Eiffel*: A Language and Environment for Software Engineering

Bertrand Meyer
Interactive Software Engineering Inc., Goleta, California

The Eiffel language and environment address the problem of building quality software in practical development environments.

Two software quality factors were deemed essential in the design of the language: reusability and reliability. They led to the following choices: language features that support the underlying bottom-up software design methodology; modular structures based on the object-oriented approach, with support for both generic parameters and multiple inheritance (including a new extension, repeated inheritance); automatic storage management; highly dynamic execution model; support for polymorphism and dynamic binding; fully static typing; information hiding facilities; assertions and invariants that may be monitored at run-time.

The Eiffel programming environment, using C as an intermediate language, supports separate compilation of classes and achieves a good run-time performance in both space and time. The environment takes care of automatically recompiling classes as needed after a change, ensuring that only up-to-date versions of classes are used, but avoiding unnecessary recompilations. A set of tools is provided to support the development of sizable software systems.

An important part of the environment is the library of reusable classes. Significant extracts of this library are given in the appendix to this article, providing a set of model reusable software components, carefully designed for robustness and extendibility.

PART 1: OVERVIEW OF THE LANGUAGE AND ENVIRONMENT

1 PRESENTATION

1.1 Background

Eiffel was initially an internal development at Interactive Software Engineering. The language was designed in late 1985 as a tool that would enable us to develop software engineering tools in accordance with our overall goal of promoting software quality. Most of Interactive's software is indeed now being produced with the Eiffel language and environment described in this report.

The decision to design and implement a new language is a far-reaching one, and it is legitimate to ask why I should have undertaken such a development. Yet an examination of available languages and environments quickly showed that none was up to the standards of modern software engineering that our products—software engineering tools—were meant to enforce. I felt that our own developments ought to observe these standards. Eiffel is the result of this decision.

The implementation of Eiffel (see Section 8) has been available since early 1986 for use within Interactive. The decision was made in December 1986 to release it as a commercial product, which is now installed at a number of industrial and academic installations in North America, Canada, Europe, and the Far East.

The system currently runs on Unix and is in the process of being ported to other environments, notably VAX-VMS. Several significant software products have already been implemented successfully using Eiffel and the basic library sketched in the appendix; applications developed at Interactive include the visual document constructor Cépage [24], the general-purpose window management system Winpack, and others.

Eiffel is not just a programming language. As a language, it can be fruitfully applied to the crucial early stages of software development: **specification** and **global design**. (Some features of the language that help in this respect are described in Section 4.10.) Beyond the language aspects, Eiffel is also a **method** of software design and as a **programming environment**:

- The method emphasizes system construction by combination of reusable and extendible modules, conceived as implementations of abstract data types; it is a bottom-up method, encouraging software devel-

Address correspondence to Bertrand Meyer, Interactive Software Engineering Inc., 270 Storke Road, Suite 7, Goleta, CA 93117.

* Eiffel is a trademark of Interactive Software Engineering Inc.

0164-1212/88/$3.50

opment by building on previous efforts rather than by starting every new effort from scratch.

- The tools of the environment, described in Section 8, support automatic recompilation, documentation, debugging, graphical design and documentation and other important tasks.

The rest of the article's main body reviews the language, method, and environment; it will enable the reader to understand the appendix, a set of programming examples from the basic Eiffel library. Section 1.2 gives an overview of the design criteria for Eiffel. Section 2 introduces some of the basic concepts of object-oriented design. Section 3 describes the fundamental Eiffel structure (the class). Section 4 presents the multiple and repeated inheritance techniques that constitute the key to reusable programming in Eiffel. The typing rules are described in Section 5, and the use of assertions for expressing correctness arguments are described in Section 6. Section 8 surveys the practical aspects of Eiffel usage and the supporting environment tools. Section 9 summarizes the main results, mentions some related efforts, and describes ongoing developments.

The appendix is a library of basic Eiffel classes defining a set of reusable software components. Although this is just a collection of Eiffel texts that may at first appear rather boring, it has been found to be invaluable to Eiffel programmers—novices and experts alike—and indeed I hope that it will prove to be the main contribution of this article in the long term. Beyond their use as models, the classes presented play a fundamental role in practical Eiffel programming. A complete documentation on the library is given in the library manual [14].

Although this article does not constitute a complete reference on Eiffel, the examples and discussions introduce all the essential features. Thus, if you understand the article, you may still have a few things to learn to become a real Eiffel designer or programmer, but not many.

Since this discussion will introduce a number of powerful language constructs, it is important to mention at the outset that Eiffel is by no means a *complex* language. Its size, as measured by such a criterion as the number of keywords (53), is only slightly higher than that of Pascal, for much more power. This is a result of a somewhat minimalist design. For example, there is no case instruction and only one form of loop. At a recent user group meeting, a speaker called the language "spartan" [29]; I have no quarrel with this characterization, although it may be more trendy to express the same idea by presenting Eiffel as a RISC language.

Other references on Eiffel include a brief overview [23], a study of the Eiffel approach to reusability [22] and a comparative analysis of Ada-like genericity with Eiffel-like inheritance [26]. Detailed technical documentation may be found in the user's manual [15]. A recent book [27] surveys object-oriented design and programming with special emphasis on the Eiffel approach.

1.2 Design Criteria

The design of Eiffel was guided by the following concerns.

- The aim is to produce software, not to do research on languages. **Efficiency** of the implementation was thus an important criterion.
- **Reliability** of the software that we produce was another fundamental aim, promoting such features as strict type checking, use of assertions, support for automatic configuration management, etc.
- Current program construction techniques too often lead to reinventing the wheel over and over again. **Reusability** of software should be a prime emphasis. Software development methods and languages should emphasize the **reusability** of software components as one of their primary goals.
- **Extendibility** of the resulting software (the ease of taking into account changes in specifications) is another essential goal if one is to take a comprehensive view of the software lifecycle.
- **Modular** language constructs should make it possible to construct and compile systems piecewise and to place strict controls on the flow of information between modules.
- A more technical requirement is the ability to create dynamic data structures and to rely on support tools for reclamation of unused space; placing the burden of space reclamation on application programmers (in the PL/I-Pascal-Modula 2 tradition) is a dangerous policy, the presence of which is unexplainable in any language whose designers have expressed concern for program reliability. (We shall see, however, that safe programmer-controlled deallocation may be provided in cases when automatic reclamation is too expensive.)
- Finally, **portability** is also a serious concern.

Of course, a solution to these issues must also involve elements that are not strictly technical. For example, the availability of good documentation and component libraries is essential to achieve reusability. However, in the current state of software technology, technical aspects such as languages are paramount.

As a picture of the language emerges in the descriptions given below, it will become clear that Eiffel is an original design, not an object-oriented extension of a classical language such as C (cf. C++ [31], Objective-C [8]), Pascal (cf. Object Pascal [32]), or Lisp (cf.

Loops [2], Flavors [7], Ceyx [13]). The use of a well-known language as stem has obvious advantages in terms of initial user acceptability, but it is more important to preserve coherence and integrity. The addition of object-oriented primitives to languages that (irrespective of their other qualities) are built on non-object-oriented principles can only, in my opinion, impair the consistency and simplicity of the result; yet these qualities are among the key criteria in language design [12].

Although it is not an extension of another language, Eiffel is not, of course, unrelated to previous efforts. The clearest conscious influences have been those of Simula, Alphard, and Ada (the latter for the the syntax). Also, it will be seen in Section 8 that the *implementation* of Eiffel is based on C, generates stand-alone C packages on option and that Eiffel software may be interfaced with software written in other languages.

2 OBJECT-ORIENTED DESIGN

The general approach to software construction that best addresses the above quality factors is the method pioneered by Simula 67 and known as object-oriented design and programming.

2.1 Overview

There are several ways to describe object-oriented design and programming, depending on the presenter's background [3, 4, 10, 20]. Because Smalltalk [10] has been so largely publicized, many current views of object-oriented programming emphasize two aspects: the concept of *messages* for communicating information between objects, and the very dynamic nature of the Smalltalk environment, which defers bindings between names and their denotations until run-time. This approach, strongly influenced by Lisp, offers much freedom to programmers, and it is useful for such application areas as artificial intelligence or rapid prototyping.

My interest in object-oriented languages comes from a more traditional software engineering perspective. I view these languages as providing key techniques for ensuring reusability, extendibility, and compatibility. However, in a software engineering context these qualities must be balanced with other criteria mentioned above, such as reliability, efficiency of the generated code, and portability. Thus, static type checking, for example, is an essential concern. In Eiffel, static typing is combined with a powerful type system, based on inheritance, and reconciled with dynamic binding.

My view was much influenced by Simula; I was particularly fortunate in having for many years access to an excellent compiler for that language, developed for IBM/MVS systems by the Norwegian Computer Center. This experience (summarized in a 1979 survey article [20]) convinced me that object-oriented programming was the right approach to produce extendible and reusable software. Eiffel improves (I hope) on the Simula concepts, but it is proper to mention my debt here.

2.2 Modularizing for Extendibility

In this discussion, object-oriented design is viewed as a **system modularization method**, relying on the idea that the structure of any software system should best be patterned, at the highest level, on the objects manipulated by the system, rather than on the system's function.

Arguments for this approach to software construction may be found in the references cited above; an analysis of its contribution to software reusability was given in [22]. Without repeating these discussions, it is useful to elaborate on another of the key criteria that justify this method: extendibility.

Observation of durable programs shows that the precise tasks performed by systems vary dramatically over their life cycle. If you take a program at a certain point of its evolution, you may well be able to describe its function as some input-to-output transformation: each run processes a batch of data and produces the corresponding results. But as the program is used and adapted, it will often evolve into a system that keeps some information between successive runs, and it may end up as an interactive system accessing a comprehensive data base, with finer-grain inputs and outputs for each individual transaction.

If they are studied from the standpoint of the tasks they perform, the initial and final versions may be very different. To realize that they are versions of the same program, you must look closer and consider the *objects* handled by the system. If they are viewed from a sufficiently high level of abstraction, these objects will in most cases turn out to be the same in both versions. For example, a payroll processing program, regardless of its precise functions, will act on data representing entities such as employees, company regulations, workload information, etc.; or a plant monitoring system will act on data representing sensors, devices, materials, and the like. In both cases, the system's identity is better characterized in the long term by these objects than by the more fluctuating functions that are applied to them.

2.3 Seven Steps Towards Object-Oriented Happiness

Based on the preceding remarks, the basic motto of object-oriented design may be formulated as follows:

Principle 1 *(object-oriented modular structure):* Ask not what the system does: ask what it does it to.

To get object-oriented design in its full sense, however, further steps must be taken. The next step takes into account the remark made above that object descriptions should be abstract enough; indeed, basing the structure of systems on the physical structure of data would produce rather disastrous results with respect to extendibility. A study of software maintenance costs by Lientz and Swanson [17] shows that, out of the more than 50% of software costs devoted to maintenance, about 17.5% arise from the need to account for changes in physical data formats. Thus, one would be ill-advised to hard-wire physical data representations into the physical structure of programs.

The answer lies in data abstraction. The theory of abstract data types provides a way to describe classes of objects by their external features rather than by their physical representations. The features in question are the operations applicable to objects of the class and the abstract properties of these operations. Note that these operations are what was called the "functions" above.

The complementarity between functions and objects is an unescapable fact of programming; object-oriented design does not contradict it, but introduces a dissymmetry by using objects, not functions, to structure software systems at the highest levels. With abstract data types, however, functions reappear as the way objects (or rather object classes) are characterized, so the loop is closed. The essential difference with classical techniques (based on procedural decomposition) is that functions are attached to data structures rather than the reverse.

The second step of object-orientedness is reached, then, through the application of the following principle:

Principle 2 *(data abstraction):* Objects should be described as implementations of abstract data types.

Most current programming languages make it possible to reach this level, i.e., to say to design modules that encapsulate the implementation of one or more abstract data types. Ada [1], CLU [18], and Modula-2 [34] are obvious examples of such languages. Even Fortran may be used for this purpose by writing subroutines with more than one entry (corresponding to the various operations on an abstract data type); however, what is provided in the Fortran case is the implementation of a fixed number of abstract objects, rather than of an abstract data type. In languages such as Pascal, Cobol, or Basic, on the other hand, it is not possible to devote a module to the implementation of an abstract data type or abstract data object.

The third step is of a less conceptual nature. It reflects an important implementation concern: how to manage space for objects. If programmers are to freely use dynamically created objects, they should not have to take care of where cells are found for newly created objects and, even more importantly, how cells are reclaimed when their objects are no longer needed. Although this is in a strict sense a property of implementations rather than languages, the language design may help or hinder the implementation of a garbage collector. Pascal and Modula-2 systems do not normally include garbage collection; the Ada standard [1] defines it as an optional feature.

On the other hand, all Lisp systems provide garbage collection, which is part of the reason why Lisp has often been used to implement object-oriented languages and has itself been subjected to object-oriented extensions.

Principle 3 (*automatic memory management*): It should be possible to let the underlying language system take care of automatically reclaiming unaccessible memory elements.

Automatic garbage collection is sometimes viewed with suspicion because of its effect on performance. As described in Section 8.8, this problem is addressed in Eiffel by using an incremental garbage collector implemented as a co-routine; also, the collector may be disabled when it is not needed.

The next step truly distinguishes object-oriented languages from the rest of the flock. It may be understood by looking at languages that are not object-oriented even though they provide facilities for data abstraction and encapsulation, such as Ada or Modula-2. In such languages, the module (package in Ada) is essentially a syntactic construct, used to group logically related program elements; but it is not itself a meaningful program element, such as a type, a variable, or a procedure, with its own semantic denotation. In contrast, the approach pioneered by the designers of Simula views modules as first-class citizens; more precisely, it all but identifies the notion of module with the notion of **type**. We may say that the defining equation of such languages is the identity *module* ≡ *type*.

This fusion of two apparently distinct notions is what gives object-oriented design its distinctive flavor, so disconcerting to programmers used to more classical approaches. In its dogmatism, it has some drawbacks. But it also gives considerable conceptual integrity to the general approach.

Principle 4 *(classes):* Every nonbasic type is a module, and every high-level module is a type.

A language construct combining the module and type aspects is called a **class**.

The qualifier "nonbasic" keeps open the possibility of having simple types (such as *INTEGER* etc.) that are not viewed as modules, and the word "high-level" makes it possible to have program structuring units such as procedures, which are not types.

The next step is a natural consequence of Principle 4. If we identify types with modules, then it is tempting to identify the reusability mechanisms provided by both concepts: on the one hand, the possibility for a module to directly rely on entities defined in another (provided in modular languages by such visibility mechanisms as the Ada "use" clause); on the other hand, the concept of subtype or derived type, whereby a new type may be defined by adding new properties to an existing type. In object-oriented languages, this is known as the inheritance mechanism, with which a new class may be declared as an extension or restriction of a previously defined one. Its realization in Eiffel is described in section 4.

Principle 5 *(inheritance):* A class may be defined as an extension or restriction of another.

We shall say in such a case that the new class is *heir* to the other.

The above techniques open the possibility of an advanced form of *polymorphism,* in which a given program entity may at run time refer to objects belonging to any of a set of different classes, all of which offer an operation with the same external specification but different implementations. The application of an operation to the entity will result in the appropriate implementation being selected, depending on the particular object associated with the entity at the time the operation is executed. For example, an entity representing a device might become associated at run-time with either a tape or a disk; the operation "read" applied to the entity will be carried out differently in each case.

Principle 6 *(polymorphism):* Program entities should be permitted to refer to objects of more than one class, and operations should be permitted to have different realizations in different classes.

This principle is implemented in different ways according to the philosophy underlying existing languages. In the design of Smalltalk, it is satisfied almost automatically because of the dynamic binding policy: Entities have no static types, so that they may at run-time refer to objects of any class; when an operation is requested on an entity, its dynamic state determines what realization, if any, is available for the operation.

In contrast, every Eiffel entity has a static type that (except for basic entities such as integers or booleans) is defined by a class; the dynamic types it may take are restricted to the descendants of that class (that is to say, the class itself and its direct and indirect heirs). The above principle is implemented in Eiffel by permitting the *redefinition* of a class operation in a descendant and by having *deferred* operations whose implementation is only given in the descendants.

The next and last step extends the notion of inheritance to enable reusing more than one context. This is the notion of multiple inheritance, developed in Section 4 below. Eiffel adds to this notion the concept of **repeated** inheritance (reusing the same structure more than once); see 4.7 below.

Principle 7 *(multiple and repeated inheritance):* It should be possible to declare a class as heir to more than one class, and more than once to the same class.

The seven above principles have alternated between high-level, design-related concepts and programming language features. One particularly interesting benefit of the object-oriented approach is indeed that the same language may be used for design and implementation. Some language traits, such as deferred features (4.10) and assertions (6), are especially useful for the application of Eiffel to system design.

2.4 Eiffel Versus Other Object-Oriented Languages

It was mentioned in the introduction that no existing language was deemed acceptable for our purposes. As we are about to explore Eiffel in some detail, it is useful to explain this claim by previewing the combination of facilities that is unique to Eiffel and its implementation:

- Multiple and repeated inheritance. Commercially available object-oriented languages, with the exception of AI-oriented languages such as LOOPS and Flavors and recent versions of Smalltalk, support single inheritance only, and no language we know of supports repeated inheritance.
- The renaming techniques (apparently unique to Eiffel) that are needed for a safe treatment of multiple inheritance.
- Generically parameterized classes, necessary to obtain truly flexible software components in the presence of type checking. (The only other object-oriented language supporting genericity appears to be Trellis-Owl [30], an internal DEC development.)
- Static type checking (not present in other languages with the exception of Object Pascal, Trellis-Owl, and, in a limited form in C++).
- Primitives for systematic program construction (not available in other object-oriented languages), consistent with the inheritance mechanism.
- Automatic configuration management within the context of object-oriented programming.
- Constant-time routine binding.

- Incremental garbage collection.
- Class documentation facilities.

3 BASIC EIFFEL CONCEPTS

The basic elements of Eiffel programming will now be introduced: run-time model, objects, classes, and export controls.

3.1 Run-Time Model

The execution of Eiffel systems (a term that is preferred to "programs" for this language) relies on a dynamic execution model. The execution of a system may be characterized at every instant by the presence of a certain number of **objects**, each of which possesses some **attributes**. Attributes are either simple values (integers, booleans, reals, or characters) or *references* to objects. Figure 1 gives a pictorial view of such a collection of objects and their attributes.

3.2 Routines

Operations, or **routines**, may be applied to objects. Routines are divided into **procedures** and **functions**. You may think of procedures as *commands* and functions as *queries*: A procedure may change the state of the associated object but does not return a value, whereas a function returns a value without normally modifying the object. A related analogy would be to see the objects as having action buttons, the procedures, and display indicators, the functions. The **features** associated with an object comprise its attributes and the routines that are applicable to it.

The execution of an Eiffel system is started by creating an object and calling one of its procedures; executing this procedure will usually trigger the creation of other objects and more routine calls.

3.3 Classes and System Structure

Every object that may be created during the execution of an Eiffel system is an instance of a **class**. An Eiffel system is an assembly of classes.

A class describes a set of potential objects (the instances of the class) through the features (attributes and routines) that are applicable to all of these objects.

In other words, a class describes the **implementation of an abstract data type**.

As implied by the above principles, classes are not

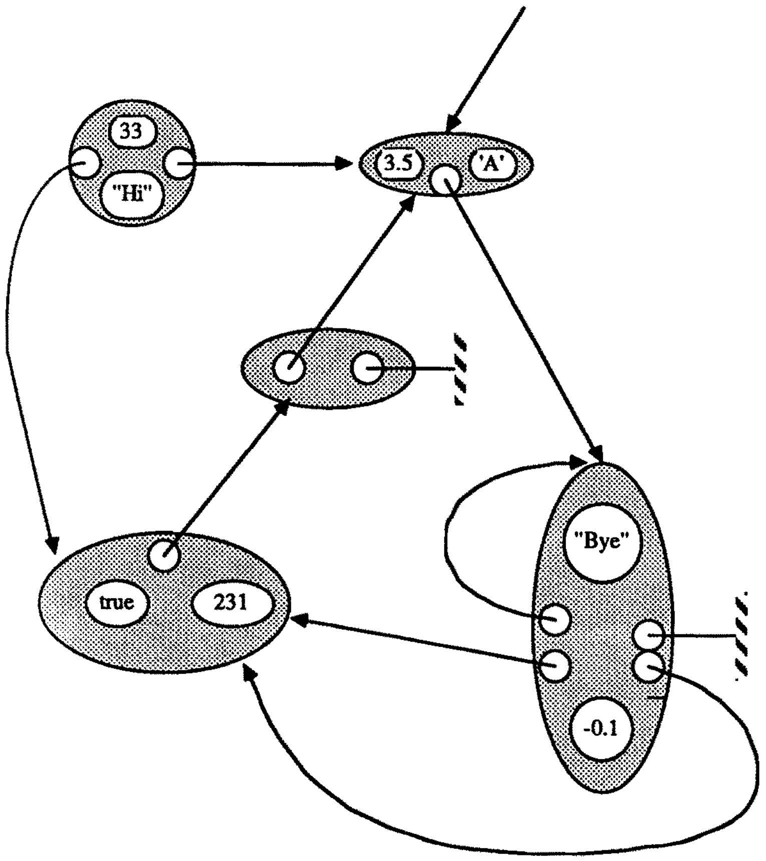

Figure 1. Objects.

only types but also modules. In fact, they constitute the only system structuring facility.

3.4 Entities

An Eiffel system contains **entities**, which may take values at run-time. Although close to the usual notion of variable, the notion of entity is more general since it includes not only local variables of routines (including the predefined variable *Result* denoting the result to be returned by a function), but also references to object attributes and routine arguments.

Eiffel is a strongly typed language: every entity is declared with a single static type. Four types, called "simple," are predefined: *BOOLEAN, CHARACTER, INTEGER,* and *REAL*. Any other type is defined by a class.

3.5 States of an Entity

Let *x* be an entity and *C* its type, assumed to be a class type. At any point during system execution, *x* may or may not be associated with an object. If it is, we say that *x* is "created," if not, that it is "void." The boolean expression *x. Void* has value true in the latter case only.

Instruction *x.Create* puts the entity *x* in the created state by creating a new object of type *C* and associating it with *x*; note that this must be done explicitly as all entities are initially void (initialization rules will be seen below).

Conversely, *x.Forget* plus *x* in the void state. It must be emphasized that *x.Forget* does not by itself deallocate the object associated with *x*, which would be a violation of principle 3 above; this instruction merely suppresses the relationship between the entity *x* and the object associated with it, making this object a candidate for automatic space reclamation if there was no other associated entity.

Figure 2 shows the two states, the transitions between them, and the allowable operations in each. As the figure shows, there are other ways to alternate between states, for example by assignment (see below).

Void, Create, and *Forget* are **predefined features** applicable to all classes. The language includes another predefined feature: *x.Clone (y)* creates a new copy of the object referenced by *y* and assigns to *x* a reference to the new object.

3.6 Initialization

Every entity has an initial value. The initialization rules are part of the language definition: they are not implementation-dependent.

By default, numbers will initially be 0, booleans will be false, characters will be null, and object references will be void.

If a different initialization is desired for the attributes of objects of a class *C*, a procedure called *Create*, with or without arguments, may be defined for that class; it will then be applied to every object of the class upon creation. This is what is done in section A.2 for the *ARRAY* class, for which a version of *Create* is defined in such a way that *a.Create (min, max)* will associate with *a* a newly allocated array with bounds *min* and *max*.

3.7 Feature Declarations

A class declaration introduces a set of features associated with objects of the class: attributes and routines, the latter comprising procedures and functions.

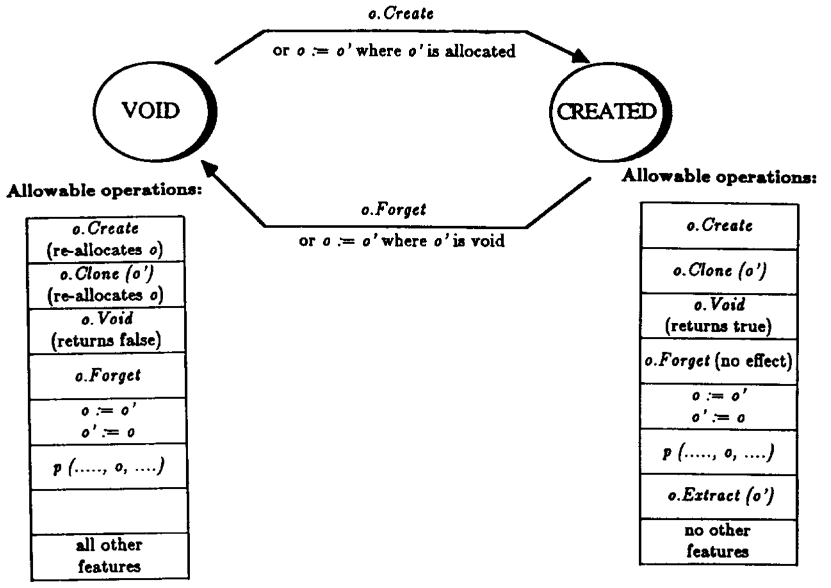

Figure 2. States of a reference, permissible operations, and transitions.

Routines may have arguments. The arguments of a routine, whether a procedure or a function, are protected in its body: The routine may not include an assignment to one of its formal arguments. However the attributes of the *object* associated with a argument may be modified in the procedure.

3.8 Expressions and Instructions

The construct expressing the application of feature f to the object associated with entity x, called a remote feature application, uses a dot notation. If f is an attribute or a routine without arguments, the notation is

$x.f$

If f is a routine with arguments, actual arguments must be provided:

$x.f(p_1, p_2, \ldots., p_n)$

Either form of remote feature application is only valid if x is declared of a class type for which f is a valid feature. Syntactically, the remote application is an instruction if f is a procedure, or an expression if f is a function or an attribute.

Assignment is written with the standard := operator. For class types, the semantics of assignment is by reference, not copy: Entities of class types represent references to objects, not the objects themselves. Thus, for entities of class types the assignment $x := y$ results in x and y being references to the same object (or x being void if y was void before the assignment).

Control structures include the loop, the conditional, and sequencing, represented by the semicolon.

3.9 A Simple Class

The example below shows the basic structure of a class. It introduces an elementary notion of "point" that could be used (with suitable extensions) in a graphics system.

Any part of a line beginning with two consecutive dashes -- is a comment.

```
class POINT export
    x, y, translate, scale, distance
feature
    x, y: REAL ;
    scale (factor: REAL) is
                -- Scale by a ratio of factor.
        do
            x := factor*x ;
            y := factor*y
        end ; -- scale
    translate (a, b: REAL) is
                -- Translate by a horizontally, b vertically.
        do
            x := x+a ;
            y := y+b
        end ; -- translate
    distance (other: POINT): REAL is
                -- Distance from current point to other.
        require
            not other.Void
        do
            Result := sqrt ((x - other.x)^2 + (y - other.y)^2)
        end -- distance
end -- class POINT
```

The features of this class comprise two attributes, x and y, and three routines: two procedures, *translate* and *scale,* and one function, *distance*.

The **export** clause says which features are public. Here all features are public, but in general classes will possess "secrets." Public features may be used by **clients** of the class, i.e., to say classes that include one or more entity declarations of the form

p: POINT

and may thus execute operations such as

```
p.Create ;                -- Allocate POINT object and associate it with p
p.translate (3.5, 2.2) ;  -- Translation
r := p.x                  -- Get abscissa of x
```

In client classes, public attributes (here x and y) are accessible in read-only mode: An assignment such as $p.x$:= ... is not permitted; the corresponding effect may only be obtained in a client class by calling a public procedure that will modify the attributes itself, such as *translate* in the *POINT* example.

It is also possible to export a feature f to a selected set of classes C_1, C_2, ... only, by listing it as $f\{C_1, C_2, ...\}$ in the export clause.

The text of an Eiffel class always refers to a **current instance** of the class. Most of the time this current instance is anonymous; in a class (like *POINT*), a feature name (like x) that appears unqualified (i.e., just x, not $p.x$ for some p of type *POINT*) denotes the corresponding feature of the current instance. If you need to refer explicitly to the current instance, the predefined entity name *Current* is available. Thus you may consider a name such as x, appearing unqualified in class *POINT*, as a synonym for *Current.x*.

The special variable *Result* is used in functions: As shown by the example of *distance*, it denotes the result to be returned by the function in which it appears. It is considered as implicitly declared of the appropriate type (*REAL* in the case of function *distance*).

3.10 Generic Parameters

The basic class structure presented so far is made more flexible by the provision for **genericity**. A class may have one or more generic parameters that represent types. For example, Section A.6 introduces a class representing linked lists of objects of an arbitrary type *T*; its declaration begins with:

class *LINKED_LIST* [*T*] **export**....

The presence of *T* as generic parameter allows the class to contain declarations of entities of type *T*. A client of the class will then declare entities of type *LINKED_LIST* [*INTEGER*], *LINKED_LIST* [*POINT*], etc.

Genericity is particularly important in connection with static type checking. Without this facility, it would be impossible to define data structures such as *LINKED_LIST* whose constituents are statically guaranteed to be all of the same type (*INTEGER*, or *POINT*, etc.).

The "horizontal" form of extendibility, as provided by generic parameters, is a useful complement to the more powerful "vertical" extendibility features offered by inheritance and described below.

The power of such a combination is evidenced by the examples of the appendix. A more detailed comparative analysis of genericity and inheritance and a rationale for the particular blend achieved in Eiffel may be found in another article [26].

4 INHERITANCE: TREES ARE LISTS AND LIST ELEMENTS

4.1 Definition

Inheritance is a key technique for reusability.

When a new class is declared as heir to a previously defined one, it posseses by birth all the features of that parent class and their associated formal properties. The inherited features are not redeclared in the new class, but new features may be added. Both the inherited features and the new ones become an integral part of the class and may be transmitted to further classes defined by inheritance.

This mechanism has a significant influence on the process of software design, as it allows software to be constructed through progressive accumulation of features rather than in a single setting. New features acquired in this process are passed along to descendants.

Syntactically, inheritance is described through the ***inherit*** clause in class declarations, as follows:

```
class C export
            ...
inherit
    P1
            ... Possible "rename" and/or "redefine" sub-clause (see 4.6–4.9 below)... ;
    P2
            ... Possible rename and/or redefine... ;
            ... Other parents ...
feature
        .... Declaration of specific features of C ...
end -- class C
```

As the syntax shows, inheritance as offered by Eiffel is **multiple**: a class may inherit from as many classes as needed. The only constraint is that the inheritance graph should be acyclic.

We rely on the following terminology, some of which has already been used above. An **heir** of a class *P* is a class *C* that lists *P* in its **inherit** clause. The **descendants** of a class *P* are *P* itself and the descendants of its heirs. The reverse notions are **parent** and **ancestor**.

4.2 An Example

The following example shows the power of multiple inheritance. Perhaps, if the reader remembers just one idea from this article, it should be this: *a tree is a list and a list element*. Let's explain.

The classes of the appendix describe *lists* of various brands. One of these classes has already been mentioned: *LINKED_LIST* [*T*] (Section A.6), describing one-way linked lists of elements; it itself inherits some of its properties from a more general class, *LIST* [*T*] (Section A.3), which introduces properties of arbitrary lists without commitment to a particular representation. As may be expected, the features declared in class *LINKED_LIST* include routines for inserting elements at various places into a list, removing elements, accessing elements, etc.

To manipulate linked lists of elements of type *T*, you need a data structure for the individual components of a linked list; such components are cells consisting of two fields, a value of type *T* and a reference to another cell. Let's use the word "linkable" to refer to such cells. Their description is given in class *LINKABLE* [*T*] (Section A.5). Among the features of "linkables" are two attributes: *value*, of type *T*, and *right*, of type *LINKABLE* [*T*].[1]

Now assume you need to define the notion of **tree**, as implemented in linked representation. You may certainly start from scratch; programming tradition, as well as fifteen years of propaganda for top-down design, indeed encourage you to do so. But the eventual result is assured to look very much, at least in part, like what was obtained for lists: insertions, deletions, access to subtrees, etc. The main difference is that here these operations apply to subtrees rather than list elements.

But from this last remark comes the light: A tree is indeed a list (since it is made of a number of subtrees), and **also** a list element (since it may be used as subtree for another tree). Hence the solution described in Section A.8, whereby trees inherit from both lists and list elements:

```
class TREE [T] export... inherit
    LINKED_LIST [T];
    LINKABLE [T]
    feature ...
```

Of course, this is not quite enough: you must add the specific features of trees, and the little mutual compromises that, as in any marriage, are necessary to ensure that life together is harmonious and prolific. But it is significant that the new data structure may essentially be engendered as the legitimate fruit of the union between lists and list elements.

This process is exactly that used in mathematics to combine theories: a *topological vector space*, for example, is a *vector space* that also is a *topological space*; here too, some connecting axioms need to be added to finish up the merger.

Multiple inheritance is a fundamental tool in our daily practice of Eiffel. Many classes have four or five parents. The following four examples of double inheritance are typical:

- Our windowing system uses a class *WINDOW*. Windows have graphical features: A height, a width, a position, etc., with associated routines to scale windows, move them and so on. Our system permits windows to be nested, so that a window also has hierarchical features: subwindows, a parent window, routines to add a subwindow, delete a subwindow, attach to another parent and so on. Rather than writing a complex class that would contain specific implementations for all of these features, it is much preferable to inherit all hierarchical features from the above *TREE* class, and all graphical features from a *RECTANGLE* class.
- In the basic library, class *FIXED_LIST* [*T*] (Section A.4) describes lists with a fixed number of elements, implemented using arrays. It is simply defined as heir to both *LIST* [*T*] (general lists, without commitment as to a specific representation) and *ARRAY* [*T*] (arrays). We call this form of multiple inheritance the "marriage of convenience": One parent brings the functionality, the other brings the implementation.
- Another class of the basic library, *TEST*, defines an environment for software testing. To test a class *X*, one may define a new class, say *X_TEST*, as heir to *X* and *TEST*, gaining access to primitives from both classes. Without multiple inheritance, this would be impossible, as *X_TEST* would have to choose

[1] Feature *right* is actually declared of type **like** *Current* for reasons explained in 5.2.

between inheriting from test and from *X*'s own ancestor.

• A basic problem in programming with complex data structure is how to store such structures in long-term memory (files). In object-oriented programming, this is the problem of persistent objects. In the Eiffel environment, a class *STORABLE* is defined, with routines *store* and *retrieve*; a whole data structure may be stored and retrieved using these routines if the root of the structure is an object whose type is a descendant of *STORABLE*.

Figure 3 gives the structure of the inheritance graph for the classes in the Appendix. Arrows show the inheritance relation.

4.3 Inheritance and Reusability

Why are inheritance techniques so crucial for the production of reusable software? One of the reasons for their superiority is that they make it possible to write software modules that are both **open** and usable as they stand, whereas these two aims are contradictory with classical methods.

Consider the typical language structure used to support these methods, the data types with "variant parts" as offered by Pascal and Ada. Such constructs do make it possible to write software elements that may exist in several versions; but as soon as you need to actually use such an element (by compiling it if it is a program element), the list of possible variants must be frozen; any later addition of new variants will imply that existing software elements, which relied on the initial version, have to be modified.

Similarly, any change in the list of formal arguments to a routine, in the set of generic parameters to an Ada package, or in the repertoire of operations available on an abstract data type, will result in tricky problems of software configuration.

In contrast, multiple inheritance makes it possible to use a class—to store it, to compile it, to execute its routines, etc.—and at the same time to leave open the possibility that the class will eventually be used as parent for an unlimited number of descendants, corresponding to all the cases that you did not envision initially. This may be stated as the **principle of openness**: Any software element, even if it is in a directly usable form, should remain amenable to future extensions.

A further example of the application of this principle to Eiffel is the fact that the language does *not* include an instruction (such as the **inspect**... **when**... instruction of Simula 67) to discriminate between the various heirs of a class. Were such an instruction to exist in Eiffel, class *LIST*, for example, could contain an instruction that chooses between several actions depending on whether

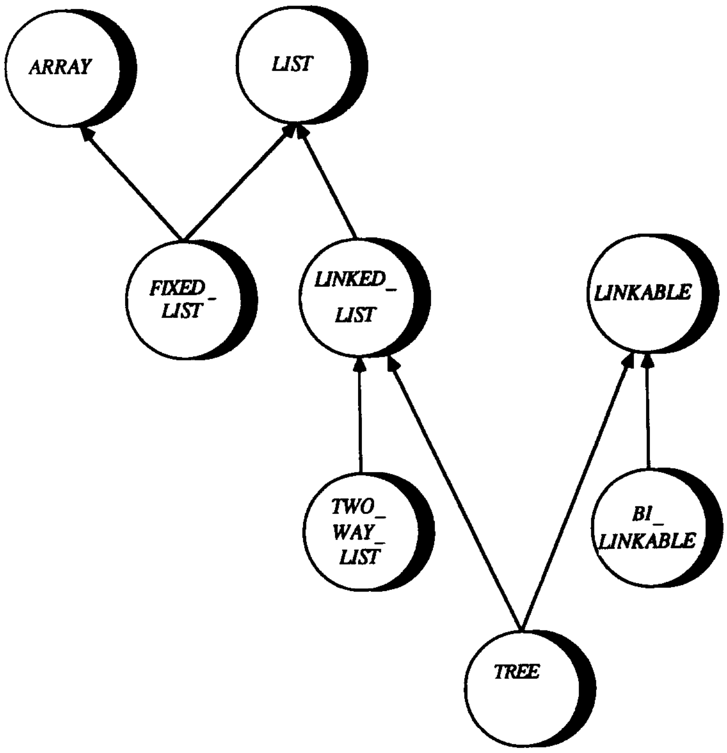

Figure 3. Inheritance graph for the examples.

the current list is a *FIXED_LIST*, a *LINKED_LIST* etc. But this would mean that *LIST*, as part of the knowledge it embodies, has information on the set of its possible heirs: thus it would no longer be open for designing new heirs without modification. To achieve the effect of **inspect** in Eiffel, you may use such mechanisms as deferred and redefined features (presented below), which preserve openness.

4.4 Inheritance and Export Controls

The Eiffel inheritance mechanism is orthogonal to the information hiding mechanism provided by export controls. Notwithstanding its export clause, a class will bequeath all its features to its descendants—the family secrets as well as the public facade. To reject part of this heritage, specific techniques must be used, such as feature renaming and redefinition, seen below; the export restrictions apply to *clients* of the class (see Section 3.9 above), not to its descendants. It is even possible for a class to export a feature inherited from another class in which that feature was secret.

I have found the orthogonality between the export and inheritance mechanisms to be a shock to some people, but a moment's reflection should convince the reader that this is indeed a correct decision.

The following example shows a case in which a feature that is secret in a class needs to be reexported in a descendant. Consider again the relationship between linked lists and trees. The notion of *LINKABLE* cell should be of no concern to clients of the class *LINKED_LIST* [*T*], which only need to deal with lists, of type *LINKED_LIST* [*T*], and values of list elements, of type *T*. Internally, class *LINKED_LIST* uses a feature called *active*, which represents the cell at the current cursor position. (A list has an associated cursor, which points to the currently active position; this is discussed in A.3.) Feature *active,* of *type LINKABLE* [*T*], it naturally secret; it is used for the implementation of exported features such as *value* (the value of the element at cursor position), *insert_right* (insert a new cell of given value at the right of cursor position) etc. The list cells themselves are none of the clients' business.

For trees, however, the picture changes. As we saw, trees are lists and list elements; the notion of cursor position transposes to that of a currently active child of a tree node. Here the child node itself is needed, not just its *T* value as returned by feature *value*; to perform tree traversal operations, you must be able to go from parent to child, both considered as tree nodes. Feature *active* is thus exported in class *TREE* [*T*], even though it is inherited from a class where it was secret. (The renaming mechanism, described below, enables class *TREE* to refer to this feature under the name *child*, more appropriate for the occasion.)

Restricting descendants' access to any of the features defined in a class would be a direct violation of the openness of classes, which has been presented above as a fundamental aspect of inheritance. Long after a class has been written, a software developer may reuse it through inheritance, with any extensions and adaptations that are needed for a new application. The power of inheritance comes from the possibility of performing these extensions and adaptations without impacting the original class or any of the other software elements that depended on it. This means that the original designer has no way of knowing what new uses will later be found for the class. Accordingly, the designer does not know which features a descendant may need to export and which it will need to hide.

To understand the relationship between inheritance and export controls, you may note that the two main reusability mechanisms of Eiffel are complementary: When class *A* is a client of class *B*, *A* only uses *B*'s specification; on the other hand, by inheriting from *B*, *A* may directly rely on *B*'s implementation, and information hiding does not apply to it. These two ways of reusing existing a software component—through its interface and through its implementation—are equally important in practice; care should be exercized to determine which one is appropriate in any given case.

4.5 Types of Entities and Objects

The inheritance relation may be viewed as an "is-a" relation [6], in the sense that a window "is-a" rectangle and also "is-a" tree, From this remark comes the rule that a language entity declared of a certain class type, say *C*, may at run-time refer to an object of any descendant type of *C*. For example, an entity declared

l: *LIST* [*INTEGER*]

may refer to a two-way list or to a tree of integers. The reverse, however, is not true.

If we call the type with which an entity is declared its "static" type and the type of the object to which the entity (if not void) refers at some point during system execution its "dynamic" type, the rule is that the dynamic type must be one of the descendants of the static type (which include the static type itself). Whenever we talk about the type of an entity, without further qualification, we always mean its static type.

4.6 Renaming

The availability of multiple inheritance raises the problem of name clashes: What happens when two or more parent classes have a feature with the same name?

The basic rule is simple. Within a class, there may be no name conflict (overloading): Any unqualified name must denote one and only one feature. This is essential for read
ability and safety. (In contrast, languages such as Loops resolve conflicts on the basis of the order in which parents are listed, a rather unsafe convention.)

Of course, it is inevitable that classes developed separately will include features with the same names; but it should still be possible to combine such classes through multiple inheritance. Renaming solves the dilemma by allowing the heir, at the point of inheritance, to resolve any name conflict by renaming selected features of the parent classes. The **inherit** clause will appear as:

```
class C export ..... inherit
        A
                rename m₁ as n₁, m₂ as n₂, .....;
        B
                rename p₁ as q₁, p₂ as q₂, .....;
        ......................
feature
        ......................
```

Within the rest of class C, the renamed features will be known by their new names (n_1, n_2, ..., q_1, q_2, ... etc.).

The ban on overloading applies to the set of names that are visible in the class after renaming has been applied and may be expressed as the following renaming principle:

> If two parents of a class possess identically named features, the inheritance clause of the class must remove any name conflicts through renaming.

Renaming also has another important application: to enhance clarity by providing more appropriate feature names in a descendant. For example:

- Class *WINDOW* inherits routine *add_child* from *TREE*, but renames it *add_subwindow* to provide consistent "window" terminology to its clients. The writer of, say, a text editor (say) needs a good window abstraction but has no business knowing that this abstraction was implemented by inheriting from a particular set of parents.
- The boolean function which tests whether a list is empty is called *empty* for lists in the strict sense (Sections A.4–A.7) and renamed *is_leaf* for trees (section A.8) to conform to usual tree terminology. Saying that a tree node is a leaf is the same as saying that, viewed as the list of its subtrees, it is empty.

4.7 Repeated Inheritance

An interesting consequence of the renaming policy is an Eiffel concept that extends multiple inheritance: **repeated** inheritance.

Repeated inheritance occurs whenever a class inherits more than once from a given ancestor. The ancestor may be a parent, or it may be a more remoteancestor (see Fig. 4). Below is an example of the second case (indirect repeated inheritance), which occurs whenever a class has two parents with a common ancestor (see part (b) of the figure).

Assume for example a class *TAXPAYER* with attributes such as

```
age: INTEGER;
address: STRING;²
bank_account: ACCOUNT;
taxpayer_id: INTEGER;
```

and routines such as

```
birthday is do age := age + 1 end;
pay_taxes is ..... ;
deposit_to_account (sum: INTEGER) is ..... ;
```

etc.

An heir of *TAXPAYER*, taking into account the specific characteristics of U.S. tax rules, may be *US_TAXPAYER*. Another may be *FRENCH_TAXPAYER* (with reference to places where taxes are payed, not citizenship).

Now we may want to consider people who pay taxes in both France and the United States, perhaps because they reside in each country for some part of the year. The natural way to express this is to use multiple inheritance: class *FRENCH_US_TAXPAYER* will be declared as heir to both *US_TAXPAYER* and *FRENCH_TAXPAYER* . This is the scheme of Figure 4 (b).

What happens with the features that are inherited twice from the common ancestor *TAXPAYER* such as *address, age, taxpayer_id,* etc.? Applied strictly, the renaming principle of in the previous section would force the programmer to rename these features in the new class.

But the principle does not seem justified here, as there

[2] Strings in Eiffel are instances of a predefined library class *STRING*.

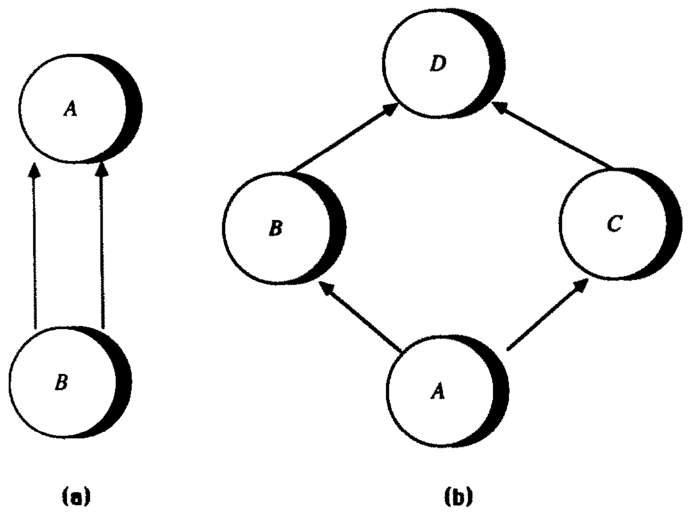

Figure 4. Repeated inheritance—direct (a) or indirect (b).

is no real name clash: The apparently conflicting features are in fact the same feature, coming from the common ancestor *TAXPAYER*. The two versions of *age*, for example, are really the same (unless you are trying to hide something, you should declare the same age to both the U.S. and French treasuries). On the other hand, the *taxpayer_id* attributes inherited from both parents should remain distinct. This will be achieved simply by renaming them at the inheritance point, as *us_taxpayer_id* and *french_taxpayer_id*.

The Eiffel convention for repeated inheritance follows from this discussion:

In repeated inheritance, any feature from the common parent is considered shared if it has not been renamed along any of the inheritance paths. Any feature that has been renamed at least once along any of the inheritance paths is considered replicated.

This rule applies to attributes as well as routines; a consequence is that it is a compile-time error for the body of a nonrenamed routine (which would thus be shared) to contain references to one or more renamed attributes or routines (which would be duplicated, leaving the meaning of the shared routine ambiguous).

This rule yields the desired flexibility in combining classes. For example the **inherit** clause of class *FRENCH_US_TAXPAYER* might look like:

```
inherit
    FRENCH_TAXPAYER
        rename
            address as french_address,
            taxpayer_id as french_taxpayer_id,
            pay_taxes as pay_french_taxes,
            bank_account as french_bank_account,
            deposit_to_account as deposit_to_french_account,
            ....................
    US_TAXPAYER
        rename
            address as us_address,
            taxpayer_id as us_taxpayer_id,
            pay_taxes as pay_us_taxes,
            bank_account as us_back_account,
            deposit_to_account as deposit_to_us_account,
            ....................
```

Note that features *age* and *birthday*, which have not been renamed along any of the inheritance paths, will be shared, which is indeed the desired effect.

With this rule, the renaming principle may be qualified by adding that the presence of identically named features in parents of a class is not considered a name conflict if the features come from a common ancestor and neither has been renamed at any point in the inheritance process.

The Eiffel implementation (see Section 8) achieves sharing or duplication of attributes according to the above rule; no space is lost (that is to say, no space needs to be reserved in class instances for unaccessible attributes). The same effect is achieved for routines. For shared routines, no code is duplicated; for routines which must be replicated according to the above rules, code must be duplicated. This is the only case in the Eiffel implementation in which code is ever duplicated.

4.8 Feature Redefinition

Another property of multiple and repeated inheritance is the possibility to **redefine** a feature of a class *C* in a descendant class, say *D*. The inheritance clause of class *D* may list some of the *C* features as being redefined in *C*, under the form

```
class D export ...
inherit
    C
        redefine f, g, h ...
```

In this case, the **feature** clause of *D* must include new declarations for the features *f, g, h* . . . listed in the **redefine** clause. These declarations override those of *C*: a feature application of the form

x.f

(possibly with arguments if *f* is a routine) will refer to the *D* version if *x* is of dynamic type *D*. This applies both when *x* is declared of type *D* and, more interestingly, when *x* is declared of type *C* but happens at run-time to be of dynamic type *D* (because of previous assignments) when feature *f* is applied to it.

Some constraints, of which the most important are described in Section 5.1, restrict the types that may be given to redefined features and (in the case of routines) to their arguments.

Feature redefinition is the basic mechanism for achieving polymorphism in Eiffel. It adds yet another element of flexibility to software design by permitting a set of related classes to provide alternative implementations of the same operation.

As a simple example, consider a set of graphic classes, including *POLYGON*, with *RECTANGLE* among its heirs, itself with heir *SQUARE*. *POLYGON* may have among its features a list of points, say *vertices*, giving the vertices of a polygon, and a function *perimeter* that returns its perimeter. The implementation of *perimeter* performs a traversal of the *vertices* list to compute and sum the distances between adjacent vertices. Class *SQUARE*, on the other hand, has a feature *side* giving the length of a square's side. It is clearly appropriate to redefine feature *perimeter* in this class to simplify the computation, which in this case just returns 4∗*side*.

Assuming the declaration

p: POLYGON

entity *p* could at run-time, as we have seen, refer to an object of type *SQUARE*. The function call *p.perimeter* would then result in the *SQUARE* version of the function being applied, whereas the same call executed when *p* refers to an object of type *POLYGON* would have triggered the execution of the *POLYGON* version.

A further degree of flexibility is provided by the ability to redefine a function feature (without arguments) as an attribute. From an information hiding viewpoint, it is useful to provide clients with a feature under such a form that it does not make any difference for them whether the feature is implemented as an attribute (that is to say, stored along with each object of the class) or a function (computed when requested); the notation for remote feature application is indeed the same in both cases: *x.f*. With inheritance coming into the picture, the idea is carried further by allowing descendants of a class to redefine as an attribute a feature declared as a function in the ancestor.

For example, one-way linked lists (class *LINKED_LIST*, Section A.5) include a function feature *last*, returning the last value of a list; here you must traverse a list to get to its last element, so a function is indeed necessary. For two-way linked lists (*TWO_WAY_LIST*, Section A.6), a reference to the last element will be permanently kept by each list, so that *last* becomes an attribute in this class.

Legitimate concerns may be voiced as to the power of the redefinition mechanism: does it not allow dangerous manipulations? A feature application

a.f (....),

where the type of *a* is a class type, say *A*, could have totally unexpected results if *a* may be assigned values of descendant types of *A* where *f* is redefined in a manner inconsistent with the original intent of *A*'s author.

Nothing indeed prevents the author of *SQUARE* to redefine *perimeter* so that it will compute, say, the area rather than the perimeter.

Although Eiffel does not provide an absolute protection against such abuses of the redefinition mechanism, it does address the problem. As will be explained in Section 6.3, a partially formal specification may be associated with a routine feature in terms of preconditions and postconditions. If this is the case, any redefinition of the routine must obey the initial specification (6.6).

4.9 Redefinition Versus Renaming

Redefinition and renaming serve different purposes and should not be confused.

Redefinition is applied to ensure that the *same* feature name refers to *different* actual features depending on the type of the object to which it is applied (that is to say, the dynamic type of the corresponding entity). It is thus an important **semantic** mechanism for providing the object-oriented brand of polymorphism.

Renaming, on the other hand, is more of a syntactic mechanism, making it possible to refer to the *same* feature under *different* names in different classes.

The two techniques are indeed orthogonal; either or both may be applied (in a descendant D of a class C) to a feature of C, say f. They address different questions:

- Redefinition corresponds to the question "can we have a different implementation for f when it is applied to entities of dynamic type D?".
- Renaming corresponds to "can we change the name under which the original (C) implementation of f may be applied to entities of static type D?"

The effect of combining these two mechanisms in various ways, summarized in the table below (Table 1), follows from this discussion. Assume that entities c and d are declared of types C and D respectively. It is important to distinguish between the name of a feature, f in the example, and the feature itself (represented for example by the body of a routine), which we call ϕ. By renaming the feature in D we associate with ϕ a new name f'; by redefining it we associate with f a new feature ϕ'.

When c is of dynamic type C, $c.f$ will always refer to feature f, and the notation $c.f'$ will always be illegal. Thus there are only three nontrivial cases: $c.f$ for c of dynamic type D; $d.f$; and $d.f'$. The table shows what actual feature is associated with each of these notations in each legal case. Note that combinations marked as illegal may be caught statically by a compiler.

Cases 5 and 6 are a little more subtle than the others and also less useful in common usage; they may be skipped on first reading.

All cases, with the exception of case 6, occur in the library of the Appendix. Note that case 4 is interesting in particular when D provides a special implementation ϕ' of the feature, but the implementation of ϕ' internally relies on the more general ϕ; thus D must be able to refer to ϕ, which is not available to it under any name in case 3 (redefinition only).

For example, the basic insertion procedure *put_between* is inherited by class *TREE* (A.7) from *LINKABLE* (A.4). To insert a new child into a tree, however, you must not only do the pointer operations for inserting an element into a list, but also set the "parent" field of the new child so that it references the correct parent. Thus, the implementation of the new *put_between* consists of a call to the original procedure, renamed

Table 1. Combining Redefinition and Renaming

No.		$c.f$	$d.f$	$d.f'$
1	f not redefined f not renamed	ϕ	ϕ	Illegal
2	f redefined ϕ' f not renamed	ϕ'	ϕ'	Illegal
3	f not redefined f renamed f'	ϕ	Illegal	ϕ
4	f redefined ϕ' f renamed f'	ϕ'	ϕ'	ϕ
5	f not redefined f renamed f' f' redefined ϕ''	ϕ''	Illegal	ϕ''
6	f redefined ϕ' f renamed f' f' redefined ϕ''	ϕ''	ϕ'	ϕ''

Note: In column 3, c is assumed to be of dynamic type D.

linkable_put_between for the occasion, followed by instructions to set the *parent* field.

4.10 Deferred Features—Eiffel as a Language for Analysis and Global Design

With redefinition, programmers can provide alternate implementations of a previously implemented feature. In some cases, you may want to define a feature without giving its implementation, passing on to the descendants the task of providing such implementations. Deferred feature declarations satisfy this need.

In such a declaration occurring in a class C, the type and arguments of the feature, if any, must be specified in C, but not its body if it is a routine. Syntactically, the **do**... part is simply replaced by the keyword **deferred**.

Various versions will be given for the body in the descendants of C. You may then apply the feature to an entity of type C (under some consistency conditions), with the understanding that the implementation used depends on the dynamic type of the entity, which will always be one of the descendants.

The syntax for deferred typed features without arguments, that is to say (in its simplest form)

```
f: T is deferred end
```

does not commit the descendants to implement the feature as an attribute rather than a function; different descendants may take different decisions in this respect.

A class that contain one or more deferred features is itself called a deferred class and must be declared as **deferred class** rather than just **class**.

As with feature redefinition, it is important to enable designers to specify properties of features even when they are declared as deferred. The techniques for specifying preconditions and postconditions of routines (6.3 and 6.6) indeed apply to deferred features.

An interesting application of deferred classes is the two-tier definition of modules (interface and implementation) as in Ada or Modula 2. You will declare an abstract data type implementation as two classes, the first of which contains deferred features (with their types, those of their arguments, as well as preconditions and postconditions), and the second, heir to the first, provides implementations. An important advantage of this technique over the method used in non-object-oriented languages such as Ada or Modula 2 is that more than one implementation may be provided for a given interface within the same system.

Deferred classes are important in connection with one of the uses for Eiffel: as a language and method for high-level **analysis** and **design** of software systems, as opposed to implementation only. The object-oriented approach is indeed particularly fruitful at these stages, where the results of classical functional methods often suffer from insufficient flexibility and reusability. Through the use of Eiffel, you may abstractly describe a system as a set of deferred classes. Note that even though implementations are not given, the routines' effects may be specified by preconditions and postconditions, and the abstract semantic properties of classes may be expressed by class invariants (see 6.2 below).

A deferred class describes not just one implementation of an abstract data type, but a set of implementations. In the extreme case where all features are deferred, the class is in fact close to a pure abstract data type specification.

A deferred class may not be instantiated (as the corresponding objects would not have implementations for some of their features), but it may be used as type of entities, to be associated at run-time with instances of descendant, nondeferred classes.

Furthermore, a deferred class is compilable, so that the Eiffel compiler may perform a number of verifications on it. To go from such a set of nonexecutable classes, viewed as a high- level system description, to an executable version, you write descendant classes, containing actual implementations of the previously deferred routines. This approach yields a much smoother development process than when a strict separation is maintained between the formalisms used at successive stages of the software development lifecycle.

5. TYPE COMPATIBILITY

5.1 Basic Constraints

Eiffel is a typed language that was designed to permit completely static (compile-time) type checking. Because of the inheritance mechanism, the type system is richer than in a language with a simpler type system. There are two basic constraints, governing assignment and feature redefinition (the discussion only addresses class types; the usual rules apply to simple types).

The first typing constraint is a direct consequence of the rule governing association between entities and objects (Section 4.5): in an assignment $x := y$, the type of y must be a descendant of the type of x (if these are class types). In other words, you may assign a "more specific" value (i.e., a value of a descendant type) to an element declared as "more general." For example, an element of type *LIST* may be assigned a value of type *TWO_WAY_LIST*. The reverse case is prohibited.

The second basic constraint applies to the redefinition of a typed feature, i.e., an attribute or a function: If such a feature, initially declared in a class C as being of a certain type T, is redefined in a descendant of C as being of another type T', then T' must be a descendant of T. For example, the feature representing the first linked element ("cell") of a list, called *first_element* and defined as *LINKABLE* in class *LINKED_LIST*, is redefined as *BI_LINKABLE* in *TWO_WAY_LIST* and as *TREE* in class *TREE*; such redefinitions are correct since each new type is a descendant of the previous one.

5.2 Declaration by Association

The second typing constraint is one of the language properties that motivate **declaration** by **association.** A declaration by association takes the form

x: **like** *y*

where y is an entity declared in the scope where this declaration appears. If T is the type associated with y, then the above declaration is equivalent to

x: *T*

with the difference that if y is redefined in a descendant of the current class with a new type T', then the corresponding redeclaration of x is implied. We say that y is an "anchor," which may be used to drag along other elements declared **like** y. The anchor itself must be declared with a "fixed" type (not by association).

This form of declaration is often needed to guarantee that a group of elements remain consistent with each other in any descendant. It is used in particular to ensure that the types of function results are properly declared, as the following simple example shows.

Assume you want to define a class *COMPLEX* to represent complex numbers. One of the features may be a function *conjugate* yielding the conjugate of the current instance, which you might declare as follows:

```
conjugate: COMPLEX is
                -- Return a copy of the conjugate of the current complex
    do
        Result.Clone (Current) ;    -- Assign to Result a copy of the current complex
        Result.change_y (-y) ;      -- Negate the y coordinate of Result
    end -- conjugate
```

It has been assumed that another feature of *COMPLEX* is the procedure *change_y* (*new_y*: *REAL*), which does what the name implies.

The solution shown is correct as along as you consider class *COMPLEX* just by itself. However, assume *COMPLEX* has a descendant—say *IMPEDANCE*, in an electrical engineering application whereby impedances are considered a special case of complex numbers. Class *IMPEDANCE* will inherit the *conjugate* feature; but with declarations such as

i1, i2: IMPEDANCE

the assignment *i1 := i2.conjugate* is typewise incorrect, since the type of the right-hand side, *COMPLEX*, is not a descendant of the type of the left-hand side, *IMPEDANCE*; in fact, the reverse holds.

The problem goes away, however, if you use a declaration by association whose anchor will be the current element itself. In other words, you will declare *conjugate* to be of type not *COMPLEX* but

like *Current*

With this declaration, *c.conjugate* is of type *COMPLEX* if *c* is declared of type *COMPLEX*, but *i1.conjugate* will now have the type of *i1*, namely *IMPEDANCE*. In all cases these types may be determined statically.

Declarations by association play an important role in the examples below. They ensure, among other properties, that list elements are consistent: for example, all elements of a doubly linked list (see class *BI_LINKABLE*, Section A.5) must include two references, to their right and left neighbors; and all members of the list of children of a tree node must themselves be tree nodes (A.8).

It is essential to emphasize that, whether or not declarations by association are used, the typing constraints are static and may be checked at compile time.

6. FEATURES FOR SYSTEMATIC PROGRAMMING

Much of the emphasis in the design of Eiffel has been on promoting such quality factors as reusability, extendibility, and compatibility. But these qualities are meaningless unless programs are also correct and robust. In fact, as techniques for the production of truly reusable software components become a reality, the concern for correctness takes on a even greater importance than in a "one-shot developments" environment, since the impact of errors will be multiplied by the reuse factor.

Eiffel includes language constructs that promote a systematic approach to software construction. The regular use of these constructs, and the general attitude they imply towards program construction, have proved extremely beneficial as to the correctness and robustness of software built with Eiffel.

6.1 Assertions

The Eiffel constructs aimed at enhancing a lucid approach to software correctness are based on the notion of **assertion**.

Syntactically, an assertion is a boolean expression, expressing some property that should be satisfied by certain entities at designated stages during a the execution of a system. Examples of assertions are:

```
i /= j            -- Note that /= is the "not equal" symbol
f(x, y) = 0
notempty: nb_elts > 0
```

As the last example shows, an assertion may have an associated label. An assertion may have more than one clause, separated by semicolons; the semicolon is semantically equivalent to an **and** here, but it allows individual identification of the components of the list, especially if they are labelled.

Eiffel does not include a full-fledged assertion language, so some properties that are not expressible as simple boolean expressions may have to be given in part as comments, as is frequently the case in the examples of the appendix. (A effort in progress, the M specification method [21], includes a specification language, LM, which might be used in conjunction with Eiffel in a fully formal approach.)

The various uses of assertions will now be described.

6.2 Class Invariants and the *Create* Procedure

The need for class invariants arises from the already voice remark that a class is in general an implementation of an abstract data type rather than the abstract data type itself (except in the case of a class with deferred features only). The implementation contains components (attributes) that are often too general for the purpose of representing the abstract type. As a trivial example, an array representation of stacks might contain an integer attributes, say *high*, which marks the topmost array position used. Although an arbitrary integer may be positive, negative or zero, an integer used as stack pointer may only be nonnegative. Thus the condition $high \geq 0$ should be a class invariant.

The notion of data type invariant is discussed in [11] and [16].

A class invariant must be satisfied after the execution of the *Create* procedure of the class; any routine of the class may be written under the assumption that the invariant is satisfied on entry, and must ensure that it is still satisfied upon exit.

For nontrivial classes invariants are strong semantic properties; by stating them explicitly, you gain in-depth insights into the fundamental properties of classes. The appendix contains significant examples of class invariants, for example the invariants for *LIST* and *LINKED_LIST*.

Syntactically, a class invariant is an assertion list, appearing in an optional clause introduced by the keyword **invariant** in a class declaration, as in

```
class ARRAY_STACK [T] export... feature
        high: INTEGER;
        ................
invariant
        high > = 0
end -- class ARRAY_STACK
```

The reader will notice in the examples of the Appendix the constant interplay between class invariants and routine preconditions and postconditions. In principle, the following should be proved for each routine body B, with precondition Q and postcondition R in a class with I as invariant:

$$\{I \wedge Q\}\ B\ \{R \wedge I\}$$

where $\{Q\}\ A\ \{R\}$ means that execution of A, starting in a state where Q is satisfied, will terminate in a state where R is satisfied). In other words, when assessing the validity of a routine body, you may assume the class invariant, and you must check that it is preserved by the routine.

The notion of class invariant is the main justification for the way object creation is handled in Eiffel through the *Create* procedure.

The conventions regarding this procedure are slightly different from those of other routines. Execution of *a.Create* (....), where *a* is of type *A*, triggers the allocation of storage for an object of type *A*, to be associated with *a*, followed by the execution of the *Create* procedure declared in class *A* if there is one (which must be the case if the call includes arguments). If *A* does not contain a *Create* procedure, *A* is still considered to have declared it with an empty body. Thus *Create* is never inherited, since every class redefines it explicitly or implicitly.

Special conventions are always disturbing and one may wonder why Eiffel does not separate object allocation from object initialization, with a syntax such as

```
        -- Warning: this is not correct Eiffel!
allocate a;
a.init (x, y, ...)
```

where **allocate** would be a universal allocation instruction and *init* some class-specific procedure (declared in *A* in the case at hand).

The advantage of the solution actually retained is that by tying initialization to allocation the designer of a class may guarantee that all objects of the class will automatically satisfy the class invariant upon creation. The alternative solution would not enable designers to prohibit clients from omitting to call *a.init* after **allocate** *a* before any other feature is applied to *a*.

From a formal viewpoint, then, the purpose of *Create* procedures is to ensure that every object of a class initially satisfies the class invariant.

6.3 Preconditions and Postconditions

Assertion lists may be associated with routines: a routine may begin with a **require** clause, stating the conditions assumed to be satisfied on entry, and end with an **ensure** clause, stating the conditions that must be enforced by the routine implementation upon exit.

The following two notations are available in **ensure** clauses: **old** *x* denotes the value of entity *x* upon routine entry; *Nochange* is a boolean expression, true if and only if no attribute of the current instance has been modified since entry.

The precondition and postcondition of a routine may be viewed as an explicit contract between the class implementer and the authors of client classes. The precondition binds the clients: a call that does not satisfy it is not valid, and the class may do what it pleases with it. The postcondition binds the class: If the precondition is satisfied, the client is entitled to expect that the routine will terminate in a state that satisfies the postcondition. An approach to software construction based on this notion of contract is developed [25].

6.4 Loop Notation

The syntax of loops includes room for loop initialization, a loop invariant (true after initialization and conserved by the loop body), and a variant (a nonnegative integer expression that decreases on each iteration, guaranteeing termination):

```
from initialization_instructions
invariant invariant
variant variant
until exit_condition
loop loop_instructions end
```

This notation (where the **invariant**... and **variant** clauses are optional) enables the program reader to check that the *initialization_instructions* ensure the *invariant,* and that the combination of this invariant and the *exit_condition* ensures the desired effect of the loop. Note that this loop is similar to a classical "while" loop, with the test reversed; it is not a **repeat...until...** since the number of iterations will be zero if the *exit_condition* is false on entry.

6.5 Check Instruction

Assertions may also be used in a special instruction of the form

```
check assertion_list end
```

whose purpose is to express that the *assertion_list* is satisfied whenever control reaches this instruction. This construct (the equivalent of the Algol W *ASSERT* instruction) is used in particular in connection with routine calls, express that a condition stronger than or equal to the routine precondition is satisfied before the call, and that a condition weaker than or equal to the postcondition may be assumed upon return. The Appendix contains numerous examples of such uses of **check**.

6.6 Assertions and Inheritance

You may use assertions to state the restrictions that apply whenever features are added or redefined in descendants of a class. As pointed out in Sections 4.8 and 4.10, class designers should have some way of providing their clients with guarantees that each class will perform according to the original contract, even if some of its features are redefined.

Such a provision is the indispensable complement to the principle of openness: Inasmuch as you strive to produce software elements that are still open to extensions and modifications, you also need a way to prescribe limits within which these future changes should remain.

The following constraints apply to the inheritance mechanism in connection with the use of assertions:

- The invariant of a class applies to all descendants of a class (thus it does not need to be repeated in their **invariant**... clauses except for clarity).
- Consequently, no two classes may be combined through multiple inheritance if their invariants are not compatible.
- If a routine is redefined in a descendant class (this includes the case when the original routine was deferred), the new precondition must be no stronger and the new postcondition must be no weaker.

In the last rule, a condition is said to be stronger than another one if it implies it. The rule expresses the requirement that whenever the original routine was applicable, the new one must also be (but it may well be less restrictive in its precondition), and it must at least ensure the original postcondition (but it may well ensure a more restrictive one).

These consistency constraints are essential for a proper use of inheritance and redefinition. They express in particular that redefinition is not arbitrary, but must instead be viewed as a semantics-preserving transformation. Further details are given in [27] and [25].

Note that these constraints could only be enforced by a system that includes a fully formal assertion language and a theorem prover. We will have to satisfy ourselves, for some time to come, with informal human verification and run-time checking.

In particular, the examples reproduced in the Appendix have been tested extensively but not formally verified and some mistakes may remain; I will be grateful to any reader reporting an error.

6.7 Use of Assertions

The primary aim of assertions is to encourage a systematic way of **writing** Eiffel classes and to help **reading** them by requiring programmers to say explicitly what mental assumptions have been made. Assertions may thus be viewed as comments of a special kind. This possibility has been used abundantly in the examples.

It is also possible, on option, to check at run-time that assertions (at least those defined formally) are satisfied. The Eiffel environment provides three compilation options for each class:

(0) • No protection: The program text is assumed to be correct and assertions have no influence at run-time. Errors are likely to result (if apparent at all) in aberrant behavior and abnormal termination (arising for example from out-of-bounds memory references).

(1) • Controlled mode: Only preconditions of routines (**require** clauses) are checked.

(2) • Total protection: All assertions (and the effective decrease of loop variants through each iteration) are checked.

Option 2 is adequate at checkout time. Option 1 is an acceptable compromise in many situations; satisfaction of the precondition is essential to the proper functioning of routines (in fact, the presence of the **require** clause allows a much simpler coding style in Eiffel than in common languages, since internal consistency checks may be factored out in routine preconditions rather than scattered throughout routine texts), yet preconditions often may be checked with reasonable efficiency. Thus, option 1 is the default.

6.8 Exceptions

In its original form, Eiffel did not have any exception handling mechanism. In particular, violation of an assertion (monitored as described above) would produce a message and halt the execution. The original version of this article reflected this decision.

This policy was based on an analysis of the limitations and dangers of exceptions as offered by such languages as CLU and Ada. Ada exceptions, in particular, are undisciplined interprocedural **goto** instructions. They encourage an irresponsible, "buck-passing" approach to the treatment of abnormal cases.

Recent research at Interactive Software Engineering has led to the design of a simple and safe exception mechanism which is currently (spring 1988) being integrated with the rest of the implementation. The following is a brief overview of this mechanism, described further in [27] and [25].

The Eiffel exception mechanism is based on the notion of "programming by contract," mentioned above. An exception is any event that prevents a routine from fulfilling its contract. This includes assertion violations when assertions are monitored, but also externally triggered events such as arithmetic overflow, memory exhaustion, or user interrupts.

When an exception occurs, only two responses make sense:

- Resumption: Attempt to fix the reason for the exception and retry the routine execution.
- "Organized panic": Concede failure, put all concerned objects back into a state satisfying the invariant, and signal the failure to the routine's caller by triggering a new exception (which the caller will have to handle in one of the same two ways).

The policy made possible by Ada of performing some instructions and returning to the caller without signaling that something wrong has happened is dangerous and must be banned.

To handle exceptions, an Eiffel routine may have a **rescue** clause that will be triggered whenever an exception occurs during the execution of the routine. The aim of the rescue clause is to bring the object back to a stable state. Unless the clause terminates by executing a **retry** instruction, the routine as a whole will be considered to have failed, and an exception will be triggered in the calling routine. (If there is no caller, that is to say at the root level, the system execution as a whole terminates with an appropriate message). The rescue clause may, however, terminate with a **retry**, in which case the routine execution is attempted again from the beginning.

A routine without a rescue clause is considered to have an empty one, so that any exception will make it fail and signal an exception to the caller.

The language extension for exceptions is limited to the mechanism just described, and to the two keywords **rescue** and **retry**. In addition, a library class *EXCEPTIONS* defines some useful features for dealing with exceptions, in particular the attribute *exception*, which gives the code of the last exception that has occurred (to enable treating various exceptions differently). Note that a programmer who wishes to explicitly trigger an exception does not need a special **raise** instruction; a routine *raise,* with precondition **false**, will do the job.

As an example of the exception mechanism, consider a routine *attempt_transmission* that transmits a message over a phone line. It is assumed that the actual transmission is performed by a routine *transmit*; once started, however, *transmit* may abruptly fail if the line is disconnected, and will then trigger an exception.

Routine *attempt_transmission* tries the transmission at most five times; before returning to its caller, it sets a boolean attribute *transmission_successful* to **true** or **false** depending on the outcome. Here is the text of the routine:

```
attempt_transmission (message: STRING) is
            -- Attempt transmission of message at most five times.
            -- Set transmission_successful accordingly.
    local
            failures: INTEGER
    do
            if failures < 5 then
                    transmit (message);
                    transmission_successful := true
            else
                    transmission_successful := false
            end
```

```
        rescue
                failures := failures + 1;
                retry
        end; -- attempt_transmission
```

Note that the integer local variable *failures* is initialized to zero on routine entry.

This example shows one of the key reasons for the simplicity of the mechanism: The rescue clause never attempts to achieve the original intent of the routine; this is the sole responsibility of the normal body (the **do** clause). Its only role is to "patch things up" and either fail or retry.

The above version never fails; it signals its inability to perform the transmission by setting an attribute. The following slightly simpler version will fail if it is unable to perform the transmission, triggering an exception in the caller, which is then charged with the responsibility of handling it in its own **rescue** clause:

```
attempt_transmission (message: STRING) is
                -- Attempt transmission of message at most five times.
                -- If impossible, signal failure by raising an exception.
        local
                failures: INTEGER
        do
                transmit (message);
        rescue
                failures := failures + 1;
                if failures < 5 then
                        retry
                end
                        -- If control reaches this point, the routine will fail.
        end; -- attempt_transmission
```

7 OTHER CONSTRUCTS

Two more language notions are needed to understand the details of the examples in the Appendix and to write software in Eiffel.

Noncommutative boolean operators use the Ada syntax: *a* **and then** *b* has value false if *a* has value false, and otherwise has the value of *b*; *a* **or else** *b* has value true if *a* has value true, and otherwise has the value of *b*. The advantage of these operators over the standard **and** and **or** (which are of course also present) is that they may be defined when the first operand gives enough information to determine the result (false for **and**, true for **or**), but the second is undefined. A simple example is the boolean expression

```
i /= 0 and then j/i = k
```

which might yield an undefined value if it used a simple **and**. The noncommutative operators are particularly useful in assertions.

Finally, **constants** are described as class attributes with fixed values. The syntax is similar to that used for routines, for example:

```
pi: REAL is 3.1415926524
```

It is common practice to encapsulate a group of related constants in a class, which is then used as ancestor by all classes needing these constants. In the Eiffel implementation, constant attributes do not occupy any space at run-time, so programmers need not be concerned about the number of such attributes.

The above notation applies to constants whose types are simple. Constants of class types are expressed as "once" functions, i.e., functions that are evaluated only once in a given system; subsequent calls will always return the same value. "Once" functions are distinguished by the keyword **once** appearing instead of **do**. For example, a class *COMPLEX* might include a declaration of the constant complex *i* (real part 0, imaginary part 1) as

```
i: COMPLEX is
                -- Pure imaginary number of modulus 1
        once
                Result.Create (0, 1)
        end -- i
```

assuming the proper *Create* procedure. "Once" procedures, as well as functions, are also permitted; any call to such a procedure beyond the first has no effect. (An example might be an *open_input* procedure, which every client might call to make sure the input has been opened; however the open operation must be executed only once during a given system's execution.)

8 IMPLEMENTATION: THE EIFFEL PROGRAMMING ENVIRONMENT

For the programmer, a programming language is no better than its implementation. We thus finish this introduction to Eiffel with a description of how the language has been implemented. Rather than just an implementation, it is appropriate to describe the set of Eiffel-related facilities as a programming environment.

8.1 Classes and Systems

There is no exact notion of "program" in Eiffel. What may be executed is a "system," defined by the name of a class, called the root, and a list of actual arguments. Executing such a system consists of allocating an object of the root class and executing its *Create* procedure, with the arguments supplied. Usually this will trigger new routine calls and the creation of other objects.

In keeping with the goals of reusability and extendibility, the primary focus of Eiffel programming is on classes rather than systems. An Eiffel class is the implementation of a useful data abstraction, but good classes should not be tied to a specific system; rather, systems should be constructed by combining existing classes and, if necessary, complementing them with new ones, again designed with generality and reusability in mind.

This concept is reflected in the implementation: Nothing binds a class to a particular system. The concept of system does not in fact belong to the language proper, but rather to the operating system level.

8.2 Implementation Policy

The current Eiffel implementation, running under the Unix system, uses C as an intermediate language. This technique enhances portability without sacrificing efficiency. C is a portable assembly language, the closest-ever realization of the old "Uncol" (Universal COmputer Language) idea.

It should be pointed out that the use of C as intermediate language is just one possible implementation technique; nothing in the design of Eiffel ties it to C.

8.3 Compilation and Assembly

Two commands are provided.

The first command, *ec*, for Eiffel Class, compiles a single class into C and then to object code. Separate compilation is of course an essential requirement for a language promoting reusability and extendibility. To compile a class, you need its ancestors, if it has any; an optional argument to *ec* lists the directories where they are to be found.

The second command, *es*, for Eiffel System, constructs a complete system from its constituent classes through a process called **assembly** and executes the result. This command refers to a System Description File of the following form:

```
ROOT: Classname
SOURCE: ... list of directories ...
EXTERNAL: ... list of files ...
NO_ASSERTION_CHECKING: ... list of classes ...
PRECONDITIONS: ... list of classes ...
ALL_ASSERTIONS: ... list of classes ...
DEBUG: ... list of classes ...
TRACE: ... list of classes ...
PAGING (Y|N)
GARBAGE_COLLECTION (Y|N)
```

Such a file describes how to assemble a system whose root is an object of type *Classname*. The *SOURCE* directories are used to locate all the necessary classes; the *EXTERNAL* files contain any needed external routines (see below).

The following lines give compilation options: list of classes to be compiled with various levels of run-time assertion checking (see Section 6.7); classes to be compiled in debug mode; classes to be traced. The keyword *ALL* may appear in lieu of a list of classes.

The last two lines of the System Description File allow selection or deselection of the built-in virtual memory and garbage collection facilities (see 8.8 below).

The format of the System Description File is generated by the first call to *es* in a given directory, so that programmers do not need to remember the details of the above syntax.

8.4 External Routines and Openness

A programming environment emphasizing extendibility and reusability should lend itself to communication with the outside world. Eiffel was specifically designed as an open environment, capable of interfacing with other languages. In fact, this requirement has made the language simpler, by allowing us to rely on external facilities in areas where we had no specific contributions to make, like physical implementation of input and output facilities (although the *packaging* of such facilities, by means of basic libraries of classes, using inheritance and information hiding, falls definitely within the province of Eiffel).

Thus routines of a class may rely on external primitives written in a language other than Eiffel. More precisely, an Eiffel routine may contain an **external**... clause listing primitives written in other languages, which may then be used within the routine's body. Examples of use of external primitives may be found in class *ARRAY* (Section A.2).

The design of this facility does not conflict with the other principles of the language. In particular, an external routine is not a class feature: Instead, it is local to an Eiffel routine that uses it for its implementation only. Thus the facilities offered by non-Eiffel primitives may be made available for use in Eiffel systems, but only once they have been encapsulated in bona fide Eiffel routines, used through the standard conventions of the language. Eiffel techniques such as preconditions and postconditions may then be applied to them.

8.5 C Package Generation

An aspect of the environment that has proved useful to many developers using Eiffel is the availability of a package generator. By using further options in the System Description File, a developer may produce a complete C package from an Eiffel system. The package contains the following elements:

- A set of C functions generated from their Eiffel counterparts (routines). (In case of name clashes, which may occur because in Eiffel, routines belonging to different classes may have the same names, the package generator chooses default names for the duplicates; the programmer may, however, specify any desired name for any generated C function.)
- A main program, generated automatically.
- A copy of the run-time system (including the garbage collector), in C form.
- An automatically generated **Make** file, allowing recompilation of the generated package in any environment.

The resulting C package is thus entirely self-contained and independent of any Eiffel environment. This makes Eiffel a powerful cross-development tool, useful for software developers whose customers have not (yet) access to an Eiffel compiler.

8.6 Efficiency

As I mentioned in Section 1.2, we were particularly concerned about efficiency of the generated code. This concern is reflected in the translation techniques used:

- As regards space for objects, each object only carries its attributes and some control information; no space is ever reserved for routines in the representation of an object (routines are associated with a class as a whole, not with individual objects of a class). As I mentioned in Section 7, constant attributes are also "free" in terms of run-time space. Thus the occupancy of an object is little more than that of an equivalent record in Pascal (without the loss that comes from reserving the largest possible space in a record with variants). This is the only acceptable solution; it means in particular that efficiency is not a serious reason for restricting the number of routines or symbolic constants in a class, or the number of parents to a class.
- As regards space for classes, the code of routines inherited directly or indirectly from ancestors is not copied, but shared. This applies to multiple as well as single inheritance: Thus, there is no need to worry about inheriting from many different worlds (networks of existing classes) when a new class is started, as the overhead per inherited routine is negligible. Neither does genericity imply any code replication; the routines are shared between all generic instances. These results should be contrasted with the Smalltalk implementation of multiple inheritance, which (if [5] is to be believed) duplicates routines on inheritance paths other than the principal path, and with the implementation of genericity in most Ada environments, which replicate code for each generic instance. As seen above, there is one exception to the "no duplication" rule in the case of repeated inheritance with renaming (Section 4.7); however this is a special and rather rare occurrence.
- As regards time, one of the serious pitfalls of object-oriented programming is the potential inefficiency of remote routine application: Since calls of the form $x.f(a, b, \ldots)$ may result in the execution of various versions of f depending on the run-time value of x, there is a danger of wasting time in looking for the appropriate version to apply. Published descriptions of object-oriented language implementations seem to consider it inevitable that the deeper the inheritance hierarchy, the longer routine search may belong at run-time. Although improvements are possible by the use of "caching" techniques [8], this is unacceptable: Programmers should not be forced to to make tradeoffs between efficiency and the qualities that are direct beneficiaries of inheritance—reusability and extendibility. Furthermore, any method based on run-time search becomes all but unapplicable with multiple inheritance, since a whole acyclic graph of ancestors would have to be searched rather than just a linear list. (The duplication of code in the Smalltalk case is precisely aimed at keeping the search linear.)

This problem has been solved in the Eiffel implementation through the use of original data structures and algorithms that ensure **constant-time** routine search. Although the overhead of a call $x.f(a, b, \ldots)$ is slightly higher than the overhead for a procedure call $f(x, a, b,$

...) in a standard programming language such as C or Pascal, it is bounded by a constant value. This, I believe, is one of the fundamental contributions of the Eiffel implementation.

Beyond the systematic application of the above techniques, a postprocessor (integrated with the package generator) performs a number of important optimizations, both in time and space. In particular:

- As I noted above, the overhead for routine calls is constant; it is also reasonably small, typically amounting to about 30% more than the overhead for function calls in C, but even this may be to much in highly time-sensitive applications. Conceptually, this overhead is a consequence of the availability of dynamic binding; this means it is only justified for routines that are redefined at least once in a system. Usually, however, a large percentage of the routines of a large system are never redefined. The postprocessor will detect them and implement all calls to such routines in the same way they would be implemented in C, removing any unjustified overhead.
- A problem that plagues many object-oriented language implementations is the useless loading of the code for all routines from ancestor classes, including routines that are never actually used. Even in today's relatively memory-rich environments, this may become a serious obstacle to the free use of inheritance. (Advances in hardware technology should never be used as an excuse for poor software performance; extra MIPS and bytes are bought to be used, not wasted.). This problem again introduces the risk that programmers will have to choose between reusability/extendibility and efficiency. The postprocessor solves the problem by removing any unneeded code, enabling programmers to use arbitrary inheritance depths without having to worry about the effect of unused routines on code size.

With these and other optimizations, we feel that the Eiffel implementation techniques have achieved our initial goal of providing the full power of object-oriented programming within the efficiency constraints of software production in ordinary programming environments.

8.7 Type Checking

Static typing was mentioned in Section 1.2 as an important concern. Eiffel is indeed a statically typed language. The language definition permits all checking to be done at compile-time; no checks are necessary at run-time. Thus if a system containing a feature application of the form *a.f* is accepted by the compiler, then *f* is guaranteed to be applicable to all objects to which *a* may refer at run-time.

This is to be contrasted with the solution taken by most object-oriented languages, in which such checks are deferred in whole or in part to run-time.

8.8 Configuration Management

The power of the reusability techniques offered by Eiffel and the emphasis on bottom-up system construction by combination of separately developed software components (classes) make it necessary to use a systematic approach to change and configuration control.

A class may depend directly or indirectly on many others. There are two direct dependency relations: A class may be the *client* or the *heir* of another. The interconnection network resulting from considering indirect dependencies as well may be quite complex. A class may depend on many others; the inheritance graph must be acyclic, but the client graph may be cyclic. Furthermore, a class may be a client of one of its ancestors or descendants.

In a development environment where classes are frequently updated, there is a serious danger of inconsistencies arising from the use of obsolete or inadequate versions. A technique that would avoid this risk would be to recompile everything every time; however, such a solution is clearly unacceptable from an efficiency viewpoint.

An automatic configuration management system has thus been integrated into the commands *ec* and *es*. Whenever a class is compiled, the system ensures that the classes on which it depends are up-to-date, triggering the necessary recompilations.

Our initial implementation of these facilities relied on the Unix Make tool [9]. However, Make soon turned out to be too limited in its capabilities. In particular, Make will not support cyclic dependencies. Even if this problem were solved, however, a major liability of Make is the necessity for the programmer to manually describe the dependencies; this is a tedious process and also a dangerous approach, since there is always the risk that a dependency will be forgotten, causing Make to generate an inconsistent or incomplete version of a system. This is not acceptable; dependency analysis should be completely automatic.

The algorithms used by **ec** and **es** will indeed perform this analysis automatically for Eiffel, freeing the programmer from any need to worry about what classes need to be recompiled after a series of changes to a system. These algorithms look for a minimum set of classes to recompile. In particular, they will detect that changes to a class only affected its secret (nonexported) features, and hence that its clients need not be recompiled.

A system that uses these facilities, even a large one, can usually be brought back to a consistent state in just a few minutes after a number of changes have been made to it.

8.9 Run-Time Support

The dynamic model described in Section 3.1 implies adequate run-time support. The implementation relies on a complete memory management system (*Dynamem*), which provides both paging and garbage collection.

Both facilities are optional, being selected at compilation time through entries in the System Description File. Both are by default disabled.

Paging should usually not be selected on standard operating systems providing their own virtual memory management.

In contrast with traditional garbage collectors that are triggered when no memory is left and then stop all execution for an often long time, the Eiffel collector is a continuous process (implemented as a co-routine) which collects unused space as the application is being executed. It uses a self-adapting mechanism that will wake up at periodic intervals, the intervals being automatically increased if memory usage is low and decreased otherwise. A form of "generation scavenging" [33] is also used by the algorithm.

When selected at compilation-time, garbage collection may be dynamically disabled and then reenabled. A collector cycle may also be triggered at times when the programmer knows that CPU time is available, for example while awaiting user input.

Garbage collection may be replaced or supplemented by programmer-controlled management; the language indeed makes it possible to implement this safely, without the well known dangers of the Pascal *dispose*, by associating a specific policy with each class of known behavior and implementing it within the language itself.

8.10 Debugging Aids

It is important to provide programmers with proper debugging tools. Although C is used as an intermediate language for Eiffel compilation, Eiffel programmers do not need to ever look at intermediate C code. (In fact they do not need to know C.) Thus, built-in C debuggers are of no use here. The following facilities are provided:

- Optional run-time checking of assertions (see the options to *es* in Section 8.3), which has proved an invaluable aid for finding logical errors if an effort is made to spell out correctness arguments as assertions.
- A class and routine tracing facility (see line *TRACE* ... in the System Description File of Section 8.3).
- A **debug** ... **end** instruction, executed only if the class has been compiled in *DEBUG* mode;
- An interactive **object viewer**, which makes it possible to traverse interactively a system's data structure at run-time, observing the objects and following the reference chains.

More debugging facilities are currently being added to this basic framework.

8.11 Short—The Class Abstracter

Another tool is important to make reuse practical: *short*, a class abstracter that produces a summarized version of any class, enabling potential users to determine whether the class provides the required capabilities, without having to look at the whole implementation of the class. The summarized version contains the **inherit** and **feature** clauses only; the latter is abstracted so that only exported features are shown and, for each exported routine, the body is not shown: Only the header, precondition, postcondition and the comment immediately following the header, if any, are reproduced.

The presence of assertions is fundamental in making this approach work for Eiffel; well-chosen preconditions and postconditions go a long way towards documenting the purpose of a routine both precisely and concisely.

For example, the abstracted version of class *ARRAY* (Section A.2) is:

```
class interface ARRAY [T] exported features
    lower, upper, size, entry, enter
inherit
    INDIRECT [T]
feature specification
    lower, upper: INTEGER
    size: INTEGER
                -- Array size
    Create (min: INTEGER, max: INTEGER)
                -- If min <= max, create array with bounds min and max;
                -- otherwise create empty array.

    entry(i: INTEGER): T
                -- Entry of index i
```

```
            require
                  lower <= i;
                  i <= upper
      enter(i: INTEGER, element: T)
                  -- Assign value element to i-th entry
            require
                  lower <= i;
                  i <= upper
invariant
      size = upper - lower + 1;
      size >= 0;
end interface -- class ARRAY
```

The **-t** option of *short* was used to produce this example; this allows the use of *short* for generating a client's manual for a class.

Note that *short* will recognize a header comment at a conventional place in a routine (after the keyword **is**) and will keep it in the output. Other comments are lost. Assertion clauses involving only public features are kept, but not those involving one or more nonexported features.

The official documentation for the Eiffel library [14] is almost entirely produced by *short*.

I believe that **short** points towards the proper solution of the documentation problem in software engineering. Most textbooks urge programmers to write extensive documentation that is, viewed as separate from the software itself. But it is hard to make sure that this advice is followed when the software is initially designed, and this is almost impossible when it comes to modifications and enhancements.

The only satisfactory approach is to make the software contain its own documentation and to rely on computer tools to extract the documentation when it is needed. In this approach, there is no clearcut boundary between implementation and documentation; the documentation for a software component is simply a more or less abstract view of the component. Various levels of abstraction are possible, from the most abstract (which would only include the component's name) to the most concrete (which is simply the full text of the component). Command **short** is in-between, yielding a class interface with the formal properties of the operations and associated comments, but no implementation details.

8.12 Graphical Tools

Beyond **short**, there remains a need for high-level documentation of interclass relationships and system structure. In this case, textual documentation must be complemented by graphical output. Graphical tools (to be made part of official releases of Eiffel in the spring of 1988, on environments supporting the X Windows graphics package) make it possible to explore the class structure in a visual form. They generalize the Smalltalk notion of a class browser.

8.13 Flattening

It was previously noted (in the *WINDOW* example, section) that the inheritance structure used for the implementation of a particular class is usually of no interest to the clients of that class; what is relevant for the clients is the complete interface specification. Thus, there is a need for a tool that will produce a functionally equivalent class with no inheritance clause.

The **flat** command is such a tool. It produces a "flattened" version of a class containing the actual text of all inherited routines; the command takes into account any renaming and redefinition that may have occurred between the original declaration of a routine and the current class. By applying **short** to the output of **flat**, one obtains the same level of documentation for all routines of a class, those that are declared in the class itself as well as those which are inherited from ancestors.

9 CONCLUSION

9.1 Further Work

Several efforts are being pursued in connection with the work described in this article:

- The language and its translator are being applied to the development of several large software systems.
- The implementation is being refined and extended. Implementations are in progress for systems other than Unix.
- The Eiffel library sketched in this article is being expanded, so as to eventually cover all the data structures and algorithms that constitute the core of programming.
- Work on new specific Eiffel tools continues; of

particular importance is the development of databases for storing and retrieving reusable software components.

• An extension of Eiffel for handling concurrency and real time is being investigated.

• Work also continues on the M formal specification method [21], applying similar ideas at a more abstract level.

9.2 Main Contributions

I believe that the main contribution of the Eiffel language and environment is to provide a consistent combination of a range of features that, to my knowledge, had never before been offered within a single language: object-oriented program modules based on data abstraction; multiple and repeated inheritance; polymorphism and dynamic binding; genericity; information hiding; fully static typing; systematic use of assertions and invariants; separate compilation; built-in automatic configuration management; automatic documentation tools enabling the documentation to be treated as part of the software itself; dynamic allocation of objects with automatic incremental garbage collection; production of code that is efficient in both time and space; portability, obtained through the use of a widely available target language, C, but without compromising the simplicity and elegance of object-oriented concepts in the source language; support for cross-development by the generation of stand-alone C packages; and, more generally, an overall concern to make the great potential of object-oriented programming available to practicing programmers in production environments.

Eiffel is a language and environment for professional programmers: people who have come to appreciate the difficulties of software design as well as the virtues of reusability, modularity, data abstraction, and assertion-guided programming; people who know that an appropriate design and programming language, supported by appropriate tools, is a key ingredient in meeting these challenging goals.

ACKNOWLEDGMENTS

This paper and the language design benefited from comments from several colleagues at Interactive Software Engineering, particularly Jean-Marc Nerson and Reynald Bouy—the first Eiffel programmers. I have a special debt to the first implementors of Eiffel: Deniz Yuksel, who started the implementation; Oliver Mallet, Frédéric Lalanne, Hervé Templereau, and Pascal Boosz, who took it to completion. I am also grateful to Jean-Claude Derniame, Jéan-Pierre Finance, Harlan D. Mills, and Jean Mendelsohn for important comments.

REFERENCES

1. ANSI and AJPO, *Military Standard: Ada Programming Language (American National Standards Institute and US Government Department of Defense, Ada Joint Program Office)*, ANSI/MIL-STD-1815A- 1983, February 17, 1983.
2. D. G. Bobrow and M. J. Stefik, *LOOPS: an Object-Oriented Programming System for Interlisp,* Xerox PARC, 1982.
3. G. Booch, *Software Engineering with Ada,* Benjamin/Cummings Publishing Co., Menlo Park, Calif., 1983.
4. G. Booch, Object-Oriented Development, *IEEE Trans. Software Engineering* 12, 211-221 (1986).
5. A. H. Borning and D. H. H. Ingalls, Multiple Inheritance in Smalltalk-80, in *Proceedings of AAAI-82,* 1982, pp. 234-237.
6. R. J. Brachman, What IS-A and isn't: An analysis of taxonomic links in semantic networks, *Computer (IEEE),* 16, 67-73 (1983).
7. H. I. Cannon, *Flavors,* Technical Report, MIT Artificial Intelligence Laboratory, Cambridge, Mass., 1980.
8. B. J. Cox, *Object-Oriented Programming: An Evolutionary Approach,* Addison-Wesley, Reading, Mass., 1986.
9. S. I. Feldman, Make—a program for maintaining computer programs, *Software Practice and Experience* 9, 255-265 (1979).
10. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, Reading, Mass., 1983.
11. C. A. R. Hoare, Proof of Correctness of Data Representations, *Acta Informatica* 1, 271-281 (1972).
12. C. A. R. Hoare, Hints on Programming Language Design, in *ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages,* October 1973.
13. Jean-Marie Hullot, *Ceyx, Version 15: I—une Initiation,* Rapport Technique no. 44, INRIA, Rocquencourt, Eté 1984.
14. Interactive Software Engineering, Inc., *Eiffel Library Manual,* Technical Report TR-EI-7/LI, Goleta, Calif., 1986.
15. Interactive Software Engineering, Inc., *Eiffel User's Manual,* Technical Report TR-EI-5/UM, Goleta, Calif., 1986.
16. C. B. Jones, *Systematic Software Development Using VDM,* Prentice-Hall, Englewood Cliffs, N.J., 1986.
17. B. P. Lientz and E. B. Swanson, Software Maintenance: A User/Management Tug of War, *Data Management,* April 1979, 26-30.
18. B. H. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder, *CLU Reference Manual,* Springer-Verlag, New York, 1981.
19. B. Meyer, *Applied Programming Methodology,* course notes, University of California, Santa Barbara, to appear as a book.
20. B. Meyer, Quelques concepts importants des langages de programmation modernes et leur expression en Simula 67,

Bulletin de la Direction des Etudes et Recherches d'Electricité de France, Série C (Informatique) 1, 89-150, (1979) Clamart, France. Also in GROPLAN 9, AFCET, 1979.

21. B. Meyer, *M: A System Description Method,* Technical Report TRCS85-15, University of California, Santa Barbara, Computer Science Department, August 1986.
22. Bertrand Meyer, Reusability: the Case for Object-Oriented Design, *IEEE Software* 4, 50-64 (1987).
23. B. Meyer, Eiffel: Programming for Reusability and Extendibility, *ACM Sigplan Notices,* 22, 85-94 (1987a).
24. B. Meyer, Cépage: Towards Computer-Aided Design of Software, *Journal of Systems and Software,* 1988, in press.
25. B. Meyer, Programming as Contracting, submitted for publication, 1988.
26. B. Meyer, Genericity, static type checking, and inheritance, *The Journal of Pascal, Ada and Modula-2,* 1988, in press (revised version of paper in OOPSLA conference, Portland, Oregon, ACM SIGPLAN Notices, September 1986, pp. 391-405).
27. B. Meyer, *Object-Oriented Software Construction,* Prentice-Hall, Englewood Cliffs, N.J., 1988.
28. H. D. Mills and R. C. Linger, Data Structured Programming: Program Design without Arrays and Pointers, *IEEE Trans. Software Engineering,* 12 2, (1986).
29. R. Rousseau, Teaching software engineering with Eiffel (in French), in *Eiffel User Group Meeting,* Paris, January 1988.
30. C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt, An Introduction to Trellis-Owl, in *OOPSLA '86 Conference Proceedings,* Portland (Or.), Sept. 29-Oct. 2, 1986, pp. 9-16, 1986 (published as SIGPLAN Notices, 21, 11, Nov. 1986.)
31. B. Stroustrup, *The C+ + Programming Language,* Addison-Wesley, Menlo Park, Calif., 1986.
32. L. Tesler, Object Pascal Report, *Structured Language World* 9, 1985.
33. D. Ungar, Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm, in *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Penn., April 23-25, 1984), pp. 157-167, ACM Software Engineering Notes, 9, 3, and SIGPLAN Notices, 19, 5, May 1984.
34. N. Wirth, *Programming in Modula-2,* Springer-Verlag, New York, 1982.

PART 2: APPENDIX—BASIC EIFFEL LIBRARY

A.1 OVERVIEW

The classes given below are extracted from the basic library that constitutes one of the fundamental assets of designing software in Eiffel. They have been somewhat simplified and some features have been omitted in the interest of space (and of providing the reader with some incentive to try his hand at Eiffel programming), but they remain faithful to the original, which serves as a basis for all of our software developments. (Applications that have been built on top of this library include Cépage, a language-oriented editor and document constructor, and Winpack, a multi-windowing display management system.)

Missing elements that the reader is invited to complete are marked ***..... ***.

These classes illustrate the bottom-up, modular, reusable programming style encouraged by Eiffel.

As the examples show, the details of data structure implementation may be rather difficult, in particular when pointer manipulations are involved. This, we think, is an important argument for taking care of these details in reusable and flexible general-purpose modules, which can be thoroughly checked and optimized once and for all; the checking and optimization are better done there than in application programs. Such professional implementations of data abstractions may be used as the basis for "data structure programming" as advocated by Mills and Linger [28], enabling programmers to write and think in terms of lists, stacks, trees and the like rather than pointers, flags, offsets, indexes, etc.

Anybody who has written software involving nontrivial data structures in languages such as Pascal or C and found himself constantly fighting to avoid being swallowed in thick pointer soup will appreciate the availability of a library of extendible, reusable implementations for the most common data structures and associated operations.

The experience of writing this library has taught us that bottom-up design, if highly promising from the reusability standpoint, is also difficult. Coming up with correct and efficient tools that will satisfy many different needs is an exacting iterative process. Much work remains to be done to capture the core of software engineering applications. The challenge—factoring out into truly reusable software components as much as possible of the tedious and repetitive side of programming—is well worth the effort.

A.2 ARRAYS

One-dimensional arrays in Eiffel are not a primitive notion but a generic class, of which an implementation is given below. The main reason for including it here is that it is used by class *FIXED_LIST* below, one of the implementations of lists. Similar classes exist for two- and three-dimensional arrays.

An array may be allocated with arbitrary bounds through the procedure *Create*; to access or modify array elements, one uses the features *entry* and *enter* of the class *ARRAY*.

The implementation shown here relies on low-level, machine-dependent primitives for dynamic memory management: *allocate* for dynamically allocating memory areas, *dynget* to access data from such areas, *dynput* to change these data. We have assumed that these primitives are written in C, as is the case with our current Unix implementation. The low-level primitives directly manipulate addresses; since "address" is not, of course, a valid Eiffel type, addresses are encoded as positive integers. The encoding and decoding are the responsibility of the low-level routines; the Eiffel level only sees "abstract" integers.

The example contains little actual Eiffel code, but shows how an Eiffel class may be used to encapsulate a group of related low-level primitives and present it to the outside world as a coherent abstraction, complete with its preconditions, postconditions, and class invariant.

```
class ARRAY [T] export
    lower, size, upper,      -- (read-only)
    entry, enter
        -- The elements of an array are called "entries"
feature
    lower: INTEGER; upper: INTEGER;
    size: INTEGER;
    area: INTEGER; -- Secret
    Create (min: INTEGER, max: INTEGER) is
            -- Allocate current array with bounds min and max;
            -- no physical allocation if min > max.
        external
            allocate (length: INTEGER) : INTEGER name "allocate" language "C";
                -- Allocate should allocate an area for length integers
                -- and return its address (0 if impossible)
        do
            lower := min; upper := max;
            size := max - min + 1;
            if max > = min then area := allocate (size) end
        end; -- Create
    entry (i: INTEGER): T is
                -- Entry of index i
        require
            lower < = i; i < = upper; area > 0
        external
            dynget (address: INTEGER; index: INTEGER) : T name "dynget" language  "C";
                -- Value of index-th element in the area of address address
        do
            Result := dynget (area, i)
        end; -- entry
    enter (i: INTEGER, t: T) is
                -- Assign the value of t to the entry of index i
        require
            lower < = i; i < = upper; area > 0
        external
            dynput (address: INTEGER; index: INTEGER; val: T) name "dynput" language "C";
                -- Replace with val the value of the index-th
                -- element in the area of address address
        do
            dynput (area, i, t)
        end; -- enter
invariant
    size = upper - lower + 1
    -- area > 0 if and only if the array has been allocated
end -- class ARRAY [T]
```

A.3 GENERAL LISTS

This section and those that follow introduce classes corresponding to lists of various brands:

LIST [*T*]
    (General notion of list)
FIXED_LIST [*T*]
    (lists represented by arrays: no insertion or deletion)
LINKED_LIST [*T*]
    (lists in linked representation; insertions and deletions are possible)
TWO_WAY_LIST [*T*]
    (like *LINKED_LIST* but providing more efficient primitives for right-to-left traversal thanks to a doubly linked representation).

These classes have undergone a fairly substantial change from a previous version of the library and the present paper. A description of what happened may be of interest to readers concerned with the methodological principles of object-oriented software specification and design and, more specifically, with finding guidelines for the specification of systems.

Our initial approach was a strictly "static" one, in which we viewed lists as sequentially ordered repositories of information (of *T* type). Features available on a list *l* were of the form *l.get_value_by_index (i),* (value of the *i*-th element of *l*), *l.get_index_by_value* (*v, j*) (index of the *j*-th element of value *v*), etc.; and, for lists in linked representation, *l.insert_by_position (v, i)* (insert value *v* at position *i*), *l.delete_by_position* (*i*) (delete *i*-th element), etc.

As we started actually using the library, however, we were confronted with a disquieting increase in the number of primitives. For example, it sometimes happens that one wants to insert an element after the *j*-th occurrence of a given value. We could in principle use *get_index_by_value* followed by *insert_by_position,* but both features entail a sequential traversal of the list, which is unacceptable in practice since the first routine internally finds the adequate inserting position.

We were thus led little by little to add features such as *insert_by_value*, *delete_by_value*, etc. But even that did not end our predicament. It turned out that in practical uses of list, there are occasions in which clients need to keep a handle on a list element, so as to use it later without having to traverse the list again. It was not clear how to specify, let alone implement such a feature at the *LIST* level. In fact, the handle does not even have the same type in all cases: for a list represented as array, it should be an integer, the index; in linked representation, the only useful handle is a reference to a *LINKABLE* element. There is no way of factoring out these cases into a deferred procedure at the *LIST* level.

To implement the handle concept in the *LINKED_LIST* case, it seemed necessary to return to clients the supposedly secrete references to "linkable" elements. So we compromised by having some functions return *LINKABLE* entities; this was still relatively safe from the information hiding viewpoint since class *LINKABLE* had all its features protected (in a fashion somewhat similar to an Ada private type). But this decision led to yet another increase in the number of features: *get_index_by_linkable*, *get_linkable_by_value*, and so on.

The prospect of getting a reasonably universal yet concise enough implementation of lists was fading away as new features came creeping in.

At that point we realized our mistake, which was to treat lists as passive objects. A list is better modelled as an abstract machine whose instantaneous state includes not only the sequence of values constituting the list, but also the indication of a currently active position or "cursor" (see Figure 5).

With this approach, the primitives becomes much simpler:

- *l.value* is the value of the currently active element of list *l*;
- *l.position* is the index of this element (that is to say, the cursor position);
- *l.forth* moves the cursor to the next position;
- *l.go* (*i*) moves the cursor to the *i*-th position;
- *l.search* (*v, j*) moves the cursor to the *j*-th occurrence of *v*;
- the cursor may move at most one position off the leftmost or rightmost elements of the list;
- to save a position and return to it later (in a last-in, first-out fashion), one will use *l.mark* and *l.return*.

And so on. For a linked list, feature *active*, of type *LINKABLE* [*T*], provides access to the active element (see Section A.5); this feature does not transpose to other representations (such as by arrays), but this poses no problem since the feature is now, as it should be, a secret one. As an added benefit of the new approach, many features that initially seemed representation-spe-

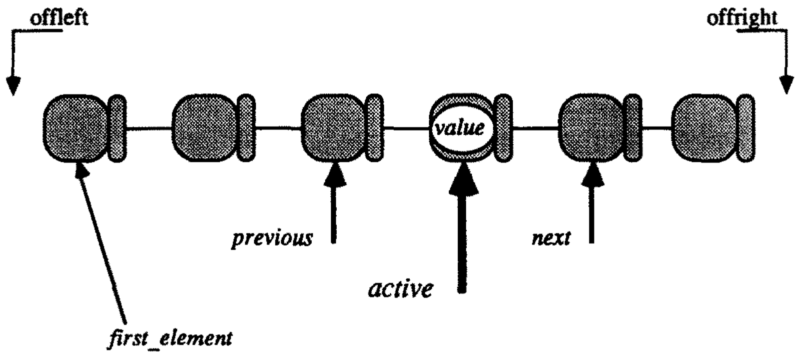

Figure 5. A list as a machine.

cific may now be lifted (sometimes in deferred form) to the generic class *LIST*.

This experience seems to lead to two conclusions, at the borderline between specification and design.

The first conclusion is the fact, mentioned above, that bottom-up construction of reusable software is a difficult, iterative process.

The second conclusion is that although the abstract data type approach may seem to imply a highly static and functional specification style, it should not preclude looking at object classes in an operational way, emphasizing the notion of state and the functions that act on the state. Some specification languages (such as LM) enforce a similar method by distinguishing between "access" and "transform" functions. Note that this does not entail any departure from a classical mathematical model based on functions.

With this background, we now introduce the *LIST* class.

```
            -- General lists, without committment as to the representation
    deferred class LIST [T] export
        nb_elements, empty,
        position, offright, offleft, isfirst, islast,
        value, i_th, first, last,
        change_value,change_i_th, swap,
        start, finish, forth, back, go, search,
        mark, return,
        index_of, present,
        duplicate

    feature
    -- Number of list elements

        nb_elements: INTEGER;

        empty: BOOLEAN is
                -- Is the list empty?
            do
                Result := (nb_elements = 0)
            end; -- empty
    -- Secret attributes for marking and retrieving
        backup: like Current;
        no_change_since_mark: BOOLEAN;

    -- Inquiring about the active position
        position: INTEGER;

        offright: BOOLEAN is
                -- Is active position off right limit?
            do
                Result := empty or (position = nb_elements + 1)
            end; -- offright
        offleft: BOOLEAN is
                    -- Is active position off left limit?
              do
                    Result := empty or (position = 0)
                              -- This formulation is for symmetry with offright: empty implies (position = 0),
                              -- so the second condition is equivalent to the entire "or" expression
              end; -- offleft
        isfirst: BOOLEAN is
                              -- Is active position first in the list?
                              -- (If so, the list is not empty)
              do
                    Result := (position = 1)
              ensure
                    not Result or else not empty
              end; -- isfirst
        islast: BOOLEAN is
                              -- Is active position last in the list?
                              -- (If so, the list is not empty)
              ***Left to the reader***
   -- Accessing list values
        value: T is
                    -- Value of active element
              require
                    not offleft; not offright -- These conditions imply not empty
              deferred
              end; -- value
        i_th (i: INTEGER): T is
                    -- Value of i-th element of the list
                    -- (Applicable only if i is a valid position for the list)
              require
                    i >= 1; i <= nb_elements;   -- These conditions imply not empty
              do
                    mark;
                    go (i); Result := value;
                    return
              ensure
                    -- Result = value of i-th element of the list
              end; --i_th
        first: T is
                    -- Value of first element in the list
              require
                    not empty
              do
                    Result := i_th (1)
              end; -- first
        last: T is
                    -- Value of last element in the list
              ***Left to the reader***
   -- Changing list values
        change_value (v: T) is
                    -- Assign v to value of active element
              require
                    not offleft; not offright -- These conditions imply not empty
              deferred
```

```
            ensure
                value = v
            end; -- change_value
        change_i_th (i: INTEGER, v: T) is
                -- Assign v to value of i-th element
                -- (Applicable only if i is a valid position for the list)
            ***Left to the reader***
        swap (i: INTEGER) is
                -- Exchange value of active element with value of element at position i.
                -- Active position is not changed.
                -- Not applicable if offleft, offright, or position i is not valid for the list.

            require
                not offleft; not offright;
                i >= 1; i <= nb_elements
                    -- These conditions imply not empty
            local
                thisvalue: T; thatvalue: T
            do
                thisvalue := value; mark;
                go (i); thatvalue := value; change_value (thisvalue);
                return;
                change_value (thatvalue)
            end; -- swap
    -- Moving along the list
        start is
                -- Make first element active (no effect if list is empty)
            deferred
            ensure
                (empty and Nochange) or else isfirst
            end; -- start
        forth is
                -- Make next position to the right active
                -- (Applicable only if not offright).
            require
                not offright -- This implies not empty
            deferred
            ensure
                position = old position + 1
            end; -- forth
        go (i: INTEGER) is
                -- Make i-th position active
                -- (Applicable only if 0 <= i <= nb_elements + 1)
            require
                i >= 0; i <= nb_elements + 1
            do
                if empty or i = 0 then
                    go_offleft
                else
                    from
                        if position > i then start end
                    invariant
                        position > 0; position <= i
                    variant i - position until position = i loop
                        check not offright end;
                        forth
                        end -- loop
                    end -- if
                ensure
                    (i = 0 and offleft) or
                    (i = nb_elements + 1 and offright) or
                    (1 <= i and i <= nb_elements and position = i)
                end; -- go
        back is
                    -- Make next position to the left active
                    -- (Applicable only if not offleft).
                    -- Warning: this version of back may be overly costly in implementations
                    -- that only provide for efficient left-to-right traversal
                require
                    not offleft
                do
                    check position >= 1 end; go (position - 1)
                end; -- back
        finish is
                    -- Make last element active (no effect if list is empty)
                do
                    go (nb_elements)
                ensure
                    (empty and Nochange) or else islast
                end; -- finish
        go_offleft is
                    -- Put the list in position offleft
                     (Secret procedure; use go (0) in clients)
                deferred
                ensure
                    offleft
                end; -- go_offleft
        search (v: T; i: INTEGER) is
                    -- Go to i-th element of value v in the list if there are at least i such elements;
                    -- else go offright.
                require
                    i > 0
                local
                    k: INTEGER
                do
                    from
                        start; k := 1
                    invariant
                        position >= 0;
                        -- k - 1 elements to the left of active position have a value equal to v
                    variant
                        nb_elements - position
                    until
                        offright or else (value = v and k = i)
                    loop
                        if value = v then k := k+1 end;
                        forth
                    end -- loop
                ensure
                    offright or else value = v
                    -- offright or else active element is the i-th element of value v
                end; -- search
    -- Marking and retrieving list positions.
```

```
        -- More than one position may be saved successively;
        -- retrieval will be done in a last-in, first-out order.
        mark is
                    -- Save active position
            do
                    backup.Clone (Current);
            end; -- mark
        return is
                    -- Make currently saved position active again
            require
                    not backup.Void; no_change_since_mark
            do
                    Extract (backup);
            end; -- return
    -- Finding information about occurrences of given elements.
        index_of (v: T; i: INTEGER): INTEGER is
                    -- Index of the i-th element of value v
                    -- (0 if fewer than i)
            require
                    i > 0
            do
                    mark;
                    search (v, i);
                    if not offright then Result := position end;
                    return
            ensure
                    -- (Result > 0 and then Result is the index of the i-th element of value v in the list)
                    -- or else (Result = 0 and there are fewer than i elements of value v in the list)
            end; -- index_of
        present (v: T): BOOLEAN is
                    -- Does v appear in the list?
            do
                    Result := index_of (v, 1) > 0
            ensure
                    -- Result = (v appears in the list)
            end; -- present
    -- Duplicating a list
        duplicate: like Current is
                    -- Complete clone of the list
            deferred
            end; -- duplicate
    -- Invariant for class LIST
    invariant

        position >= 0; position <= nb_elements + 1;
        not empty or else (position = 0);
        empty = (offleft and offright);
        offright = (empty or (position = nb_elements + 1));
        offleft = (empty or (position = 0));
            -- Note that empty implies (position = 0), so that also:
        offleft = (position = 0);

        isfirst = (position = 1);
        islast = (not empty and (position = nb_elements));
        not empty or else (not isfirst and not islast);
    end -- class LIST
```

A.4 LISTS IMPLEMENTED BY ARRAYS

Class *FIXED_LIST* [*T*] provides an array implementation of lists; only limited operations are available (no insertions or deletions). The array is created with fixed bounds, given as parameters to the version of procedure *Create* redefined for this class.

```
        -- Lists with a fixed number of elements
class FIXED_LIST [T] export
    ***Same exported features as in class LIST***
inherit
    ARRAY [T]
        rename Create as array_Create;
    LIST [T]
        redefine i_th, change_i_th, swap, go;
feature
    Create (n: INTEGER) is
                    -- Allocate fixed list with n elements
        do
            array_Create (1, n);
                check n = size end;
            nb_elements := n;
        end; -- Create
    value: T is
            -- Value of active element
        do
            Result := entry (position)
        end; -- value
    change_value (v: T) is
            -- Assign v to value of current element
        do
            enter (position, v)
        ensure
            value = v; entry (position) = v
        end; -- change_value
    i_th (i: INTEGER): T is
            -- Value of i-th element of the list
            -- (Applicable only if i is a valid position for the list)
        ***Left to the reader***
    change_i_th (i: INTEGER, v: T) is
            -- Assign v to value of i-th element
            -- (Applicable only if i is a valid position for the list)
        ***Left to the reader***
    swap (i: INTEGER) is
            -- Exchange value of active element with value of element at position i.
            -- Active position is not changed.
            -- Not applicable if offleft, offright, or position i is not valid for the list.

        ***Left to the reader***
    start is
            -- Make first element active (no effect if list is empty)
        do position := min (nb_elements, 1) end; -- start
    forth is
            -- Make next position to the right active
            -- (Applicable only if not offright).
        require
            not offright
```

```
          do
                position := position + 1
          ensure
                position = old position + 1
          end; -- forth
     go (i: INTEGER) is
                -- Make i-th position active
                -- (Applicable only if 0 < = i < = nb_elements+1)
          ***Left to the reader***
     go_offleft is
                -- Put the list in position offleft
                 (Secret procedure; use go (0) in clients)
          ***Left to the reader***
     duplicate: like Current is
                -- Complete clone of the list
          do
                Result.Create (nb_elements);
                      -- Result.Clone would be inappropriate here
                mark;
                from
                      start; Result.start
                invariant
                      -- position - 1 values have been copied
                variant
                      nb_elements - position
                until
                      offright        -- thus Result.offright too
                loop
                      Result.change_value (value);
                      forth; Result.forth
                end; -- loop
                return; Result.go (position)
          end; -- duplicate
invariant
          -- The class invariant adds nothing to the invariant of class LIST
end -- class FIXED_LIST
```

A.5 LINKED LIST ELEMENTS

This section introduces classes *LINKABLE* [*T*] and *BI_LINKABLE* [*T*] corresponding to "linkable" list components of two different brands: right-linked only and doubly-linked. Objects of such types have two fields: a value and a "right" pointer to another similar object. Bi-linkable objects also have a "left" field. Such component structures are designed for use in connection with classes representing linked lists: *LINKED_LIST* [*T*] and *TWO_WAY_LIST* [*T*].

```
          -- Linked list elements
          -- (for use in connection with LINKED_LIST [T] and TWO_WAY_LIST [T])
class LINKABLE T]
export
     value, change_linkable_value {LINKED_LIST},
     right, change_right {LINKED_LIST}, put_between {LINKED_LIST}
feature
     Create (initial: T) is
                      -- Initialize with value initial
          do
                value := t
          end; -- Create
```

```
        value: T;
        change_linkable_value (new: T) is
                        -- Assign value new to current list element
            do
                value := t
            end; -- change_linkable_value
        right: like Current;
        change_right (other: like Current) is
                        -- Put other to the right of the Current element
            do
                right := other
            end; -- change_right
        put_between (before: like Current; after: like Current) is
                -- Insert current element between before and after (if it makes sense)
                -- This procedure is used in LINKED_LIST every time an insertion is performed.
            do
                if not before.Void then before.change_right (Current) end;
                change_right (after);
            end; -- put_between
    end; -- class LINKABLE [T]
    class BI_LINKABLE [T]
        -- Same as LINKABLE [T], plus "left" field
    export
        value, change_bilinkable_value {TWO_WAY_LIST},
        right, change_right {BI_LINKABLE, TWO_WAY_LIST},
        left, change_left {BI_LINKABLE, TWO_WAY_LIST}
    inherit
        LINKABLE [T]
            rename change_linkage_value as change_bilinkable_value,
                -- Renaming is only to ensure consistent terminology.
            redefine right, change_right
    feature
        left: like Current;
        right: like Current;
        change_right (other: like Current) is
                        -- Put other to the right of current element
            do
                right := other;
                if not other.Void then
                    other.change_left (Current)
                end
            end -- change_right;
        change_left (other: like Current) is
                        -- Put other to the left of the current element
            do
                left := other;
                if not other.Void
                    -- Avoid infinite recursion with change_right!
                and then other.right /= Current
                then
                    other.change_right (Current)
                end
            end -- change_left
    invariant
        right.Void or else right.left = Current;
        left.Void or else left.right = Current;
  end -- class BI_LINKABLE [T]
```

A.6 LINKED LISTS

Class *LINKED_LIST [T]* introduces singly linked lists. All operations of insertion and deletion are possible; however, since the lists are chained one way only, operations such as *back,* implying a complete traversal, will be inefficient. They are provided, however, for completeness.

The representation keeps references not only to the active element but also to its left and right neighbors (*active, left, right*). This allows, for example, efficient insertions both just before and just after the active element.

A note to the courageous reader: an excellent test of your understanding of the present set of basic classes and the general principles of Eiffel design is to write two procedures patterned after *insert_right* and *insert_left* below, namely

```
    merge_after (l: like Current)
    merge_before (l: like Current)
```

which insert a linked list *l* to the right and left (respectively) of the currently active position. The precise conditions (**require**...) under which they are applicable should be spelled out. The guiding criteria should be simplicity (no auxiliary procedure is necessary), preservation of the class invariant, perfect symmetry between left and right, and elegance. It will be even better if the procedures also apply to two-way lists (next section) without redefinition.

```
                    -- One-way linked lists
  class LINKED_LIST [T] export
              --Features from LIST:
        nb_elements, empty,
        position, offright, offleft, isfirst, islast,
        value, i_th, first, last,
        change_value, change_i_th, swap,
        start, finish, forth, back, go, search,
        mark, retrieve,
        index_of, present,
        duplicate,
              -- Plus new features permitted by linked list representation:
        insert_right, insert_left,
        delete, delete_right, delete_left,
        delete_all_occurrences, wipe_out
  inherit
        LIST [T]
              redefine first
  feature
        first: T;     -- Value of first element (redefined here as attribute)
  -- Secret attributes specific to linked list representation
        first_element: LINKABLE [T];
        active:, previous, next: like first_element;
  -- Linked list implementations of features deferred in LIST
        value: T is
                    -- Value of active element
              require
                    not offleft; not offright -- These conditions imply not empty
              do
                    Result := active.value
              end; -- value
        change_value (v: T) is
                    -- Assign v to value of current element
```

```
            require
                  not offleft; not offright -- These conditions imply not empty
            do
                  active.change_linkable_value (v)
            ensure
                  value = v
            end; -- change_value
      start is
                  -- Make first element active (no effect if list is empty)
            do
                  if not empty then
                        previous.Forget; active := first_element;
                        check not active.Void end;
                        next := active.right; position :=:= 1
                  end
            ensure
                  empty or else isfirst
            end; -- start

      forth is
                  -- Make next position to the right active
                  -- (Applicable only if not offright).
            require
                  not offright
            do
                  if offleft then
                        check not empty end; start
                  else
                        check not active.Void end;
                        previous := active; active := next;
                        if not active.Void then next := active.right end;
                        position := position + 1
                  end
            ensure
                  position = old position + 1
            end; -- forth

      go_offleft is
                  -- Put the list in position offleft
                   (Secret procedure; use go (0) in clients)
            do
                  active.Forget; previous.Forget; next := first_element;
                  position := 0
            ensure
                  offleft
            end; -- go_offleft
      duplicate: like Current is
                  -- Complete clone of the list
            ***Left to the reader (go through the list, duplicating every list element)***
            ***(See the corresponding procedure for FIXED_LIST)***
-- Deletion and insertion procedures specific to linked lists
      insert_right (v: T) is
                  -- Insert an element of value v to the right of active position if there is one;
                  -- Active position is unchanged.
                  -- Applicable only if list is empty or not offright
            require
                  empty or else not offright
            local
                  new: like first_element
```

```
        do
            new.Create (v); insert_linkable_right (new)
        ensure
            nb_elements = old nb_elements + 1;
            active = old active; position = old position;
            not next.Void; next.value = v
        end; -- insert_right
    insert_left (v: T) is
            -- Insert an element of value v to the left of active position if there is one.
            -- Active position is unchanged.
            -- Applicable only if list is empty or not offleft
        ***Left to the reader***
    delete is
            -- Delete active element and make its right neighbor, if any, active
            -- (List becomes offright if no right neighbor)
            -- Not applicable if offleft or offright
        require
            not offleft; not offright
        do
            active := next;
            if not previous.Void then previous.change_right (active) end;
            if not active.Void then next := active.right end;
                -- else next is void already
            nb_elements := nb_elements - 1;
            no_change_since_mark := false;
            check
                position - 1 >= 0; position - 1 <= nb_elements;
                empty or else position - 1 > 0 or else not active.Void;
            end;
            update_after_deletion (previous, active, position - 1);
        ensure
            nb_elements = old nb_elements - 1;
            empty or else (position = old position)
        end; -- delete
    delete_right is
            -- Delete element immediately to the right of active position; active position is unchanged.
            -- (No effect if active position is last in list).
            -- Not applicable if offright
        ***Left to the reader (imitate delete)***
    delete_left is
            -- Delete element immediately to the left of active position;
            -- active position is unchanged (but its index is decremented by 1).
            -- (No effect if active position is first in list)
            -- Not applicable if offleft
            -- Inefficient for one-way lists: included for completeness
        ***Left to the reader (use back and delete)***
    delete_all_occurrences (v: T) is
            -- Delete all occurrences of v from the list
        do
            from start until offright loop
                if value = v then delete else forth end
            end;
            no_change_since_mark := false
        end; -- delete_all_occurrences
    wipe_out is
            -- Empty the list
        do
              nb_elements := 0; position := 0;
              active.Forget; first_element.Forget; previous.Forget; next.Forget;
              no_change_since_mark := false
        ensure
              empty
        end -- wipe_out
-- Secret routines for implementing insertion and deletion
    insert_linkable_right (new: like first_element) is
              -- Insert new to the right of active position if there is one;
              -- Active position is unchanged.
              -- Secret procedure.
              --Applicable only if list is empty or not offright
        require
              not new.Void; empty or else not offright
        do
              new.put_between (active, next); next := new;
              nb_elements := nb_elements + 1;
              no_change_since_mark := false;
              check
                    position + 1 >= 1; position + 1 <= nb_elements
              end;
              update_after_insertion (new, position + 1)
        ensure
              nb_elements = old nb_elements + 1; position = old position
              previous = new
        end; -- insert_linkable_right
    insert_linkable_left (new: like first_element) is
              -- Insert new to the left of active position if there is one;
              -- Active position is unchanged (but its index is increased by one).
              -- Secret procedure.
              -- Applicable only if list is empty or not offleft
        require
              not new.Void; empty or else not offleft
        do
              if empty then position := 1 end;
              new.put_between (previous, active); previous := new;
              nb_elements := nb_elements + 1; position := position + 1;
              no_change_since_mark := false
              check
                    position - 1 >= 1; position - 1 <= nb_elements
              end;
              update_after_insertion (new, position - 1);
        ensure
              nb_elements = old nb_elements + 1; position = old position + 1;
              previous = new
        end; -- insert_linkable_left

    update_after_insertion (new: like first_element; index: INTEGER) is
              -- Check consequences of insertion of element new at position index:
              -- does it become the first element?
        require
              not new.Void; index >= 1; index <= nb_elements
        do
              if index = 1 then
                    first_element := new; first := new.value
              end
        end; -- update_after_insertion
        update_after_deletion (one: like first_element; other: like first_element; index: INTEGER) is
                    -- Check consequences of deletion of element between one and other,
                    -- where index is the position of one.
                    -- Update first_element if necessary.
                require
                    index >= 0; index <= nb_elements;
                    empty or else index > 0 or else not other.Void;
                    -- the element deleted was between one and other
                do
                    if empty then
                        first_element.Forget; position := 0
                    elsif index = 0 then
                        check not other.Void end;          -- See precondition
                        first_element := other; first := other.value
                    -- else do nothing special
                    end
                end; -- update_after_deletion
        -- Invariant for class LINKED_LIST
    invariant
        -- The invariant of class LIST plus the following:
        empty = first_element.Void;
        empty or else first_element.value = first;
        active.Void = (offleft or offright);
        previous.Void = (offleft or isfirst);
        next.Void = (offleft or islast);
        previous.Void or else (previous.right = active);
        active.Void or else (active.right = next);
        -- (offleft or offright) or else active is the position-th element
end -- class LINKED_LIST
```

A.7 TWO-WAY LISTS

Class *TWO_WAY_LIST* [*T*] introduces doubly linked lists. Features *back* and *forth* now have the same efficiency; in fact the whole class is almost entirely symmetric with respect to "left" and "right."

```
                -- Two-way linked lists
class TWO_WAY_LIST [T] export
    ***Same exported features as in LINKED_LIST***
    -- Some features, however, are redefined more efficiently
inherit
    LINKED_LIST [T]
        rename go as reach_from_left, wipe_out as simple_wipe_out,
        redefine
            first_element, last, back, go, wipe_out, last
            update_after_deletion, update_after_insertion
feature
    first_element: BI_LINKABLE [T];        -- Redefined from LINKED_LIST
                    -- For two-way lists, we also keep a reference
                    -- to the last element and its value:
    last_element: like first_element;
    last: T;
                    -- Redefined here as an attribute
                    -- (It was a function in LINKED_LIST).
    back is
            -- Make next position to the left active
            -- (Applicable only if not offleft).
        require
                not offleft
        do
                if offright then
                        check not empty end; finish
                else
                        check not active.Void end;
                        next := active; active := previous;
                        if not active.Void then previous := active.left end;
                        position := position - 1
                end
        ensure
                position = old position - 1
        end; -- back
    go (i: INTEGER) is
                -- Make i-th position active
                -- (Applicable only if 0 <= i <= nb_elements+1)
        require
                i >= 0; i <= nb_elements+1
        do
                if i = nb_elements+1 then
                        -- Go offright
                        active.Forget; next.Forget; previous := last_element;
                        position := nb_elements+1
               elsif i <= position/2 or (i >= position and i <= (position+nb_elements)/2) then
                        reach_from_left (i)
               else
                                -- Reach from the right
                        from
                                if position < i then
                                        -- Finish (revised for two-way_lists)
                                        active := last_element; previous := active.left; next.Forget
                                end
                        invariant
                                position <= nb_elements; position >= i
                        variant position - i until position = i loop
                                check not offleft end;
                                back
                        end -- loop
                end -- if
        ensure
                position = i
        end; -- go
    update_after_insertion (new: like first_element; index: INTEGER) is
                -- Check consequences of insertion of element new at position index:
                -- does it become the first element?
        ***Redefinition left to the reader***
        ***Hints: make the routine symmetric with respect to right and left;***
        ***last_element and last may need to be updated as well as first_element and first***

    update_after_deletion (one: like first_element; other: like first_element; index: INTEGER) is
                -- Check consequences of deletion of element between one and other,
                -- where index is the position of one.
                -- Update first_element if necessary.
        ***Redefinition left to the reader***
        ***Hints: see update_after_insertion***

    wipe_out is
                -- Empty the list
        do
              simple_wipe_out; last_element.Forget
        ensure
              empty
        end -- wipe_out

     -- Invariant for class TWO_WAY_LIST
invariant
        -- The invariant of class LINKED_LIST, plus the following:
        empty = last_element.Void;
        empty or else last_element.value = last;
        active.Void or else (active.left = previous);
        next.Void or else (next.left = active);
        -- (offleft or offright) or else active is the position-th element
end -- class TWO_WAY_LIST
```

A.8 TREES AND THEIR NODES

The following class is an implementation of trees, using linked representation. Note that no distinction is made between trees and tree nodes.

As explained in Section 4.2 of the main text, tree nodes are implemented as a combination of lists and list elements. The list features make it possible to obtain the children of a node; the list element features make it possible to access the value associated with each node and its right sibling (the class may be redefined using two-way lists and "bi-linkable" elements to allow access to the left sibling as well). The added feature *parent* makes it possible to access the parent of each node.

Since each node of the tree is—among other things—a list in the sense defined above; so it keeps a record of which of its children is the "active" one. To change the active child of a node, procedures inherited from *LIST* (through *LINKED_LIST*) are available: *back, forth, go,* etc.

```
class TREE [T] export
     position, offright, offleft, isfirst, islast, start, finish, forth, back, go, mark, return,
     is_leaf, arity,
     node_value, child_value, change_node_value, change_child_value,
     child, change_child, right_sibling, first_child,
     insert_child_right, insert_child_left,
     delete_child, delete_child_left, delete_child_right
     parent, is_root
inherit
     LINKABLE [T]
          rename
               right as sibling,
               value as node_value, change_value as change_node_value,
               put_between as linkable_put_between;
          redefine put_between;
     LINKED_LIST [T]
          rename
               empty as is_leaf, nb_elements as arity,
               value as child_value, change_value as change_child_value,
               active as child, first_element as first_child,
               insert_linkable_right as insert_child_right, insert_linkable_left as insert_child_left,
               delete as delete_child, delete_left as delete_child_left, delete_right as delete_child_right,
               update_after_insertion as linked_update_after_insertion;
          redefine first_child, update_after_insertion
feature
     first_child: like Current;
     parent: like Current;
```

```
attach_to_parent (n: like Current) is
                -- Make n the parent of current node.
                -- Secret procedure.
    do
        parent := n
    ensure
        parent = n
    end; -- attach_to_parent

update_after_insertion (new: like first_element; index: INTEGER) is
        -- Check consequences of insertion of element new at position index:
        -- does it become the first element?
        -- Secret procedure redefined from LINKED_LIST
    require
        not new.Void; index >= 1; index <= nb_elements
    do
        linked_update_after_insertion (new, index);
        if index = 1 then
            new.attach_to_parent (Current)
        end
    end; -- update_after_insertion
change_child (n: like Current) is
                -- Replace by n the active child
    require
        not offleft; not offright; -- Thus not child.Void
        not n.Void
    do
        insert_child_right (n);
        check
            n.parent = Current
                -- Because of the redefinition of put_between
        end;
        delete_child
        check
            child = n
                -- Because of the convention for the new active element after delete
        end
            -- A direct implementation (not using insert and delete) is also possible
    ensure
        child = n;
        n.parent = Current
    end; -- change_child

is_root: BOOLEAN is
                -- Is current node a root?
    do
        Result := parent.Void
    end; -- is_root

put_between (before: like Current; after: like Current) is
        -- Insert current element between before and after (if it makes sense)
        -- Redefined from class LINKED_LIST
        -- to ensure that Current will have the same parent as its new siblings.
    require
        (before.Void or after.Void) or else (before.parent = after.parent)
    do
        linkable_put_between;
        if not before.Void then attach_to_parent (before.parent) end;
        if not after.Void then attach_to_parent (after.parent) end;
    end; -- put_between
```

```
invariant
        -- The invariants of the parent classes, plus the following:
    is_root = parent.Void;
    sibling.Void or else sibling.parent = parent;
    child.Void or else child.parent = Current;
    previous.Void or else previous.parent = Current;
    next.Void or else next.parent = Current;
    first_child.Void or else first_child.parent = Current;
end -- TREE [T]
```