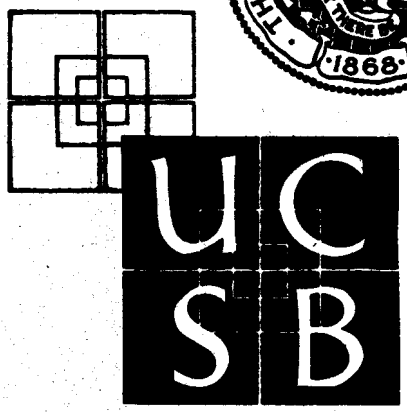

university of california • santa barbara

Department of Computer Science

TRCS85-1

Eiffel: A Language for Software Engineering

Bertrand Meyer

Department of Computer Science

(December 1985)

College of Engineering

Eiffel:

A Language for Software Engineering

Bertrand Meyer

Department of Computer Science, University of California
Santa Barbara, CA. 93106 (USA)

and

Interactive Software Engineering, Inc.
270 Storke Road, Suite #7
Goleta, CA. 93117 (USA)

ABSTRACT

The Eiffel language addresses the problem of designing quality software in practical development environments.

Two software quality factors were deemed essential in the design of Eiffel: reusability and reliability. Consideration of the technical implications of these factors led to the following choices: language features that encourage bottom-up software design; modular structures based on the object-based approach pioneered by Simula 67, but including both generic parameters and multiple inheritance (with a new extension, repetitive inheritance); highly dynamic execution model; information hiding facilities; assertions and invariants that may be monitored at run-time.

The current Eiffel system runs under Unix and uses C as an intermediate language; it also takes care of configuration management aspects, automatically performing tasks similar to those of the Unix Make tool.

Keywords: software reusability, software reliability, modularity, object-oriented design and programming, object-oriented languages, bottom-up software design, assertions, invariants, Eiffel.

TABLE OF CONTENTS

Eiffel: A Language for Software Engineering

Bertrand Meyer

PART 1: LANGUAGE OVERVIEW

1.1 - PRESENTATION

Our main interest is in designing software, not programming languages. But when we looked for a language allowing the development of production-quality software according to the modern principles of software engineering, we realized that no existing language provided the needed facilities; this simple realization led to the design of Eiffel.

Eiffel is intended for use at both the design and implementation stages of software development. The language has been implemented and currently runs on Unix; the basic library described in Part 2 is used as support for what constitutes our real aim: producing advanced software tools such as Cépage, a structural and visual editor [17, 19], Winpack (a general-purpose screen handling package) and other products.

This article is a general presentation of the language and the underlying design method. Part 1 covers the fundamental concepts of the language and will allow the reader to understand Part 2, a set of programming examples from the basic Eiffel library. The emphasis on this library reflects the fact that a mere presentation of individual language characteristics would not suffice to show what may be the most important aspect of the Eiffel approach: a method of software design and implementation which views software systems as structured networks of extendible and reusable abstract data type implementations.

The rest of Part 1 reviews the language. In section 1.2 , we give an overview of the design criteria for Eiffel. Section 1.3 introduces some of the basic concepts of object-based design. Section 1.4 describes the fundamental Eiffel structure (the class). Section 1.5 presents the multiple and repeated inheritance techniques that constitute the key to reusable programming in Eiffel. We describe the typing rules in section 1.6 and the use of assertions for expressing correctness arguments in section 1.7. Section 1.8 briefly surveys the practical aspects of Eiffel usage and our current implementation techniques.

Part 2 is a library of Eiffel classes defining reusable software components that play an essential role in our current developments.

Part 3 summarizes the main results and mentions some related efforts.

Although a few details may remain hazy, we hope that the reader will form a good idea of Eiffel programming. The examples use the essential features of Eiffel, and thus provide a good view of the language: if you understand this article, you may still have a few things to learn to become a real Eiffel programmer, but not many.

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** is thus an important criterion, precluding the use of functional or interpreted languages.
- **Reliability** of the software we produce is another fundamental aim, promoting such features as strict type checking, limitations on side-effects, etc.
- Current program construction techniques, which amount to re-inventing the wheel over and over again, seem unacceptable to us. Software development methods and languages

should emphasize the **reusability** of software components as one of their primary goals.

- **Extendibility** (the ease of adapting software to changes and specifications) and **compatibility** (the ease of combining separately produced software elements) are also essential in the long term.
- A particularly important requirement of the main type of software we develop (software engineering tools) is the need to manipulate complex, highly **dynamic** data structures. It should be possible to create as many of these objects as needed and to rely on support tools for automatic memory management and reclamation of unused space; programmer-controlled de-allocation (in the PL/I-Pascal-Modula 2 tradition) is an awkward and dangerous feature whose presence is unexplainable in any language whose designers have expressed any concern for program reliability.
- **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.
- Finally, **portability** is also a serious concern.

Of course, a solution to these issues must involve elements other than programming languages; for example, reusability raises questions of specification (to be reusable on a large scale, software components should be formally specified), design (one needs to reuse designs, not just codes), documentation (to reuse elements, one must find out about them) and tools (to retrieve and combine components). Reusability also raises non-technical questions: training, economic incentives and others.

However the rôle of technical aspects, and more precisely of language features, should not be underestimated: all software designs must eventually be expressed in terms of programs, and it is well known that programming languages exert a strong influence on the way software developers think. Thus we felt that choosing a proper language was a key step towards meeting the above criteria. But it does not take a very long analysis to realize that no widely available language satisfies all of them.

1.3 - OBJECT-BASED DESIGN

In our experience, the general approach to software construction that best answers the above concerns is the method known as object-based design, which may carried over to the implementation stage through object-based (or "object-oriented") programming languages applying the ideas introduced by Simula 67.

1.3.1 - Background

There are several ways to describe object-based design, depending on the individual presenter's background. Because Smalltalk has been so largely publicized, many current views of object-based programming emphasize two aspects: the concept of *messages* for communicating information between objects, and the very *dynamic* nature of the Smalltalk environment, which makes it possible to defer bindings between names and their denotations until run-time and, as in Lisp, provides programmers with great freedom. Such features are particularly useful from the perspective of Artificial Intelligence applications, for example, or for rapid prototyping.

Our interest in object-based languages comes from a more traditional software engineering perspective. We 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, and more generally static binding, are essential concerns.

In this respect, the author's view was much influenced by Simula 67; 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 convinced me that object-based 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.

1.3.2 - Modularizing for extendibility

With this background, we may present the definition of object-based design which underlies Eiffel. Object-based design is viewed as a **system modularization technique**, relying on the idea that the structure of software systems should best be patterned, at the highest level, on the objects manipulated by the system, rather than on the system's function.

Several arguments may be used to support this approach. Here we shall only elaborate on one: extendibility.

Observation of durable programs shows that the precise tasks performed by systems vary dramatically over their lifecycle. For example, a program may simply perform, at a certain stage of its evolution, an input-to-output transformation, each run processing a batch of data and producing corresponding results; as such programs are used and adapted, they often evolve into systems that keep some information between successive runs, and may end up as interactive systems accessing a comprehensive data base, with finer-grain inputs and outputs for each individual transaction.

Studied from the standpoint of the tasks they perform, the initial and the final versions may be very different. To realize that they are versions of the same program, one has to look closer and consider the *objects* handled by the system. If 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 which are applied to them.

1.3.3 - Seven steps towards object-based happiness

Th above remark is one of the main reasons for basing the module structure of the system, at the highest level, on the data rather than the actions. The basic motto of object-based design could thus be formulated as follows:

> **Principle 1** (*object-based modular structure*): do not ask what the system does: ask what it does it to.

To get object-based design in its full sense, however, further steps must be taken. The next one 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. In fact, a study of software maintenance costs by Lientz and Swanson [15] shows that, out of the approximately 50% of software costs devoted to maintenance, more than 17% arise from the need to account for changes in physical data formats. Thus one would be ill-advised to 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 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 duality between functions and objects is an unescapable fact of programming; object-based 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.

We thus reach our second step towards object-based happiness:

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, that is to say to design modules that encapsulate the implementation of one or more abstract data types. Ada and Modula-2 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. On the other hand, languages which do *not* provide any such possibility are Pascal, Cobol and Basic.

The third step is of a less conceptual nature. It reflects an important implementation concern: how objects are created. To be able to freely use objects, programmers should not have to take care of their physical allocation and deallocation. Here most of our language friends leave us; although this is in a strict sense a property of language systems 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], section 4.8) 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-based languages, and has itself been subjected to object-based extensions.

Principle 3 (*automatic memory management*): Objects should be created and deallocated by the underlying language system, without programmer intervention.

The next step is the one which, in our opinion, truly distinguishes object-based languages from the rest of the world. It may be understood by looking at languages which are *not* object-based 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 a purely 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* $\equiv$ *type*.

This fusion of two apparently distinct notions is what gives object-based 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 non-basic type is a module, and every high-level module is a type.

The qualifier "non-basic" keeps open the possibility of having simple predefined types (such as *INTEGER*, etc.) which 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.

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

The next step is a natural consequence from 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, whereby a new type may be defined by adding new properties to an existing type (as a Pascal integer range, whose elements are integers subject to some

restrictions). In object-based 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 1.5.

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. Applying 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" in the ordinary sense, 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 (class) and 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.

Existing languages that (in our opinion) are worthy of the name "object-based" satisfy all principles above; they include Simula [10, 3, 16], C++ [24], Object Pascal [25], Objective-C [8] and Smalltalk [12].

The next and last step extends the notion of inheritance to enable reuse of more than one context. This is the notion of multiple inheritance, developed in section 1.5 below. Of the languages mentioned above, only Smalltalk, to our knowledge, offers it in its recent versions (although the basic reference on Smalltalk, cited above, excludes this feature). Eiffel adds to this notion the concept of **repeated** inheritance (reusing the same structure more than once); see 1.5.5 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 reader may have noted that in our seven "principles" we have alternated between high-level, design-related concepts and programming language features. One particularly interesting benefit of the object-based approach is indeed that the same language may be used for design and implementation. Some language traits, such as deferred features (1.5.8) are especially useful for the application of Eiffel to system design.

1.4 - BASIC EIFFEL CONCEPTS

We now introduce the basic elements of Eiffel programming: run-time model, objects, classes, export controls.

1.4.1 - Run-time model

Before introducing the syntax of the language, it is best to first present the model that underlies its dynamic semantics.

Eiffel relies on an entirely dynamic execution model: the execution of a *system* (a term which is preferred to "program" for this language) may be characterized at each instant by the presence of a certain number of **objects**, each of which possesses some **attributes**. Attributes are either simple values (integers, booleans, real numbers or character strings) or *references* to objects. Figure 1 gives a pictorial view of such a collection of objects and their attributes.

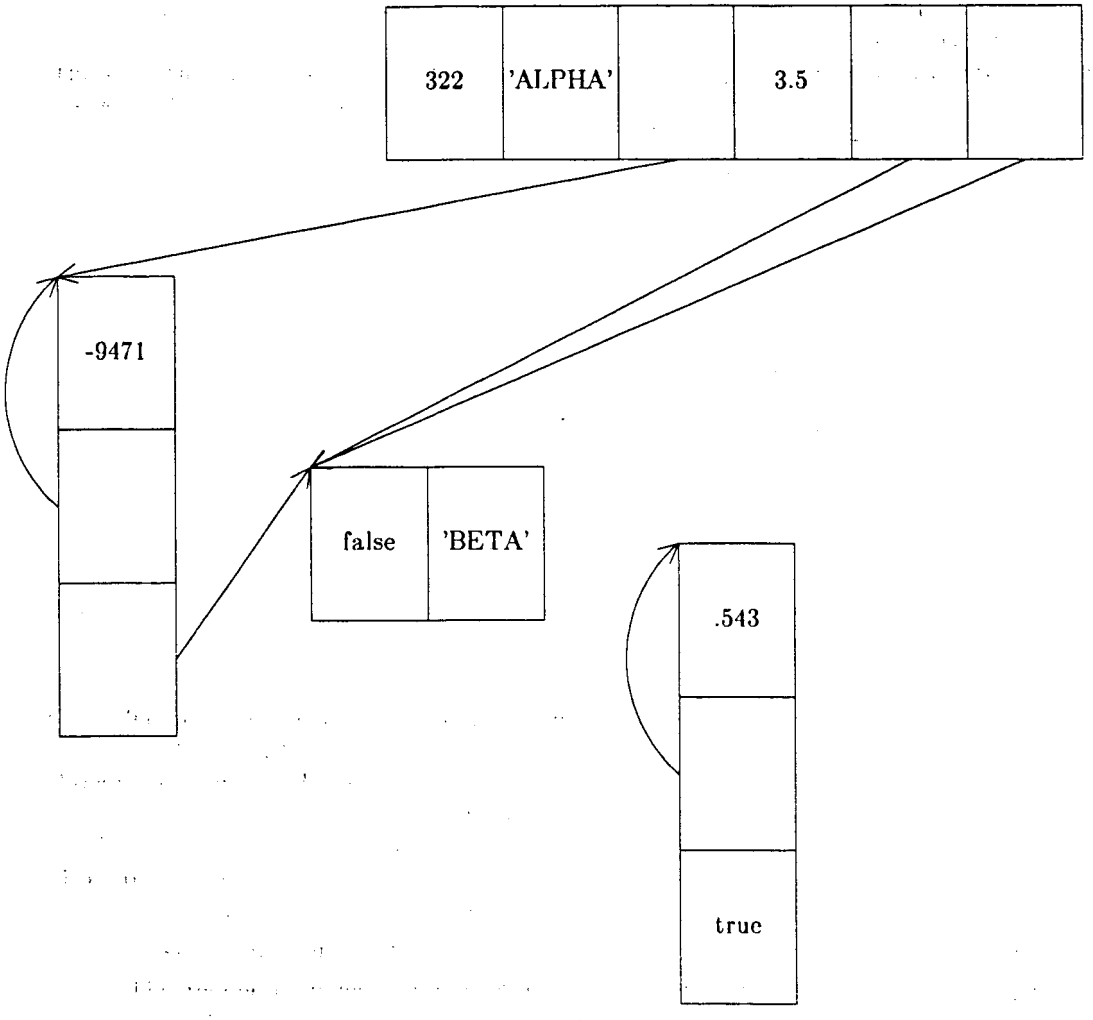

Figure 1: Objects

1.4.2 - Routines

Operations, or **routines**, may be applied to objects. Routines are divided into **procedures** and **functions**. One may think of procedures as *commands* and functions as *questions*: 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 procedure features; executing this procedure will usually trigger the creation of other objects and more routine calls.

1.4.3 - Classes and system structure

An Eiffel system consists of one or more **classes**, each describing a set of potential objects with the same features (attributes and routines), that is to say, with the same structure and operations.

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

As implied by the above principles, classes are not only types but also modules. In fact, they constitute the only system structuring facility.

1.4.4 - Entities

The program elements that may take on values at run-time are called **entities.** The notion of entity is more general than that of variable, since it includes (in Eiffel) local variables of routines (including the predefined variable *Result* denoting the result to be returned by a function), routine parameters, and constructs denoting object attributes.

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.* All other types are class types.

1.4.5 - States of an entity

Let x be an entity of a class type C. 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.

The following two instructions change the state of an entity: *x.Forget* puts x in the void state; *x.Create* puts x in the created state by creating a new object of type C and associating it with x. 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).

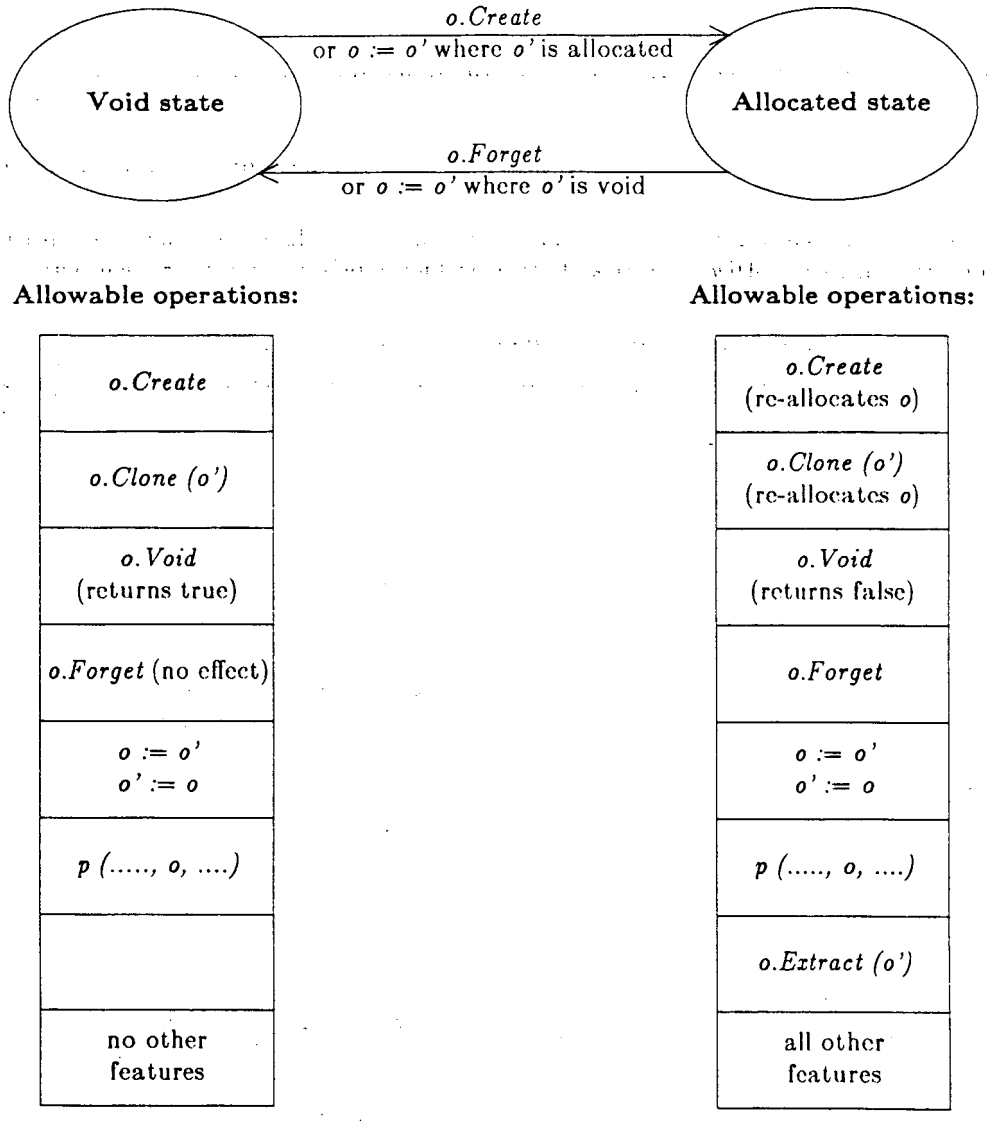

Figure 2: States of a reference, permissible operations and transitions

Void, *Create* and *Forget* are **predefined features** applicable to all classes. The language includes two other predefined features: *x.Clone* (*y*) creates a new copy of the object referenced by *y* and assigns to *x* a reference to the new object; procedure *Extract* performs conversions between objects of different classes and will be described in section 1.6.1.

1.4.6 - 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 true, character strings will be blank 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 parameters, 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 2.2 for the *ARRAY*

class, for which a version of *Create* is defined in such a way that *t.Create* (*min*, *max*) will associate with *t* a new array with bounds *min* and *max*.

1.4.7 - 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.

All parameters to a routine are protected: more precisely, a routine, whether a procedure or a function, may not include an assignment whose target is a formal parameter of the routine. However, a procedure may, by applying procedures to its parameters, change the attributes of the associated objects.

1.4.8 - Expressions and instructions

The construct used to express 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 parameters, the notation is

$x.f$

If f is a routine with parameters, actual parameters 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.

Either form is syntactically 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 entity of class types the assignment $x := y$ results in x and y being a reference to the same object (or x being void if y was).

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

1.4.9 - A simple class

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

Following the Ada convention, any part of a line beginning with two consecutive dashes -- is a comment.

```
class POINT export
        x, y, translate, scale, distance
feature
        x: REAL ;
        y: REAL ;
        scale (factor: REAL) is
                        - - Scale by a ratio of factor.
                do
                        x := factor*x ;
                        y := factor*y
                end ; - - scale
        translate (a: REAL ; 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, that is to say classes that include one or more entity declarations of the form

```
p: POINT
```

and may thus execute operations such as

```
p.Create ;  - - Dynamic allocation of 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 which 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.

As in other true object-based languages, the text of an Eiffel class always refers to a *current object* of the class. Most of the time this current object is anonymous; in a class (like *POINT*), a feature name (like x) which appears unqualified (i.e. just x, not $p.x$ for some p of type *POINT*) denotes the corresponding feature of the current object. If one needs to refer explicitly to the current object, the predefined entity name *Current* is available. Thus we 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 right type (*REAL* in the case of function *distance*).

1.4.10 - Generic parameters

One more basic property of classes belongs in this overview: a class may have one or more **generic parameters** that represent types. For example, section 2.6 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 declaration of entities of type T. A client of the class will then declare entities of type *LINKED_LIST* [*INTEGER*], *LINKED_LIST* [*POINT*], etc.

The "horizontal" form of genericity, as provided by class parameters, is a useful complement to the more powerful "vertical" reusability features offered by inheritance and described below. Another language that combines these two approaches is LPG (Language for Generic Programming), developed in Grenoble [2].

The power of such a combination is evidenced by the examples of Part 2. A more detailed comparative analysis of genericity and inheritance and a rationale for their use in Eiffel may be found in reference [18].

1.5 - INHERITANCE: TREES ARE LISTS AND LIST ELEMENTS

1.5.1 - Principles of inheritance

Inheritance, introduced by Simula 67, is one of the key techniques for reusability. It makes it possible to entrust a new class with the features of previously defined classes.

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.

The following example shows the whole power of this notion. If the reader remembers just one idea from this article, we would like it to be this: *a tree is a list and a list element.* Let's explain.

In the classes of Part 2, we define *lists* of various brands. One of these classes has already been mentioned: *LINKED_LIST* [*T*] (section 2.6), describing one-way linked lists of elements; it itself inherits some of its properties from a more general class, *LIST* [*T*] (section 2.3), 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.

In order to manipulate linked lists of elements of type T, one needs 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. We use the word "linkable" to refer to such cells. Class *LINKABLE* [*T*] (section 2.5) describes their features, particularly two attributes: *value*, of type T, and *right*, of type *LINKABLE* [*T*][1].

Now assume we need to define the notion of **tree**, as implemented in linked representation. We may certainly start from scratch; programming tradition, as well as fifteen years of propaganda for top-down design, indeed encourage us 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

[1] Feature *right* is actually declared of type **like** *Current* for reasons explained in 1.6.2.

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 2.8, 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: one must add the specific features of trees, and the little mutual compromises which, 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.

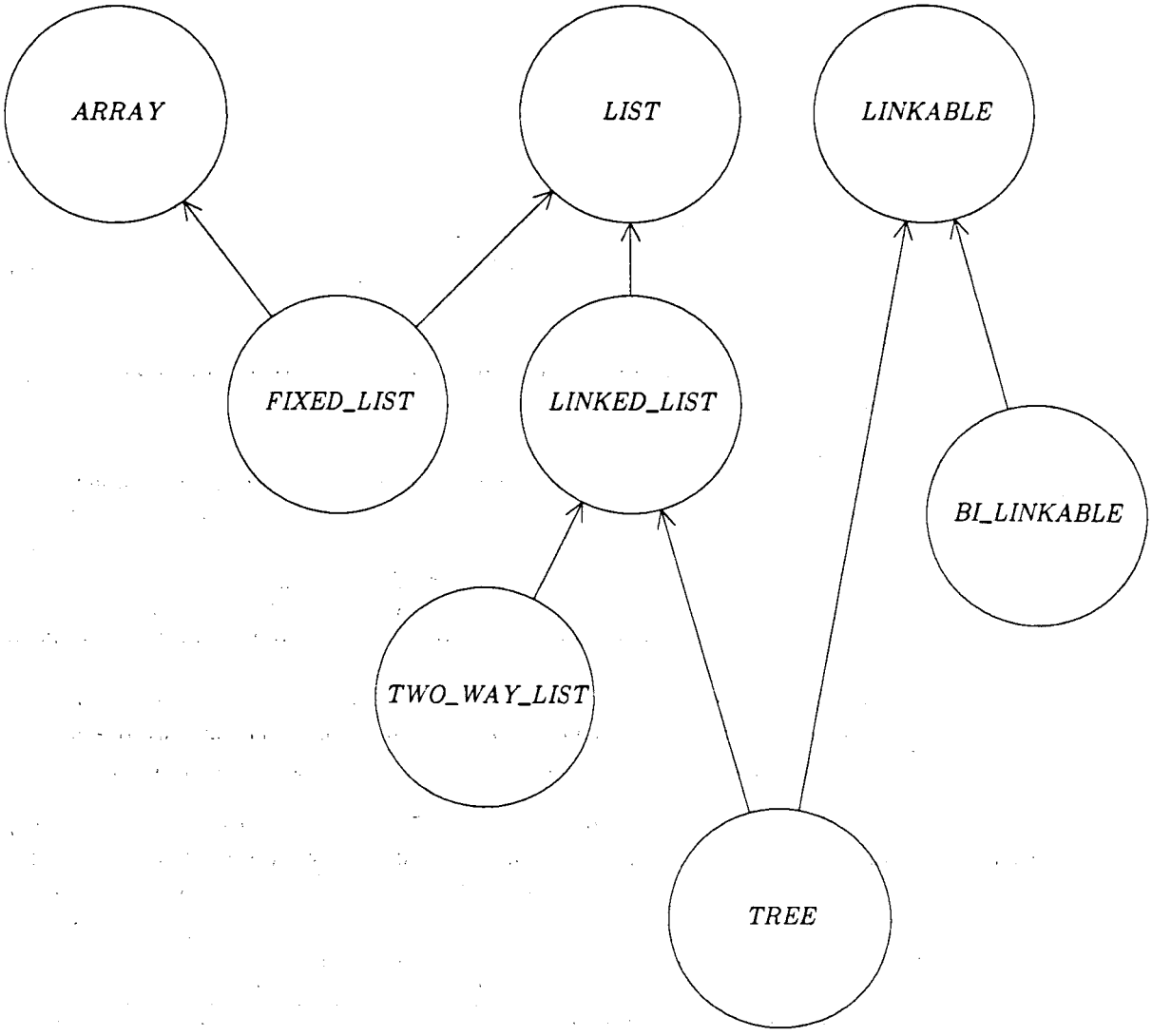

Figure 3: Inheritance graph for the examples

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.

Such is the power of multiple inheritance. Few languages have permitted it so far; it is available in Xerox's experimental Traits system [9] and, as mentioned above, in recent versions of Smalltalk.

Figure 3 gives the structure of the inheritance graph for the classes in Part 2.

We use the following terminology. An **heir** of a class C is a class which lists C in its **inherit** clause. On figure 3, this is shown by an arrow from A to B; for example, the heirs of *LINKABLE* are *BI_LINKABLE* and *TREE*. The **descendants** of a class C are C itself and the descendants of its heirs; for example, the descendants of *LIST* are *FIXED_LIST*, *LINKED_LIST*, *TWO_WAY_LIST* and *TREE*. The reverse notions are **parent** and **ancestor**.

1.5.2 - Inheritance and export controls

One aspect of inheritance worth noting here is that this mechanism is independent from export controls. Notwithstanding its export clause, a class will bequeath all its features to its descendants — lock, stock and barrel or, if you wish, 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 1.4.9 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.

We 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 the right decision.

As an example, 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], who are only interested in dealing 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 currently active list position. This feature, of type *LINKABLE* [T] , is naturally secret; it is used for the implementation of exported features such as *value* (the value at the currently active position), *insert_right* (insert a new cell of given value at the right of currently active 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 currently active list position transposes to "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, we 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).

1.5.3 - Inheritance and reusability

Why are inheritance techniques so crucial for the production of reusable software? In our opinion, what explains 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.

Let us look for example at a typical language structure used to support these methods, the data types with "variant parts" as offered by Pascal and Ada. Sure enough, such constructs make it possible to write software elements that may exist in several versions; but as soon as one needs to actually use such an element (by compiling it if it is a program element, or having it approved by management and baselined if it is some part of the design documentation), 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 parameters to a procedure, 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 as offered by Eiffel 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 one did not envision initially. This may be stated as the **principle of openness**, which we view as one of the essential laws of software design: whatever you design, keep it open for future extensions.

A further example of the application of this principle to Eiffel is the fact that the language does *not* include a construct (such as the **inspect... when...** instruction of Simula 67) which acts as a case instruction 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 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, one may use such mechanisms as deferred and redefined features, which preserve openness.

1.5.4 - Types of entities and objects

The inheritance relation may be viewed as an "is-a" relation [4], in the sense that an elephant "is-a" mammal and also "is-a" gray thing. 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 a descendant of the static type. Remember that a class is included in its own descendants, so the two may of course be the same. Whenever we talk about the type of an entity, without further qualification, we always mean its static (declared) type.

1.5.5 - Renaming and repeated inheritance

The basic Eiffel rule for resolving name clashes is simple. Within a class, there may be no name conflict (overloading): any unqualified name must denote one and only one feature. This, in our view, is essential for readability and safety.

With the emphasis on reusability and bottom-up construction, however, 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 the multiple inheritance mechanism. 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 thus in its most general form appear as:

```
class C export ..... inherit
        A
            rename f1 as g1, f2 as g2, .....;
        B
            rename m1 as n1, m2 as n2, .....;
        .....
feature
        .....
```

Class *C* will refer to the renamed features under their new names (g_1, g_2, ..., n_1, n_2, ..., etc.). The ban on overloading applies to the set of names that are visible in the class after renaming has been applied.

An interesting consequence of the renaming policy is an Eiffel concept that extends multiple inheritance: **repeated** inheritance[2]. Repeated inheritance makes it possible to inherit more than once from the same class, sharing some features and duplicating others.

Assume for example that a class *SORTABLE_LIST* [*T*] has been defined as a descendant of *LINKED_LIST* [*T*]; among the features we single out the following two:

```
nb_elements: INTEGER;
            -- The number of elements in the list
            -- (this feature comes from LINKED_LIST

sort is
            -- Sort the list
    do ... end;
```

We may want to use this class to define, say, a stock inventory list of materials, kept sorted in two different ways: from each element in the list, one may access its successors by either unit price or available stock quantity. This is achieved simply by inheriting twice from the sorted list class, and removing name clashes through **renaming** clauses, as in

```
class INVENTORY_LIST export ..... inherit
    SORTABLE_LIST [MATERIAL]
        rename sort as sort_by_price;

    SORTABLE_LIST [MATERIAL]
        rename sort as sort_by_quantity
feature ....
```

Here feature *sort* is inherited twice under different names, so *sort_by_price* and *sort_by_quantity* are really two different features for class *INVENTORY_LIST*; their effect, corresponding to their names, should indeed be different (for this to work properly, of course, other features necessary for the execution of *sort*, such as the attributes representing the element keys used in the comparisons and the procedure for swapping elements, should also be renamed separately).

---

[2] Repeated inheritance exists in the LM specification language associated with the M method [20], which in many respects may be viewed as the formal specification method associated with the use of Eiffel for design and programming. On the other hand, the semantics given by Sannella [23] as well as Burstall and Goguen [5] for another specification language, Clear [6, 7], use techniques designed to *merge* the instances of a single feature when it has been inherited more than once; thus it seems that the designers of this language overlooked the potential interest of repeated inheritance.

On the other hand, feature *nb_elements* is inherited from both sides without being renamed. This is not a violation of the "no overloading" rule, however, since no conflict is involved: *nb_elements* comes in both cases from the same class (*LINKED_LIST*). Such a feature inherited more than once, directly or indirectly, from the same class, is shared. Thus, as fits the example, attribute *nb_elements* is unique at the *INVENTORY_LIST* level.

This example is representative of the power of the mechanisms associated with inheritance. Achieving the same effect (that of a list kept sorted according to two different criteria) without these techniques involves declaring complex data structures and performing tedious and tricky pointer manipulations.

This discussion shows the two main applications of renaming:

- to remove potential name conflicts in normal *multiple* inheritance, when combining classes with identically named features;
- to replicate a common feature in *repeated* inheritance, as with the *sort* procedure above.

A third, more "cosmetic" use of renaming is to enhance clarity by providing more appropriate feature names in a descendant: for example, the boolean function which tests whether a list is empty is called *empty* for lists in the strict sense (sections 2.4 to 2.7) and renamed *is_leaf* for trees (section 2.8) to conform to usual tree terminology.

1.5.6 - 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 the C features redefined in C, under the form

inherit *C*
redefine *f, g, h ...*

allowing it to introduce new declarations of features *f, g, h, ...* which, for an object of type D, will override the corresponding declarations given in C. Some constraints, of which the most important are described in section 1.6.1, restrict the types that may be given to such redefined features and (in the case of routines) to their arguments.

Feature redefinition 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* which 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 parameters) 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 2.5) include a function feature *last*, returning the last value of a list; here one 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 2.6), 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 mechanisms: does it not allow dangerous manipulations? A feature application

$a.f$ (....),

where a is of a class type 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 1.7.3, a partially formal specification may be associated with a routine feature in terms of preconditions and postcondition. If this is the case, any redefinition of the routine must obey the initial specification (1.7.6).

1.5.7 - Redefinition vs. 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: for redefinition, "Can we have a different implementation for f when it is applied to entities of dynamic type D?"; for renaming, "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 (figure 4), follows from this discussion. We 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 our 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 the only interesting cases are the interpretations of $c.f$ when the dynamic type of c is 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 "illegal" combinations are statically so and may be caught 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.

#		*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''

Figure 4: Combining redefinition and renaming

(Note: in column 3, *c* is assumed to be of dynamic type *D*).

All cases, with the exception of case 6, occur in the library of part 2. 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* (2.7) from *LINKABLE* (2.4). To insert a new child into a tree, however, one 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 *linkable_put_between* for the occasion, followed by code to set the *parent* field.

1.5.8 - Deferred features

The redefinition mechanism allows providing alternate implementations of a previously implemented feature. In some cases, one wants to define a feature without giving its implementation, relying on descendants to provide implementations. Deferred feature declarations satisfy this need.

In such a declaration occurring in a class *C*, the type and parameters 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**.

Versions of the body, usually distinct from one another, will be given in the descendants of *C*. One may then apply the feature to an object of type *C* (under some consistency conditions), with the understanding that the implementation used depends on the descendant to which the object belongs at execution time.

In keeping with the remark made in the presentation of the redefinition mechanism, the syntax for deferred typed features without parameters, 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.

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 (1.7.3 and 1.7.6) indeed apply to deferred features.

An interesting application of deferred features is to provide for two-tier definition of modules (interface and implementation) as in Ada or Modula 2. One will declare an abstract data type implementation as two classes, the first of which contains only 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 the inheritance mechanism over the techniques of non object-based languages such as Ada or Modula 2 is that more than one implementation may be provided for a given interface within the same system.

Such a use of deferred features is particularly interesting when Eiffel is used as a design language ("PDL"). The global design of the system may be expressed as a set of classes where all non-trivial features are deferred; preconditions and postconditions should be stated whenever possible. At the implementation stage, deferred features will be expanded into actual code. Such an approach makes the development process smoother and more continuous than when different languages are used for design and implementation.

1.6 - TYPE COMPATIBILITY

1.6.1 - Basic constraints

Eiffel was designed to permit strict type checking. Because of the inheritance mechanism, the type rules are more flexible 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 1.5.4): in an assignment $x := y$, the class of y must be a descendant of the class of x. In other words, one 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. However, if the class of x is a descendant of the class of y, then *x.Extract* (y) will assign the values of the attributes of y common to both classes to the corresponding attributes of x, leaving other attributes of x untouched. This operation is only permitted if both x and y are created. (*Extract* is the last of the predefined features of the language, common to all class types, the others being *Create, Void, Forget* and *Clone*).

The second basic constraint applies to the redefinition of a typed feature, that is to say 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.

1.6.2 - 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 we define a class *COMPLEX* to represent complex numbers. One of the features may be a function *conjugate* yielding the conjugate of the current object, which might be declared 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
```

We have 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 long as we 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: IMPEDANCE; 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 we use a declaration by association whose anchor will be the current element itself. In other words, we will declare *conjugate* and *temp* 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 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.

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.

1.6.3 - Side-effects in functions

One more constraint is worth discussing here. In section 1.4.2, we introduced procedures as "commands" and functions as "questions". In keeping with this definition, it would seem wise to forbid functions from performing any operation ("side-effect") that could alter the state of the current object; such a restriction excludes procedure calls as well as assignment to attributes of the current object.

This constraint is **not**, however, a language rule. The reason is that a class is the *implementation* of an abstract data type, not the abstract data type itself. The state of an object of the class should be seen as one particular representation of a more abstract state. What is required of functions is that they do not alter the *abstract* state; however, the mapping from concrete to abstract states is not necessarily injective; there may be more than one concrete representation of a given abstract state. There are indeed cases when, for efficiency reasons, a function will have to change the concrete state without changing the abstract state.

As an example, assume we have a class representing complex numbers (or points). Depending on the operations requested on a given number, we might want to alternate between representations: multiplication, for example, is best done in polar coordinates, whereas addition demands that the cartesian representation be available. If we adopt such an implementation, which has three kinds of possible states ("cartesian representation available only", "polar only" and "both"), a function such as x, which returns the current abscissa, might trigger a state change to make the cartesian representation available if it is not. From a methodological point of view, such a change of the concrete state is acceptable, since a function call of the form $z.x$ will not change the underlying abstract state: even though some attributes of z may be changed, the representation of z still corresponds to the same complex number.

Thus the prohibition of concrete side-effects in functions, which is an easy check for a compiler, is not part of the language definition. The theoretical rule is that functions are barred from any change to the abstract state. Such a constraint, however, cannot be enforced mechanically in the absence of the proper tools for formal specification. It is a *methodological* guideline, not an a language rule.

In practice, compilers may be expected to check for side-effects anyway and produce warning messages.

1.7 - 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. Of course, these qualities are meaningless unless programs are also correct and reliable. 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 as in a "one-shot developments" environment, since the impact of errors will be multiplied by the reuse factor.

Eiffel provides no revolutionary solution to the issue of program correctness but includes language constructs that promote a systematic approach to software construction.

1.7.1 - Assertions

These constructs are based on the notion of **assertion**.

An assertion is a list of boolean expressions, separated by semicolons; a semicolon is semantically equivalent to an **and** here, but it allows individual identification of the components of the assertion. The following is an assertion:

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

We shall see below (1.7.7) that other elements may be associated with assertion components (labels, messages, actions).

Eiffel does not include a full-fledged assertion language, so some properties which 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 part 2 (a related effort, the M specification method [20], includes a specification language, LM, which may be used in conjunction with Eiffel in a fully

formal approach).

The various uses of assertions will now be described.

1.7.2 - Class invariants and the *Create* procedure

The need for class invariants arises from the already voiced remark that a class is an implementation of an abstract data type rather than the abstract data type itself. The implementation contains components (attributes) which are often too general for the purpose of representing the abstract type. As a trivial example, an array representation of stacks may contain an integer attribute, 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 non-negative. Thus the condition $high \geq 0$ should be a class invariant.

The notion of data type invariant is discussed in [13] and [14]. From the viewpoint of section 1.6.3, the class invariant characterizes the domain of the mapping from concrete to abstract states.

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.

What the simplistic example of *high* in *STACK* does not show is that for interesting classes invariants are strong semantic properties; by stating them explicitly, one gains in-depth insights into the fundamental properties of classes. Part 2 contains significant examples of class invariants, for example the invariants for *LIST* and *LINKED_LIST*.

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

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

The reader will notice in the examples of Part 2 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, one may assume the class invariant, and one 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 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 parameters). If A does not contain a *Create* procedure, A is still considered to have redefined 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 this discussion stems an important principle of Eiffel design: the purpose of *Create* procedures is to ensure that every object of a class initially satisfies the class invariant.

1.7.3 - Preconditions and postconditions

Assertions may be associated with routines: a routine may begin with a **require** clause, stating the conditions assumed 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 object has been modified since entry.

1.7.4 - Loop notation

The syntax of loops (taken from [21], chapter 3) includes room for loop initialization, a loop invariant (true after initialization and conserved by the loop body), and a variant (a non-negative integer expression which decreases on each iteration, guaranteeing termination):

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

This notation (where the **keep**... and **decrease** 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 Pascal "while" loop, with the test reversed; it is not a Pascal **repeat...until....**

1.7.5 - Check instruction

An assertion may also be used in a special instruction of the form

```
check assertion end
```

whose purpose is to express that the *assertion* 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, to 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. Part 2 contains numerous examples of such uses of **check**.

1.7.6 - Assertions and inheritance

Using assertions, one may state the restrictions that apply whenever features are added or redefined in descendants of a class. As pointed out in sections 1.5.6 and 1.5.8, 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 one strives to produce software elements which are still open to extensions and modifications, one also needs 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 **keep...** clauses except for clarity).
- Consequently, no two classes may be combined through multiple inheritance if their invariants are not compatible (note that they could only be incompatible if the two classes share some features and are thus themselves descendants of the same class).
- 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).

1.7.7 - 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. Eiffel systems should provide at least three compilation options:

1 • 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).

3 • total protection: all assertions (and the effective decrease of loop variants through each iteration) are checked.

2 • controlled mode: only preconditions of routines (**require** clauses) are checked.

Option 3 is adequate at checkout time. Option 2 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 in the current Eiffel compiler option 2 is the default.

With options 2 and 3, when an assertion is found to be violated, the Eiffel system should react as follows. In all cases, the current routine will terminate and control will be transferred back to its calling routine. (Note that since function results, like all other entities, are initialized by default, a function will return a well-specified result even if no programmer-defined instruction has been executed). Before passing control back to the calling routine, the system may produce an error message and/or perform an action; this will be the case in particular if

the programmer has taken advantage of the possibility to associate a *label*, a *message* and an *action* to components of the assertion. The syntax of assertion components, with all option present, is illustrated by the following example from a hypothetical function:

positive: $x > 0$ **message** "*Argument must be positive*" **do** *Result* := *0* **end**;

In this form, the label (*positive*) and the message will be used to produce an error report should the assertion be violated at run-time; note that the message written by the programmer does not mention the class and routine names, since it is the responsibility of the Eiffel system to identify them properly in the report actually generated. The instructions between **do** and **end** will be executed before control is returned to the calling routine. Note that the **do...end** part was probably not necessary here since the result of the function, which the assignment shows to be of a numerical type, is initialized to zero anyway.

This facility is, in our opinion, simpler and safer than a general exception handling mechanism as offered, for example, by Ada, with its potential for remote transfer of control. The facility does not disrupt the normal inter-routine flow of control; it allows catching errors, dealing with them locally and producing appropriate error messages.

Assertions and the associated language constructs should not be misused. Clauses such as **require**... and **check**... are in no way appropriate for dealing with run-time situations that fall outside the "normal" cases but are nevertheless possible and, as such, part of the specification. Examples of such situations include expected errors in the input to a program: if a certain kind of input cannot be processed normally, but the program is prepared to deal with it in some fashion, for example by outputting a message and requesting new input, then the erroneous case belongs to the specification; such cases should be dealt with through standard language constructs such as conditional instructions. This is not what assertions are for. Assertions express properties that should *always* be satisfied when the program is executed; thus violation of an assertion signifies a *program* error, not a special run-time condition. The error may have been made by the programmer who wrote the class containing the assertion; or it may be the responsibility of the writer of a client class, resulting in a routine call that failed to observe the advertised precondition.

Thus by writing assertions, the programmer is documenting his design and defending his belief in its correctness; by choosing to have these assertions monitored at run-time, he is showing his distrust not of the system's users, but of his clients and, just as importantly, of himself.

This view of assertions explains why we have not included any **message**... or **do**... clause in the library examples of part 2; such clauses are not meant to affect the official semantics of library classes, and consequently do not belong in a published version.

One more note is in order with respect to assertions. The consistency constraints on feature redefinition, mentioned in the preceding section, could only be enforced by a system including a fully formal assertion language and a theorem prover. We will have to satisfy ourselves, for some time to come, with informal human checking.

In particular, the examples of part 2 have been tested but not formally verified and we expect that some mistakes remain; we will be grateful to any reader reporting an error.

1.8 - MISCELLANEOUS

Two more explanations will help the reader understand the examples and write his own Eiffel programs.

Non-commutative 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 has 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 non-commutative 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 our 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 of simple types. Constants of class types are object references of which a single copy exists in any system (as opposed to normal attributes, of which there is an instance for every object of a class). Such constants are treated as normal attributes but included in the only clause of classes not yet presented, the **freeze...** clause, ensuring that the corresponding object is shared over a whole system. This possibility is useful in some application programs but less frequently in libraries and thus does not occur in the examples of part 2.

1.9 - ON THE IMPLEMENTATION OF EIFFEL

We finish this introduction to Eiffel with a brief overview of how the language has been implemented.

1.9.1 - Classes and systems

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

1.9.2 - Translation techniques

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

Two commands are provided.

The first command, *ec*, for Eiffel Class, compiles a single class into C and then object code. Separate compilation is of course an essential requirement for a language promoting reusability and extendibility. To compile a class, one needs 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 (param1, param2, ...)
SOURCES: dir1 dir2 .....
LIBRARIES: dir'1 dir'2 .....

Such a file describes how to assemble a system whose root is an object of type *Classname*, created with the actual parameters given (these parameters must be of simple types); the *SOURCES* directories are used to locate all the necessary classes; the *LIBRARIES* contain any needed external routines. (External routines are routines written in a language other than Eiffel; examples of use of such routines may be found in class *ARRAY*, section 2.2).

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.

1.9.3 - Binding and type-checking

Binding is the association of names to their denotations — for example, the association of attribute names to memory offsets and of routine names to actual code. A related task in a typed language is type-checking: verifying that every entity is only used in accordance with the constraints imposed by its declared type.

As mentioned in sections 1.2 and 1.6, one of the major design criteria of Eiffel was to allow **static** binding and type-checking; this is crucial for efficiency as well as safety reasons.

Almost all binding is indeed done at translation time: intra-class binding in *ec*, inter-class in *es*. The only binding that remains to be done at execution time is the binding of names of deferred or redefined features to the appropriate code. The techniques used make it possible to limit the corresponding loss in efficiency to a minimal amount.

With respect to type-checking, the language definition permits all checking to be done at compile-time; no checks are necessary at run-time (in contrast, other object-based languages either take a lax attitude towards typing or, as in the Simula case, leave some checking to be done at execution time). However our system currently performs only intra-class checking. Inter-class type-checking (to be done during assembly, by *es*) is under way.

1.9.4 - 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.

Classes are interconnected by two dependency relations: "descendant" and "client", with inverses "ancestor" and "supplier". Since a given class may be connected directly or indirectly to many others, there is a serious danger that obsolete or inadequate versions might be inadvertently used. Some automated support should be provided to avoid this risk. Commands *ec* and especially *es* address this concern by enforcing time consistency of the dependency relations.

The optional argument to command *ec* specifies where to look for ancestors of a class to be compiled separately; the *SOURCES* line in the System Description File used by command *es* specifies where to look for direct and indirect ancestors and suppliers of a system's root. In both cases, the commands check that every needed class has been re-compiled after any modification of the classes to which it is related; if not, they automatically trigger the necessary re-compilations. In particular, command *es* will itself call *ec* for classes that have been modified but not re-compiled.

Our initial implementation of these facilities relied on the Unix Make tool [11]. However Make turned out to be too limited in its capabilities and we substitued specific tools.

1.9.5 - Run-time support

The dynamic model described in section 1.4.1 implies adequate run-time support. Our Eiffel implementation relies on a complete memory management system (*Dynamem*), which provides both paging and garbage collection; the latter is done in parallel ("on-the-fly") if the operating system supports multiple user processes, and is otherwise called as a coroutine.

It is regrettable to have to resolve such issues just for the implementation of a design and programming language; the state of the art in commonly available programming environments did not leave us any other choice.

1.9.6 - Other tools

This article covers the language and the associated method rather than programming tools. However two categories of tools are important in practice and should be mentioned briefly here.

The first tool is a **class abstracter** that produces a summarized version of any class. A 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 and postcondition and the comment immediately following the header, if any, are reproduced. For example, the abstracted version of function *index_of* in lists (section 2.3 below) is:

```
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
    deferred
    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
```

The form shown is that produced by our current Eiffel class abstracter, which generates **deferred** routine bodies, so that the result of running the abstracter on a class is still a syntactically correct class.

The other necessary tool, on which we shall not elaborate any further, is a database system for keeping track of available classes and their features, and enabling Eiffel programmers to find the classes adapted to their needs.

PART 2: BASIC EIFFEL LIBRARY

2.1 - OVERVIEW

The classes given below are extracted from the basic library of classes used in our developments. 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 own hand at Eiffel programming), but they remain faithful to the original, which serves as a basis for such applications as structural (language-based) editing and multiple-windowing display management.

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

These classes illustrate the bottom-up, modular, reusable programming style (cf. [21], chapter 6) 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 such as the ones below, which can be thoroughly checked and optimized once and for all; the checking and optimization are better done here than in application programs. Such professional implementations of data abstractions may be used as the basis for "data structure programming", free from tricky pointer manipulations, as advocated by Mills [22].

Anybody who has written software involving non-trivial 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 very difficult. Coming up with correct and efficient tools that will satisfy many different needs is an exacting iterative process; we make no pretense that the classes below are in their final state. 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.

2.2 - ARRAYS

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.

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 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 have been written in C, an easy task indeed on Unix.

Similar classes exist for two- and three-dimensional arrays. Other implementations are also possible.

```
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 ; -- allocate

        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

keep
        size = upper - lower + 1
        -- area > 0 if and only if the array has been allocated
end -- class ARRAY [T]
```

2.3 - 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-based 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 element 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 secret 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 started to fade away as new features came creeping in.

Fortunately we realized our mistake, which was to treat lists as passive objects. As others would perhaps have known right away, 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).

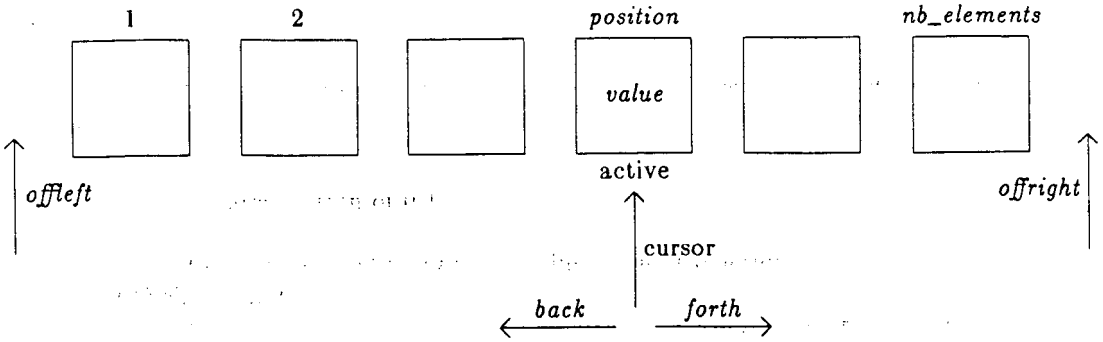

Figure 5: A list as machine

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 retrieve it later (in a last-in, first-out fashion), one will use *l.mark* and *l.retrieve*.

And so on. For a linked list, feature *active*, of type *LINKABLE* [*T*], provides access to the active element (see section 2.5); 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-specific may now be lifted (sometimes in deferred form) to the generic class *LIST*.

Apart from the author's personal shortcomings, 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 remark 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

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, retrieve,
    index_of, present,
    duplicate

feature

-- Number of list elements

    nb_elements: INTEGER;

    empty: BOOLEAN is
            -- Is the list empty?
        do
            Result := (nb_elements = 0)
        ensure
            Result = (nb_elements = 0)
        end ; -- empty

    backup: like Current  -- (SECRET: for marking and retrieving)
    no_change_since_mark: BOOLEAN -- (SECRET: for marking and retrieving)

-- Inquiring about the current 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
                Result = (position = 1);
                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)
            do
                Result := not empty and (position = nb_elements)
            ensure
                Result = (not empty and (position = nb_elements));
                not Result or not empty
            end; -- islast

                -- Complete symmetry between isfirst and islast would be achieved
                -- by writing the result of isfirst as
                    -- not empty and (position = nb_elements);
                -- however the first operand is redundant since it is implied by the first
                -- (see second clause of the class invariant).

    -- 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;
                retrieve
            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
            require
                not empty
            do
                Result := i_th (nb_elements)
            end; -- last

-- Changing list values

        change_value (v: T) is
                -- Assign v to value of current 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)
            require
                i >= 1; i <= nb_elements;  -- These conditions imply not empty
            do
                ****** Left to the reader (see function i_th above and procedure swap below) ******
            ensure
                i_th (i) = v
            end; -- change_i_th
```

```
        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);
                retrieve;
                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 i = 0 then
            go_offleft
        else
            from
                if position > i then start end
            keep
                position > 0 ; position <= i
            decrease i − position until position = i loop
                check not offright end;
                forth
            end -- loop
        end -- if
    ensure
        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
            keep
                position >= 0;
                -- k - 1 elements to the left of active position have a value equal to v
            decrease
                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 current position
        do
            backup.Clone (Current);
        end; -- mark

    retrieve is
            -- Make currently saved position active again
        require
            not backup.Void; no_change_since_mark := true
        do
            Extract (backup);
        end; -- retrieve
```

```
-- 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;
                retrieve
            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

keep
        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
```

2.4 - 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;

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)
        require
            i >= 1; i <= nb_elements; -- These conditions imply not empty
        do
            Result := entry (i)
        ensure
            -- Result = value of i-th element of the list
        end; --i_th
```

```
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)
    require
        i >= 1; i <= nb_elements; -- These conditions imply not empty
    do
        enter (i, v)
    ensure
        i_th (i) = v
    end; -- change_i_th

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 := entry (position) ; enter (position, entry (i)); enter (i, thisvalue)
    end; -- swap

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)
    require
        i >= 0; i <= nb_elements+1
    do
        position := i
    ensure
        position = i
    end; -- go
```

```
        go_offleft is
                -- Put the list in position offleft
                (Secret procedure; use go (0) in clients)
            do
                position := 0
            ensure
                offleft
            end; -- go_offleft

        duplicate: like Current is
                -- Complete clone of the list
            local
                new: like Current
            do
                new.Create (nb_elements);
                    -- new.Clone would be inappropriate here
                mark;
                from
                    start; new.start
                keep
                    -- position − 1 values have been copied
                decrease
                    nb_elements − position
                until
                    offright      -- thus new.offright too
                loop
                    new.change_value (value);
                    forth; new.forth
                end; -- loop
                retrieve; new.go (position)
            end; -- duplicate
    keep
            -- The class invariant adds nothing to the invariant of class LIST
    end -- class FIXED_LIST
```

2.5 - 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 (t: T) is
                    -- Initialize with value t
        do
            value := t
        end ; -- Create

    value: T ;

    change_linkable_value (t: T) is
                    -- Assign value of t 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_linkable_value as change_bilinkable_value,
            -- Renaming is to ensure consistent terminology;
            -- the procedure does not need redefinition.
        redefine right, change_right
```

```
        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
    keep
        right.Void or else right.left = Current ;
        left.Void or else left.right = Current ;
    end ; -- class BI_LINKABLE [T]
```

2.6 - 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: like first_element;
    previous: like first_element;
    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_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
    do
        ****** Left to the reader (go through the list, duplicating every list element) ******
        ****** (See the corresponding procedure for FIXED_LIST) ******
    end; -- duplicate
```

```
-- 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
        require
            empty or else not offleft
        local
            new: like first_element
        do
            new.Create (v); insert_linkable_left (new)
        ensure
            nb_elements = old nb_elements + 1;
            active = old active; position = old position + 1;
            not next.Void; next.value = v
        end; -- insert_left
```

```
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 := 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
    require
        not offright

    do
        ******* Left to the reader (imitate delete) ******
    ensure
        (old islast and Nochange) or else (nb_elements := nb_elements - 1);
        active = old active;
        position = old position
    end; -- delete_right
```

```
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
    require
        not offleft
    do
        ******* Left to the reader (use back and delete) ******
    ensure
        active = old active;
        (old isfirst and Nochange) or else
            ((nb_elements := nb_elements − 1) and (position = old position − 1))
    end; -- delete_left

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
    keep
            -- 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
```

2.7 - 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 export clause as in LINKED_LIST ******

        -- Some features, however, are redefined

    inherit
        LINKED_LIST [T]
            rename go as reach_from_left, wipe_out as simple_wipe_out,

            redefine
                first_element, last, back, go, wipe_out,
                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;

    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
            keep
                position <= nb_elements ; position >= i
            decrease 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?
    require
        not new.Void
    do
        ****** 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 ******
    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
                ****** Redefinition left to the reader ******
                ****** Hints: see update_after_insertion ******
                end
            end; -- update_after_deletion

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

        -- Invariant for class TWO_WAY_LIST
    keep
            -- 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
```

2.8 - 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 1.5.1, 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, retrieve,
        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;
                redefine first_child

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
```

```
        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

    keep
            -- 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]
```

PART 3: CONCLUSION

We would like to emphasize that Eiffel is a *small* language (which is not the same as "easy"); we feel that its size, to the extent that such a measure exists, is about equivalent to that of Pascal, for much more power, flexibility and safety.

The comparison is not entirely fair, since the design of Eiffel concentrated on "programming-in-the-large" features and we are quite happy, at least in the current version of the language, to rely on external C or Fortran routines (encapsulated in a few standardized basic classes) for such relatively mundane tasks as input and output. But in general paucity is one of the main properties of the design, and the language includes little redundancy and few, if any, of the "bell and whistles" found in many languages.

Non-indispensable features have been avoided: for example, we do not see any use for a "repeat...until..." loop, for a "for" loop (except when programming an SIMD machine) or for enumerated types (in a language that includes the notions of class and inheritance); arrays are not part of the language proper but may be defined (see section 2.2) as a basic class, relying on external procedures for memory allocation.

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

- The language and its translator (ETC) are being applied to the development of several software products.
- The implementation is being refined and extended.
- The basic library sketched in this article is being expanded and its scope put to test.
- Work on specific Eiffel tools (beyond the translator and the associated configuration management facilities) has not yet begun, but the document constructor Cépage [19] will have an Eiffel version.
- A separate paper [18] explores in more detail the relationship of inheritance to Ada-like genericity; work on the formal specification of Eiffel, in particular the inheritance mechanism, is also in progress.
- A formal specification method, M [20], relying on similar ideas at a more abstract level, is being further investigated.

Many (although not all) of the individual language traits present in Eiffel have appeared in other languages. We believe that the main contribution of Eiffel and the associated system is that they provide a consistent combination of a range of features which, to our knowledge, had never before been offered within a single language: object-oriented program modules based on data abstraction, multiple and repeated inheritance, genericity, information hiding, fully static typing, systematic use of assertions and invariants, separate compilation, dynamic allocation of objects with automatic garbage collection, efficient compiled code, portability through the use of a widely available intermediate target language, built-in automatic configuration management — and, more generally, an overall concern to cater to the needs of serious software practitioners in production environments.

We view Eiffel as a language for professional programmers: people who have come to appreciate the difficulties involved in software design as well as the virtues of reusability, modularity, data abstraction, genericity and assertion-guided programming; people who know that an appropriate design and programming language 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 and students at UCSB: Reynald Bouy, Vincent Cazala, Kam Chow, Michael Mansur, Susan Murphy and Jean-Marc Nerson; Deniz Yuksel, author of the first Eiffel translator, deserves special thanks (important contributions were also made by Olivier Mallet). I also grateful to Jean-Claude Derniame, Jean-Pierre Finance and Harlan D. Mills for useful 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. Didier Bert, "Manuel de Référence du Langage LPG, Version 1.2," Rapport R-408, IFIAG, IMAG Institute (Grenoble University), Grenoble, December 1983.
3. Graham Birtwistle, Ole-Johan Dahl, Bjorn Myrhaug, and Kristen Nygaard, *Simula Begin,* Studentliteratur and Auerbach Publishers, 1973.
4. Ronald J. Brachman, "What IS-A and isn't: An Analysis of Taxonomic Links in Semantic Networks," *Computer (IEEE),* vol. 16, no. 10, pp. 67-73, October 1983.
5. Rod M. Burstall and Joe A. Goguen, "The Semantics of Clear, a Specification Language," in *Proceedings of Advanced Course on Abstract Software Specifications,* pp. 292-332, Springer Lecture Notes on Computer Science, 86, Copenhagen (Denmark), 1940.
6. Rod M. Burstall and Joe A. Goguen, "Putting Theories Together to Make Specifications," in *Proceedings of 5th International Joint Conference on Artificial Intelligence,* pp. 1045-1058, Cambridge (Mass.), 1977.
7. Rod M. Burstall and Joe A. Goguen, "An Informal Introduction to Specifications using Clear," in *The Correctness Problem in Computer Science,* ed. R. S. Boyer and J. S. Moore, pp. 185-213, Springer-Verlag, New York, 1981.
8. Brad J. Cox, "Message/Objet Programming: An Evolutionary Change in Programming Technology," *IEEE Software,* vol. 1, no. 1, pp. 50-69, January 1984.
9. Gael A. Curry and Robert M. Ayers, "Experience with Traits in the Xerox Star Workstation," *IEEE Transactions on Software Engineering,* vol. SE-10, no. 5, pp. 519-527, September 1984.
10. Ole-Johan Dahl, Bjørn Myrhaug, and Kristen Nygaard, (Simula) *Common Base Language,* Norsk Regnesentral (Norwegian Computing Center), Oslo, February 1984.
11. Stuart I. Feldman, "Make - A Program for Maintaining Computer Programs," *Software, Practice and Experience,* vol. 9, pp. 255-265, 1979.
12. Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, Reading (Massachusets), 1983.
13. C.A.R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica,* vol. 1, pp. 271-281, 1972.
14. Cliff B. Jones, *Software Development: A Rigorous Approach,* Prentice-Hall, Englewood Cliffs (New-Jersey), 1980.
15. B.P. Lientz and E.B. Swanson, "Software Management: A User/Management Tug of War," *Data Management,* pp. 26-30, April 1979.
16. Bertrand 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),* no. 1, pp. 89-150, Clamart (France), 1979. Also in GROPLAN 9, AFCET, 1979
17. Bertrand Meyer and Jean-Marc Nerson, "CEPAGE, a Full-Screen Structured Editor," in *Software Engineering: Practice and Experience,* ed. Emmanuel Girard, pp. 60-65., North Oxford Academic, Oxford, 1984.
18. Bertrand Meyer, "Genericity versus inheritance," in *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications,* Portland (Oregon), September 29 - October 2, 1986. (To appear).
19. Bertrand Meyer, "Cépage: Towards Computer-Aided Design of Software," *Computer Language,* September 1986. To appear.

20. Bertrand Meyer, "M: A System Description Method," Technical Report TRCS85-15, University of California, Santa Barbara, Computer Science Department, May 1985.

21. Bertrand Meyer, *Applied Programming Methodology,* Courses notes, UC Santa Barbara, to appear as a book., 1986.

22. Harlan D. Mills, "Software Methodology and Tools," in *Softfair Conference*, San Francisco, California, December 3-5, 1985. (Verbal presentation).

23. D.T. Sannella, "A Set-Theoretic Semantics for Clear," *Acta Informatica*, vol. 21, no. 5, pp. 443-472, December 1984.

24. Bjarne Stroustrup, "Data Abstraction in C," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, Part 2, pp. 1701-1732, October 1984.

25. Larry Tesler, "Object Pascal Report," *Structured Language World*, vol. 9, no. 3, 1985.