16

Inheritance techniques

From the last two chapters we have learned to appreciate inheritance as a key ingredient in the object-oriented approach to reusability and extendibility. To complete its study we must explore a few more facilities — something of a mixed bag, but all showing striking consequences of the beauty of the basic ideas:

- How the inheritance mechanism relates to assertions and Design by Contract.
- The global inheritance structure, where all classes fit.
- Frozen features: when the Open-Closed principle does not apply.
- Constrained genericity: how to put requirements on generic parameters.
- Assignment attempt: how to force a type — safely.
- When and how to change type properties in a redeclaration.
- The mechanism of anchored declaration, avoiding redeclaration avalanche.
- The tumultuous relationship between inheritance and information hiding.

Two later chapters will pursue inheritance-related topics: the review of *typing* issues in chapter 17, and a detailed methodological discussion of *how to use inheritance* (and how not to misuse it) in chapter 24.

Most of the following sections proceed in the same way: examining a consequence of the inheritance ideas of the last two chapters; discovering that it raises a challenge or an apparent dilemma; analyzing the problem in more depth; and deducing the solution. The key step is usually the next-to-last one: by taking the time to pose the problem carefully, we will often be led directly to the answer.

16.1 INHERITANCE AND ASSERTIONS

Because of its very power, inheritance could be dangerous. Were it not for the assertion mechanism, class developers could use redeclaration and dynamic binding to change the semantics of operations treacherously, without much possibility of client control. But assertions will do more: they will give us deeper insights into the nature of inheritance. It is in fact not an exaggeration to state that only through the principles of Design by Contract can one finally understand what inheritance is really about.

The basic rules governing the rapport between inheritance and assertions have already been sketched: in a descendant class, all ancestors' assertions (routine preconditions and postconditions, class invariants) still apply. This section gives the rules more precisely and uses the results obtained to take a new look at inheritance, viewed as subcontracting.

Invariants

We already encountered the rule for class invariants:

Parents' Invariant rule

The invariants of all the parents of a class apply to the class itself.

The parents' invariants are added to the class's own, "addition" being here a logical **and then**. (If no invariant is given in a class, it is considered to have *True* as invariant.) By induction the invariants of all ancestors, direct or indirect, apply.

As a consequence, you should not repeat the parents' invariant clauses in the invariant of a class (although such redundancy would be semantically harmless since *a* **and then** *a* is the same thing as *a*).

The flat and flat-short forms of the class will show the complete reconstructed invariant, all ancestors' clauses concatenated.

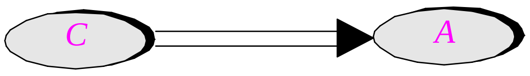
See *"FLATTENING THE STRUCTURE", 15.3, page 541*.

Preconditions and postconditions in the presence of dynamic binding

The case of routine preconditions and postconditions is slightly more delicate. The general idea, as noted, is that any redeclaration must satisfy the assertions on the original routine. This is particularly important if that routine was deferred: without such a constraint on possible effectings, attaching a precondition and a postcondition to a deferred routine would be useless or, worse, misleading. But the need is just as bad with redefinitions of effective routines.

The exact rule will follow directly from a careful analysis of the consequences of redeclaration, polymorphism and dynamic binding. Let us construct a typical case and deduce the rule from that analysis.

Consider a class and one of its routines with a precondition and a postcondition:



```
r is
    require
        α
    …
    ensure
        β
    end
```

The routine, the client and the contract

The figure also shows a client *C* of *A*. The typical way for *C* to be a client is to include, in one of its routines, a declaration and call of the form

```
a1: A
…
a1.r
```

For simplicity, we ignore any arguments that *r* may require, and we assume that *r* is a procedure, although the discussion applies to a function just as well.

Of course the call will only be correct if it satisfies the precondition. One way for *C* to make sure that it observes its part of the contract is to protect the call by a precondition test, writing it (instead of just *a1*.*r*) as

```
if a1.α then
    a1.r
        check a1.β end        -- i.e. the postcondition holds
    … Instructions that may assume a1.β …
end
```

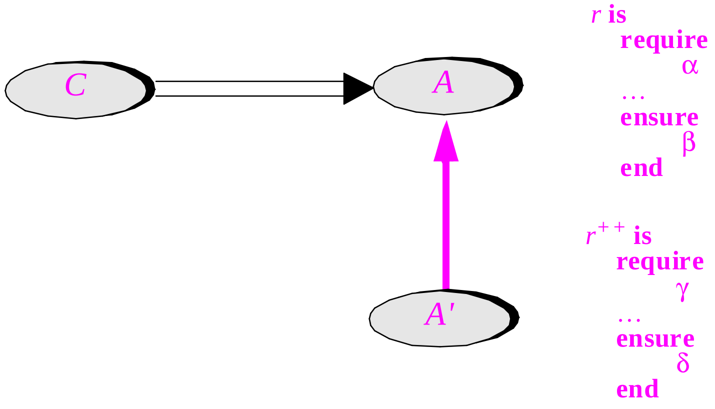
(As noted in the discussion of assertions, this is not required: it suffices to guarantee, with or without an **if** instruction, that α holds before the call. We will assume the **if** form for simplicity, and ignore any **else** clause.)

Having guaranteed the precondition, the client *C* is entitled to the postcondition on return: after the call, it may expect that *a1*.β will hold.

All this is the basics of Design by Contract: the client *must* ensure the precondition on calling the routine and, as a recompense, *may* count on the postcondition being satisfied when the routine exits.

What happens when inheritance enters the picture?

The routine, the client, the contract and the descendant



Assume that a new class *A'* inherits from *A* and redeclares *r*. How, if at all, can it change the precondition α into a new one γ and the postcondition β into a new one δ?

To decide the answer, consider the plight of the client. In the call *a1*.*r* the target *a1* may now, out of polymorphism, be of type *A'* rather than just *A*. But *C* does not know about this! The only declaration for *a1* may still be the original one:

```
a1: A
```

which names *A*, not *A'*. In fact *C* may well use *A'* without its author ever knowing about the existence of such a class; the call to *r* may for example be in a routine of *C* of the form

```
some_routine_of_C (a1: A) is
    do
        …; a1.r; …
    end
```

Then a call to *some_routine_of_C* from another class may use an actual argument of type *A'*, even though the text of *C* contains no mention of class *A'*. Dynamic binding means that the call to *r* will in that case use the redefined *A'* version.

So we can have a situation where *C* is only a client of *A* but in fact will at run time use the *A'* version of some features. (We could say that *C* is a "dynamic client" of *A'* even though its text does not show it.)

What does this mean for *C*? The answer, unless we do something, is: trouble. *C* can be an honest client, observing its part of the deal, and still be cheated on the result. In

```
if a1.α then a1.r end
```

if *a1* is polymorphically attached to an object of type *A'*, the instruction calls a routine that expects γ and guarantees δ, whereas the client has been told to satisfy α and expect β. So we have a potential discrepancy between the client's and supplier's views of the contract.

How to cheat clients

To understand how to satisfy the clients' expectations, we have to play devil's advocate and imagine for a second how we could fool them. It is all for a good cause, of course (as with a crime unit that tries to emulate criminals' thinking the better to fight it, or a computer security expert who studies the techniques of computer intruders).

If we, the supplier, wanted to cheat our poor, honest *C* client, who guarantees α and expects β, how would we proceed? There are actually two ways to evil:

- We could *require more* than the original precondition α. With a stronger precondition, we allow ourselves to exclude (that is to say, not to guarantee any specific result) for cases that, according to the original specification, were perfectly acceptable.

  Remember the point emphasized repeatedly in the discussion of Design by Contract: making a precondition stronger facilitates the task of the supplier ("the client is more often wrong"), as illustrated by the extreme case of precondition **false** ("the client is always wrong").

- We could *ensure less* than the original postcondition β. With a weaker postcondition, we allow ourselves to produce less than what the original specification promised.

As we saw, an assertion is said to be stronger than another if it logically implies it, and is different; for example, $x >= 5$ is stronger than $x >= 0$. If *A* is stronger than *B*, *B* is said to be weaker than *A*.

How to be honest

From understanding how to cheat we deduce how to be honest. When redeclaring a routine, we may keep the original assertions, but we may also:

- Replace the precondition by a *weaker* one.
- Replace the postcondition by a *stronger* one.

The first case means being more generous than the original — accepting more cases. This can cause no harm to a client that satisfies the original precondition before the call. The second case means producing more than what was promised; this can cause no harm to a client call that relies on the original postcondition being satisfied after the call.

Hence the basic rule:

Assertion Redeclaration rule (1)

A routine redeclaration may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger.

The rule expresses that the new version must accept all calls that were acceptable to the original, and must guarantee at least as much as was guaranteed by the original. It may — but does not have to — accept more cases, or provide stronger guarantees.

As its name indicates, this rule applies to both forms of redeclaration: redefinitions and effectings. The second case is particularly important, since it allows you to take seriously the assertions that may be attached to a deferred feature; these assertions will be binding on all effective versions in descendants.

For a more rigorous definition see "A mathematical note", page 580

The assertions of a routine, deferred or effective, specify the essential semantics of the routine, applicable not only to the routine itself but to any redeclaration in descendants. More precisely, they specify a **range of acceptable behaviors** for the routine and its eventual redeclarations. A redeclaration may specialize this range, but not violate it.

A consequence for the class author is the need to be careful, when writing the assertions of an effective routine, not to *overspecify*. The assertions must characterize the intent of the routine — its abstract semantics —, not the properties of the original implementation. If you overspecify, you may be closing off the possibility for a future descendant to provide a different implementation.

An example

Assume I write a class *MATRIX* implementing linear algebra operations. Among the features I offer to my clients is a matrix inversion routine. It is actually a combination of a command and two queries: procedure *invert* inverts the matrix, and sets attribute *inverse* to the value of the inverse matrix, as well as a boolean attribute *inverse_valid*. The value of *inverse* is meaningful if and only if *inverse_valid* is true; otherwise the inversion has failed because the matrix was singular. For this discussion we can ignore the singularity case.

Of course I can only compute an approximation of the inverse of a matrix. I am prepared to guarantee a certain precision of the result, but since I am not very good at numerical analysis, I shall only accept requests for a precision not better than 10^{-6}. The resulting routine will look like this:

```
invert (epsilon: REAL) is
        -- Inverse of current matrix, with precision epsilon
    require
        epsilon >= 10 ^ (–6)
    do
        "Computation of inverse"
    ensure
        ((Current ∗ inverse) |–| One) <= epsilon
    end
```

The postcondition assumes that the class has a function **infix** "|–|" such that *m1* |–| *m2* is |*m1* — *m2*|, the norm of the matrix difference of *m1* and *m2*, and a function **infix** "∗" which yields the product of two matrices; *One* is assumed to denote the identity matrix.

I am not too proud of myself, so for the summer I hire a bright young programmer-numerician who rewrites my *invert* routine using a much better algorithm, which approximates the result more closely and accepts a smaller *epsilon*:

```
    require
        epsilon >= 10 ^ (–20)
    …
    ensure
        ((Current ∗ inverse) |–| One) <= (epsilon / 2)
```

Warning: syntactically not valid as a redefinition. See next.

The author of this new version is far too clever to rewrite a full *MATRIX* class; only a few routines need adaptation. They will be included in a descendant of *MATRIX*, say *NEW_MATRIX*.

If the new assertions are in a redefinition, they must use a different syntax than shown above. The rule will be given shortly.

The change of assertions satisfies the Assertion Redeclaration rule: the new precondition *epsilon >= 10 ^ (–20)* is weaker than (that is to say, implied by) the original *epsilon >= 10 ^ (–6)*; and the new postcondition is stronger than the original.

This is how it should be. A client of the original *MATRIX* may be requesting a matrix inversion but, through dynamic binding, actually calling the *NEW_MATRIX* variant. The client could contain a routine

```
some_client_routine (m1: MATRIX; precision: REAL) is
    do
        … ; m1.invert (precision); …
            -- May use either the MATRIX or the NEW_MATRIX version
    end
```

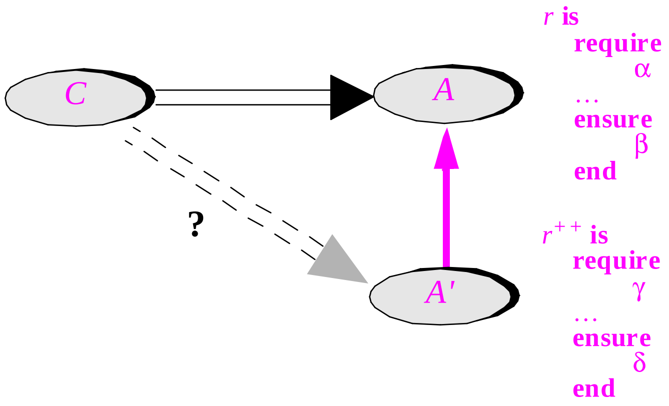
to which one of its own clients passes a first argument of type *NEW_MATRIX*.

NEW_MATRIX must be able to accept and handle correctly any call that *MATRIX* would accept. If we made the precondition of the new *invert* stronger than the original (as in *epsilon* >= ^ (–5)), calls which are correct for *MATRIX* would now be incorrect; if we made the postcondition weaker, the result returned would not be as good as guaranteed by *MATRIX*. By using a weaker precondition and a stronger postcondition we correctly treat all calls from clients of *MATRIX*, while offering a better deal to our own clients.

Cutting out the middleman

The last comment points to an interesting consequence of the Assertion Redeclaration rule. In our general scheme

The routine, the client and the sub-contractor



the assertions of the redeclared version, γ and δ, if different from α and β, are more favorable to the clients, in the sense explained earlier (weaker precondition, stronger postcondition). But a client of *A* which uses *A'* through polymorphism and dynamic binding cannot make good use of this improved contract, since its only contract is with *A*.

Only by becoming a direct client of *A'* (the shaded link with a question mark on the last figure) can you take advantage of the new contract, as in

```
a1: A'
...
if a1.γ then a1.r end
    check a1.δ end          -- i.e. the postcondition holds
```

But then of course you have specialized *a1* to be of type *A'*, not the general *A*; you have lost the polymorphic generality of going through *A*.

The tradeoff is clear. A client of *MATRIX* must satisfy the original (stronger) precondition, and may only expect the original (weaker) postcondition; even if its request gets served dynamically by *NEW_MATRIX* it has no way of benefiting from the broader tolerance of inputs and tighter precision of results. To get this improved specification it must declare the matrix to be of type *NEW_MATRIX*, thereby losing access to other implementations represented by descendants of *MATRIX* that are not also descendants of *NEW_MATRIX*.

Subcontracting

The Assertion Redeclaration rule fits nicely in the Design by Contract theory introduced in the chapter bearing that title.

We saw that the assertions of a routine describe the contract associated with that routine: the client is bound by the precondition and entitled to the postcondition, and conversely for the class implementer.

Inheritance, with redeclaration and dynamic binding, means *sub*contracting. When you have accepted a contract, you do not necessarily want to carry it out yourself. Sometimes you know of somebody else who can do it cheaper and perhaps better. This is exactly what happens when a client requests a routine from *MATRIX* but, through dynamic binding, may actually call at run time a version redefined in a proper descendant. Here “cheaper” refers to routine redefinition for more efficiency, as in the rectangle perimeter example of an earlier chapter, and “better” to improved assertions in the sense just seen.

The Assertion Redeclaration rule simply states that if you are an honest subcontractor and accept a contract, you must be willing to do the job originally requested, or better than the requested job, but not less.

The scheme described in the last section — declaring *a1* of type *A'* to benefit from the improved contract — is similar to the behavior of a customer who tries to get a better deal by bypassing his contractor to work directly with the contractor’s own subcontractor

In the Design by Contract view, class invariants are general constraints applying to both contractors and clients. The parents’ invariant rule expresses that all such constraints are transmitted to subcontractors.

It is only with assertions, and with the two rules just seen, that inheritance takes on its full meaning for object-oriented design. The contracting-subcontracting metaphor is a powerful analogy to guide the development of correct object-oriented software; certainly one of the central deas.

Abstract preconditions

The rule on weakening preconditions may appear too restrictive in the case of an heir that restricts the abstraction provided by its parent. Fortunately, there is an easy workaround, consistent with the theory.

A typical example arises if you want to make a class *BOUNDED_STACK* inherit from a general *STACK* class. In *BOUNDED_STACK* the procedure for pushing an element onto the stack, *put*, has a precondition, which requires *count <= capacity*, where *count* is the current number of stack elements and *capacity* is the physically available size.

For the general notion of *STACK*, however, there is no notion of *capacity*. So it seems we need to *strengthen* the precondition when we move down to *BOUNDED_STACK*. How do we build this inheritance structure without violating the Assertion Redeclaration rule?

The answer is straightforward if we take a closer look at client needs. What needs to be kept or weakened is not necessarily the concrete precondition as implemented by the supplier (which is the supplier's business), but the precondition *as seen by the client*. Assume that we write *put* in *STACK* as

```
put (x: G) is
            -- Push x on top.
      require
            not full
      deferred
      ensure
            …
      end
```

with a function *full* defined always to return false, so that by default stacks are never full:

```
full: BOOLEAN is
            -- Is representation full?
            -- (Default: no)
      do Result := False end
```

Then it suffices in *BOUNDED_STACK* to redefine *full*:

```
full: BOOLEAN is
            -- Is representation full?
            -- (Answer: if and only if number of items is capacity)
      do Result := (count = capacity) end
```

A precondition such as **not** *full*, based on a property that is redefinable in descendants, is called an abstract precondition.

This use of abstract preconditions to satisfy the Assertion Redeclaration rule may appear to be cheating, but it is not: although the concrete precondition is in fact being strengthened, the abstract precondition remains the same. What counts is not how the assertion is implemented, but how it is presented to the clients as part of the class interface (the short or flat-short form). A protected call of the form

```
if not s.full then s.put (a) end
```

will be valid regardless of the kind of *STACK* attached to *s*.

There is, however, a valid criticism of this approach: it goes against the Open-Closed principle. We must foresee, at the *STACK* level, that some stacks will have a bounded capacity; if we have not exerted such foresight, we must go back to *STACK* and change its interface. But this is inevitable. Of the following two properties

- A bounded stack is a stack.
- It is always possible to add an element to a stack.

one must go. If we want the first property, permitting *BOUNDED_STACK* to inherit from *STACK*, we must accept that the general notion of stack includes the provision that a *put* operation is not always possible, expressed abstractly by the presence of the query *full*.

It would clearly be a mistake, in class *STACK*, to include *Result* = *False* as a postcondition for *full* or (equivalently but following the recommended style) an invariant clause **not** *full*. This would be a case of overspecification as mentioned earlier, hampering the descendants' freedom to adapt the feature.

The language rule

The Assertion Redeclaration rule as given so far is a conceptual guideline. How do we transform it into a safe, checkable language rule?

We should in principle rely on a logical analysis of the old and new assertions, to verify that the old precondition logically implies the new one, and that the new postcondition implies the old one. Unfortunately, such a goal would require a sophisticated *theorem prover* which, if at all feasible, is still far too difficult (in spite of decades of research in artificial intelligence) to be integrated routinely among the checks performed by a compiler.

Fortunately a low-tech solution is available. We can enforce the rule through a simple language convention, based on the observation that for any assertions α and β:

- α implies α **or** γ, regardless of what γ is.
- β **and** δ implies β, regardless of what δ is.

So to be sure that a new precondition is weaker than or equal to an original α, it suffices to accept it *only* if it is of the form α **or** γ; and to be sure that a new postcondition is stronger than or equal to an original β, it suffices to accept it only if it is of the form β **and** δ. Hence the language rule implementing the original methodological rule:

Assertion Redeclaration rule (2)

In the redeclared version of a routine, it is not permitted to use a **require** or **ensure** clause. Instead you may:

- Use a clause introduced by **require else**, to be or-ed with the original precondition.
- Use a clause introduced by **ensure then**, to be and-ed with the original postcondition.

In the absence of such a clause, the original assertion is retained.

Note that the operators used for or-ing and for and-ing are the non-strict boolean operators **or else** and **and then** rather than plain **or** and **and**, although in most cases the difference is irrelevant.

See "Non-strict boolean operators", page 454

Sometimes the resulting assertions will be more complicated than strictly necessary. For example in our matrix routine, where the original read

```
invert (epsilon: REAL) is
        -- Inverse of current matrix, with precision epsilon
    require
        epsilon >= 10 ^ (–6)
    …
    ensure
        ((Current ∗ inverse) |–| One) <= epsilon
```

the redefined version may not use **require** and **ensure** but will appear as

```
…
    require else
        epsilon >= 10 ^ (–20)
    …
    ensure then
        ((Current ∗ inverse) |–| One) <= (epsilon / 2)
```

so that formally the precondition is *(epsilon >= 10* ^ (–*20*)) **or else** (*epsilon >= 10* ^ (–*6*)), and similarly for the postcondition. But this does not really matter, since a weaker precondition or a stronger postcondition takes over: if α implies γ, then α **or else** γ has the same value as γ; and if δ implies β, then β **and then** δ has the same value as δ. So mathematically the precondition of the redefined version is *epsilon >= 10* ^ (–*20*) and its postcondition is ((*Current* ∗ *inverse*) |–| *One*) <= (*epsilon* / *2*), even though the software assertions (and probably, in the absence of a symbolic expression simplifier, their evaluation at run time if assertion checking is enabled) are more complicated.

Redeclaring into attributes

See "Redeclaring a function into an attribute", page 491.

The Assertion Redeclaration rule needs a small complement because of the possibility of redeclaring a function into an attribute. What happens to the original's precondition and postcondition, if any?

An attribute is always accessible, and so may be considered to have precondition *True*. This means that we may consider the precondition to have been weakened, in line with the Assertion Redeclaration rule.

An attribute, however, does not have a postcondition. Since it is necessary to guarantee that the attribute satisfy any property ensured by the original function, the proper convention (an addition to the Assertion Redeclaration rule) is to consider that the postcondition is automatically added to the class invariant. The flat form of the class will include the condition in its invariant.

When expressing a property of the value of a function without arguments, you always have the choice between including it in the postcondition or in the invariant. As a matter of style it is considered preferable to use the invariant. If you follow this rule there will not be any change of assertions if you later redeclare the function as an attribute.

A mathematical note

An informal comment on the Assertion Redeclaration rule stated: “A redeclaration may specialize the range of acceptable behaviors, but not violate it”. Here, to conclude this discussion, is a rigorous form of that property (for mathematically inclined readers only).

Page 573.

Consider that a routine implements a partial function r from the set of possible input states I to the set of possible output states O. The routine’s assertions define rules as to what r and its possible redeclarations may and may not do:

- The precondition specifies the domain DOM of r (the subset of I in which r is guaranteed to yield a result).
- The postcondition specifies, for each element x of DOM, a subset $RESULTS\ (x)$ of O such that $r\ (x) \in RESULTS\ (x)$. This subset may have more than one element, since a postcondition does not have to define the result uniquely.

The Assertion Redeclaration rule means that a redeclaration may broaden the domain and restrict the result sets; writing the new sets in primed form, the rule requires that

$DOM' \supseteq DOM$
$RESULTS'\ (x) \subseteq RESULTS\ (x)$ for any x in DOM

A routine’s precondition specifies that the routine and its eventual redeclarations *must at least* accept certain inputs (DOM), although redeclarations may accept more. The postcondition specifies that the outputs produced by the routine and its eventual redeclarations *may at most* include certain values ($RESULTS\ (x)$), although redeclarations’ postconditions may include fewer.

In this description a state of a system’s execution is defined by the contents of all reachable objects; in addition, input states (elements of I) also include the values of the arguments. For a more detailed introduction to the mathematical description of programs and programming languages see [M 1990].

16.2 THE GLOBAL INHERITANCE STRUCTURE

A few references have been made in earlier discussions to the universal classes *GENERAL* and *ANY* and to the objectless class *NONE*. It is time to clarify their role and present the global inheritance structure.

Universal classes

It is convenient to use the following convention.

Universal Class rule

Any class that does not include an inheritance clause is considered to include an implicit clause of the form

inherit *ANY*

referring to a Kernel library class *ANY*.

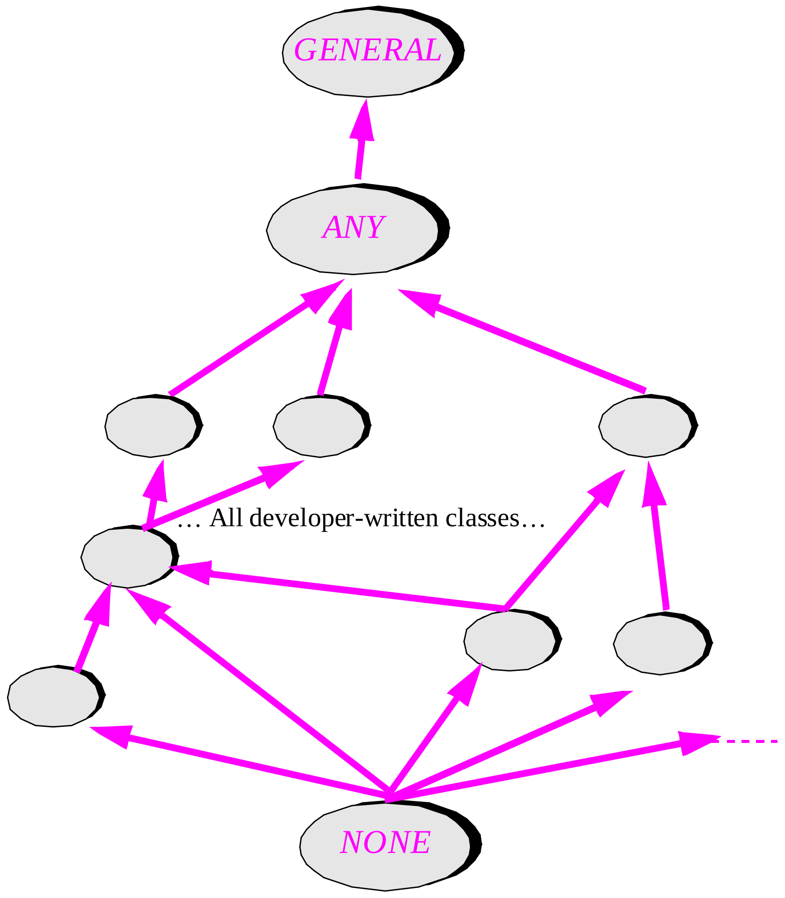
This makes it possible to define a certain number of features that will be inherited by all classes. These features provide operations of universal interest: copy, clone, comparison, basic input and output.

For more flexibility, we will not put these features in *ANY* but in a class *GENERAL* of which *ANY* itself is an heir. *ANY*, in its default form, will have no features (being simply of the form **class** *ANY* **inherit** *GENERAL* **end**); but then a project leader or corporate reuse manager who wants to make a certain number of features available across the board can adapt *ANY* for local purposes without touching *GENERAL*, which should be the same in Versailles, Vanuatu, Venice and Veracruz.

To build a non-trivial *ANY*, you may want to use inheritance. You can indeed make *ANY* inherit from some class *HOUSE_STYLE*, or several such classes, without introducing any cycles in the inheritance hierarchy or violating the universal class rule: just make *HOUSE_STYLE* and its consorts explicit heirs of *GENERAL*. In the following figure, “All developer-written classes” means more precisely: all developer-written classes that do not explicitly inherit from *GENERAL*.

Here then is a picture of the general structure:

The global inheritance structure



The bottom of the pit

Also included in the figure is a class *NONE*, the nemesis of *ANY*: it inherits from any class that does not have any other heir and makes the global inheritance class a lattice. You probably do not want to see the **rename** subclauses of *NONE* and, be relieved, you will not. (It changes anyway each time someone writes a new class.) *NONE* is just a convenient fiction. But its theoretical existence serves two practical purposes:

- The type of *Void*, the void reference used among other things to terminate linked structures, is by convention *NONE*. (*Void* is in fact one of the features of *GENERAL*.)
- To hide a feature from all clients, export it to *NONE* only (in a feature clause of the form **feature** {*NONE*}, equivalent in practice to **feature** { } but more explicit, or in an inheritance subclause **export** {*NONE*}, also with the same practical effect as **export** { }). This will make it unavailable to any developer class, since *NONE* has no proper descendants. Note that *NONE* hides all its features.

On the first property, note that you may assign the value *Void* to an entity of any reference type; so until now the status of *Void* was a little mysterious, since it had somehow to be compatible to all types. Making *NONE* the type of *Void* makes this status clear, official, and consistent with the type system: by construction, *NONE* is a descendant of all classes, so that we can use *Void* as a valid value of any reference type without any need to tamper with the type rules.

On the second property note that, symmetrically, a feature clause beginning with just **feature**, which exports its features to all developer classes, is considered a shorthand for **feature** {*ANY*}. To reexport to all classes a parent feature which had tighter availability, you may use **export** {*ANY*}, or the less explicit shorthand **export**.

ANY and *NONE* ensure that our type system is closed and our inheritance structure complete: the lattice has a top and it has a bottom.

Universal features

Here is a small sampling of the features found in *GENERAL* and hence available to all classes. Several of them were introduced and used in earlier chapters:

- *clone* for duplicating an object, and its deep variant *deep_clone* for recursively duplicating an entire object structure.
- *copy* for copying the contents of an object into another.
- *equal* for field-by-field object comparison, and its deep variant *deep_equal*.

See "Object cloning and equality", page 245, and subsequent sections.

Other features include:

- *print* and *print_line* to print a simple default representation of any object.
- *tagged_out*, a string containing a default representation of any object, each field accompanied by its tag (the corresponding attribute name).
- *same_type* and *conforms_to*, boolean functions that compare the type of the current object to the type of another.

- *generator*, which yields the name of an object's generating class — the class of which it is a direct instance.

16.3 FROZEN FEATURES

The presentation of inheritance has repeatedly emphasized the Open-Closed principle: the ability to take any feature from an ancestor class and redefine it to do it something different. Can there be any reason for shutting off this possibility?

Prohibiting redefinition

The discussion of assertions at the beginning of this chapter has provided us with the theoretical understanding of redefinition: the "open" part of the Open-Closed principle — the ability to change features in descendants — is kept in check by the original assertions. The only permitted redefinitions change the implementation while remaining consistent with the specification given by the precondition and postcondition of the original.

In some rare cases, you may want to guarantee to your clients, and to the clients of your descendants, not only that a feature will satisfy the official specification, but also that it will use the exact original implementation. The only way to achieve this goal is to forbid redeclarations altogether. A simple language construct provides this possibility:

```
frozen feature_name … is … The rest of the feature declaration as usual …
```

With this declaration, no descendant's **redefine** or **undefine** subclause may list the feature, whether under its original name or (since renaming remains of course permitted) another. A deferred feature — meant, by definition, for redeclaration — may not be frozen.

Fixed semantics for copy, clone and equality features

The most common use of frozen features is for general-purpose operations of the kind just reviewed for *GENERAL*. For example there are two versions of the basic copy procedure:

```
copy, frozen standard_copy (other: …) is
            -- Copy fields of other onto fields of current object.
    require
            other_not_void: other /= Void
    do
            …
    ensure
            equal (Current, other)
    end
```

This declares two features as synonyms. (A general convention allows us to declare two features together so that they can share the same declaration; just separate their names with commas as here. The effect is as if there had been two separate declarations with identical declaration bodies.) But only one of the features is redefinable. So a descendant class can redefine *copy*; this is necessary for example for classes *ARRAY* and *STRING*, which redefine *copy* so as to compare actual array and string contents, not the array or string descriptors. It is convenient in such cases to have a frozen version as well, so that we can use the default operation, *standard_copy*, guaranteed to be the original.

In class *GENERAL*, feature *clone* also has a similar doppelgänger *standard_clone*, but here both versions are frozen. Why should *clone* be frozen? The reason is not to prevent the definition of a different cloning operation, but to ensure that clone and copy semantics remain compatible, and as a side benefit to facilitate the redefiner's task. The declaration of *clone* is of the general form

```
frozen clone (other: …): … is
        -- Void if other is void; otherwise new object with contents copied from
other.
    do
        if other /= Void then
            Result := "New object of the same type as other"
            Result.copy (other)
        end
    ensure
        equal (Result, other)
    end
```

If other is void the default initializations yield Void for Result.

"New object of the same type as *other*" informally denotes a call to some function that creates and returns such an object, as provided by the implementation.

So even though *clone* is frozen, it will follow any redefinition of *copy*, for example in *ARRAY* and *STRING*. This is good for safety, as it would be a mistake to have different semantics for these operations, and convenience, as you will only need to redefine *copy* to change the copy-clone semantics in a descendant.

Although you need not (and cannot) redefine *clone*, you will still need, in step with a redefinition of *copy*, to redefine the semantics of equality. As indicated by the postconditions given for *copy* and *clone*, a copy must yield equal objects. Function *equal* itself is in fact frozen in the same way that *clone* is — to ensure its dependency on another, redefinable feature:

```
frozen equal (some, other: …): BOOLEAN is
        -- Are some and other either both void
        -- or attached to objects considered equal?
    do
        Result := ((some = Void) and (other = Void)) or else some.is_equal (other)
    ensure
        Result = ((some = Void) and (other = Void)) or else some.is_equal (other)
    end
```

The matter was discussed in "The form of clone and equality operations", page 274.

Function *equal* is called under the form *equal (a, b)*, which does not quite enjoy the official O-O look of *a.is_equal (b)* but has the important practical advantage of being applicable when *a* or *b* is void. The basic feature, however, is *is_equal*, not frozen, which you should redefine in any class that redefines *copy*, to keep equality semantics compatible with copy and clone semantics — so that the postconditions of *copy* and *clone* remain correct.

Besides *equal* there is a function *standard_equal* whose semantics is not affected by redefinitions of *is_equal*. (It uses the above algorithm but using *standard_is_equal*, frozen, rather than *is_equal*.)

Freeze only when needed

The examples of freezing that have just been given are typical of the use of this mechanism: guaranteeing the exact semantics of the original.

See "Static binding as an optimization", page 511.

It is never appropriate to freeze a feature out of efficiency concerns. (This is a mistake sometimes made by developers with a C++ or Smalltalk background, who have been told that dynamic binding is expensive and that they must manually avoid it if possible.) Clearly, a call to a frozen feature will never need dynamic binding; but this is a side effect of the **frozen** mechanism rather than its purpose. As discussed in detail in an earlier chapter, applying static binding safely is a compiler optimization, not a concern for software developers. In a well-designed language the compiler will have all it needs to perform this optimization when appropriate, along with even more far-reaching optimizations such as routine inlining. Determining the appropriate cases is a job for machines, not humans. Use **frozen** in the rare although important cases in which you need it for conceptual purposes — to guarantee the exact semantics of the original implementation — and let the language and the compiler do their job.

16.4 CONSTRAINED GENERICITY

"Polymorphic data structures", page 470.

Inheritance and genericity have been presented as the two partners in the task of extending the basic notion of class. We have already studied how to combine them through the notion of *polymorphic data structure*: into a container object described by an entity of type *SOME_CONTAINER_TYPE* [*T*] for some *T*, we can insert objects whose type is not just *T* but any descendant of *T*. But there is another interesting combination, in which inheritance serves to define what is and is not acceptable as actual generic parameter to a certain class.

Addable vectors

A simple and typical example will allow us to see the need for constrained genericity — and, as everywhere else in this book, to deduce the method and language construct as a logical consequence of the problem's statement.

Assume we want to declare a class *VECTOR* to describe vectors of elements, with an addition operation. There are vectors of elements of many different types, so we clearly need a generic class. A first sketch may look like

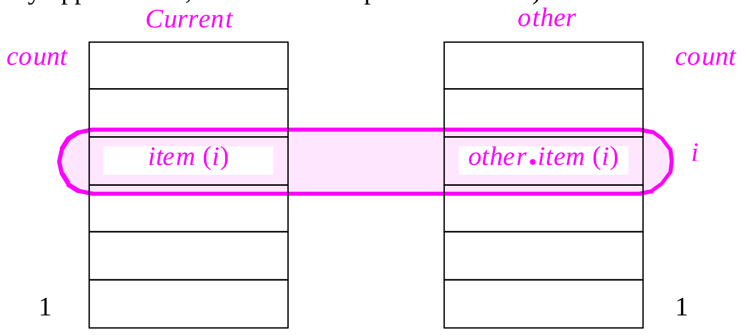
```
indexing
    description: "Addable vectors"
class
    VECTOR [G]
feature -- Access
    count: INTEGER
        -- Number of items

    item, infix "@" (i: INTEGER): G is
            -- Vector element of index i (numbering starts at 1)
        require … do
            …
        end
feature -- Basic operations
    infix "+" (other: VECTOR [G]): VECTOR is
            -- The sum, element by element, of current vector and other
        require … do
            …
        end
    … Other features …
invariant
    non_negative_count: count >= 0
end -- class VECTOR
```

The use of an infix feature is convenient for this class, but does not otherwise affect the discussion. Also for convenience, we have two synonyms for the basic access feature, so that we can denote the *i*-th element of a vector (as in the *ARRAY* class, which could be used to provide an implementation) as either *v*.*item* (*i*) or just *v* @ *i*.

Now let us see how we could write the "+" function. At first it seems straightforward: to add two vectors, we just add one by one their elements at corresponding positions. The general scheme is

```
    infix "+" (other: VECTOR [G]): VECTOR is
            -- The sum, element by element, of current vector and other
        require
            count = other.count
        local
            i: INTEGER
        do
            "Create Result as an array of count items"
            from i := 1 until i > count loop
                Result.put (item (i) + other.item (i), i)
                i := i + 1
            end
        end
```

The boxed expression is the sum of the items at index *i* in the current vector and *other*, as illustrated by the figure on the facing page. The enclosing call to *put* assigns its value to the *i*-th item of *Result*. (Procedure *put* has not been shown in class *VECTOR*, but must obviously appear there, like its counterpart in *ARRAY*.)



Adding two vectors, item by item

But this does not work! The + operation in the boxed expression is an addition of vector elements (not vectors); it is intended to add values of type *G*, the generic parameter. By definition a generic parameter represents an unknown type — the actual generic parameter, to be provided only when we decide to use the generic class for good, through what has been called a **generic derivation**. If the generic derivation uses, as actual generic parameter, a type such as *INTEGER*, or some other class which includes a function **infix** "+" with the right signature, everything will work fine. But what if the actual generic parameter is *ELLIPSE*, or *STACK* [*SOME_TYPE*], or *EMPLOYEE*, or any other type that does not have an addition operation?

We did not have such a problem with the generic classes encountered previously — general container classes such as *STACK*, *LIST* and *ARRAY* — since the only operations they needed to apply to container elements (represented by entities of type *G*, the formal generic parameter) were universal, type-independent operations: assignment, comparison, use as argument in feature calls. But for an abstraction such as addable vectors we need to restrict the permissible actual generic parameters to make sure certain operations are available.

This is by no means an exceptional example. Here are two other typical ones:

- Assume you want to describe sortable structures, with a procedure *sort* that will order the elements according to some criterion. You need to ensure the availability of a comparison operation **infix** "<=", representing a total order, on the corresponding objects.

- In building basic data structures such as dictionaries, you may want to use a *hash-table*, where the position of each element is determined by a key derived from the value of the element. This assumes the availability of a “hashing function” which computes the key (also known as the “hash value”) of any element.

A non-O-O approach

Although there have been enough hints in the preceding paragraphs to suggest the almost inevitable solution to our problem, it is useful to pause for a moment and examine how another approach, not object-oriented, has addressed the same problem.

The hurried reader may skip directly to the O-O solution in the next section, "Constraining the generic parameter", page 588.

Ada does not have classes, but has packages which serve to group related operations and types. A package may be generic, with generic parameters representing types. So the same problem arises: a package *VECTOR_PROCESSING* might include a declaration of type *VECTOR* and the equivalent of our **infix** "+" function.

The solution in Ada is to treat the needed operations, such as **infix** "+", as generic parameters themselves. The parameters of a package may include not only types, as in the object-oriented approach, but also routines (called subprograms). For example:

```
generic
    type G is private;
    with function "+" (a, b: G) return G is <>;
    with function "∗" (a, b: G) return G is <>;
    zero: G; unity: G;
package VECTOR_HANDLING is
    … Package interface …
end VECTOR_HANDLING
```

Note that along with the type *G* and the subprograms the package also uses, as generic parameter, a value *zero* representing the zero element of addition. A typical use of the package will be

```
package BOOLEAN_VECTOR_HANDLING is
    new VECTOR_HANDLING (BOOLEAN, "or", "and", false, true);
```

which uses boolean "or" as the addition and boolean "and" as the multiplication, with corresponding values for zero and unity. We will study a more complete solution to this example in a later chapter, as part of a systematic discussion of genericity *vs*. inheritance.

See "Constrained genericity", page 1170.

Although appropriate for Ada, this technique is not acceptable in an O-O context. The basic idea of object technology is the primacy of data types over operations in software decomposition, implying that there is no such thing as a stand-alone operation. *Every operation belongs to some data type, based on a class*. So it would be inconsistent with the rest of the approach to let a function such as **infix** "+", coming out of nowhere, serve as actual generic parameter along with types such as *INTEGER* and *BOOLEAN*. The same holds for values such as *zero* and *unity*, which will have to find their place as features of some class — respectable members of object-oriented society.

Constraining the generic parameter

These observations yield the solution. We must work entirely in terms of classes and types.

What we are requiring is that any actual parameter used for *VECTOR* (and similarly for the other examples) be a type equipped with a set of operations: **infix** "+", perhaps *zero* to initialize sums, and possibly a few others. But since we studied inheritance we know how to equip a type with certain operations: just make it a descendant of a class, deferred or effective, that has these operations.

A simple syntax is

class *C* [*G* –> *CONSTRAINING_TYPE*] … The rest as for any other class …

where *CONSTRAINING_TYPE* is an arbitrary type. The –> symbol, made of a hyphen and a "greater than", evokes the arrow of inheritance diagrams. *CONSTRAINING_TYPE* is called the generic constraint. The consequences of such a declaration are two-fold:

Conformance was defined in "Limits to polymorphism", page 474.

- Only types that conform to *CONSTRAINING_TYPE* will be acceptable as actual generic parameters; remember that a type conforms to another if, roughly speaking, it is based on a descendant.
- Within the text of class *C*, the operations permitted on an entity of type *G* are those which would be permitted on an entity of *CONSTRAINING_TYPE*, that is to say features of the base class of that type.

See "Numeric and comparable values", page 522.

In the *VECTOR* case, what should we use as a generic constraint? A class introduced in the discussion of multiple inheritance, *NUMERIC*, describes the notion of objects to which basic arithmetic operations are applicable: addition and multiplication with zero and unity. (The underlying mathematical structure, as you may recall, is the ring.) This seems appropriate even though for our immediate purposes we only need addition. So the class will be declared as

```
indexing
        description: "Addable vectors"
class
        VECTOR [G –> NUMERIC]
… The rest as before (but now valid!) …
```

Then within the class text, the loop instruction that was previously invalid

Result.*put* ($\boxed{\textit{item } (i) + \textit{other.item } (i)}$, *i*)

has become valid since *item* (*i*) and *other*.*item* (*i*) are both of type *G*, so that all *NUMERIC* operations such as **infix** "+" are applicable to them.

Generic derivations such as the following are all correct, assuming the classes given as actual generic parameters are all descendants of *NUMERIC*:

VECTOR [*NUMERIC*]
VECTOR [*REAL*]
VECTOR [*COMPLEX*]

If, however, you try to use the type *VECTOR* [*EMPLOYEE*] you will get a compile-time error, assuming class *EMPLOYEE* is not a descendant of *NUMERIC*.

NUMERIC is a deferred class; this causes no particular problem. A generic derivation can use an effective actual parameter, as in the preceding examples, or a deferred one, as in *VECTOR* [*NUMERIC_COMPARABLE*], assuming the class given is a deferred heir of *NUMERIC*.

Similarly, a dictionary class could be declared as

class *DICTIONARY* [*G*, *H* –> *HASHABLE*] …

where the first parameter represents the type of the elements and the second represents the type of their keys. A class supporting sorting may be declared as

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

Playing it recursively

A nice twist of the *VECTOR* example appears if we ask whether it is possible to have a vector of vectors. Is the type *VECTOR* [*VECTOR* [*INTEGER*]] valid?

The answer follows from the preceding rules: only if the actual generic parameter conforms to *NUMERIC*. Easy — just make *VECTOR* itself inherit from *NUMERIC*:

Exercise E16.2, page 610.

```
indexing
    description: "Addable vectors"
class
    VECTOR [G –> NUMERIC]
inherit
    NUMERIC
… The rest as before …
```

It is indeed justified to consider vectors "numeric", since addition and multiplication operations give them a ring structure, with *zero* being a vector of *G* zeroes and *unity* a vector of *G* ones. The addition operation is precisely the vector **infix** "+" discussed earlier.

We can go further and use *VECTOR* [*VECTOR* [*VECTOR* [*INTEGER*]]] and so on — a pleasant recursive application of constrained genericity.

Unconstrained genericity revisited

Not all cases of genericity are constrained, of course. The original form of genericity, as in *STACK* [*G*] or *ARRAY* [*G*], is still available and is called unconstrained genericity. As the example of *DICTIONARY* [*G*, *H* –> *HASHABLE*] shows, a class can have both constrained and unconstrained generic parameters.

The discussion of constrained genericity enables us to understand the unconstrained case better. You have certainly come up with the rule by yourself as you were reading the above: from now on, **class** *C* [*G*] will be understood as a shorthand for **class** *C* [*G* –> *ANY*]. So if *G* is an unconstrained generic parameter (say in *STACK*) and *x* is an entity of type *G*, we know exactly what we can do with *x*: assign to or from it, compare it through = and /=, pass it as argument, and apply to it any of the universal features *clone*, *equal*, *deep_clone* and the like.

16.5 ASSIGNMENT ATTEMPT

Our next technique addresses regions of Object Land in which, for fear of tyrannical behavior, we cannot let simplistic type rules reign without opposition.

When type rules become obnoxious

The aim of the type rules introduced with inheritance is to yield statically verifiable dynamic behavior, so that a system that passes the compiler's checks will not end up applying inadequate operations to objects at run time.

See "TYPING FOR INHERITANCE", 14.3, page 472.

The two basic rules were introduced in the first inheritance chapter:

- The *Feature Call rule*: *x*.*f* is only valid if the base class of *x*'s type includes and exports a feature *f*.
- The *Type Conformance rule*: to pass *a* as argument to a routine, or to assign it to a certain entity, requires that *a*'s type conform to the expected type, that is to say, be based on a descendant class.

The Feature Call rule will not cause any problem; it is the fundamental condition for doing business with objects. Certainly, if we call a feature on an object, we need the reassurance that the corresponding class offers and exports such a feature.

The Type Conformance rule requires more attention. It assumes that we have all the type information that we need about the objects that we manipulate. Usually that is the case; after all, we create the objects, so we know who they are. But sometimes part of the information may be missing. In particular:

- In a polymorphic data structure we are only supposed to know the information that is common to all objects in the structure; but we may need to take advantage of some specific information that applies only to a particular object.
- If an object comes to our software from the outside world — a file, a network — we usually cannot trust that it has a certain type.

Let us explore examples of these two cases. First consider a polymorphic data structure such as a list of figures:

figlist: *LIST* [*FIGURE*]

This refers to the figure inheritance hierarchy of earlier chapters. What if someone asks us to find out what is the longest diagonal of all rectangles in the list (with some convention, say –1, if there are no rectangles)? We have no easy way of answering the request, since the expression *item* (*i*).*diagonal*, where *item* (*i*) is the *i*-th list element for some integer *i*, violates the Feature Call rule; *item* (*i*) is of type *FIGURE*, and there is no feature *diagonal* in class *FIGURE* — only in its proper descendant *RECTANGLE*.

The only solution with what we have seen so far is to change the class definitions so as to associate with each *FIGURE* class a code, different for each class, indicating the figure type. This is not an attractive approach.

Now for an example of the second kind. Assume a mechanism to store objects into a file, or transmit them over a network, such as the general-purpose *STORABLE* facility described in an earlier chapter. To retrieve an object or object structure you would use

See "Deep storage: a first view of persistence", page 250.

```
my_last_book: BOOK
…
my_last_book := retrieved (my_book_file)
```

WARNING: type-invalid assignment.

The result of function *retrieved* is of the Kernel library type *STORABLE*, but it might just as well be of type *ANY*; in either case it is only an ancestor of the object's generating type (that is to say, the type of which it is a direct instance), presumably *BOOK* or a descendant. But you are not expecting an *ANY* or a *STORABLE*: you are expecting a *BOOK*. The assignment to *my_last_book* violates the Type Conformance rule.

Even if instead of a general-purpose mechanism *retrieved* were a retrieval function specific to your application and declared with the intended type, you could still not trust its result blindly. Unlike an object that the software creates and then uses during the same session, guaranteeing type consistency thanks to the type rules, this one comes from the outside world. You may have chosen the wrong file name and retrieved an *EMPLOYEE* object rather than a *BOOK* object; or someone may have tampered with the file; or, if this is a network access, the transmission may have corrupted the data.

The challenge

It is clear from such examples that we may need a way to ascertain the type of an object.

The challenge is to satisfy this need — which arises only in specific cases, but in those cases is crucial — without sacrificing the benefits of the object-oriented style of development. In particular, we do not want to go back to the decried scheme

```
if "f is of type RECTANGLE" then
    …
elseif "f is of type CIRCLE" then
    …
etc.
```

the exact antithesis of such principles of modularity as Single Choice and Open-Closed. Two insights will help us avoid this risk:

- We do not need a general mechanism to determine the type of an object, at least not for the purposes described. In the cases under discussion we *know the expected type* of the object. So all we require is a way to test our expectation. We will check an object against a designated type; this is much more specific than asking for the object's type. It also means that we do *not* need to introduce into our language any operations on types, such as type comparisons — a frightening thought.

- As already noted, we should not tamper with the Feature Call rule. Under no circumstances is there any justification for applying a feature (“sending a message”) to an object unless we have statically ascertained that the corresponding class is equipped to deal with it. All that we will need is a looser version of the other rule, type conformance, allowing us to “try a type” and check the result.

The mechanism

Once again the notational mechanism follows directly from the analysis of the issue. We will use a new form of assignment, called **assignment attempt**, and written

target ?= *source*

to be compared with the usual assignment, *target* := *source*. The question mark indicates the tentative nature of the assignment. The effect of the assignment attempt, assuming that the entity *target* has been declared with type *T*, is the following:

- If *source* is attached to an object of a type conforming to *T*, attach that object to *target* exactly as a normal assignment would do.
- Otherwise (that is to say if the value of *source* is void, or is a reference to an object of a non-conforming type), make *target* void.

There is no type constraint on the instruction, except that the type *T* of the target must be a reference type. (Assignment attempt is polymorphic by nature, so an expanded target would not make sense.)

This instruction immediately and elegantly solves problems of the kind mentioned above. First, type-specific access to objects of a polymorphic structure:

```
maxdiag (figlist: LIST [FIGURE]): REAL is
        -- Maximum value of diagonals of rectangles in list; –1 if none
    require
        list_exists: figlist /= Void
    local
        r: RECTANGLE
    do
        from
            figlist.start; Result := –1.0
        until
            figlist.after
        loop
            r ?= figlist.item                  The assignment attempt
            if r /= Void then
                Result := Result.max (r.diagonal)
            end
            figlist.forth
        end
    end
```

This routine uses the usual iteration mechanisms on sequential structures: *start* to position the traversal on the first element if any, *after* to determine whether there is any element left to examine, *forth* to advance by one position, *item* (defined if **not** *after*) to yield the element at the current cursor position.

See "ACTIVE DATA STRUCTURES", 23.4, page 774.

The assignment attempt uses a local entity *r* of the appropriate type *RECTANGLE*. We know whether it succeeded by testing *r* against *Void*. Only if *r* is not void do we have a rectangle; then we can safely access *r.diagonal*. This scheme of testing for *Void* right after an assignment attempt is typical.

Note again that we never violate the Feature Call rule: any call of the form *r.diagonal* is guarded, statically, by a compiler check that *diagonal* is a feature of class *RECTANGLE*, and, dynamically, by a guarantee that *r* is not void — has an attached object.

A list element of type *SQUARE*, or some other descendant of *RECTANGLE*, will make *r* non-void, so that its diagonal will, rightly, participate in the computation.

The other example, using a general-purpose object retrieval function, is immediate:

```
my_last_book: BOOK
…
my_last_book ?= retrieved (my_book_file)
if my_last_book /= Void then
        … "Proceed normally with operations on my_last_book" …
else
        … "What we expected is not what we got"…
end
```

Compare with := in the first try (page 592)

Using assignment attempt properly

Assignment attempt is an indispensable tool for those cases — typically of the two kinds shown: elements of polymorphic data structures, and objects coming from the outside world — in which you cannot trust the statically declared type of an entity but need to ascertain at run time the type of the object actually attached to it.

Note how carefully the mechanism has been designed to discourage developers from using it to go back to the old case-by-case style. If you really want to circumvent dynamic binding, and test separately for each type variant, you can — but you have to work really hard at it; for example instead of the normal *f*.*display*, using the O-O mechanisms of polymorphism and dynamic binding, you would write

Warning: *this is* **not** *the recommended style*!

```
display (f: FIGURE) is
                -- Display f, using the algorithm adapted to its exact nature.
        local
                r: RECTANGLE; t: TRIANGLE; p: POLYGON; s: SQUARE
                sg: SEGMENT; e: ELLIPSE; c: CIRCLE; …
        do
                r ?= f; if r /= Void then "Apply the rectangle display algorithm" end
                t ?= f; if t /= Void then "Apply the triangle display algorithm" end
                c ?= f; if c /= Void then "Apply the circle display algorithm" end
                … etc …
        end
```

This scheme will in practice be even worse than it seems because the inheritance structure has several levels; for example an object of type *SQUARE* will make an assignment attempt *x* ?= *f* succeed for *x* of type *POLYGON* and *RECTANGLE* as well as *SQUARE*. So you must complicate the control structure to avoid multiple matches.

Because of the difficulty of writing such contorted uses of the assignment attempt, there is little risk that novice developers will mistakenly use it instead of the normal O-O scheme. But even advanced developers must remain alert to the possibility for misuse.

Java offers a mechanism called "narrowing" similar in some respects to assignment attempt. But in case of a type mismatch, instead of yielding a void value, it produces an exception. This looks like overkill, since an unsuccessful assignment is not an abnormal case, simply one of several possible and expected cases; it does not justify adding exception-handling code and setting in motion the exception machinery. Java also offers the *instanceof* operator to test for type conformance.

These mechanisms are used particularly extensively in Java because of the absence of genericity: you may have to rely on them, when retrieving elements from container data structures (even single-type), to check the elements' type against an expected type. Part of the reason may be that, in the absence of multiple inheritance, Java has no *NONE* class and hence no easy way to give the equivalent of *Void* a stable place in the type system.

16.6 TYPING AND REDECLARATION

When you redeclare a feature, you are not constrained to keep exactly the same signature. The precise rule will give us a further degree of flexibility.

So far we have seen redeclaration as a mechanism for substituting an algorithm for another — or, in the case of effecting a previously deferred routine, providing an algorithm where only a specification was originally given.

But we may also need to change the types involved, to support the general idea that a class may offer a more specialized version of an element declared in an ancestor. Let us study two typical examples, which will suggest the precise Type Redeclaration rule.

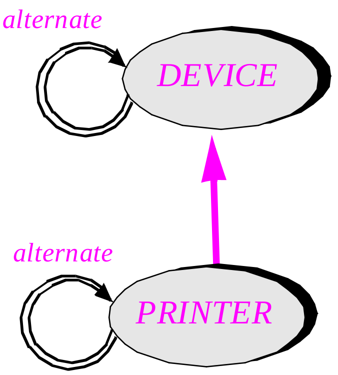
Devices and printers

Here is a simple example of type redefinition. Consider a notion of device including the provision that for every device there is an alternate, to be used if for some reason the first one is not available:

```
class DEVICE feature
    alternate: DEVICE

    set_alternate (a: DEVICE) is
            -- Designate a as alternate.
        do
            alternate := a
        end
    … Other features …
end -- class DEVICE
```

Printers are a special kind of device, justifying the use of inheritance. But the alternate of a printer can only be a printer — not a CD-ROM reader or a network transceiver! — so we must redefine the types:

```
class PRINTER inherit
    DEVICE
        redefine alternate, set_alternate
feature
    alternate: PRINTER
    set_alternate (a: PRINTER) is
            -- Designate a as alternate.
        … Body as in DEVICE …
    … Other features …
end -- class DEVICE
```



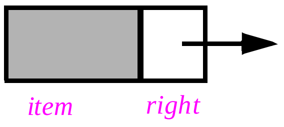
These redefinitions reflect the specializing nature of inheritance.

Linkable and bi-linkable elements

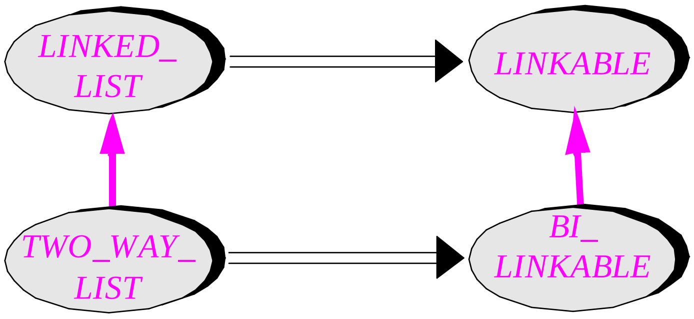
Here is another example, involving fundamental data structures. Consider the library class *LINKABLE* describing the linked list elements used in *LINKED_LIST*, one of the implementations of lists. A partial view of the class is:

```
indexing
    description: "Cells to be linked in a list"
class LINKABLE [G] feature
    item: G
    right: LINKABLE [G]
    put_right (other: LINKABLE [G]) is
            -- Put other to the right of current cell.
        do right := other end
    … Other features …
end -- class LINKABLE
```

A linkable cell

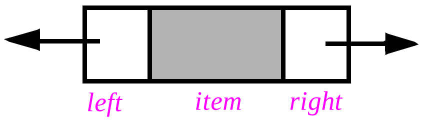


Some applications need lists chained both ways (each element linked to its successor and its predecessor). The corresponding class, *TWO_WAY_LIST*, is an heir of *LINKED_LIST*, and will need an heir *BI_LINKABLE* of *LINKABLE*:



Parallel hierarchies

A bi-linkable element is like a linkable but with one more field:



A bi-linkable cell

In a two-way list, bi-linkables should only be chained to bi-linkables (although it is harmless to introduce bi-linkables in a one-way list: this is polymorphism). So we should redefine *right* and *put_right* to guarantee that two-way lists remain homogeneous.

```
indexing
        description: "Cells to be linked both ways in a list"
class BI_LINKABLE [G] inherit
        LINKABLE [G]
                redefine right, put_right end
feature
        left, right: BI_LINKABLE [G]
        put_right (other: BI_LINKABLE [G]) is
                        -- Put other to the right of current element.
                do
                        right := other
                        if other /= Void then other.put_left (Current) end
                end
        put_left (other: BI_LINKABLE [G]) is
                        -- Put other to the left of current element
                … Left to the reader …
        … Other features …
invariant
        right = Void or else right.left = Current
        left = Void or else left.right = Current
end
```

(Try writing *put_left*. There is a pitfall! See appendix A.)

The Type Redeclaration rule

Although addressing abstractions of widely different kinds, the two examples show the same need for type redeclaration. Going down an inheritance hierarchy means specializing, and some types will follow that change pattern: types of routine arguments, such as *a* in *set_alternate* and *other* in *put_right*; types of queries, such as the attributes *alternate* and *right*, as well as functions.

The following rule captures this type aspect of redeclaration:

Type Redeclaration rule

A redeclaration of a feature may replace the type of the feature (if an attribute or function), or the type of a formal argument (if a routine), by any type that conforms to the original.

Here "conforms to" refers to the notion of type conformance, as defined on the basis of the descendant relation. The rule uses "or" non-exclusively: a function redeclaration may change both the type of the function's result and the type of one or more arguments.

The diagram is on page 597.

The permitted forms of redeclaration all go in the same direction: the direction of specialization. As illustrated by the last inheritance diagram, when you go down from *LINKED_LIST* to *TWO_WAY_LIST*, arguments and results will concomitantly go down from *LINKABLE* to *BI_LINKABLE*. In the first example, when you go from *DEVICE* to *PRINTER*, the attribute *alternate* and the argument of *set_alternate* follow. This explains the name often use to characterize this type redeclaration policy: **covariant typing**, where the "co" indicates that as we descend the inheritance diagram all the types go down in step.

Covariant typing, as we will see in the next chapter, creates for the compiler writer a few headaches which, fortunately, he can often avoid passing on to the software developer.

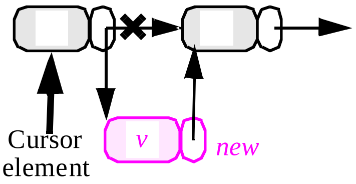
16.7 ANCHORED DECLARATION

The Type Redeclaration rule could make life quite unpleasant in some cases, and even cancel some of the benefits of inheritance. Let us see how and discover the solution — anchored declaration.

Type inconsistencies

As an example of the problems that may arise with the Type Redeclaration rule, consider the following example from *LINKED_LIST*. Here is the procedure for inserting a new element with a given value to the right of the current cursor position. Although there is nothing mysterious with the details, all you need to note at this stage is the need for a local entity *new* of type *LINKABLE*, representing the list cell to be created and added to the list.

```
put_right (v: G) is
        -- Insert an element of value v to the right of cursor position.
        -- Do not move cursor.
    require
        not after
    local
        new: LINKABLE [T]
    do
        !! new.make (v)
        put_linkable_right (new)
        …
    ensure
        … See appendix A …
    end
```



To insert a new item of value *v*, we must create a cell of type *LINKABLE* [*G*]; the actual insertion is carried out by the secret procedure *put_linkable_right*, which takes a *LINKABLE* as argument (and chains it to the cursor item using the *put_right* procedure of class *LINKABLE*.) This procedure performs the appropriate reference manipulations.

In proper descendants of *LINKED_LIST*, such as *TWO_WAY_LIST* or *LINKED_TREE*, procedure *put_right* should still be applicable. Unfortunately, it will not work as given: although the algorithm is still correct, the entity *new* should be declared and created as a *BI_LINKABLE* or a *LINKED_TREE* rather than a *LINKABLE*. So we must redefine and rewrite the whole procedure for each descendant — a particularly wasteful task since the new body will be identical to the original except for a single declaration (for *new*). For an approach meant to solve the reusability issue, this is a serious deficiency.

Application-oriented examples

It would be a mistake to believe that the spurious redefinition problem only arises for implementation-oriented structures such as *LINKED_LIST*. With any scheme of the form

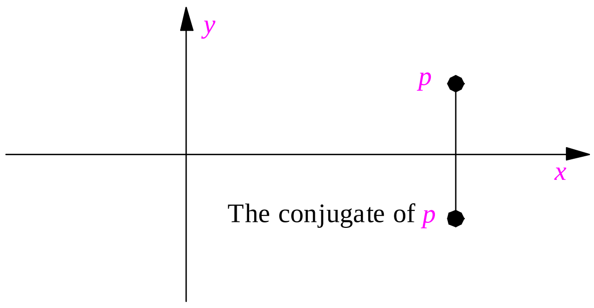
```
some_attribute: SOME_TYPE
set_attribute (a: SOME_TYPE) is do … end
```

a redefinition of *some_attribute* will imply the corresponding redefinition of *set_attribute*. In the case of *put_right* for *BI_LINKABLE*, the redefinition actually changed the algorithm (because of the necessity, if you chain O1 right to O2, also to chain O2 left to O1), but in many other cases, such as *set_alternate*, the new algorithm is identical to the original. This pattern is so common that we may expect to have to write many redundant routine bodies.

BI_LINKABLE's put_right (not to be confused with put_right for LINKED_LIST) was on page 597.

Here is one more example, showing how general the problem is (and not just tied to *set_xxx* procedures, themselves a result of information hiding principles). Assume we add to class *POINT* a function yielding the conjugate of a point, that is to say its mirror image across the horizontal axis:

A point and its conjugate



The function may appear as follows in *POINT*:

```
conjugate: POINT is
        -- Conjugate of current point
    do
        Result := clone (Current)   -- Get a copy of current point
        Result.move (0, –2*y)       -- Translate result vertically
    end
```

Now consider a descendant of *POINT*, perhaps *PARTICLE*, where particles have attributes other than *x* and *y*: perhaps a mass and a speed. Conceptually, *conjugate* is still applicable to particles; it should yield a particle result when applied to a particle argument. The conjugate of a particle is identical to that particle except for the *y* coordinate. But if we leave the function as it stands, it will not work for particles, since instructions such as the following violate the conformance rule:

```
p1, p2: PARTICLE; !! p1.make (…); …
p2 := p1.conjugate
```

In the underlined assignment, the source (right-hand side) is of type *POINT*, but the target is of type *PARTICLE*; the Type Conformance rule would require the reverse. So we must redefine *conjugate* in *PARTICLE*, for no purposes but type conformance.

Assignment attempt is not the solution here: although valid, it will result in a void *p2*, since the source object's type will, at execution time, be of type *POINT*, not *PARTICLE*.

A serious problem

If you look more closely at class *LINKED_LIST* in appendix A you will realize that the problem is of even greater scope. *LINKED_LIST* contains more than a few declarations referring to type *LINKABLE [G]*, and most will need to be redefined for two-way lists. For example a possible representation of a list keeps four references to linkable elements:

```
first_element, previous, active, next: LINKABLE [G]
```

All of these must be redefined in *TWO_WAY_LIST*, and similarly for other descendants. Many routines such as *put_right* take linkables as arguments, and must also be redefined. It seems that we will end up repeating in *TWO_WAY_LIST*, for purposes of declaration only, most of the features written for *LINKED_LIST*.

The notion of anchor

Unlike other type-related problems solved earlier in this chapter — such as the problems whose analysis led to constrained genericity and assignment attempt — the Case of the Useless Code Duplication is not that the type system prevents us from doing something that we need: thanks to the covariant Type Redeclaration rule we can redefine types to our heart's content, but this forces us to perform tedious code duplication.

To obtain a solution, we may note that the examples do require a type redefinition, but only one: all others ensue from it. The answer follows: provide a mechanism to declare an entity's type not absolutely, but *relative* to another entity.

This will be called an anchored declaration. An anchored type has the form

like *anchor*

where *anchor*, called the anchor of the declaration, is either a query (attribute or function) of the current class or the predefined expression *Current*. To declare *my_entity*: **like** *anchor* in a class *A*, where *anchor* is a query, means to declare it as being of the same type as *anchor*, but with the provision that any redefinition of *anchor* in a proper descendant will implicitly cause the same redefinition for *my_entity*.

So, assuming that *anchor* has been declared of some type *T*, the anchored declaration will cause *my_entity* to be treated within the text of class *A* as if it too had been declared of type *T*. If you only consider *A* there is no difference between the two declarations

- *my_entity*: **like** *anchor*
- *my_entity*: *X*

The difference only comes up in descendant classes of *A*. Being declared "like" *anchor*, *my_entity* will automatically follow any redefinition of the type of *anchor*, without the need for explicit redefinition by the author of the descendant class.

So if you find that a class includes a group of entities — attributes, function results, formal routine arguments, local entities — which descendants will have to redefine identically, you can dispense with all but one of the redefinitions: just declare all elements **like** the first one, and redefine only that first one. All others will automatically follow.

Let us apply this technique to *LINKED_LIST*. We can choose *first_element* as anchor for the other entities of type *LINKABLE* [*G*]. The attribute declarations become:

first_element: *LINKABLE* [*G*]
previous, *active*, *next*: **like** *first_element*

In the *put_right* procedure of *LINKED_LIST*, the local entity *new* should also be declared of type **like** *first_element*; this is the only change to the procedure. With these declarations, it suffices to redefine *first_element* as a *BI_LINKABLE* in class *TWO_WAY_LIST*, as a *LINKED_TREE* in class *LINKED_TREE* etc.; all entities declared **like** it follow automatically and need not be listed in the **redefine** clause. Neither is redefinition necessary any more for procedure *put_right*.

put_right from LINKED_LIST is on page 599.

Anchored declarations are an essential tool to preserve reusability in a statically typed object-oriented context.

Current as anchor

See "The current instance", page 181.

Instead of the name of a query you can use *Current* as anchor. The expression *Current*, as you know, denotes the current instance. An entity declared **like** *Current* in a class *A* will be treated within the class as being of type *A* and, in any descendant *B* of *A*, as being of type *B* — without any need for redefinition.

Original: page 600.

This form of anchored declaration addresses the remaining examples. To get the correct type for function *conjugate* in class *POINT*, amend its declaration to read

```
conjugate: like Current is
            … The rest exactly as before …
```

Then the result type of *conjugate* gets automatically redefined, in every descendant, to the associated type, for example type *PARTICLE* in class *PARTICLE*.

In class *LINKABLE*, you should similarly, in the earlier declarations

```
right: LINKABLE [G]
put_right (other: LINKABLE [G]) is…
```

replace *LINKABLE* [*G*] by **like** *Current*. Feature *left* in *BI_LINKABLE* should also be declared as **like** *Current*.

Class DEVICE, followed by PRINTER, appeared on page 595.

This scheme applies to many *set_attribute* procedures. In the *DEVICE* case we get:

```
class DEVICE feature
    alternate: like Current

    set_alternate (a: like Current) is
                -- Designate a as alternate.
            do
                alternate := a
            end
    … Other features …
end -- class DEVICE
```

No redefinition is then necessary in a descendant such as *PRINTER*.

Base classes revisited

With the introduction of anchored types, we need to extend the notion of base class of a type.

See "Types and classes", page 325.

You will remember the idea. At the beginning, classes and types were a single concept. That property, the starting point of the object-oriented method, remains *essentially* true, but we have had to extend the type system a little by adding generic parameters to classes. Every type is still fundamentally based on a class; for a generically derived type such as *LIST* [*INTEGER*] you obtain the base class by removing the actual generic parameters, giving *LIST* in this example. We also added expanded types, again based on classes; the base type of **expanded** *SOME_CLASS* […] is *SOME_CLASS*.

With anchored types we have another extension of the type system which, like the previous two, leaves intact the property that each type directly follows from a class. The base class of **like** *anchor* is the base class of the type of *anchor* in the current class; if *anchor* is *Current*, the base class is the enclosing class.

Rules on anchored types

There is no theoretical obstacle to accepting **like** *anchor* for an *anchor* that is itself of an anchored type; we must simply add a rule that prohibits cycles in declaration chains.

Initially the notation disallowed anchored anchors; although this rule is acceptable, the more liberal one that only prohibits anchor cycles allows more flexibility.

Let *T* be the type of *anchor* (given by the current class if *anchor* is *Current*). The type **like** *anchor* conforms to itself, and to *T*.

In the other direction, the only type that conforms to **like** *anchor* is itself. In particular *T* does not conform to **like** *anchor*. If we allowed

anchor, other: *T*; *x*: **like** *anchor*

…

!! *other*

x := *other*

WARNING: invalid assignment.

then in a descendant class where *anchor* is redefined to be of type *U* (conforming to *T* but based on a proper descendant) the assignment would attach *x* to an object of type *T*, whereas we should only accept objects of type *U* or conforming to *U*.

Of course you may assign to and from the anchor, as in *x* := *anchor* and *anchor* := *x*, and more generally between anchor-equivalent elements, defining *x* to be anchor-equivalent to *y* if it is *y* or declared as **like** *z* where *z* is (recursively) anchor-equivalent to *y*.

In the case of anchoring a formal argument or result of a routine, as in

r (*other*: **like** *Current*)

the actual argument in a call, such as *b* in *a*•*r* (*b*), must be anchor-equivalent to the target *a*.

The discussion of typing issues in chapter 17 will further explore the conformance properties of anchored types.

When not to use anchored declaration

Not every declaration of the form *x*: *A* within a class *A* should be replaced by *x*: **like** *Current*, and not every pair of features with the same type should be declared **like** one another.

An anchored declaration is a commitment: it indicates that whenever the anchor changes types in the future, the anchored entity must change too. As we just saw with the type rules, this commitment is not reversible: once you have declared an entity of type **like** *anchor* you cannot redefine its type any further (since the new type would have to conform to the original, and no type conforms to an anchored type but itself). As long as you have not chosen an anchored type, everything is still possible: if *x* is of type *T*, you can redeclare *x* as being of a conforming type *U* in a descendant; and you can in fact redeclare it as **like** *anchor* for some compatible *anchor* to close off further variations.

The pros and cons are clear. Anchoring an entity guarantees that you will never have to redeclare it for type purposes; but it binds it irrevocably to the type of the anchor. It is a typical case of trading freedom for convenience — like signing up with the military, or taking vows. (In a certain sense Faust declared himself **like** *Mephistopheles*.)

As an example of when anchoring may not be desirable, consider a feature *first_child* of trees, describing the first child of a given tree node. (In the construction of trees explained in the last chapter it comes from *first_element* of lists, originally of type *CELL* [*G*] or *LINKABLE* [*G*].) In a tree class it must be declared or redeclared to denote a tree. It may seem appropriate to use an anchored declaration:

first_child: **like** *Current*

This may, however, be too restrictive in practice. The tree class may have descendants, representing various kinds of tree (or tree node). Examples may include *UNARY_TREE* (nodes with just one child), *BINARY_TREE* (nodes with two children) and *BOUNDED_ARITY_TREE* (nodes with a bounded number of children). If *first_child* is anchored to *Current*, every node must have children of the same type: unary if it is unary, and so on.

This is probably not the desired effect, since you may want more flexible structures, permitting for example a binary node to have a unary child. This is obtained by declaring the feature not by an anchored declaration but simply as

first_child: *TREE* [*G*]

This solution is not restrictive: if you later need trees with nodes guaranteed to be all of the same type, you may leave *TREE* as it is and give it a new descendant *HOMOGENEOUS_TREE* which redefines *first_child* as

first_child: **like** *Current*

ensuring consistency of all the nodes in a tree.

To facilitate such a redefinition the other features of *TREE* representing nodes, such as *parent* and *current_child*, may and probably should be declared as **like** *first_child*; but *first_child* itself is not anchored in *TREE*.

A static mechanism

One last comment on anchored declaration, to dispel any possible misunderstanding that might remain about this mechanism: it is a purely static rule, not implying any change of object forms at run-time. The constraints may be checked at compile time.

Anchored declaration may be viewed as a syntactic device, avoiding many spurious redeclarations by having the compiler insert them. As it stands, it is an essential tool for reconciling reusability and type checking.

16.8 INHERITANCE AND INFORMATION HIDING

One last question needs to be answered to complete this panorama of inheritance issues: how inheritance interacts with the principle of information hiding.

For the other intermodule relation, client, the answer is clear: the author of each class is responsible for granting access privileges to the clients of the class. He specifies a policy for every feature: exported (generally available); selectively available; secret.

The policies

What happens to the export status of a feature when it is passed on to a descendant? Whatever you want to happen. Information hiding and inheritance are orthogonal mechanisms. A class *B* is free to export or hide any feature *f* that it inherits from an ancestor *A*. All possible combinations are indeed open:

- *f* exported in both *A* and *B* (although not necessarily to the same clients).
- *f* secret in both *A* and *B*.
- *f* secret in *A*, but exported, generally or selectively, in *B*.
- *f* exported in *A*, but secret in *B*.

The language rule is the following. By default — reflecting the most common case — *f* will keep the export status it had in *A*. But you may change this by adding an **export** subclause to the **inheritance** clause for *A*, as in

```
class B inherit
    A
        export {NONE} f end -- Makes f secret (it may have been exported in A)
    …
```

or

```
class B inherit
    A
        export {ANY} f end -- Makes f exported (it may have been secret in A)
    …
```

or

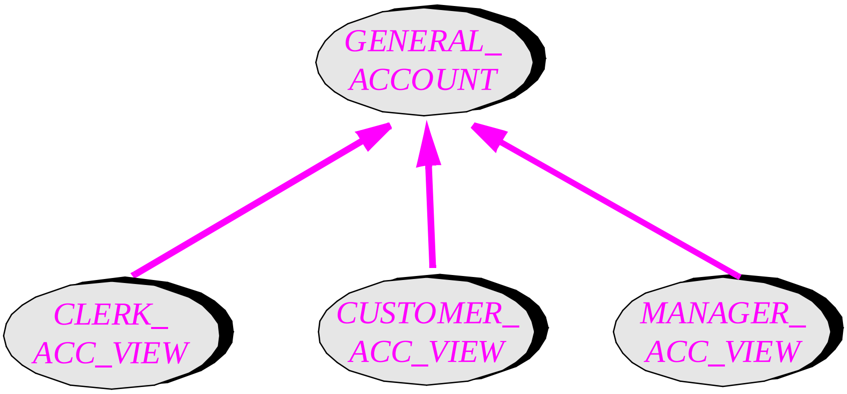
```
class B inherit
    A
        export {X, Y, Z} f end -- Makes f selectively available to certain classes
    …
```

Applications

A typical application of this flexibility is to provide several views of a certain basic notion.

Imagine a class *GENERAL_ACCOUNT* containing all the necessary tools for dealing with bank accounts, with procedures such as *open*, *withdraw*, *deposit*, *code* (for withdrawal from automatic teller machines), *change_code* etc.; but this class is not meant to be used directly by clients and so does not export anything. Descendants provide various views: they do not add any features, but simply differ in their export clauses. One will export *open* and *deposit* only, another will also include *withdraw* and *code*, and so on.

Views of a basic abstraction



See "Facility inheritance", page 832.

This scheme belongs to what the discussion of inheritance methodology will call "facility inheritance".

The notion of view is a classical one in databases, where it is often necessary to provide different users with different abstract notions of an underlying set of data.

"Trees are lists and list elements", page 525.

Classes sketched the discussion of multiple inheritance provide another application. Feature *right* of class *CELL* is secret in this class or, more precisely, is exported only to *LIST*; this is in fact true of all the features of *CELL*, since this class was initially designed only for the purpose of lists. But in class *TREE*, implemented as heir to *CELL* as well as *LIST*, *right* now denotes access to the right sibling of a node, a respectable public feature which should be exported.

Why the flexibility?

The policy of letting each descendant choose its own export policy (only by overriding the default, which keeps the parent's policy) makes type checking more difficult, as discussed in the next chapter, but provides the necessary flexibility to the class developer. Anything more restrictive hinders the goals of object-oriented software development.

Other solutions have been tried. Some O-O languages, beginning with a revision of Simula, let a class specify not only whether a feature will be exported to its clients, but whether it will be available to its descendants. The benefits are not clear. In particular:

- I am not aware of any published methodological advice on how to use this facility: when to bequeath a feature to descendants, when to hide it from them. A notational mechanism with no accompanying theory is of dubious value. (In comparison, the methodological rule governing information hiding policy for clients is limpid: what belongs to the underlying ADT should be exported; the rest should be secret.)

- More pragmatically, it seems that few developers in Simula and languages offering similar descendant restriction mechanisms bother to use them.

On closer examination, the lack of clear methodological guidelines is not surprising. Inheritance is the embodiment of the Open-Closed principle: a mechanism that enables you to pick an existing class, written yesterday or twenty years ago by you or by someone else, and discover that you can do something useful with it, far beyond what had been foreseen by the original design. Letting a class author define what eventual descendants may or may not use would eliminate this basic property of inheritance.

The example of *CELL* and *TREE* is typical: in the design of *CELL*, the only goal was to satisfy the needs of *LIST* classes, so *right* and *put_right* served only internal purposes. Only later did these features suddenly find a new application for a descendant, *TREE*. Without such openness, inheritance would lose much of its appeal.

If a class designer has no basis for deciding which features the class should pass on to its descendants, it would be even more preposterous for him to predict what they may or may not export to their *own clients*. Any such attempt is guesswork, with the knowledge that a wrong guess will make the descendant developers' task impossible.

These descendant developers have only one task: to provide their clients with the best possible class. In such an effort, inheritance is only a tool, enabling the developers to get a good result faster and better. The only rules of the game are the typing constraints and the assertions. Beyond that, anything goes. A useful ancestor feature is a godsend; whether the ancestor exported it or not is a matter between the ancestor and its own clients: the descendant developer could not care less.

In summary, the only policy compatible with the fundamental openness of inheritance seems to be the one described: let every descendant developer take its pick of ancestor features, and decide on its own export policy in the interest of its own clients.

Interface and implementation reuse

If you have read some of the more superficial O-O presentations, or follow newsgroup discussions, you may have been subjected to warnings against "inheriting implementation". But (as we shall see in more detail in the inheritance methodology chapter) there is nothing wrong about using inheritance for implementation.

See the figure on page 144.

There are two forms of reuse: reuse through interface, and reuse of implementation. We can understand them as follows from the theoretical picture. Any class is an implementation (possibly partial) of an abstract data type. It contains both the interface, as expressed by the ADT specification — the tip of the iceberg, if you remember the pictures that accompanied the presentation of information hiding and ADTs — and a set of implementation choices. Interface reuse means that you are content to rely on the specification; implementation reuse, that you need to rely on properties that belong to the class but not to the ADT.

You will not use these two possibilities for the same purposes. If you can reuse a certain set of facilities through their abstract properties only, and want to be protected against future changes in the reused elements, go for interface reuse. But in some cases you will just fall in love with a certain implementation because it provides the right basis for what you are building.

These forms of reuse are complementary, and are both perfectly legitimate.

The two inter-module relations of object-oriented software construction cover them: client provides interface reuse, inheritance supports implementation reuse.

Reusing an implementation is, of course, a more committing decision than just reusing an interface: you cannot reasonably expect, as in the other case, to be protected against changes in implementation! For that reason, inheriting is a more committing decision than just being a client. But in some cases it is what you need.

"WOULD YOU RATHER BUY OR INHERIT?", 24.2, page 812.

It is not always easy in practice to determine which one of the client and inheritance relations is appropriate in a certain case. A later chapter contains a detailed discussion of how to choose between them.

Rehabilitating implementation

Why the distrust of implementation inheritance? I have come to think that the answer is less technical than psychological. A thirty-year legacy of less-than-pristine programming has left us with a distrust of the very idea of implementation. The word itself has in some circles come to take on an almost indecent character, as if it were an insult to abstraction. (H.L. Mencken, in *The American Language*, similarly tells of how words such as *leg* came to be banished from late-nineteenth-century polite conversation for fear of the immodest connotations they evoke, even when the matter was limbs of a piano or of a chicken.) So we talk of analysis and design, and when we mention implementation at all we make sure to precede it by "but", "just" or "only", as in "this is just an implementation issue".

Object technology, of course, is the reverse of all that: producing implementations that are so elegant, useful and clearly correct that we do not have to watch our language. What for us is a program is often more abstract, more high-level, more understandable than much of what the analysis and design view presents as the highest of the high.

The two styles

In the picture that comes out of this discussion, we merge a set of originally separate distinctions.

We have two relations, client and inheritance; two forms of reuse, interface and implementation; information hiding, or not; protection against internal changes in provider modules, or not.

In each case the existence of a choice is not controversial, and both of the opposing options are defensible depending on the context. The slightly bolder step is to treat all these oppositions as just one:

Client	::	Inheritance
Reuse through interface	::	Reuse of implementation
Information hiding	::	No information hiding
Protection against changes in original implementation	::	No protection against original's changes

Merging four oppositions

Other approaches may be possible. But I do not know of any that is as simple, easy to teach and practical.

Selective exports

As a consequence of the information hiding properties of inheritance we must clarify the effects of selective exports. A class *A* which exports *f* selectively to *B*, as in

```
class A feature {B, …}
    f …
    …
```

makes *f* available to *B* for the implementation of *B*'s own features. What about the descendants of *B*? As we have just seen, they have access to *B*'s implementation; so they should be able to access whatever is accessible to *B* — for example *f*.

Experimental observation confirms this theoretical reasoning: what a class needs, its descendants tend to need too. But we do not want to have to come back and modify *A* (to extend its export clause) whenever a new descendant is added to *B*.

Here the principle of information hiding should be combined with the Open-Closed principle. The designer of *A* is entitled to decide whether or not to make *f* available to *B*; but he has no right to limit the freedom of the designer of the *B* line of classes to provide new extensions and implementation variants. In fact, what descendants *B* has, if any, is none his business. Hence the rule:

Selective Export Inheritance rule

A feature selectively exported to a class is available to all its descendants.

16.9 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- Invariants of parents are automatically added to a class's invariant.
- In the Design by Contract approach, inheritance, redefinition and dynamic binding introduce the concept of subcontracting.
- A routine redeclaration (redefinition or effecting) may keep or weaken the precondition; it may keep or strengthen the postcondition.
- An assertion redeclaration may only use **require else** (for or-ing of preconditions) and **and then** (for and-ing of postconditions). It may not use just **require** or **ensure**. In the absence of these clauses the routine keeps the original assertions.

- The universal class *GENERAL* and its customizable heir *ANY* provide redefinable features of interest to all developer-defined classes. *NONE* closes down the lattice.
- It is possible to freeze a feature to guarantee eternal semantic uniqueness.
- To entrust generic parameters with specific features, use constrained genericity.
- Assignment attempt makes it possible to verify dynamically that an object has the expected type. It should not be used as a substitute for dynamic binding.
- A descendant may redefine the type of any entity (attribute, function result, formal routine argument). The redefinition must be covariant, that is to say replace the original type with a conforming one, based on a descendant.
- Anchored declaration (**like** *anchor*) is an important part of the type system. facilitating the application of covariant typing and avoiding redundant redeclarations.
- Inheritance and information hiding are orthogonal mechanisms. Descendants may hide features that were exported by their ancestors, and export features that were secret.
- A feature available to a class is available to its descendants.

16.10 BIBLIOGRAPHICAL NOTE

See [Snyder 1986] for a different viewpoint on the relationship between inheritance and information hiding.

EXERCISES

E16.1 Inheriting for simplicity and efficiency

"A tolerant module", page 360.

Rewrite and simplify the protected stack example of an earlier chapter, making class *STACK3* a descendant rather than a client of *STACK* to avoid unneeded indirections. (**Hint**: see the rules governing the relationship between inheritance and information hiding.)

E16.2 Vectors

Write a class *VECTOR* describing vectors of a numeric type (ring), with the usual mathematical operations, and itself treated recursively as a numeric type. You may have to complete class *NUMERIC* for yourself (or get a version from [M 1994a]).

E16.3 Extract?

The assignment *y1* := *x1* is not permitted if *x1* is of a type *X*, *y1* of type *Y*, and *X* is a proper ancestor of *Y*. It might seem useful, however, to include a universal feature *extract* such that the instruction *y1*.*extract* (*x1*) copies the values of the fields of the object attached to *x1* to the corresponding fields in the object attached to *y1*, assuming neither reference is void.

Explain why the notation does not include such an *extract* feature. (**Hint**: examine correctness issues, in particular the notion of invariant.) Examine whether it is possible to design a satisfactory mechanism that achieves the same general goal in a different way.