B

Genericity versus inheritance

The material that follows, and its appearance in an appendix, deserve some background explanation. Part of the original impetus for the work that eventually led to this book was a study that I performed in 1984; in preparation for a graduate course that I was to teach on "*advanced concepts in programming languages*", I compared the "horizontal" module extension mechanism of genericity, illustrated by Ada, Z, LPG and other generic languages, with the "vertical" mechanism of inheritance introduced by Simula: how these techniques differ, to what extent they compete, and to what extent they complement each other. This led to an article on "Genericity versus Inheritance" [M 1986], presented at the first OOPSLA conference, and to a chapter in the first edition of the present book.

When preparing this new edition I felt that both genericity and inheritance were now understood well enough, and their treatment detailed enough in the rest of the book, to make the chapter appear too specialized: useful mostly to readers interested in issues of language design or O-O theory. So I removed it. But then I found out that a regular flow of articles in the software press still showed must puzzlement over the issue, especially in the context of C++ for which many people seem to be searching for general guidelines on when to use "templates" and when to use inheritance. This means the discussion still has its place in a general presentation of object technology, although it is perhaps best severed from the main part of the text. Hence this appendix.

The topics reviewed are, in order: genericity; inheritance; how to emulate each of these mechanisms through the other; and, as a conclusion, how best to reconcile them.

If you have read carefully the remainder of this book, you will find the beginning of this discussion familiar since we must restart with the basics to get a full picture of each mechanism, of its contribution, and of its limitations. As we probe deeper and deeper, perhaps stepping briefly into a few dead ends along the way, the ideal combination of genericity and inheritance will progressively unfold before our eyes, imposing itself in the end as almost inevitable and letting us understand, in full detail, the fascinating relationship between the two principal methods for making software modules open to variation and adaptation.

B.1 GENERICITY

We begin our review by appraising the merits of genericity as it exists in a number of languages, object-oriented or not. Let us rely for convenience on the notations — semicolons and all — of the best known non-O-O generic language, Ada (meaning by default, as elsewhere in this book, Ada 83). So for the rest of this section we forget about O-O languages and techniques.

Only the most important form of Ada genericity will be considered: *type parameterization*, that is to say the ability to parameterize a software element (in Ada, a package or routine) by one or more types. Generic parameters have other, less momentous uses in Ada, such as parameterized dimensions for arrays. We may distinguish between *unconstrained* genericity, imposing no specific requirement on generic parameters, and *constrained* genericity, whereby a certain structure is required.

Unconstrained genericity

Unconstrained genericity removes some of the rigidity of static typing. A trivial example is a routine (in a language with Ada-like syntax but without explicit type declarations) to swap the values of two variables:

This extract and the next few are in Ada or Ada-like syntax.

```
procedure swap (x, y) is
        local t;
begin
        t := x; x := y; y := t;
end swap;
```

This form does not specify the types of the elements to be swapped and of the local variable *t*. This is too much freedom, since a call *swap* (*a, b*), where *a* is an integer and *b* a character string, will not be prohibited even though it is probably an error.

To address this issue, statically typed languages such as Pascal and Ada require developers to declare explicitly the types of all variables and formal arguments, and enforce a statically checkable type compatibility constraint between actual and formal arguments in calls and between source and target in assignments.The procedure to exchange the values of two variables of type *G* becomes:

```
procedure G_swap (x, y: in out G) is
        t: G;
begin
        t := x; x := y; y := t;
end swap;
```

Demanding that *G* be specified as a single type averts type incompatibility errors, but in the constant haggling between safety and flexibility we have now erred too far away from flexibility: to correct the lack of safety of the first solution, we have made the solution inflexible. We will need a new procedure for every type of elements to be exchanged, for example *INTEGER_swap*, *STRING_swap* and so on. Such multiple declarations lengthen and obscure programs. The example chosen is particularly bad since all the declarations will be identical except for the two occurrences of *G*.

Static typing may be considered too restrictive here: the only real requirement is that the two actual arguments passed to any call of *swap* should be of the same type, and that their type should also be applied to the declaration of the local variable *t*. It does not matter what this type actually is as long as it satisfies these properties.

> In addition the arguments must be passed in **in out** mode, so that the procedure can change their values. This is permitted in Ada.

Genericity provides a tradeoff between too much freedom, as with untyped languages, and too much restraint, as with Pascal. In a generic language you may declare *G* as a generic parameter of *swap* or an enclosing unit. Ada indeed offers generic routines, along with the generic packages described in chapter 33. In quasi-Ada you can write:

```
generic
        type G is private;
procedure swap (x, y: in out G) is
        t: G;
begin
        t := x; x := y; y := t;
end swap;
```

The only difference with real Ada is that you would have to separate interface from implementation, as explained in the chapter on Ada. Since information hiding is irrelevant for the discussion in this chapter, interfaces and implementations will be merged for ease of presentation.

The **generic**... clause introduces type parameters. By specifying *G* as "private", the writer of this procedure allows himself to apply to entities of type *G* (*x*, *y* and *t*) operations available on all types, such as assignment or comparison, and these only.

The above declaration does not quite introduce a routine but rather a routine pattern; to get a directly usable routine you will provide actual type parameters, as in

```
procedure int_swap is new swap (INTEGER);
procedure str_swap is new swap (STRING);
```

etc. Now assuming that *i* and *j* are variables of type *INTEGER*, *s* and *t* of type *STRING*, then of the following calls

```
int_swap (i, j); str_swap (s, t); int_swap (i, s); str_swap (s, j); str_swap (i, j);
```

all but the first two are invalid, and will be rejected by the compiler.

More interesting than parameterized routines are parameterized packages. As a minor variation of our usual stack example, consider a queue package, where the operations on a queue (first-in, first out) are: add an element; remove the oldest element added and not yet removed; get its value; test for empty queue. The interface is:

```
generic
        type G is private;
package QUEUES is
        type QUEUE (capacity: POSITIVE) is private;
        function empty (s: in QUEUE) return BOOLEAN;
        procedure add (t: in G; s: in out QUEUE);
        procedure remove (s: in out QUEUE);
        function oldest (s: in QUEUE) return G;
private
        type QUEUE (capacity: POSITIVE) is
                -- The package uses an array representation for queues
            record
                implementation: array (0 .. capacity) of G;
                count: NATURAL;
            end record;
end QUEUES;
```

Again this does not define a package but a package pattern; to get a directly usable package you will use generic derivation, as in

```
package INT_QUEUES is new QUEUES (INTEGER);
package STR_QUEUES is new QUEUES (STRING);
```

Note again the tradeoff that generic declarations achieve between typed and untyped approaches. *QUEUES* is a pattern for modules implementing queues of elements of all possible types *G*, while retaining the possibility to enforce type checks for a specific *G*, so as to rule out such unholy combinations as the insertion of an integer into a queue of strings.

The form of genericity illustrated by both of the examples seen so far, swapping and queues, may be called *unconstrained* since there is no specific requirement on the types that may be used as actual generic parameters: you may swap the values of variables of any type and create queues of values of any type, as long as all the values in a given queue are of the same type.

Other generic definitions, however, only make sense if the actual generic parameters satisfy some conditions. This form may be called *constrained* genericity.

Constrained genericity

As in the unconstrained case, the examples of constrained genericity will include both a routine and a package.

Assume first you need a generic function to compute the minimum of two values. You can try the pattern of *swap*:

```
generic
        type G is private;
function minimum (x, y: G) return G is begin
            if x <= y then return x; else return y; end if;
end minimum;
```

From here on most routine declarations omit the **in** *mode specification for arguments, which is optional.*

Such a function declaration, however, does not always make sense; only for types *G* on which a comparison operator <= is defined. In a language that enhances security through static typing, we want to enforce this requirement at compile time, not wait until run time. We need a way to specify that type *G* must be equipped with the right operation.

In Ada this will be written by treating the operator <= as a generic parameter of its own. Syntactically it is a function; as a syntactic facility, it is possible to invoke such a function using the usual infix form if it is declared with a name in double quotes, here "<=". Again the following declaration becomes legal Ada if the interface and implementation are taken apart.

```
generic
        type G is private;
        with function "<=" (a, b: G) return BOOLEAN is <>;
function 0(x, y: G) return G is begin
                if x <= y then return x; else return y end if;
end minimum;
```

The keyword **with** introduces generic parameters representing routines, such as "<=".

You may perform a generic derivation *minimum* for any type, say *T1*, such that there exists a function, say *T1_le*, of signature **function** (*a*, *b*: *T1*) **return** *BOOLEAN*:

```
function T1_minimum is new minimum (T1, T1_le);
```

If function *T1_le* is in fact called "<=", more precisely if its name and type signature match those of the corresponding formal routine, then you do not need to include it in the list of actual parameters to the generic derivation. So because type *INTEGER* has a predefined "<=" function with the right signature, you can simply declare

```
function int_minimum is new minimum (INTEGER);
```

This use of default routines with matching names and types is made possible by the clause **is** <> in the declaration of the formal routine, here "<=". Operator overloading, as permitted (and in fact encouraged) by Ada, plays an essential role: many different types will have a "<=" function.

This discussion of constrained genericity for routines readily transposes to packages. Assume you need a generic package for handling matrices of objects of any type *G*, with matrix sum and product as basic operations. Such a definition only makes sense if type *G* has a sum and a product of its own, and each of these operations has a zero element; these features of *G* will be needed in the implementation of matrix sum and product. The public part of the package may be written as follows:

```
generic
        type G is private;
        zero: G;
        unity: G;
        with function "+" (a, b: G) return G is <>;
        with function "*" (a, b: G) return G is <>;
```

```
package MATRICES is
    type MATRIX (lines, columns: POSITIVE) is private;
    function "+" (m1, m2: MATRIX) return MATRIX;
    function "*" (m1, m2: MATRIX) return MATRIX;
private
    type MATRIX (lines, columns: POSITIVE) is
        array (1 .. lines, 1 .. columns) of G;
end MATRICES;
```

Typical generic derivations are:

```
package INTEGER_MATRICES is new MATRICES (INTEGER, 0, 1);

package BOOLEAN_MATRICES is
    new MATRICES (BOOLEAN, false, true, "or", "and");
```

Again, you may omit actual parameters corresponding to formal generic routines (here "+" and "*") for type *INTEGER*, which has matching operations; but you will need them for *BOOLEAN*. (It is convenient to declare such parameters last in the formal list; otherwise keyword notation is required in derivations that omit the corresponding actuals.)

It is interesting here to take a look at the body (implementation) of such a package:

```
package body MATRICES is
    … Other declarations …
    function "*" (m1, m2: G) is
        result: MATRIX (m1'lines, m2'columns);
    begin
        if m1'columns /= m2'lines then
            raise incompatible_sizes;
        end if;
        for i in m1'RANGE(1) loop
            for j in m2'RANGE(2) loop
                result (i, j) := zero;
                for k in m1'RANGE(2) loop
                    result (i, j) := result (i, j) + m1 (i, k) * m2 (k, j)
                end loop;
            end loop;
        end loop;
        return result
    end "*";
end MATRICES;
```

This extract relies on some specific features of Ada:

- For a parameterized type such as *MATRIX (lines, columns: POSITIVE)*, a variable declaration must provide actual parameters, e.g. *mm: MATRIX (100, 75)*; you may then retrieve their values using apostrophe notation, as in *mm'lines* which in this case has value 100.

- If *a* is an array, *a'RANGE*(*i*) denotes the range of values in its *i*-th dimension; for example *m1'RANGE*(*1*) above is the same as *1 .. m1'lines*.
- If requested to multiply two dimension-wise incompatible matrices, the extract raises an exception, corresponding to the violation of an implicit precondition.

The minimum and matrix examples are representative of Ada techniques for constrained genericity. They also show a serious limitation of these techniques: only syntactic constraints can be expressed. All that a programmer may require is the presence of certain routines ("<=", "+", "*" in the examples) with given types; but the declarations are meaningless unless the routines also satisfy some semantic constraints. Function *minimum* only makes sense if "<=" is a total order relation on *G*; and to produce a generic derivation of *MATRICES* for a type *G*, you should make sure that operations "+" and "*" have not just the right signature, $G \times G \to G$, but also the appropriate properties: associativity, distributivity, *zero* a zero element for "+" and *unity* for "*" etc. We may use the mathematical term **ring** for a structure equipped with operations enjoying these properties.

B.2 INHERITANCE

So much for pure genericity. The other term of the comparison is inheritance. To contrast it with genericity, consider the example of a general-purpose module library for files. First here is the outline of an implementation of "special files" in the Unix sense, that is to say, files associated with devices:

This extract and the next few are in the O-O notation of the rest of this book.

```
class DEVICE feature
        open (file_descriptor: INTEGER) is do … end
        close is do … end
        opened: BOOLEAN
end -- class DEVICE
```

An example use of this class is:

```
d1: DEVICE; f1: INTEGER; …
!! d1.make; d1.open (f1);
if d1.opened then …
```

Consider next the notion of a tape device. For the purposes of this discussion, a tape unit has all the properties of devices, as represented by the three features of class *DEVICE*, plus the ability to rewind its tape. Rather than building a class from scratch, we may use inheritance to declare class *TAPE* as an extension-cum-modification of *DEVICE*. The new class extends *DEVICE* by adding a new procedure *rewind*, describing a mechanism applicable to tapes but not necessarily to other devices; and it modifies some of *DEVICE*'s properties by providing a new version of *open*, describing the specifics of opening a device that happens to be a tape drive.

Objects of type *TAPE* automatically possess all the features of *DEVICE* objects, plus their own (here *rewind*). Class *DEVICE* could have more heirs, for example *DISK* with its own specific features such as direct access read.

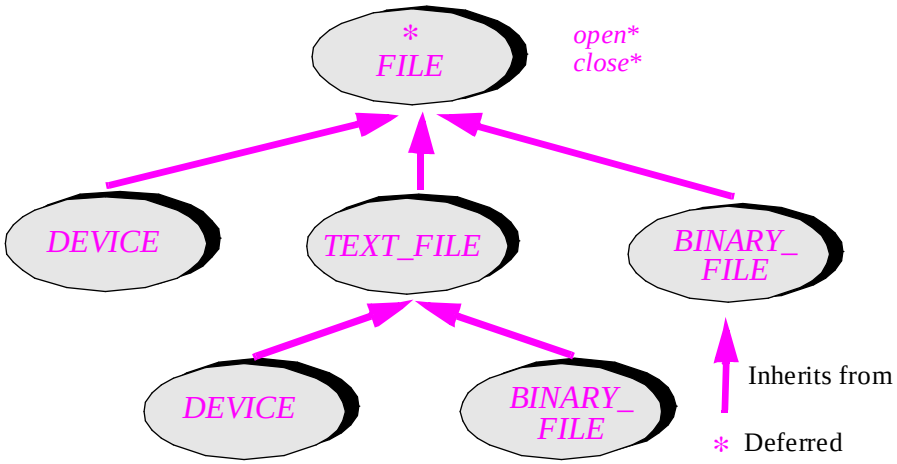
Objects of type *TAPE* will possess all the features of type *DEVICE*, possibly adapted (in the case of *open*), and complemented by the new feature *rewind*.

With inheritance comes polymorphism, permitting assignments of the form $x := y$, but only if the type of x is an ancestor of the type of y. The next associated property is dynamic binding: if x is a device, the call $x.open\ (f1)$ will be executed differently depending on the assignments performed on x before the call: after $x := y$, where y is a tape, the call will execute the tape version.

This is approximate terminology; "is an ancestor of" stands for "conforms to". Precise rules appear in earlier chapters.

We have seen the remarkable benefits of these inheritance techniques for reusability and extendibility. A key aspect was the Open-Closed principle: a software element such as *DEVICE* is both usable as it stands (it may be compiled as part of an executable system) and still amenable to extensions (if used as an ancestor of new classes).

Next come deferred features and classes. Here we note that Unix devices are a special kind of file; so you may make *DEVICE* an heir to class *FILE*, whose other heirs might include *TEXT_FILE* (itself with heirs *NORMAL* and *DIRECTORY*) and *BINARY_FILE*. The figure shows the inheritance graph, a tree in this case.



A simple inheritance hierarchy, with deferred and effective classes

Although it is possible to open or close any file, how these operations are performed depends on whether the file is a device, a directory etc. So *FILE* is a deferred class with deferred routines *open* or *close*, making descendants responsible for implementing them:

```
deferred class FILE feature
        open (file_descriptor: INTEGER) is deferred end
        close is deferred end;
end -- class FILE
```

Effective descendants of *FILE* will provide effective implementations of *open* and *close*.

B.3 EMULATING INHERITANCE WITH GENERICITY

To compare genericity with inheritance, we will study how, if in any way, the effect of each feature may be emulated in a language offering the other.

First consider a language such as Ada (again meaning Ada 83), offering genericity but not inheritance. Can it be made to achieve the effects of inheritance?

The easy part is name overloading. Ada, as we know, allows reusing the same routine name as many times as needed for operands of different types; so you can define types such as *TAPE*, *DISK* and others, each with its own version of the routines:

This extract and the next few are in Ada syntax.

```
procedure open (p: in out TAPE; descriptor: in INTEGER);
procedure close (p: in out DISK);
```

No ambiguity will arise if the routines are distinguished by the type of at least one operand. But this solution does not provide polymorphism and dynamic binding, whereby *d*.*close*, for example, would have a different effect after assignments *d* := *di* and *d* := *ta*, where *di* is a *DISK* and *ta* a *TAPE*.

To obtain the same effect, you have to use records with variant fields: define

```
type DEVICE (unit: DEVICE_TYPE) is
        record
                … Fields common to all device types …
                case unit is
                        when tape => … fields for tape devices …;
                        when disk => … fields for disk devices …;
                        … Other cases …;
                end case
        end record
```

where *DEVICE_TYPE* is an enumerated type with elements *tape*, *disk* etc. Then there would be a single version of each the procedures on devices (*open*, *close* etc.), each containing a case discrimination of the form

```
case d'unit is
        when tape => … action for tape devices …;
        when disk => … action for disk devices …;
        … other cases …;
end case
```

See "Single Choice", page 61.

This uses explicit discrimination in each case, and closes off the list of choices, forcing every routine to know of all the possible variants; addition of new cases will cause changes to all such routines. The Single Choice principle expressly warned against such software architectures.

So the answer to the question of this section is essentially no:

Emulating inheritance

It appears impossible to emulate inheritance through genericity.

B.4 EMULATING GENERICITY WITH INHERITANCE

Let us see if we will have more luck with the reverse problem: can we achieve the effect of Ada-style genericity in an object-oriented language with inheritance?

The O-O notation introduced in earlier chapters does provide a generic parameter mechanism. But since we are comparing pure genericity versus pure inheritance, the rule of the game for some time, frustrating as it may be, is to pretend we have all but forgotten about that genericity mechanism. As a result the solutions presented in this section will be substantially more complex than those obtainable with the full notation, described in the rest of this book and in later sections. As you read this section, remember that the software extracts are not final forms, but for purposes of discussion only.

Surprisingly perhaps, the simulation turns out to be easier, or at least less artificial, for the more sophisticated form of genericity: constrained. So we begin with this case.

Emulating constrained genericity: overview

The idea is to associate a class with a constrained formal generic type parameter. This is a natural thing to do since a constrained generic type may be viewed, together with its constraining operations, as an abstract data type. Consider for example the Ada generic clauses in our two constrained examples, minimum and matrices:

This extract is in Ada syntax.

```
generic
        type G is private;
        with function "<=" (a, b: G) return BOOLEAN is <>

generic
        type G is private;
        zero: G; unity: G;
        with function "+" (a, b: G) return G is <>;
        with function "*" (a, b: G) return G is <>;
```

We may view these clauses as the definitions of two abstract data types, *COMPARABLE* and *RING_ELEMENT*; the first is characterized by a comparison operation "<=", and the second by features *zero*, *unity*, "+" and "*".

In an object-oriented language, such types may be directly represented as classes. We cannot define these classes entirely, for there is no universal implementation of "<=", "+" etc.; rather, they are to be used as ancestors of other classes, corresponding to actual generic parameters. Deferred classes provide exactly what we need:

This extract and all remaining ones are in the O-O notation of this book.

```
deferred class COMPARABLE feature
        infix "<=" (other: COMPARABLE): BOOLEAN is deferred end
end -- class COMPARABLE
```

```
deferred class RING_ELEMENT feature
    infix "+" (other: like Current): like Current is
        deferred
        ensure
            equal (other, zero) implies equal (Result, Current)
        end;
    infix "*" (other: like Current): like Current is deferred end
    zero: like Current is deferred end
    unity: like Current is deferred end
end -- class RING_ELEMENT
```

Unlike Ada, the O-O notation allows us here to express abstract semantic properties, although only one of them has been included as an example (the property that $x + 0 = x$ for any x, appearing as a postcondition of **infix** "+").

"ANCHORED DECLARATION", 16.7, page 599.

The use of anchored types (**like** *Current*) makes it possible to avoid some improper combinations, as explained for the *COMPARABLE* example next. At this stage replacing all such types by *RING_ELEMENT* would not affect the discussion.

Constrained genericity: routines

We can write a routine such as *minimum* by specifying its arguments to be of type *COMPARABLE*. Based on the Ada pattern, the function would be declared as

```
minimum (one: COMPARABLE; other: like one): like one is
        -- Minimum of one and other
    do ... end
```

In O-O development, however, every routine appears in a class and is relative to the current instance of that class; we may include *minimum* in class *COMPARABLE*, argument *one* becoming the implicit current instance. The class becomes:

COMPARABLE becomes a "behavior class", with an effective feature relying on a deferred one. See "Don't call us, we'll call you", page 505.

```
deferred class COMPARABLE feature
    infix "<=" (other: like Current): BOOLEAN is
            -- Is current object less than or equal to other?
        deferred
        end
    minimum (other: like Current): like Current is
            -- Minimum of current object and other
        do
            if Current <= other then Result := Current else Result := other end
        end
end -- class COMPARABLE
```

To compute the minimum of two elements, you must declare them of some effective descendant type of *COMPARABLE*, for which **infix** "<=" has been effected, such as

```
class INTEGER_COMPARABLE inherit
        COMPARABLE
creation
        put
feature -- Initialization
        put (v: INTEGER) is
                        -- Initialize from v.
                do item := new end
feature -- Access
        item: INTEGER;
                -- Value associated with current object
feature -- Basic operations
        infix "<=" (other: like Current): BOOLEAN is
                        -- Is current object less than or equal to other?
                do Result := (item <= other.item) end;
end -- class INTEGER_COMPARABLE
```

To find the minimum of two integers, you may now apply function *minimum* to entities *ic1* and *ic2*, whose type is not *INTEGER* but *INTEGER_COMPARABLE*:

```
ic3 := ic1.minimum (ic2)
```

To use the generic **infix** "<=" and *minimum* functions, you must renounce direct references to integers, using *INTEGER_COMPARABLE* entities instead; hence the need for attribute *item* and routine *put* to access and modify the associated integer values. You will introduce a similar heirs of *COMPARABLE*, such as *STRING_COMPARABLE*, and *REAL_COMPARABLE*, for each type requiring a version of *minimum*.

Note that the mechanism of anchored declaration is essential to ensure type correctness. If the argument to *minimum* in *COMPARABLE* had been declared as a *COMPARABLE*, rather than **like** *Current*, then the following call would be valid:

```
ic1.minimum (c)
```

even if *c* is a *COMPARABLE* but not an *INTEGER_COMPARABLE*. Clearly, such a call should be disallowed. This also applies to the previous example, *RING_ELEMENT*.

Having to declare features *item* and *put* for all descendants of *COMPARABLE*, and hence sacrificing the direct use of simple types, is unpleasant. There is also a performance cost: rather than manipulating integers or strings we must create and use **wrapper objects** of types such as *INTEGER_COMPARABLE*. But by paying this fixed price in both ease of use and efficiency we do achieve the full emulation of constrained genericity by inheritance. (In the final notation, of course, there will be no price at all to pay.)

Emulating constrained genericity (1)

It is possible to emulate constrained genericity through inheritance, by using wrapper classes and the corresponding wrapper objects.

Constrained genericity: packages

The previous discussion transposes to packages. To emulate the matrix abstraction which Ada implemented through the *MATRICES* package, we can use a class:

```
class MATRIX feature
        anchor: RING_ELEMENT is do end
        implementation: ARRAY2 [like anchor]
        item (i, j: INTEGER): like anchor is
                        -- Value of (i, j) entry
                do Result := implementation.item (i, j) end
        put (i, j: INTEGER; v: like anchor) is
                        -- Assign value v to entry (i, j).
                do implementation.put (i, j, v) end
        infix "+" (other: like Current): like Current is
                        -- Matrix sum of current matrix and other
                local
                        i, j: INTEGER
                do
                        !! Result.make (…)
                        from i := … until … loop
                                from j := … until … loop
                                        Result. put ((item (i, j) + other.item (i, j)), i, j)
                                        j := j + 1
                                end
                                i := i + 1
                        end
                end
        infix "*" (other: like Current): like Current is
                        -- Matrix product of current matrix by other
                local … do … end
end -- class MATRIX
```

The type of the argument to *put* and of the result of *item* raises an interesting problem: it should be *RING_ELEMENT*, but redefined properly in descendant classes. Anchored declaration is the solution; but here for the first time no attribute of the class seems to be available to serve as anchor. This should not stop us, however: we declare an **artificial anchor**, called *anchor*. Its only purpose is to be redefined to the proper descendant types of *RING_ELEMENT* in future descendants of *MATRIX* (that is to say, to *BOOLEAN_RING* in *BOOLEAN_MATRIX* etc.), so that all associated entities will follow. To avoid any space penalty in instances, *anchor* is declared as a function rather than an attribute. This technique of artificial anchors is useful to preserve type consistency when, as here, there is no “natural” anchor among the attributes of the class.

A few loop details have been left out, as well as the body of **infix** "*", but they are easy to fill in. Features *put* and *item* as applied to *implementation* will come from the library class *ARRAY2* describing two-dimensional arrays.

To define the equivalent of the Ada generic package derivation shown earlier

```
package BOOLEAN_MATRICES is
        new MATRICES (BOOLEAN, false, true, "or", "and");
```

we must first declare the "ring element" corresponding to booleans:

```
class BOOLEAN_RING_ELEMENT inherit
        RING_ELEMENT
                redefine zero, unity end
creation
        put
feature -- Initialization
        put (v: BOOLEAN) is
                        -- Initialize from v.
                do item := v end
feature -- Access
        item: BOOLEAN
feature -- Basic operations
        infix "+" (other: like Current): like Current is
                        -- Boolean addition: or
                do !! Result.put (item or other.item) end
        infix "*" (other: like Current): like Current is
                        -- Boolean multiplication: and
                do !! Result.put (item and other.item) end
        zero: like Current is
                        -- Zero element for boolean addition
                once !! Result.put (False) end
        unity: like Current is
                        -- Zero element for boolean multiplication
                once !! Result.put (True) end
end -- class BOOLEAN_RING_ELEMENT
```

Note how *zero* and *unity* are effected as once functions.

Then to obtain an equivalent to the Ada package derivation, just define an heir *BOOLEAN_MATRIX* of *MATRIX*, where you only need to redefine *anchor*, the artificial anchor; all the other affected types will follow automatically:

```
class BOOLEAN_MATRIX inherit
        MATRIX
                redefine anchor end
feature
        anchor: BOOLEAN_RING_ELEMENT
end -- class BOOLEAN_MATRIX
```

See the box on page 1178.

This construction achieves the effect of constrained genericity using inheritance, confirming for packages the emulation result initially illustrated for routines.

Unconstrained genericity

The mechanism for simulating unconstrained genericity is the same; we can simply treat this case as a special form of constrained genericity, with an empty set of constraints. As above, formal type parameters will be interpreted as abstract data types, but here with no relevant operations. The technique works, but becomes rather heavy to apply since the dummy types do not correspond to any obviously relevant data abstraction.

Let us apply the previous technique to both our unconstrained examples, swap and queue, beginning with the latter. We need a class, say *QUEUABLE*, describing objects that may be added to and retrieved from a queue. Since this is true of any object, the class has no other property than its name:

```
class QUEUABLE end
```

We may now declare a class *QUEUE*, whose operations apply to *QUEUABLE* objects. (Remember that this class is not offered as a paragon of good O-O design: we are still voluntarily playing with an impoverished version of the O-O notation, devoid of genericity.) Routine postconditions have been left out for brevity. Although in principle function *item* could serve as an anchor, its body will not change in descendants, so it is better to use an artificial anchor *item_anchor* to avoid having to redefine *item*.

```
indexing
        description: "First-in-first out queues, implemented through arrays"
class QUEUE creation
        make
feature -- Initialization
        make (m: INTEGER) is
                        -- Create queue with space for m items.
                require
                        m >= 0
                do
                        !! implementation.make (1, m); capacity := m
                        first := 1; next := 1
                end
```

```
feature -- Access
    capacity, first, next, count: INTEGER
    item: like item_anchor is
            -- Oldest element in queue
        require
            not empty
        do
            Result := implementation.item (first)
        end
feature -- Status report
    empty: BOOLEAN is
            -- Is queue empty?
        do Result := (count = 0) end
    full: BOOLEAN is
            -- Is representation full?
        do Result := (count = capacity) end
feature -- Element change
    put (x: like item_anchor) is
            -- Add x at end of queue
        require
            not full
        do
            implementation.put (x, next); count := count + 1; next := successor (next)
        end
    remove is
            -- Remove oldest element
        require
            not empty
        do
            first := successor (first); count := count – 1
        end
```

```
feature {NONE} -- Implementation
        item_anchor: QUEUABLE is do end
        implementation: ARRAY [like item_anchor]
        successor (n: INTEGER): INTEGER is
                        -- Next value after n, cyclically in the interval 1 .. capacity
                require
                        n >= 1; n <= capacity
                do
                        Result := (n \\ capacity) + 1
                end
invariant
        0 <= count; count <= capacity; first >= 1; next >= 1
        (not full) implies ((first <= capacity) and (next <= capacity))
        (capacity = 0) implies full
        -- Items, if any, appear in array positions first, … next – 1 (cyclically)
end -- class QUEUE
```

For an alternative technique see e.g. "A buffer is a separate queue", page 990.

Bounded queue implementations elsewhere in this book rely on the technique of keeping one position open. Here, we allocate *capacity* elements and keep track of *count*. There is no particular reason, other than to illustrate alternative implementation techniques.

To get the equivalent of generic derivation (so as to obtain queues of a specific type) you must, as with the *COMPARABLE* example, define descendants of *QUEUABLE*:

```
class INTEGER_QUEUABLE inherit
        QUEUABLE
creation
        put
feature -- Initialization
        put (n: INTEGER) is
                        -- Initialize from n.
                do item := n end
feature -- Access
        item: INTEGER
feature {NONE} -- Implementation
        item_anchor: INTEGER is do end
end -- class INTEGER_QUEUABLE
```

and similarly *STRING_QUEUABLE* etc.; then declare the corresponding descendants of *QUEUE*, redefining *item_anchor* appropriately in each.

Emulating unconstrained genericity

It is possible to emulate unconstrained genericity through inheritance, by using wrapper classes and the corresponding wrapper objects.

B.5 COMBINING GENERICITY AND INHERITANCE

It appears from the previous discussion that inheritance is the more powerful mechanism since we have not found a reasonable way to simulate it with genericity. In addition:

- You can express the equivalent of generic routines or packages in a language with inheritance, but this requires some duplication and complication. The verbosity is particularly hard to justify for unconstrained genericity, which requires just as much emulation effort even though it is theoretically simpler.
- Type checking introduces difficulties in the use of inheritance to emulate genericity.

Anchored declaration solves the second problem. (The reader familiar with the detailed discussion of typing in an earlier chapter will, however, have noted the potential for system validity problems, which we do not need to explore further since they will disappear in the solutions finally retained below.) *Chapter 17.*

Let us see how we can solve the first problem by introducing (reintroducing, that is) the appropriate form of genericity.

Unconstrained genericity

Since the major complication arises for unconstrained genericity even though it should be the simpler case, it seems adequate to provide a specific genericity mechanism for this case, avoiding the need to rely on inheritance. Consequently, we allow our classes to have unconstrained generic parameters: as we are now (at last) allowed to remember from earlier chapters, a class may be defined as

class *C* [*G, H, …*] …

where the parameters represent arbitrary types. To obtain a directly usable type you use a generic derivation, using types as actual generic parameters:

x: *C* [*DEVICE, RING_ELEMENT, …*]

This immediately applies to the queue class, which we can simply declare as

```
indexing
        description: "First-in-first out queues, implemented through arrays"
class QUEUE [G] creation
        … The rest as before, but removing the declaration of item_anchor
            and replacing all occurrences of type like item_anchor by G …
end -- class QUEUE
```

We get rid of class *QUEUABLE* as well as *INTEGER_QUEUABLE* and other such descendants; to have a queue of integers, we simply use type *QUEUE* [*INTEGER*], manipulating integers directly rather than through intermediate wrapper objects.

This is a remarkable simplification, suggesting that in spite of the theoretical possibility of emulating unconstrained genericity through inheritance, it is desirable in practice to introduce a generic mechanism into the object-oriented framework.

Providing unconstrained genericity

Along with inheritance, it is desirable to provide a specific notation for declaring classes as generic (unconstrained).

Constrained genericity

For constrained genericity we can explore the same general scheme. In the matrix example:

```
class MATRIX [G] feature
        anchor: RING_ELEMENT [G]
        …Other features as before …
end -- class MATRIX
```

with ring elements now declared as

```
deferred class RING_ELEMENT [G] feature
        item: G
        put (new: G) is do item := new end
        …Other features as before …
end -- class RING_ELEMENT
```

Using the same a generic parameter in two related classes, *RING_ELEMENT* and *MATRIX*, ensures type consistency: all the elements of a given matrix will be of type *RING_ELEMENT* [*G*] for the same *G*.

We can similarly make class *COMPARABLE* generic:

```
deferred class COMPARABLE [G] feature
        item: G
        put (new: G) is do item := new end

        …Other features (infix "<=", minimum) as before …

end -- class COMPARABLE
```

The features of the class (**infix** "<=", *minimum*) represent the constraints (the **with** routines of the Ada form). The earlier descendants become extremely simple:

```
class INTEGER_COMPARABLE inherit
        COMPARABLE [INTEGER]
creation
        put
end
```

(Note that this is the whole class, not a sketch with features to be added!) The same scheme immediately applies to all other variants such as *STRING_COMPARABLE*.

The technique is indeed fairly simple to apply, leading to one more emulation principle:

Emulating constrained genericity (2)

It is possible to emulate constrained genericity through inheritance and unconstrained genericity, by using wrapper classes and the corresponding wrapper objects.

But we are again paying a price: we need to reintroduce wrapper classes such as *INTEGER_COMPARABLE*. This is less shocking than in the earlier solution, because then we had to pay that price for the unconstrained case as well, even though it is conceptually very simple. Here it seems easier to justify the need for wrapper classes and objects since constrained genericity is a relatively sophisticated idea.

Based on these observations, the notation of this book and compilers for it did not initially — for a little over two years, late 1985 to early 1988 — have special support for constrained genericity. The first edition of this book mentioned the possibility of such support, proposing as an exercise the exact design of an appropriate language construct. But it did not take very long afterwards to realize that most applications were not ready to pay the price of wrapper classes and objects, and to integrate the exercise's solution into the notation; the compilers soon followed.

Exercise 19.5, page 422 of [M 1988]. *Later printings mentioned that the extension had been integrated into the language.*

The notation in question is, of course, the one earlier chapters have used to specify constrained genericity, as in

class *MATRIX* [*G –> RING_ELEMENT*] …

and

class *SORTABLE_LIST* [*G –> COMPARABLE*] …

where *RING_ELEMENT* and *COMPARABLE* are the original versions, deferred and non-generic. As noted in the first presentation of this notation in an earlier chapter, it is a remarkable combination of genericity and inheritance, avoiding all the extra baggage of earlier solutions:

- We do not need, like Ada, to use routines as generic parameters (**with** clauses). Only types can be generic parameters; this is simple, consistent and easy to learn.
- We do not need any special wrapper classes and objects. If you want a matrix of integers, you declare it as *MATRIX* [*INTEGER*] and use plain integers to set and retrieve its elements; if you want a sortable list of strings, you declare it as *SORTABLE_LIST* [*STRING*] and use plain strings.

The semantics, as you will remember, is that *G* represents not an arbitrary type any more, but a type that must conform to the constraint (be based on a descendant class). A generic derivation such as *MATRIX* [*T*] is valid if and only if *T* is such a type; this is true of *INTEGER* but not, for example, of *STRING*. Similarly, *STRING* will inherit from *COMPARABLE* and hence will be acceptable as an actual generic parameter for the class *SORTABLE_LIST*; but this is not true of a class *COMPLEX* (for complex numbers) which has no associated order relation. The symbol –> was chosen, as you will also remember, to evoke the arrow of inheritance diagrams.

Providing constrained genericity

Along with unconstrained genericity, it is desirable to provide constrained genericity by relying on inheritance rules (through the notion of type conformance) to define constraints on permissible actual generic parameters.

As a last detail, you will remember that in this scheme constrained genericity becomes the more basic facility: the unconstrained case, as in *QUEUE* [*G*], is understood as an abbreviation for *QUEUE* [*G* –> *ANY*] where *ANY* denotes the class that serves as ancestor to all developer-defined classes. This has the consequence of defining precisely the operations applicable to *G*: those, coming from *ANY*, which are applicable to all classes, including general-purpose features such as *clone*, *print* and *equal*.

The introduction of constrained genericity provides the final touch to the delicate combination of inheritance and genericity detailed in this chapter. I hope that you will find the result consistent, elegant, and *minimal* in the sense that although no component of the edifice is redundant (as it should indeed always be immediately clear, for any particular circumstance, which of the various possibilities is the appropriate one), removing any one of them would lead us to one of the situations that we found unacceptable or unpleasant in the earlier sections of this appendix: unacceptable because we cannot do what we want, as when we were trying to emulate inheritance with genericity; unpleasant when we could do what we want but at the price of such complications as the use of artificial wrapper classes and inefficient wrapper objects. The proper combination of inheritance and genericity should help make our choices not only acceptable but pleasant too.

B.6 KEY CONCEPTS INTRODUCED IN THIS APPENDIX

- Both genericity and inheritance aim to increase the flexibility of software modules.
- Genericity is a static technique, applicable in O-O and non-O-O contexts, permitting the definition of modules parameterized by types.
- There are two forms of genericity: unconstrained, imposing no requirements on the parameters; constrained, requiring parameters to be equipped with specific operations.
- Inheritance permits incremental module construction, by extension and specialization. It opens the way to polymorphism and dynamic binding.
- It does not seem possible to obtain the power of inheritance through genericity.
- Pure inheritance can be used to emulate genericity, but at the expense of heaviness in expression, performance penalties (mostly space) and type difficulties.
- A good compromise is to combine the full power of inheritance and redefinition with genericity, at least in its unconstrained form. This is achieved by permitting classes to have generic parameters.
- It is also desirable to provide constrained genericity, which relies on the notion of type conformance, itself following from inheritance. Unconstrained genericity can then be viewed as a special case, using the universal class *ANY* as the constraint.
- The resulting construction seems elegant and minimal.

B.7 BIBLIOGRAPHICAL NOTES

The material for this chapter originated with an article at the first OOPSLA conference [M 1986]. The Trellis language [Schaffert 1986] also offered the combination of multiple inheritance with constrained and unconstrained genericity.

EXERCISES

E-B.1 Artificial anchors

The artificial anchor *anchor* is declared as an attribute of class *MATRIX* and thus entails a small run-time space overhead in instances of the class. Is it possible to avoid this overhead by declaring *anchor* as a "once function", whose body may be empty since it will never need to be evaluated? (**Hint**: consider type rules.)

E-B.2 Binary trees and binary search trees

Write a generic "binary tree" class *BINARY_TREE*; a binary tree (or binary node) has some root information and two optional subtrees, left and right. Then consider the notion of "binary search tree" where a new element is inserted on the left of a given node if its information field is less than or equal to the information of that node, and to the right otherwise; this assumes that there is a total order relation on "informations". Write a class *BINARY_SEARCH_TREE* implementing this notion, as a descendant of *BINARY_TREE*. Make the class as general as possible, and its use by a client, for an arbitrary type of "informations" with their specific order relation, as easy as possible.

E-B.3 More usable matrices

Add to the last version obtained for class *MATRIX* two functions, one for access and one for modification, which in contrast to *item* and *put* will allow clients to manipulate a matrix of type *MATRIX* [*G*] in terms of elements of type *G* rather than *RING_ELEMENT* [*G*].

E-B.4 Full queue implementations

Expand the queue example by defining a deferred class *QUEUE*, completing the class of this chapter (now called *ARRAYED_QUEUE*, inheriting from *QUEUE* and *ARRAY*, and with proper postconditions), and adding a class *LINKED_QUEUE* for the linked list implementation (based on inheritance from *LINKED_LIST* and *QUEUE*).