraint may be imposed on the client; the postcondition rule, that the client is entitled to a result which is as good or better as promised by the original contractor.)
- "Except" since an heir may decide not to export a feature exported by a parent if it does not make sense for its own clients. (The reverse, exporting a previously hidden feature, is also possible.)

This combination yields a highly flexible classification mechanism. (It also makes static type checking less trivial than it might appear at first.)

Inheritance in Eiffel is **multiple**: a class may have any number of parents. This is necessary whenever a simple tree-structured hierarchy would not provide a satisfactory classification. Many library classes use this possibility. For example, class *POPUP_MENU* in the Graphics Library inherits both from *MENU* and from *POPUP* (describing "pop-up" objects).

Multiple inheritance is also commonly used to combine a parent describing an abstract behavior (a deferred class, see below) and one providing the implementation. For example, *LINKED_QUEUE* inherits from *QUEUE* and *LINKED_LIST*.

A corollary of multiple inheritance is **repeated inheritance**, whereby a class inherits from another, directly or indirectly, more than once. A precise policy based on the language's renaming mechanism allows the descendant to select, for each repeatedly inherited feature, whether it should be just one feature (*sharing*) or one per inheritance path (*duplication*). This is particularly useful for obtaining several variants of a common notion, as with an iterator on trees which achieves several traversal policies (see the sidebar entitled "Classifying Data Structures.")

Only two relations may exist between classes: inheritance and the client relationship. In particular, there is no sharing of information through global variables or equivalent mechanisms. This is fundamental to achieve the decentralized nature of component-based software development.

Deferred Classes

A deferred class provides only a partial implementation of an abstract data type, or no implementation at all. (Deferred classes are close to the Simula notion of class with virtual procedures, and to the Smalltalk notion of abstract class.)

A deferred class will have one or more deferred routines, which have no implementation (the keyword **deferred** is used in lieu of a **do** clause). Behavior may still be specified, however, through assertions. Actual implementations in descendants are bound by these assertions (through the rules given above for routine redefinition). An example is class *QUEUE*, from the Data Structure Library, whose "flat" form (with inheritance expanded) is shown in the figure below.

QUEUE is almost entirely deferred (implementation-independent). Classes which are only partially deferred are also extremely useful. They describe a set of components with common properties. This ability to freeze some elements of behavior, while leaving others open, is essential to software reuse where (in contrast, perhaps, with what happens in hardware) reusable components with an entirely fixed behavior are of little practical scope. The ability to reuse must be combined with the ability to extend and adapt.

As an example, the Graphics Library includes a class *FIGURE*, which is fully deferred. Its heir *CLOSED_FIGURE* remains general enough to be also deferred; but it is more specific and includes non-deferred features such as *set_fill_style* (choose a filling pattern for a closed figure).

Polymorphism and Dynamic Binding

Inheritance, serving as the basis for the type system, allows an entity (variable) to become attached at run-time, to objects of more than one type. The basic type rule allows assignments or routine calls of the form

$x := y$

$r(y)$ for a formal argument x

if and only if the type of y is a descendant of the type of x. This means that x can become attached to an object of any descendant type. An entity such as x is said to be **polymorphic.**

If the type used to declare an entity is called its static type, and the type of the object associated with it at some run-time instant is called its dynamic type, the type rule expresses that the dynamic type must be a descendant of the static type.

A Deferred Class.

```
deferred class QUEUE [T] export
    count, empty, full, item, put, remove,
    wipe_out, nb_elements,
    oldest, add
feature

    count: INTEGER is
         --Number of items in queue
         deferred
         end;--count

    empty: BOOLEAN is
         --Is queue empty?
      deferred
      end;--full

    full: BOOLEAN is
         --Is queue full?
      deferred
      end; --full

    item: T is
         --Only accessible item
      require
         not_empty: not empty
      deferred
      end;--item

    put (v: like item) is
         --Put v in queue.
      require
         not_full: not full
      deferred
      ensure
         not empty;
         (old empty) implies (item = v);
         count = old count+1
      end; --put

    remove is
    --Remove the accessible item.
      require
         not_empty: not empty
      deferred
      ensure
         not_full: not full:
         count =old count-1
      end;--remove

    wipe_out is
         --Remove all items.
      deferred
      ensure
         empty
         end;--wipe_out

invariant
    count> =0

    end--class QUEUE
```

A key property associated with polymorphism is dynamic binding, which ensures that if a feature with more than one variant is called on a polymorphic entity (for example in a call of the form *x.f* where *f* is redefined in *D* to have an implementation different from the default *C* implementation), then the appropriate variant is selected on the basis of *x*'s dynamic type (that is to say, the type of the object actually associated with *x*), not its static type.

Static binding (the reverse choice) would be a grave conceptual mistake: it would mean that the wrong version of an operation could be applied to an object, for example the *C* version of *f* to an object of type *D*. This could result in inconsistent objects, since the *C* version of *f* is only constrained to preserve *C*'s invariant, not *D*'s, which may be stronger. Once you start producing inconsistent objects, you cannot guarantee any property about your programs. (The only case in which static binding is justified is when it yields the **same** result as dynamic binding, that is to say, when *f* is not redefined in *D*. But then applying static binding becomes a matter of optimization, which should be applied by the compiler, and indeed is in the current Eiffel implementation.)

Dynamic binding is important from a client's perspective; it means that an application class that manipulates polymorphic entities does not need to test repeatedly for their dynamic types. For example, a class that uses a tree *t* may include a call of the form

t.postorder

to perform a postorder traversal of the tree; even though the details of the algorithm differ for various implementations of trees, the client need not discriminate explicitly between them. Similarly, a call of the form

w.display

will automatically select the appropriate *display* operation for a window *w*, even though various kinds of windows will be displayed in different ways.

Flattened Forms

The use of inheritance raises a potential problem for developers who may want to deliver a class to a user without necessarily delivering the entire inheritance diagram that led to the class. This is also a documentation problem: **short** does not know about inheritance and will not provide useful information about the inherited features of a class.

These issues are solved by the notion of **class flattening.** The flattened form of a class is the text of the class with the same features as the original, but with no inheritance clause. Features inherited directly or indirectly are put in the flattened form at the same level as the features declared locally. Of course, the flattening process takes renaming and redefinition into account.

From a client's perspective, there is no difference between using a class and using its flattened form, with the exception of polymorphism and dynamic binding. (If *C* is a descendant of *B* and *b* is of type *B*, then the assignment *b* := *c* is valid for *c* of type *C*, but not if the type of *c* is a class obtained by flattening *C*.)

Command **flat** may be used in conjunction with **short** to produce a full interface documentation of a class, with local and inherited features treated on an equal footing. The result is called the "flat-short" version of a class. Using Unix-style "piping," this can be obtained through the command

flat *class_name* | **short**

This technique is the one used to produce final library documentation (as discussed in the text).

Typing

Dynamic binding is often confused with dynamic *typing*.

Dynamic typing would mean having to wait until run-time to determine whether operations are applicable to their arguments. In contrast, the Eiffel approach to typing is static: whenever possible, the applicability of *f* to *x* in *x.f* is determined statically, by examination of the class text in which such a call appears. Static typing and dynamic binding are equally important for the reliability of software systems based on reusable components: the former means, for every call of the form *x.f*, a static guarantee that there will be *at least one* version of *f* applicable to *x*; the former a static guarantee that, if more than one version is in fact available, *the right one* (based on the type of the object attached to *x*) will be used.

The presence of a statically-typed language and the use of a type-checking compiler are considerable assets in producing correct systems. (The current Eiffel compiler misses some cases of type mismatches [15]. They are, however, of little practical consequence and work is proceeding to correct them.)

Genericity

Typing would be meaningless without the possibility of defining **generic** classes. A generic class is one which has one or more parameters representing types. This is particularly useful for classes representing **container** data structures, used to gather objects; most of the classes of the Data Structure Library, covering sets, lists, trees and the like, fall into that category.

A class with one generic parameter is declared under the form

class *C* [*T*] ...

and used by clients in declarations of the form

x: *C* [*A*]

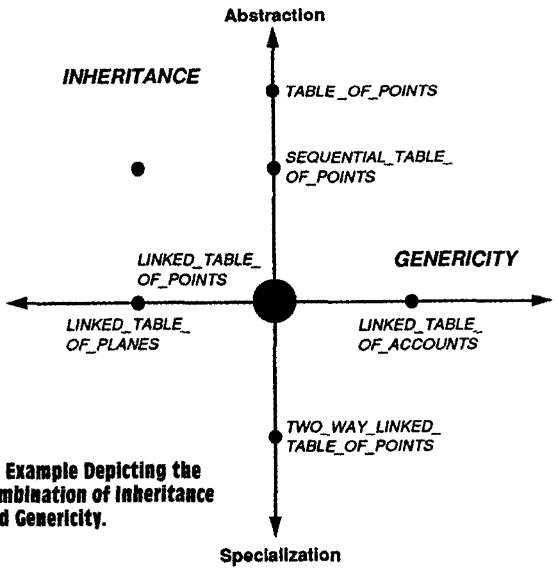
where *A* is some type. For example, with a generic class *LIST* [*T*], you may declare entities of types *LIST* [*SIGNAL*], *LIST* [*POPUP_MENU*], *LIST* [*LIST* [*POPUP_MENU*]] etc.

Combining inheritance and genericity

Of particular interest is the combination of inheritance and genericity. One of its applications is the possibility to define "polymorphic data structures:" with a declaration of the form

a: *LINKED_TABLE* [*C*]

the type rules allow calls of the form *a.put* (*x*) for *x* not just of type *C* but of any descendant type of *C*. This means that the corresponding lists may contain objects of different types, subject to the consistency rules of the inheritance-based typing mechanism. The figure below illustrates this combination of inheritance and genericity by showing how a class describing a specific data structure, say linked tables of points, may be generalized both horizontally (to linked tables of objects or other types) and vertically (to more abstract data structures such as sequential tables, or more specific ones such as two-way linked tables).



An Example Depicting the Combination of Inheritance and Genericity.

Another way to combine genericity and inheritance is offered by the mechanism of constrained genericity. This allows a generic parameter to be restricted to descendants of a given class. For example, a *VECTOR* class needing a generic parameter on which arithmetic operations are available may be declared as

VECTOR [*T* -> *NUMERIC*]

meaning that acceptable actual generic parameters must be descendants of the Kernel Library class *NUMERIC*. This class, used as ancestor by the classes describing standard arithmetic types (*INT, FLOAT* etc.) has a short form beginning with

```
deferred class interface NUMERIC exported
features
    infix "+", infix "–",
    infix "*", infix "/", prefix "+",
    prefix "–"
  feature specification
    infix "+" (other: NUMERIC): NUMERIC
          --Sum of current element and "other"
      deferred
    infix "–" (other: NUMERIC): NUMERIC
          --Difference between current element and
          "other"
      deferred
  ... (etc.) ...
```

Any actual parameter corresponding to *T* must here be a descendant of *NUMERIC*, meaning that it is equipped with the appropriate operations. (Because there is no restriction on multiple inheritance, any class that a programmer desires to use as actual generic parameter may be made a descendant of *NUMERIC* if it was not already one.)

This example also illustrates the use of "infix" and "prefix" features, which will be called by clients, not through the usual dot notation (as in *v.plus* (*w*)), but in operator form (as in *v* + *w*).

Other Aspects

Other important facilities also play a role in building and using quality libraries:

- Persistence: When a session terminates, not all objects should go away. The environment supports the automatic storage of objects with all their dependents (including cyclic structures), and their retrieval in later sessions.
- Garbage collection: Writing serious object-oriented applications, which typically need to manipulate complex dynamic data structures, requires a good garbage collector to reclaim space automatically for unused objects. The language was designed to make efficient garbage collection possible and the current implementation supports an incremental, tunable garbage collector.
- Exception handling: It is essential to offer programmers a way to recover from abnormal cases, or at least to terminate execution gracefully when no other recovery scheme is possible. One of the contributions of Eiffel is a disciplined exception mechanism, built on the contracting theory, which provides for both recovery (resumption) and graceful termination. The exceptions handled may be hardware signals or malfunctions, violated assertions, software bugs, etc.
- Tools: Practical usage of the approach requires a number of tools such as automatic recompilation after a change, source-level debugging etc. The tools of the environment run on top of a modern operating system (such as Unix).
- Simplicity: A programming language should be easy to learn and use, enabling client programmers to concentrate on putting library components to good use. Eiffel's design focuses on a small number of powerful constructs. In particular, it does not try to be "compatible" with older languages which would destroy its conceptual integrity and simplicity.
- Openness: Refusal of compatibility at the language level by no means precludes compatibility with previously written software elements and openness to other tools. Both of the latter goals are essential for reuse. As a consequence, Eiffel supports both call-out of utilities written in other languages, and call-in (of Eiffel routines from those other languages). This makes it possible to use the languages' structuring capabilities (classes, information hiding, multiple inheritance, genericity) as an encapsulating mechanism for software whose actual "meat" is written in other languages. For example, a relational database system can be packaged in one or more classes. (We have found that this approach to interfacing with older languages, which keeps each world separate and forces communication to occur through well-defined bridges, permits more effective reuse and exchange than an approach which would, in a single language, mix the object-oriented paradigm with incompatible ideas.)
- Cross-development: The implementation supports generation of final code in other programming languages (currently C). This complements the openness techniques described above: one can develop library components on a certain platform in Eiffel and use them on a different platform in their C form.

people would buy an amplifier without at least three elements of information:

- What range of input voltage is acceptable—the precondition.
- What the corresponding output voltages will be—the postcondition.
- What general conditions, such as the temperature range, will be both expected and maintained—the invariant.

Without such information, there would be no way to use the amplifier other than by resorting to internal implementation information (such as wiring diagrams).

The same applies in software. Although some reusability may be achieved (as the Smalltalk example shows) in a context where "reusers" must peruse the source code of the modules themselves, large-scale reuse in the industrial sense seems to require mechanisms for understanding the purpose of reusable elements based on their external, abstract properties.

Assertions are not a control structure; in other words, they are not a substitute for conditional instructions (such as "**if** the node is a root **then**..."). In the execution of a correct system, no assertion should ever be violated; a violated assertion is always the manifestation of an error in the software—a bug.

As a consequence, assertions provide an excellent debugging tool; on option, assertion checking may be enabled at runtime, making it possible to catch many defects. It is also possible to check preconditions only (PRECONDITIONS compilation option).

The Contract Theory

The presence of assertions illustrates the underlying theory of software construction, "programming by contract" [13], which plays an important role in the design and use of libraries.

Reusing a software component, rather than writing a new component, is similar to contracting for a job, rather than doing the job yourself. In software contracts, as in human ones, both parties are entitled to some benefits and subject to some obligations. Assertions are the contract document, which expressly specifies each party's obligations and benefits as follows:

- The precondition is an obligation for the client and a benefit for the supplier.
- The postcondition is a benefit for the client and an obligation for the supplier.

This is illustrated for the above example in Table I.

The bottom-right box of Table I is particularly important. It shows the precondition as a protection for the supplier—limiting the set of cases that the supplier must be prepared to handle. Without such limitations, it would be difficult to write effective reusable components; the sheer burden of dealing with all possible cases (those which make sense and those which do not) would make components far too complicated, decreasing the likelihood that they are correct, efficient—or just usable.

This goes against much of the conventional wisdom in software engineering, which favors "defensive programming"—the principle according to which programs elements should be as general as possible. For example, a well-known text about "abstraction and specification in program development" [8] warns sternly against partial routines (that is to say, routines with preconditions other than **true**) by stating (page 53) that:

Partial [*routines*] *are not as safe as total ones, since they leave it to the* [*client*] *to satisfy the constraint in the* [*precondition*].

As the only argument in favor of partial routines, the authors cite efficiency.

Our experience with designing reusable software leads to a different view. Although restricting the scope of routines certainly helps improve their efficiency, the main argument is the very one used *against* them in the above quotation: reliability. By taking a system approach to the construction of reliable software, one realizes that reliability is not obtained by trying to make every software element responsive to every kind of possible input, a futile pursuit which usually results in elements that are too complex—and hence in *less* reliability, since in software complexity breeds bugs. More conducive to the production of reliable systems is an approach which ensures that every element is characterized by a precise indication of its duties as well as its rights.

The programmer of a client module does not expect the supplier to perform in every imaginable case; he knows this is unrealistic. Much more important to him is the precise definition of what constraints must be satisfied by the client, and the knowledge that this performance will be guaranteed if the client abides by those constraints.

This assurance that the precondition is *sufficient* to guarantee correct functioning—in other words, that the contract has no hidden clauses—is what makes it possible to write correct client modules. This appears to be a more fruitful approach to software reliability than an endless race for more general supplier modules.

Software may have bugs, of course, leading to contract violations. This justifies the presence of a general-purpose mechanism to monitor satisfaction of assertions—observance of contracts. This software equivalent of the "Better Business Bureau" is the run-time assertion-checking mechanism. If assertion monitoring is on, the result of an assertion violation is to trigger an exception (see the sidebar entitled "Major Eiffel Techniques").

TABLE I: *A Contract.*

	Obligations	Benefits
Client	Provide node which is not a root and has a left sibling.	Make *other* the new left sibling.
Supplier	Get tree updated so that *other* is the current node's left sibling.	No need to care about roots, or nodes which have no left sibling.

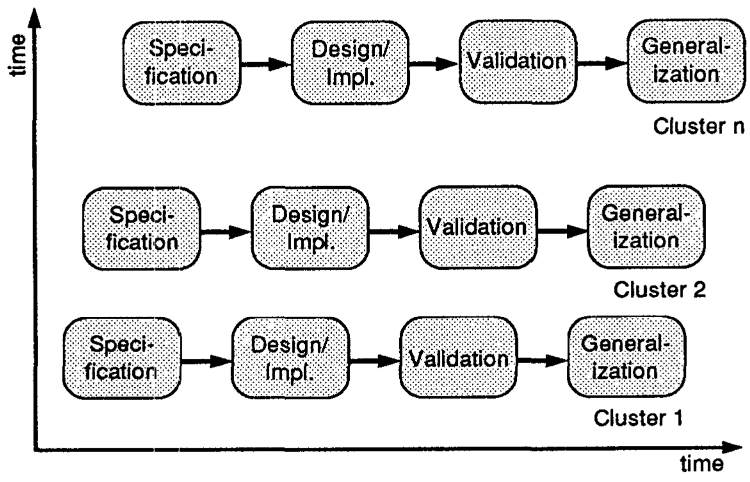


FIGURE 1. The Cluster Model of the Software Life Cycle.

Bottom-up Development

The preceding discussion also illustrates the precise role of assertions in the general bottom-up strategy of software development which library-based object-oriented design naturally implies.

A violated precondition is a bug in the **client** (which has not observed the consistency condition on calling a routine); a violated postcondition is a bug in the **supplier** (which has been unable to produce the expected result).

This has an important consequence on the run-time checking of assertions. Checking all assertions (preconditions, postconditions, invariants) may imply significant overhead. If, however, classes are developed in clusters, as suggested above, and the clusters are built in a bottom-up order. According to the "Cluster Model" of the software life cycle (see [14] and Figure 1), you will release a cluster for general use as part of a library only once it has been thoroughly validated and you have good confidence in its reliability. This means that you may be prepared to switch off the run-time monitoring of its postconditions (and also invariants). But the client modules (the higher-level clusters) may not have reached the same degree of reliability yet and may still contain bugs, which would manifest themselves as violated preconditions. In such a case it is useful to monitor preconditions only, as obtained with the PRECONDITIONS compilation option.

As a trivial but typical example, an incorrect client class could call an array operation with an out-of-bounds index, violating the following routine precondition in class *ARRAY*:

```
    require
        lower <= i; i <= upper
```

Precondition monitoring will catch the error by triggering an exception. Of course, the exception is raised in the client: class *ARRAY* will not even see the call.

Interface Documentation

One of the major problems confronting the designers of reusable components is how to document them. Beyond documentation of a general explanatory nature on clusters and classes, there is a need for very precise documentation which spells out the way each class and feature may be used by clients—the contracts.

A method which would handle this second part of the documentation as a product separate from the classes themselves would face severe obstacles. First, producing such detailed documentation is a tedious process, requiring as much attention as actual programming but intellectually far less rewarding. Then, perhaps even more importantly, it is next to impossible to guarantee that the documentation will be updated when the components evolve. Yet (as will be discussed below) library evolution is an inevitable phenomenon.

The solution used in Eiffel is to extract the documentation, as much as possible, from the class texts. This is made possible by the structure of the language and in particular by the presence of assertions. The **short** form of a class (also called its abstract form) shows the interface properties which are relevant to client programmers—but no implementation details. This excludes any information on non-exported features and, for exported features, includes only the signature declarations (types of arguments and result), the assertions and header comments.

For example, the routine *put_left_sibling* given above appears in the short form of its class as

```
put_left_sibling (other: TREE [T])
        --Make other the left sibling
        --of current node
    require
        not is_root;
        not left_sibling.Void
    ensure
        left_sibling = other
```

A short form of a complete class, extracted from the Library Reference, is given in Figure 2. (This is in fact a "flat-short" form, as explained below.)

In the Eiffel environment, the short form is produced automatically from a class text by a command called **short**. For an Ada or Modula-2 programmer, this would amount to having the "interface" or "definition" part of a module produced automatically on demand by a software tool rather than being written and maintained by programmers. All standard documentation on the Eiffel libraries in the library reference book [16] is produced in this way. Of course, chapters still begin with general explanations, corresponding to the first kind of documentation mentioned at the beginning of this section.

Inheritance

It is not possible to manage a library, with its potentially large number of components, without a classification scheme for these components. The arguments made in the discussion of

The Status of Eiffel

The Eiffel language was designed by the author and his group at Interactive Software Engineering. The basic language specification can be found in "Eiffel: The Language" (reference [17]).

The language specification is in the public domain and anyone is welcome to write Eiffel compilers, interpreters, tools or specialized libraries, which will of course remain the property of their developers.

To ensure wide support and accessibility, the original designers and a number of users have started the International Eiffel Consortium. Operational on August 31, 1990, the consortium will take full control over the evolution of the language, free from proprietary concerns. Fundamental elements of the Basic Libraries' specification, as described in this article, will also be transferred to the consortium. In addition, Interactive Software Engineering will relinquish the Eiffel trademark to the consortium.

documentation apply even more forcefully here: no classification can be successful unless it is built into the components themselves. This is shown *a contrario* by the difficulty of building satisfactory libraries in languages such as Ada, which do not include any classification mechanism.

In Eiffel, inheritance provides the basic classification mechanism, and one of the two reuse mechanisms, the other being the client relation (see the sidebar entitled "Major Eiffel Techniques"). Two main properties distinguish these mechanisms:

- Being a client means reusing the specification. You access the features of a class through its official interface.
- Being an heir (or more generally a descendant) means having access to the implementation. You can access all of the class properties and redefine them as needed to adapt them to a more specific context.

Design Issues

Producing and using the libraries has taught us a number of lessons. This section and the accompanying sidebar discuss some of the principles that seem to have been successful as well as some of what we have learned from our mistakes.

Classification

The organization of the libraries is not arbitrary. In particular, the architecture of the Data Structure Library is the direct result of an ongoing theoretical effort to provide a general taxonomy of the fundamental data structures of computer programming.

The taxonomy uses several orthogonal criteria:

- Access method: Do clients access elements through keys (as with arrays or hash tables), on the basis of the order of insertions (as with stacks or queues), with respect to a client-controlled cursor position (as with lists and other "active data structures", as discussed below), or through some other access method?
- Traversing: Is the data structure traversable? If so, what defines the

```
--One-dimensional arrays

class interface ARRAY [T] exported features
  item, put, lower, upper, count resize, force, clear_all,
  all_cleared, wipe_out, empty

feature specification

  item (i: INTEGER): T
      --Entry of index i Applicable only if i is
      --between currently defined bounds.
    require
        index_large_enough: lower <= i;
        index_small_enough: i <= upper

  put (v. like item, i: INTEGER)
        --Assign item v to i-th entry. Applicable only
        --if i is between currently defined bounds.
    require
        index_large_enough; lower <= i;
        index_small_enough: i <= upper

  Create (minindex, maxindex: INTEGER)
    --If minindex <= maxindex, allocate array with
    --bounds lower and upper; otherwise create empty
    --array.
  ensure
    (not (upper < lower and count = 0)) implies
    (upper > = lower and count = maxindex - minindex + 1)

  lower: INTEGER
   --Minimum current legal index.

  upper: INTEGER
    --Maximum current legal index.

  count: INTEGER
    --Current available entries.

  resize (minindex, maxindex: INTEGER)
    --Rearrange array so that it can accommodate indices
    --down to minindex and up to maxindex. Do not
    --lose any previously entered item.

  force (v: T, I: INTEGER)
    --Assign item v to i-th entry. Always
    --applicable: resize the array if i falls out of
    --currently defined bounds.
  ensure
    inserted: item (i) = v;
    higher_count: count > =old count

  clear_all
    --Reset all items to default values.

  all_cleared: BOOLEAN
    --Are all items set to default values?

  wipe_out
    --Empty the array: discard all items.
      ensure
          wiped_out: empty

  empty: BOOLEAN
    --Is array empty?

invariant
  consistent_size: count = upper--lower + 1;
  non_negative_size: count > = 0;

end interface--class ARRAY
```

FIGURE 2. A Class Interface.

traversal order or orders?
- Storage: Does the representation use a fixed storage structure, one that is initially fixed but resizable, an unbounded one?

Each of these criteria gives rise to an inheritance hierarchy. Classes describing specific structures—for example, "linked tables," with a key-based access method, a linearly traversable structure, and an unbounded representation—are obtained by combining classes from all three hierarchies, using multiple inheritance.

The three inheritance hierarchies corresponding to the above criteria are shown in the sidebar entitled "Classifying Data Structure." (They are fully used in the version of the Data Structure library to be released at the end of 1990.)

Other criteria could also have been used: for example, some structures are read-only, others are read-write; some are persistent (files), others only exist for the duration of a session; and so on. But this would have pushed the granularity of the classification too far. Any classification results from a set of choices: deciding which criteria are essential, and which are secondary. The decisions we have made are not the only possible ones; the guiding principle has been to try to get the simplest and most convincing structures.

The taxonomy process has been one of trial and error, and more work remains to be done. This effort at a multi-threaded classification of data structures and the associated algorithms—in other words, of some of the fundamental tools of our discipline—has been one of the most challenging and exciting aspects of the library design.

Naming

An interesting issue is the choice of names for class features.

Name choices illuminate the problems that library designers and users face when libraries reach industrial size. At that stage, the concern for consistency and regularity takes over the concern over the individual properties of each class and feature.

When you only have a few dozen classes, you tend to select names based on the precise functionality of each feature. Standard names for operations are also an immediate influence.

The "container" classes of the Data Structure Library (in other words, classes describing various ways of storing and accessing objects such as arrays, trees, stacks and the like) provide good examples. Some of the classes in the original version of the library had among their features those shown in Table II.

A typical call, *a.enter (i, x),* would enter value *x* at index *i* in array *a*; the call *h.insert (x, k)* would enter value *x* associated with key *k* in the hash-table *h*; and so on.

These name choices reflected the traditional terminology employed for the corresponding data structures in computer science textbooks. In other words, the naming criterion was an internal one, adapted to each structure.

When the usage of the libraries and its size started to grow, however, we realized that it was preferable to use more external criteria. All the above container data structures have a basic mechanism for inserting an element, another for accessing an element, yet another for removing an element and so on. Because the overall goal of each of these operations is the same regardless of the variant chosen, it is preferable in the long term to forget about traditional, specific terminology and to use consistent names. As a result, a small set of standard names was chosen; for example, the names of the above example are replaced by those of Table III.

Table IV shows some of the standardized names used throughout the libraries.

After the initial shock of seeing a *STACK* module without a *push* operation, the change appeared to be welcomed by users for the consistency and regularity it brought.

At first sight the use of a single name, such as *item,* for operations which have a quite different practical behavior might seem confusing for client programmers. What is important is the difference of signature and specification. For example

- *item* for stacks takes no argument and returns an element (the stack top) chosen by the supplier.
- *item* for hash tables takes a string argument chosen by the client and returns the element associated with the corresponding key in a hash table.

These differences are expressed clearly by the signature and specification (assertions) as they appear in the flat-short form of the class. A client programmer will have to understand them to use the classes effectively. Having to learn different names would bring no benefit to the client programmer, but would only add to the effort of understanding and remembering the interface. With the new conventions, a client programmer can approach a new class and recognize the feature names; this helps him grasp quickly what each feature is about and removes the need to learn unfamiliar terminology. This is confirmed by the experience of Smalltalk libraries, where the recommended style also favors consistency over specificity.

One exception to the generally favorable response to the above name changes was the use of *put* for operations which appear to add an element in some cases and merely replace an element in others. This caused some confusion. Further analysis has led to the following finer-grain characterization:

- A routine which replaces an existing element, associated with a certain key *k,* with a new value *v,* will be called *replace.* This applies for example to arrays and hash tables. The rough postcondition in this case is

  $$item\ (k) = v$$
- A routine which adds a new value *v* will be called *add.* The rough precondition in this case is that the structure now includes one more occurrence of *v* than before. Examples are lists or "dispenser" structures (stacks, queues).
- There is still a need for a basic *put* operation which simply ensures

Classifying Data Structures

Data Structures are classified according to three criteria: access method, traversal and storage.

Access Method

An essential property of "container" data structures is the way elements are accessed. The following classification captures some of the most important variants. (The figure below illustrates the top of the hierarchy.)

The only two operations on "containers" are *has,* the membership test; and *fill,* which fills a certain structure with the elements of another. Procedure *fill* is variusly redefined at different levels of the hierarchy and provides a universal conversion mechanism

Containers may be "collections" or "tables."

In a table, every element is stored with and retrievable through a certain key. This covers hash tables and "indexable" structures such as arrays. The declaration of HASH_TABLE begins with

deferred class *HASH_TABLE[T, KEY -> HASHABLE]*

using constrained genericity (see the sidebar entitled "Major Eiffel Techniques") to express that the type of the key must be a descendant of class *HASHABLE,* which has a function *hash* delivering a hash value. An example of such a descendant is class *STRING.*

In a collection, elements are retrieved through some criterion other than a key. In a "set," the only significant property regarding an element is whether or not it appears in the collection. For a "bag," in contrast, the number of times an element appears is significant. Most of our bags are **active data structures** which have a notion of current position or cursor; most operations are then relative to the cursor position.

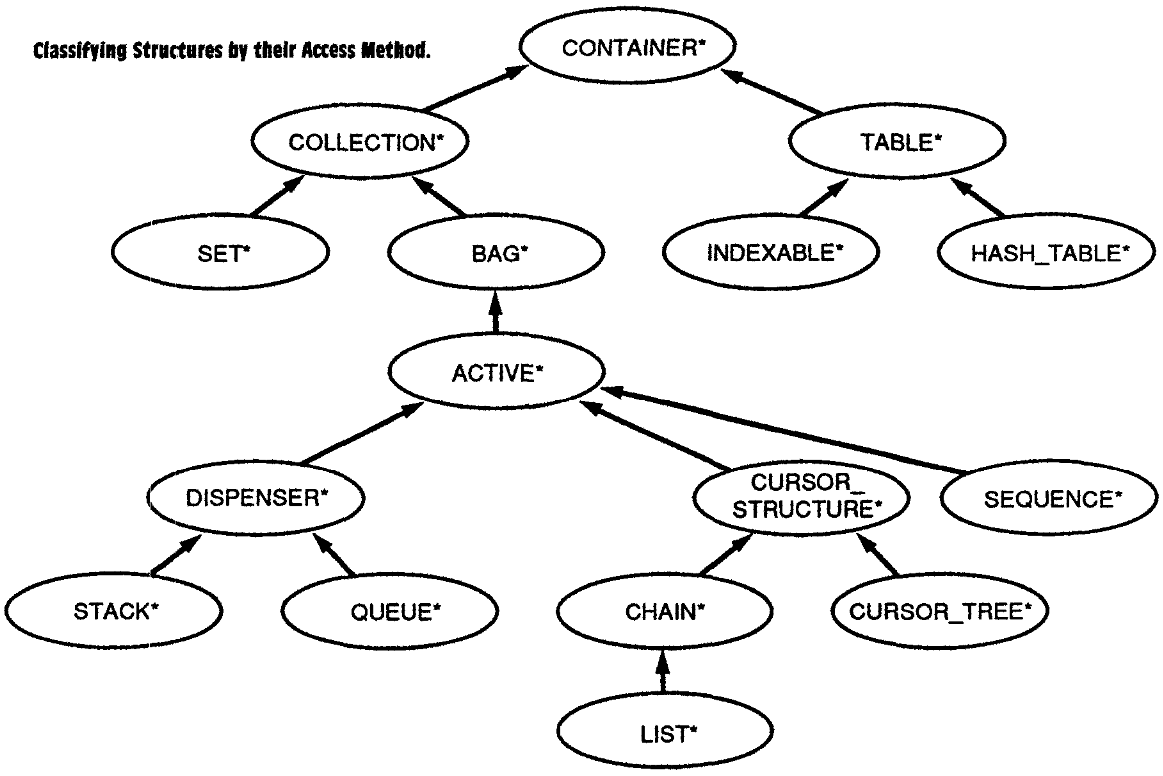
Examples of active data structures include "dispensers," where the client has no control over the cursor: insertions and retrievals occur at positions determined by the structure's properties. Typical examples are stacks (last-in, first-out) and queues (first-in, first-out).

Other data structures such as lists are "cursor data structures," where the client has explicit control over the cursor (see the figure entitled "Active Data Structure with Cursor."). For example, operations on a chain (a general notion including non-circular lists as a special case) include

position	Current cursor position (integer)
forth	Move cursor ahead one position
item	Element at cursor position
before	Is cursor at the left of the first element, if any?
after	Is cursor at the right of the first element, if any?
count	Number of elements in list

As the specification for *before* and *after* indicates, the cursor is allowed to go one position off the right or

Classifying Structures by their Access Method.



left edge. Such properties are captured as invariant clauses such as

0 <= *position* <= *count* +1;
not (*before* **and** *after*);

More generally, invariants and other assertions are the principal guide for making sure that the conventions (regarding default initial states, borderline cases, compatibility between the various features) are sound, consistent and easy to teach.

The notion of cursor as it exists for chains is generalized in *CURSOR_TREE* to two-dimensional cursors. Here the features also include the Boolean queries *above* and *below* (to test whether the cursor has been taken higher than the root or below a leaf), the procedures *up* (to parent) and *down* (i) (to *i*-th child), etc. Again, assertions play a key role in getting the conventions right.

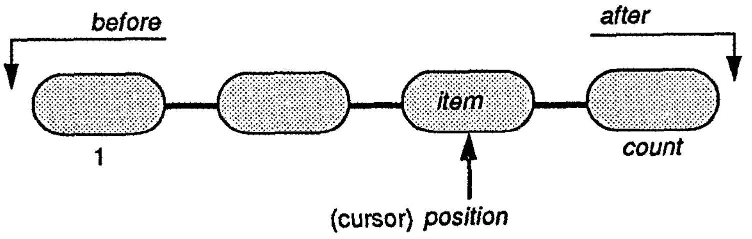
Traversal

Many data structures are traversable. The Data Structure Library includes a set of "iterator" classes which define traversal mechanisms, allowing programmers to avoid writing loops; instead, they define the actions to be applied to every element. The top of the "traversable" hierarchy is shown in the figure below.

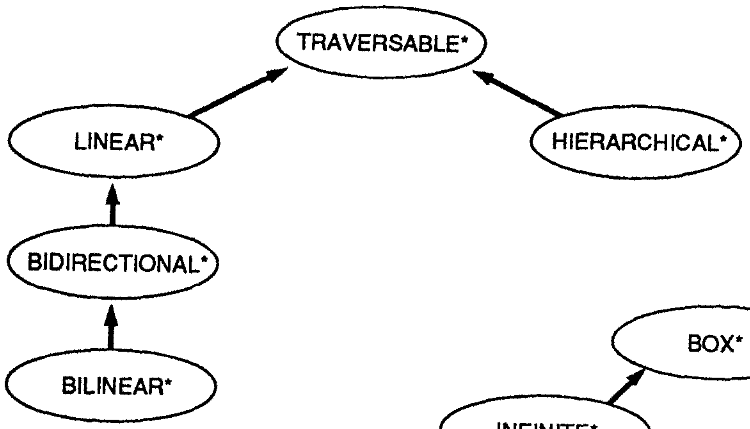
As an interesting use of "repeated inheritance" (inheriting twice or more from the same parent), tree iterators (inorder, preorder, postorder) are obtained by inheriting repeatedly from the same basic iterator, with a different redefinition of the basic stepping procedure in each case.

Storage

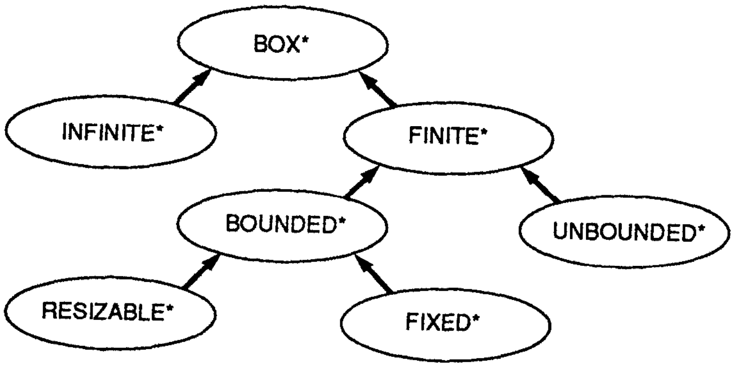
The top of the "storage" hierarchy is shown in the figure below. A "box" is finite or infinite (infinite structures cannot be fully constructed, of course, but may be approximated using "lazy" techniques, and are also useful to describe predefined concepts such as the set of integers); a finite box may be "bounded" or "unbounded." A bounded structure may be fixed or resizable. The tendency in the library is to avoid fixed structures as much as possible; built-in size limits are a plague of traditional programming methods. In the libraries, even arrays are resizable. (More precisely, the *put* operation requires an index within the current bounds, as specified by a precondition; but the *force* operation will accept any index, and will resize the array if needed.)



Active Data Structure with Cursor.



Classifying Structures by their Traversal Policy.



The Top of the "Storage" Hierarchy.

that its argument, *v*, is present in the structure. In other words, the visible postcondition in this case is *has* (*v*). This specification is less strong than for the other two operations; indeed, *put* will normally be a synonym for either *replace* (for arrays, etc.) or *add* (for stacks, etc.). Having such a generally available feature, with a well-understood semantics, is essential to enable client programmers to grasp the essentials of a new class quickly and feel immediately at ease with it.

TABLE II: *Some Original Feature Names.*

Class	Features		
STACK	*push*	*pop*	*top*
ARRAY	*enter*		*entry*
QUEUE	*add*	*remove_oldest*	*oldest*
H_TABLE	*insert*	*delete*	*value*

TABLE III: *A More Uniform Terminology.*

Class	Features		
STACK	*put*	*remove*	*item*
ARRAY	*put*		*item*
QUEUE	*put*	*remove*	*item*
H_TABLE	*put*	*remove*	*item*

TABLE IV: *Some Standard Feature Names.*

item	Basic access operation.
count	Number of significant items in the structure.
has	Basic membership test: does a given item appear in the structure?
put	Basic operation to insert or replace an item.
force	Like *put*, but will always succeed when it can. For example, if may resize the structure if full.
remove	Basic operation for removing an item.
wipe_out	Basic operation for removing all items.
empty	Test for absence of any significant items. Should return the same value as *count* = *0*.
full	Test for lack of space for more items.

Note how the reasoning which led to this solution required (as almost always in such cases) a precise analysis based on assertions, here postconditions.

Other Naming Issues

Further criteria must be applied to the choice of names in a successful library.

Names should be both simple (which usually implies that they should be short) and chosen according to consistent conventions.

One consequence is that library authors should resist the temptation to over-qualify names (a typical beginner's mistake). For example a procedure for handling an event in class *EVENT* in a graphics system should not be called *handle_event* or *event_handle*, but just *handle*.[1]

This would not necessarily be true in a less-typed language because of ambiguities that might result if many classes use the same simple names such as *handle*, *put*, *item*, etc. Typing averts these problems. When you see

e.handle (...)

the declared type of *e* immediately tells you which version of *handle* is meant (while leaving the desirable ambiguity provided by dynamic binding).

To facilitate quick recognition and understanding of the role of each feature, the Eiffel libraries usually follow uniform rules as to the syntactic category of feature names:

- Names for procedures are verbs in the imperative, as in *put*.
- Names for attributes or functions of type other than Boolean are nouns, as in *item*.
- Names of Boolean queries are adjectives, as with *full*, or verbs suggesting a question, as with *is_leaf*.

Because English is the default lan-

[1] The fashion of using in-word capitalization, as in EventHandle, does not conform to normal English usage and is frowned upon in the recommended Eiffel style.

guage for the libraries, these rules cannot be absolute; we need additional conventions regarding the use of words such as *empty,* which may be used both as an adjective and as a verb.

Feature Obsolescence

Name changes such as the ones experienced when the library moved to the new naming system, as described above, are only a special case of changes to the interface to a class. Internal implementation changes do not affect clients (the Eiffel automatic recompilation mechanism in fact guarantees that clients will not be recompiled in such a case); but of course interface changes will affect them.

This raises a key question, which surprisingly does not seem to have been addressed in the reuse literature: feature obsolescence.

Perfect reusable components are not obtained at the first shot. Yet if one is aiming at a full-fledged industry of reusable software components, perfection is what we should eventually strive for.

This raises the question of what you do when you have produced a first version of a reusable class, or even a second and a third, and you realize that you could have done better. Two spirits are at odds:

- The Great Tempter of Perfectionism exhorts: "Correct it here and now before it is too late!"
- The Guardian Angel of the Installed Base warns: "Think of the current users!"

To try to placate both, the library designer or maintainer needs a mechanism to phase out obsolete features progressively without impairing the correct functioning of existing client classes whose programmers may not wish to "migrate" immediately.

Eiffel includes a language mechanism devised to support this process. A routine may be declared as "obsolete." For example, the new *ARRAY* class still has a feature *enter* of the form

```
enter (i: V; v: T)
obsolete " Use 'put (value, index)' "
is
    do
        put (v, i)
    end--enter
```

Such a feature is normal in every respect but two. It can be used by clients (if it is exported) and by descendants, but such uses will trigger compile-time warnings, listing the message given after the **obsolete** keywords. Furthermore, an obsolete feature does not appear in the short form of the class.

Because they cease to be documented in the official reference, obsolete routines pose no immediate threat to the simplicity of the class as perceived by new users. This is different from what would occur if both old and new features were merely kept as synonyms.

More sophisticated effects could have been devised for obsolete features; for example, one may imagine a mechanism which would on option take care of updating client calls (although this is not so trivial when the new routine has different arguments or, as in the above example, changes the order of arguments). As it is, however, the mechanism has played a key role in allowing Eiffel library developers to take advantage of bouts of *esprit de l'escalier* without disturbing existing clients too much. (Esprit de l'escalier, or "wit of the staircase," is a great thought which unfortunately is an afterthought, like a clever reply that would have stunned all the other dinner guests—if only it had occurred to you before you started walking down the stairs after the party was over.)

Of course, if you uncover a serious design mistake in the original version of the class you should not leave it around but just rewrite the class. In this class, the first spirit (the Tempter) wins handily. Feature obsolescence is useful in the following cases:

1) You can think of a better name for a routine.
2) You want to advise programmers not to use the routine any more.
3) You can think of a better signature or specification (assertions).

Situation 1 may occur as you are doing an after-the-fact cleanup of your library and realize that naming conventions could be made more consistent (as discussed above).

Situation 2 may arise when the routine's action is not needed any more (as with a routine which performed some initialization which you later realize can be carried out automatically on object creation).

Situation 3 may occur (among other cases) when you realize that a routine has too many arguments and should be split into two or more routines. For example, you may have a procedure adding a subwindow to a window, under the form

w.add_subwindow (other_window, horizontal_position, vertical_position)

but then you realize that it would have been better to omit the last two arguments and have the subwindow be initially positioned at the top left corner of the parent window, and let clients move it if necessary by using a specific *move* procedure, which is needed anyway.

Cases 1 and 3 often involve changes small enough that it is tempting to heed the Angel's advice and resist any change at all. But in the long term this is dangerous. Here the Angel is really a front man for the hideous Devil of Eternal Compatibility with the Horrors of the Past, whose nefarious influence is all too visible in the computer industry.

Arguments and Options

The last example illustrates a general guideline about choosing the proper arguments for library routines.

In general, a routine should only include among its arguments what may be called "indispensable arguments," as opposed to "options." An option is recognized by the class's ability to set reasonable default values, as for *horizontal_position* and *vertical_position* above. In contrast, there is no reasonable default for *other_window,* which should thus be an indispensable argument.

A widely applicable guideline is to

avoid including options among the arguments to a routine. Rather, the creation procedures of the class should set defaults for each option, and there should be separate routines to change the option's values. (Some of these issues were discussed, in the context of much more primitive technology, in [11].)

Obtaining the Proper Inheritance Structures

By reading theoretical discussions of object-oriented techniques, it would seem that one always gets the inheritance structure—the classification—right from the start. The reality, however, is usually more painful.

Classification tends to be the result of hard work as much as of immediate insight. This work may be called **generalization** and is worth more attention.[2]

Class Abstraction

Object-oriented library design is a quest for abstraction. Using inheritance means that one writes classes that are more general than what is immediately needed for the problem at hand. *Deferred* classes are particularly useful here. Once you have captured a general pattern through a deferred class, you or others may produce specific variants by writing 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.[3]

In practice, however, the scheme is not always as smooth and intellectually satisfying as the theory would have it. Even library developers 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 inherently problem solvers. Few will complain if they get the job done first.

If reusable products 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. There have been numerous examples of this type of reorganization in the evolution of the Eiffel Libraries:

- The Data Structure Library originally contained a *TREE* class, which has proved powerful and useful, serving as a basis for the hierarchical windowing system of the Graphics and Winpack libraries, for the data structures of the Parsing library, and for the abstract syntax tree of our Cépage structural editor. But it was too specific, describing just one implementation of trees rather than the general concept. Recognition of this situation led to a deferred class, of which the original became an heir.
- In the version of the library available at the time of this writing, files and strings are still treated as special classes, instead of inheriting from more general "chain" or "stream" classes (used for example as ancestors to classes describing lists). Here the taxonomy effort mentioned above obviously did not go far enough. After taking a closer look, we came to the realization that strings should be treated just as sequences of characters, based on a *SEQUENCE* Data Structure Library class. As for text files, they came out just as a specific variant of strings, with only one clearly distinctive property: their persistence. The class hierarchy has been both enriched and simplified as a result.

The need for an a posteriori abstraction process was discussed in the Smalltalk context by Johnson and Foote[4] [7].

The process 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 *class_name,* with all implementation details removed. This is usually a good basis for obtaining a more abstract class while keeping the interface for clients.

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.

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, both the Winpack non-graphical windowing library and the Graphics library use hierarchically structured windows, with many concepts in common. The two *WINDOW* classes are not, however, part of the same inheritance hierarchy. This is clearly a mistake, which is in the process of being corrected. The result should yield a library which supports the execution of the same applications both on a graphical terminal and, in somewhat degraded mode, on a character-oriented terminal, at the cost of a minimal change to the client software (such as a different call at initialization time).

[2] This section draws heavily on an earlier publication [12]

[3] Because of the common graphical representations 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.

[4] Many of the design rules of that article are confirmed by our experience. One point of divergence is its recommendation that inheritance hierarchies be narrow and deep. Although it is always rewarding to obtain deep classifications, in some cases inheritance just serves to classify a potentially large set of alternate cases, all at the same level. For example, class *EVENT* in the graphical library has many heirs describing various event types. The same situation occurs in classifications of natural objects such as plants or animals: sometimes the categories are complex; at other times they are just numerous.

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. It is best to 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 effective object-oriented reuse of components.

An important aspect of both abstraction and extraction is that they normally do not affect the clients of

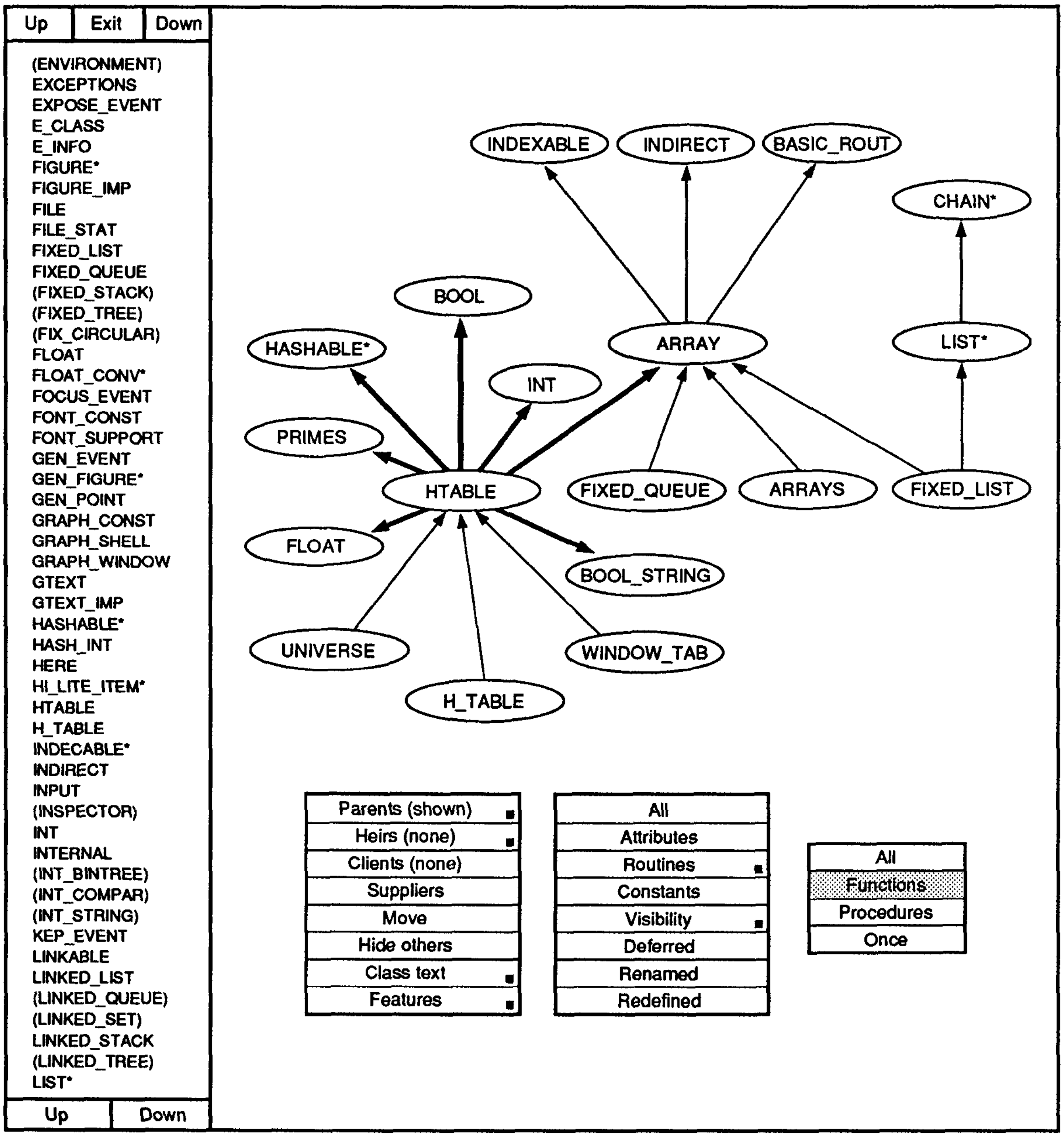


FIGURE 3. A Browser Screen.

the classes being restructured, since the interface of a class will not change if it is rewritten 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 a supplier's mechanism, not a client's mechanism; it does not affect the interface. For the clients of a class, what the class inherits from is irrelevant. Such tools as the flattener 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 (such as our own) that have made a serious effort at producing, using and maintaining libraries. This phenomenon, also noted in [7], is a progressive elevation of the level of abstraction of the classes produced by a group or organization committed to object-oriented programming. As one starts reusing previous classes, cataloging them, archiving them into libraries, the need for more general versions becomes apparent. 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.

Storing and Accessing Components

A common problem in the component-based approach to software development is determining how to enable client programmers to find out about available components and retrieve them easily. Obviously, the seriousness of this problem grows with the number of components.

The concern over this issue, especially among managers, is exaggerated. Compared to the need for reusable components, the libraries that now exist are only a small beginning. A manager or programmer who hesitates about reuse for fear of being overwhelmed by the potential number of resulting components is similar to someone who refuses a pay raise for fear of not knowing what to do with the money. The natural reaction of a coworker (in such an unlikely situation) is: "why don't you give *me* the money as well as the troubles—I'll handle both." Similarly, the first problem in introducing the new culture is not to keep on top of the components, but to build enough high-quality components initially.

Browsing

This being understood, the retrieval problem must of course be addressed. The first step was provided by "browsers," a concept introduced by the Smalltalk environment [4]. The graphical Eiffel browser (**GOOD**) makes it possible to obtain information about all the classes in a "universe" (set of directories). The information is displayed in graphical form (see Figure 3). By clicking on a class bubble, one can request the display of other bubbles and their relation to the original class: parents, ancestors, clients, suppliers etc. One can also obtain information about the class, for example the list and signatures of its features or the class text in its various forms (full, short, flat-short).

Beyond the browser stage, what is required are veritable databases of software components. Standard database technology seems directly applicable here to support archival operations and queries.

The use of database tools is consistent with a principle stated in the above discussion of documentation: all information about a class should be deducible from the class text—as opposed to information kept separately, for which it would be difficult to guarantee that consistency is maintained as the class evolves.

Indexing

This suggests a need for including in class texts higher-level information than is given just by the executable class text. Examples of such information include keywords, hardware requirements, and more generally elements of "domain analysis" [1]. To cater to this need, Eiffel classes may include an initial **indexing** clause, which is part of the language. This is a clause of the form

indexing
    index: value, value, value, ...
    ...

where each subclause lists the values associated with a given index. For example, the *ARRAY_LIST* class from the Data Structure Library has the following clause:

indexing
    names: list, sequence;
    representation: array, linked;
    access: fixed, cursor;
    size: resizable;
    contents: generic

Once included in a class, such information may be used by various query tools. Such tools are currently being written for the Eiffel environment.

Indexing Guidelines

The choice of indices and values is free (values may be identifiers, integers etc.). This makes it possible to define a precise style for a given library or installation. Such a standardized style has been defined for and applied to the current libraries [17]. It includes the following guidelines:

- Keep the Indexing clauses short (2 to 5 entries is typical).
- Avoid repeating information which is in the rest of the class text.
- Use a set of standardized indices for properties that apply to many structures, such as choice of representation. (Examples of such indices are given below.)
- For values, define a set of standardized possibilities for the common cases.
- Include positive information only. For example, a *representation* index is used to describe the choice of representation (linked, array, ...). A

deferred class does not have a representation. For such a class the clause should not contain the entry *representation: none* but simply no entry with the index *representation*. A reasonable query language will make it possible to use a query pair of the form <*representation, NONE*>.

The indices chosen for the library, along with typical values, are the following.

An entry of index *names* is used to record the names under which the corresponding data structures are known. Although a class has only one official name, the abstraction it implements may be known under other names. For example, a "list" is also known as a "sequence." Also, the official name may need to be of an abbreviated form; in such a case, the *names* entry may give the expanded form of the abbreviation.

An entry of index *access* records the mode of access of the data structures. Standard values include:

- *fixed* (only one element is accessible at any given time, as in a stack or queue).
- *fifo* (first-in-first-out policy).
- *lifo* (last-in-first-out).
- *index* (access by an integer index).
- *key* (access by a non-integer key).
- *cursor* (access through a client-controlled cursor, as with the list classes).
- *membership* (availability of a membership test).
- *min, max* (availability of operations to access the minimum or the maximum).

Obviously, more than one of these values may be used.

An entry of index *size* indicates a size limitation. Among common values:

- *fixed* means the size of the structure is fixed at *Create* time and cannot be changed later (there are few such cases in the library).
- *resizable* means that an initial size is chosen but the structure may be resized (possibly at some cost) if it outgrows that size; for extendible structures without size restrictions this entry should not be present.

Graphical Conventions

The class diagrams in this article use some simple elements of a formalism under development by Jean-Marc Nerson and the author for graphical representations of object-oriented system analysis and design. Classes are represented by elliptic bubbles; an asterisk indicates a deferred class; single arrows indicate inheritance; double arrows indicate the client relation. The full formalism also includes conventions (not used here) for representing the class's features, its preconditions, postconditions and invariant.

An entry of index *representation* indicates a choice of representation. Value *array* indicates representation by contiguous, direct-access memory areas. Value *linked* indicates a linked structure.

An entry of index *contents* is appropriate for "container" data structures. It indicates the nature of the contents. Possible values include *generic* (for generic classes), *int, real, bool, char* (for classes representing containers of objects of basic types).

For example, the *ARRAY_LIST* class describes lists implemented by one or more arrays, chained to each other. Its indexing clause, as given above, reflects the preceding guidelines.

Evolution of the Libraries

Much work remains to be done, of course, on the Eiffel libraries. The main areas of improvement are the following:

- Improve the regularity and consistency of the existing classes, applying methods of "abstraction" and "extraction of commonality" as described above.
- Build more archival and query tools, based on indexing clauses and related concepts.
- Add uniform mechanisms to the classes, for example iterators on every data structure.
- Add many classes used as examples in the object-oriented literature (in particular [12]) in the form of full-fledged reusable components.
- Extend the Data Structure Library to cover the essential part of the classical textbooks on this subject.
- Extend the graphical and user-interface classes.
- Develop more specialized libraries: database access, numerical software, etc.

The list of attractive new areas to cover is considerable and beyond the reach of any single group. We do hope that a real "software components subindustry" (using McIlroy's terms) will join us in producing the high-quality components which are needed to make the new culture a reality.

Acknowledgments.

My greatest debt is to the people who contributed classes or just ideas to the libraries. They are too numerous to mention here but I am particularly grateful to Jean-Marc Nerson, Reynald Bouy, Frédéric Deramat (who originated some of the above material on classification in the Data Structure library). Jean-Daniel Dumas, Vincent Kraemer, Philippe Lahire, Jean-François Macary, John Potter, Kim Rochat, Roxanne Rochat and Philippe Stephan.

I also thank the other contributors to this special issue, especially Ralph Johnson, Rebecca Wirfs-Brock, Brian Henderson-Sellers, Gul Agha and the editors (John D. McGregor and Tim Korson) for their many constructive comments. ■

References

1. Biggerstaff, T.J. and Perlis, A.J. Eds. *Software Reusability.* ACM Press, Addison-Wesley, Reading, Mass., 1989.
2. Dahl, O.-J., Myrhaug, B. and Nygaard, K. (Simula) *Common Base Language.* Norsk Regnesentral Norwegian Computing Center, Oslo, February 1984.
3. Floyd, R.W. Assigning meanings to programs. In *Proceedings of the American Mathematical Society Symposium in Applied*

Mathematics, vol. 19 (1967), pp. 19-31.
4. Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison Wesley, Reading Mass., 1983.
5. Hoare, C.A.R. An axiomatic basis for computer programming. *Commun. ACM, 12,* 10 (October 1969), 576-580, 583.
6. Hoare, C.A.R. and Wirth, N. A Contribution to the development of ALGOL. *Commun. ACM, 9,* 6, (June 1966), 413-431.
7. Johnson, E. and Foote, B. Designing reusable classes. *J.Object-Oriented Prog. 1,* 2 (June-July 1988), 22-35.
8. Liskov, B. and Guttag, J. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, Mass., 1986.
9. Luckham, D. and von Henke, F.W. An overview of Anna, a specification language for Ada. *IEEE Softw. 2,* 2 (March 1985), 9-22.
10. McIlroy, M.D. Mass-produced Software Components. In *Software Engineering Concepts and Techniques (1968 NATO Conference on Software Engineering),* J.M. Buxton, P. Naur and B. Randell, Eds., Van Nostrand Reinhold, 1976, pp. 88-98.
11. Meyer, B. Principles of package design. *Commun. ACM, 25,* 7 (July 1982), 419-428.
12. Meyer, B. *Object-Oriented Software Construction.* Prentice-Hall 1988.
13. Meyer, B. Programming as contracting. Tech. Rep. TR-EI-12/CO, Interactive Software Engineering, Santa Barbara, Calif, 1988.
14. Meyer, B. The new culture of software development: Reflections on the practice of object-oriented design. In *TOOLS '89 Technology of Object-Oriented Languages and Systems,* (Paris, November 1989), pp. 13-23.
15. Meyer, B. Static typing for Eiffel. Tech. Rep. TR-EI-18/ST, Interactive Software Engineering Inc., 1989.
16. Meyer, B. Eiffel: The libraries. Tech. Rep. TR-EI-7/LI, Interactive Software Engineering Inc., Santa Barbara, Calif., October 1986 (version 2.2, August 1989). To be published by Prentice-Hall in 1990.
17. Meyer, B. Eiffel: The language. Tech. Rep. TR-EI-17/RM, Interactive Software Engineering Inc., Santa Barbara, Calif, 1989. To be published by Prentice-Hall in 1990.
18. Redwine, S.T. and Riddle, W.E. Software technology maturation. In *Proceedings of the Eighth International Conference on Software Engineering* (London, August 1985), pp. 189-200.
19. Sada, F. and Gindre, C. A development in Eiffel: Design and implementation of a network simulator. *J. Object-Oriented Prog, 2,* 2 (May 1989).
20. Tracz, W. Software Reuse: Emerging Technology (Tutorial). Catalog number EH0278-2, IEEE, 1988.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*Software Libraries;* D.2.4 [**Software Engineering**]: Program Verification—*Reliability;* D.2.7 [**Software Engineering**]: Distribution and Maintenance—*Corrections, Documentation, Enhancement, Extendibility;* D.2.9 [**Software Engineering**]: Management—*Life Cycle;* D.2.10 [**Software Engineering**]: Design—Methodologies; D.2.m [**Software Engineering**]: Miscellaneous—Reusability; D.3.3 [**Programming Languages**]: Language Constructs—Abstract Data Types; I.3.4 [**Computer Graphics**]: Graphics Utilities—*Graphics Packages, Software Support;* K.6.3 [**Management**]: Software Management—Software Maintenance

General Terms: Design, Documentation, Languages, Management, Reliability, Verification

Additional Key Words and Phrases: Assertion, class, cluster, cluster life-cycle model, component, defensive programming, invariant, library design, object-oriented design, object-oriented programming, postcondition, precondition, software obsolescence.

About the Author

BERTRAND MEYER is president of Interactive Software engineering (Santa Barbara) and Société des Outils du Logiciel (Paris). His research has covered several aspects of software engineering, particularly design methods, reusability, programming langauges, formal specification, interactive systems and object-oriented techniques. He is chairman of the TOOLS conference (Technology of Object-Oriented Languages and Systems). His two latest books are *Object-Oriented Software Construction* and *Introduction to the Theory of Programming Languages,* both published by Prentice-Hall. The next two, *Eiffel: The Language* and *Eiffel: The Libraries* are scheduled to be published at the end of 1990. **Author's Present Address:** Interactive Software Engineering Inc., 270 Storke Road Suite 7, Goleta, CA 93117; email: bertrand@eiffel.com.