14

Introduction to inheritance

*I*nteresting systems are seldom born into an empty world.

Almost always, new software expands on previous developments; the best way to create it is by imitation, refinement and combination. Traditional design methods largely ignored this aspect of system development. In object technology it is an essential concern.

The techniques studied so far are not enough. Classes do provide a good modular decomposition technique and possess many of the qualities expected of reusable components: they are homogeneous, coherent modules; you may clearly separate their interface from their implementation according to the principle of information hiding; genericity gives them some flexibility; and you may specify their semantics precisely thanks to assertions. But more is needed to achieve the full goals of reusability and extendibility.

For *reusability*, any comprehensive approach must face the problem of repetition and variation, analyzed in an earlier chapter. To avoid rewriting the same code over and over again, wasting time, introducing inconsistencies and risking errors, we need techniques to capture the striking commonalities that exist within groups of similar structures — all text editors, all tables, all file handlers — while accounting for the many differences that characterize individual cases.

For *extendibility*, the type system described so far has the advantage of guaranteeing type consistency at compile time, but prohibits combination of elements of diverse forms even in legitimate cases. For example, we cannot yet define an array containing geometrical objects of different but compatible types such as *POINT* and *SEGMENT*.

Progress in either reusability or extendibility demands that we take advantage of the strong conceptual relations that hold between classes: a class may be an extension, specialization or combination of others. We need support from the method and the language to record and use these relations. Inheritance provides this support.

A central and fascinating component of object technology, inheritance will require several chapters. In the present one we discover the fundamental concepts. The next three chapters will describe more advanced consequences: multiple inheritance, renaming, subcontracting, influence on the type system. Chapter 24 complements these technical presentations by providing the methodological perspective: how to use inheritance, and avoid misusing it.

14.1 POLYGONS AND RECTANGLES

To master the basic concepts we will use a simple example. The example is sketched rather than complete, but it shows the essential ideas well.

Polygons

Assume we want to build a graphics library. Classes in this library will describe geometrical abstractions: points, segments, vectors, circles, ellipses, general polygons, triangles, rectangles, squares and so on.

Consider first the class describing general polygons. Operations will include computation of the perimeter, translation, rotation. The class may look like this:

```
indexing
        description: "Polygons with an arbitrary number of vertices"
class POLYGON creation
        …
feature -- Access
        count: INTEGER
                        -- Number of vertices
        perimeter: REAL is
                        -- Length of perimeter
                do … end
feature -- Transformation
        display is
                        -- Display polygon on screen.
                do … end
        rotate (center: POINT; angle: REAL) is
                        -- Rotate by angle around center.
                do
                        … See next …
                end
        translate (a, b: REAL) is
                        -- Move by a horizontally, b vertically.
                do … end
        … Other feature declarations …
feature {NONE} -- Implementation
        vertices: LINKED_LIST [POINT]
                        -- Successive points making up polygon
invariant
        same_count_as_implementation: count = vertices.count
        at_least_three: count >= 3
                -- A polygon has at least three vertices (see exercise 14.2)
end
```

The attribute *vertices* yields the list of vertices; the choice of a linked list is only one possible implementation. (An array might be better.)

See also exercise E24.4, page 869.

Here is a possible implementation for a typical procedure, *rotate*. The procedure performs a rotation by a certain angle around a certain rotation center. To rotate a polygon, it suffices to rotate every vertex in turn:

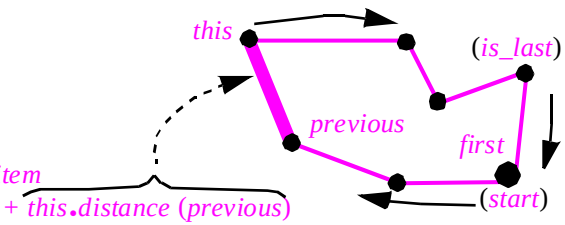
```
rotate (center: POINT; angle: REAL) is
        -- Rotate around center by angle.
    do
        from
            vertices.start
        until
            vertices.after
        loop
            vertices.item.rotate (center, angle)
            vertices.forth
        end
    end
```

The text of class POINT appeared on page 176.

To understand this procedure, note that feature *item* from *LINKED_LIST* yields the value of the currently active list element (where the cursor is). Since *vertices* is of type *LINKED_LIST* [*POINT*], *vertices.item* denotes a point, to which we may apply procedure *rotate* defined for class *POINT* in an earlier chapter. It is valid — and common — to give the same name, here *rotate*, to features of different classes, as the target of any feature always has a clearly defined type. (This is the O-O form of overloading.)

Another routine, more important for our immediate purposes, is the function to compute the perimeter of a polygon. Since our polygons have no special properties, the only way to compute their perimeter is to loop through their vertices and sum the edge lengths. Here is an implementation of *perimeter*:

```
perimeter: REAL is
        -- Sum of edge lengths
    local
        this, previous: POINT
    do
        from
            vertices.start; this := vertices.item
                check not vertices.after end -- A consequence of at_least_three
        until
            vertices.is_last
        loop
            previous := this
            vertices.forth
            this := vertices.item
            Result := Result + this.distance (previous)
        end
        Result := Result + this.distance (vertices.first)
    end
```



The loop simply adds the successive distances between adjacent vertices. Function *distance* was defined in class *POINT*. *Result*, representing the value to be returned by the function, is automatically initialized to 0 on routine entry. From class *LINKED_LIST* we use features *first* to get the first element, *start* to move the cursor to that first element, *forth* to advance it to the next, *item* to get the value of the element at cursor position, *is_last* to know whether the current element is the last one, *after* to know if the cursor is past the last element. As recalled by the **check** instruction the invariant clause *at_least_three* will guarantee that the loop starts and terminates properly: since it starts in a **not** *after* state, *vertices.item* is defined, and applying *forth* one or more time is correct and will eventually yield a state satisfying *is_last*, the loop's exit condition.

The list interface will be discussed in "ACTIVE DATA STRUCTURES", 23.4, page 774.

Rectangles

Now assume we need a new class representing rectangles. We could start from scratch. But rectangles are a special kind of polygon and many of the features are the same: a rectangle will probably be translated, rotated or displayed in the same way as a general polygon. Rectangles, on the other hand, also have special features (such as a diagonal), special properties (the number of vertices is four, the angles are right angles), and special versions of some operations (to compute the perimeter of a rectangle, we can do better than the above general polygon algorithm).

We can take advantage of this mix of commonality and specificity by defining class *RECTANGLE* as an **heir** to class *POLYGON*. This makes all the features of *POLYGON* — called a **parent** of *RECTANGLE* — by default applicable to the heir class as well. It suffices to give *RECTANGLE* an **inheritance clause**:

```
class RECTANGLE inherit
        POLYGON
feature
        … Features specific to rectangles …
end
```

The **feature** clause of the heir class does not repeat the features of the parent: they are automatically available because of the inheritance clause. It will only list features that are specific to the heir. These may be new features, such as *diagonal*; but they may also be redefinitions of inherited features.

The second possibility is useful for a feature that was already meaningful for the parent but requires a different form in the heir. Consider *perimeter*. It has a better implementation for rectangles: no need to compute four vertex-to-vertex distances; the result is simply twice the sum of the two side lengths. An heir that redefines a feature for the parent must announce it in the inheritance clause through a **redefine** subclause:

```
class RECTANGLE inherit
        POLYGON
                redefine perimeter end
feature
        …
end
```

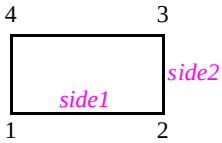
This allows the **feature** clause of *RECTANGLE* to contain a new version of *perimeter*, which will supersede the *POLYGON* version for rectangles. If the **redefine** subclause were not present, a new declaration of *perimeter* among the features of *RECTANGLE* would be an error: since *RECTANGLE* already has a *perimeter* feature inherited from *POLYGON*, this would amount to declaring a feature twice.

The *RECTANGLE* class looks like the following:

```
indexing
        description: "Rectangles, viewed as a special case of general polygons"
class RECTANGLE inherit
                POLYGON
                        redefine perimeter end
creation
        make
feature -- Initialization
        make (center: POINT; s1, s2, angle: REAL) is
                        -- Set up rectangle centered at center, with side lengths
                        -- s1 and s2 and orientation angle.
                do ... end
feature -- Access
        side1, side2: REAL
                -- The two side lengths

        diagonal: REAL
                -- Length of the diagonal

        perimeter: REAL is
                        -- Sum of edge lengths
                        -- (Redefinition of the POLYGON version)
                do
                        Result := 2 * (side1 + side2)
                end
invariant
        four_sides: count = 4
        first_side: (vertices.i_th (1)).distance (vertices.i_th (2)) = side1
        second_side: (vertices.i_th (2)).distance (vertices.i_th (3)) = side2
        third_side: (vertices.i_th (3)).distance (vertices.i_th (4)) = side1
        fourth_side: (vertices.i_th (4)).distance (vertices.i_th (1)) = side2
end
```



For a list, i_th (i) gives the element at position i (the i-th element, hence the name of the query).

Because *RECTANGLE* is an heir of *POLYGON*, all features of the parent class are still applicable to the new class: *vertices*, *rotate*, *translate*, *perimeter* (in redefined form) and any others. They do not need to be repeated in the new class.

This process is transitive: any class that inherits from *RECTANGLE*, say *SQUARE*, also has the *POLYGON* features.

Basic conventions and terminology

The following terms will be useful in addition to “heir” and “parent”.

Inheritance terminology

A **descendant** of a class *C* is any class that inherits directly or indirectly from *C*, including *C* itself. (Formally: either *C* or, recursively, a descendant of an heir of *C*.)

A **proper descendant** of *C* is a descendant other than *C* itself.

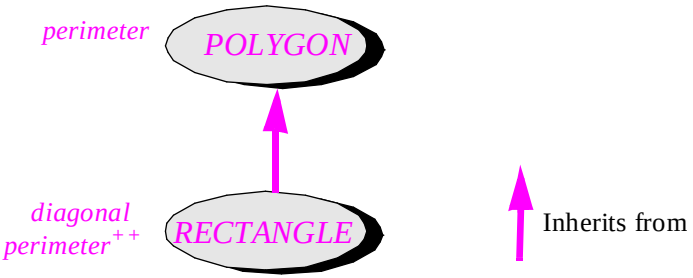
An **ancestor** of *C* is a class *A* such that *C* is a descendant of *A*. A **proper ancestor** of *C* is a class *A* such that *C* is a proper descendant of *A*.

In the literature you will also encounter the terms “subclass” and “superclass”, but we will stay away from them because they are ambiguous; sometimes “subclass” means heir (immediate descendant), sometimes it is used in the more general sense of proper descendant, and it is not always clear which. In addition, we will see that the “subset” connotation of this word is not always justified.

Associated terminology applies to the features of a class: a feature is either *inherited* (coming from a proper ancestors) or *immediate* (introduced in the class itself).

In graphical representations of object-oriented software structures, where classes are represented by ellipses (“bubbles”), inheritance links will appear as single arrows. This distinguishes them from links for the other basic inter-class relation, client, which as you will recall uses a double arrow. (For further distinction this book uses black for client and color for inheritance.)

An inheritance link



A redefined feature is marked ++, a convention from the Business Object Notation (B.O.N.).

The arrow points upward, from the heir to the parent; the convention, easy to remember, is that it represents the relation “inherits from”. In some of the literature you will find the reverse practice; although in general such choices of graphical convention are partly a matter of taste, in this case one convention appears definitely better than the other — in the sense that one suggests the proper relationship and the other may lead to confusion. An arrow is not just an arbitrary pictogram but indicates a unidirectional link, between the two ends of the arrow. Here:

- Any instance of the heir may be viewed (as we shall see in more detail) as an instance of the parent, but not conversely.
- The text of the heir will always mention the parent (as in the **inherit** clause above), but not conversely; it is in fact an important property of the method, resulting among others from the Open-Closed principle, that a class does not "know" the list of its heirs and other proper descendants.

Mathematically, the direction of the relationship is reflected in algebraic models for inheritance, which use a *morphism* (a generalization of the notion of function) from the heir's model to the parent's model — not the other way around. One more reason for drawing the arrow from the heir to the parent.

Although with complex systems we cannot have an absolute rule for class placement in inheritance diagrams, we should try whenever possible to position a class above its heirs.

Invariant inheritance

You will have noticed the invariant of class *RECTANGLE*, which expresses that the number of sides is four and that the successive edge lengths are *side1*, *side2*, *side1* and *side2*.

Class *POLYGON* also had an invariant, which still applies to its heir:

Invariant inheritance rule

The invariant property of a class is the boolean **and** of the assertions appearing in its **invariant** clause and of the invariant properties of its parents if any.

Because the parents may themselves have parents, this rule is recursive: in the end the full invariant of a class is obtained by **and**ing the invariant clauses of all its ancestors.

The rule reflects one of the basic characteristics of inheritance: to say that *B* inherits from *A* is to state that one may view any instance of *B* also as an instance of *A* (more on this property later). As a result, any consistency constraint applying to instances of *A*, as expressed by the invariant, also applies to instances of *B*.

In the example, the second clause (*at_least_three*) invariant of *POLYGON* stated that the number of sides must be at least three; this is subsumed by the *four_sides* subclause in *RECTANGLE*'s invariant clause, which requires it to be exactly four.

You may wonder what would happen if the heir's clause, instead of making the parent's redundant as here (since *count = 4* implies *count >= 3*), were incompatible with it, as with an heir of *POLYGON* that would introduce the invariant clause *count = 2*. The result is simply an inconsistent invariant, not different from what you get if you include, in the invariant of a single class, two separate subclauses that read *count >= 3* and *count = 2*.

Inheritance and creation

Although it was not shown, a creation procedure for *POLYGON* might be of the form

```
make_polygon (vl: LINKED_LIST [POINT]) is
        -- Set up with vertices taken from vl.
    require
        vl.count >= 3
    do
        … Initialize polygon representation from the items of vl …
    ensure
        -- vertices and vl have the same items (can be expressed formally)
    end
```

This procedure takes a list of points, containing at least three elements, and uses it to set up the polygon.

The procedure has been given a special name *make_polygon* to avoid any name conflict when *RECTANGLE* inherits it and introduces its own creation procedure *make*. This is not the recommended style; in the next chapter we will learn how to give the standard name *make* to the creation procedure in *POLYGON*, and use renaming in the inheritance clause of *RECTANGLE* to remove any name clash.

See "FEATURE RENAMING", 15.2, page 535.

The creation procedure of class *RECTANGLE*, shown earlier, took four arguments: a point to serve as center, the two side lengths and an orientation. Note that feature *vertices* is still applicable to rectangles; as a consequence, the creation procedure of *RECTANGLE* should set up the *vertices* list with the appropriate point values (the four corners, to be computed from the center, side lengths and orientation given as arguments).

The creation procedure for general polygons is awkward for rectangles, since only lists of four elements satisfying the invariant of class *RECTANGLE* would be acceptable. Conversely, the creation procedure for rectangles is not appropriate for arbitrary polygons. This is a common case: a parent's creation procedure is not necessarily right as creation procedure for the heir. The precise reason is easy to spot; it follows from the observation that a creation procedure's formal role is to establish the class invariant. The parent's creation procedure was required to establish the parent's invariant; but, as we have seen, the heir's invariant may be stronger (and usually is); we cannot then expect that the original procedure will guarantee the new invariant.

In the case of an heir adding new attributes, the creation procedures might need to initialize these attributes and so require extra arguments. Hence the general rule:

Creation Inheritance rule

An inherited feature's creation status in the parent class (that is to say, whether or not it is a creation procedure) has no bearing on its creation status in the heir.

An inherited creation procedure is still available to the heir as a normal feature of the class (although, as we shall see, the heir may prefer to make it secret); but it does not by default retain its status as a creation procedure. Only the procedures listed in the heir's own **creation** clause have that status.

In some cases, of course, a parent's creation procedure may still be applicable as a creation procedure; then you will simply list it in the creation clause:

```
class B inherit
        A
creation
        make
feature
        ...
```

where *make* is inherited — without modification — from *A*, which also listed it in its own **creation** clause.

An example hierarchy

For the rest of the discussion it will be useful to consider the *POLYGON*-*RECTANGLE* example in the context of a more general inheritance hierarchy of geometrical figure types, such as the one shown on the next page.

Figures have been classified into open and closed variants. Along with polygons, an example of closed figure is the ellipse; a special case of the ellipse is the circle.

Various features appear next to the applicable classes. The symbol $^{++}$, as noted, means "redefined"; the symbols $^{+}$ and * will be explained later.

In the original example, for simplicity, *RECTANGLE* was directly an heir of *POLYGON*. Since the sketched classification of polygons is based on the number of vertices, it seems preferable to introduce an intermediate class *QUADRANGLE*, at the same level as *TRIANGLE*, *PENTAGON* and similar classes. Feature *diagonal* can be moved up to the level of *QUADRANGLE*.

Note the presence of *SQUARE*, an heir to *RECTANGLE*, characterized by the invariant *side1 = side2*. Similarly, an ellipse  has two focuses (or foci), which for a circle  are the same point, giving *CIRCLE* an invariant property of the form *equal* (*focus1 = focus2*).

14.2 POLYMORPHISM

Inheritance hierarchies will give us considerable flexibility for the manipulation of objects, while retaining the safety of static typing. The supporting techniques, polymorphism and dynamic binding, address some of the fundamental issues of software architecture discussed in part B of this book. Let us begin with polymorphism.

Polymorphic attachment

"Polymorphism" means the ability to take several forms. In object-oriented development what may take several forms is a variable entity or data structure element, which will have the ability, at run time, to become attached to objects of different types, all controlled by the static declaration.

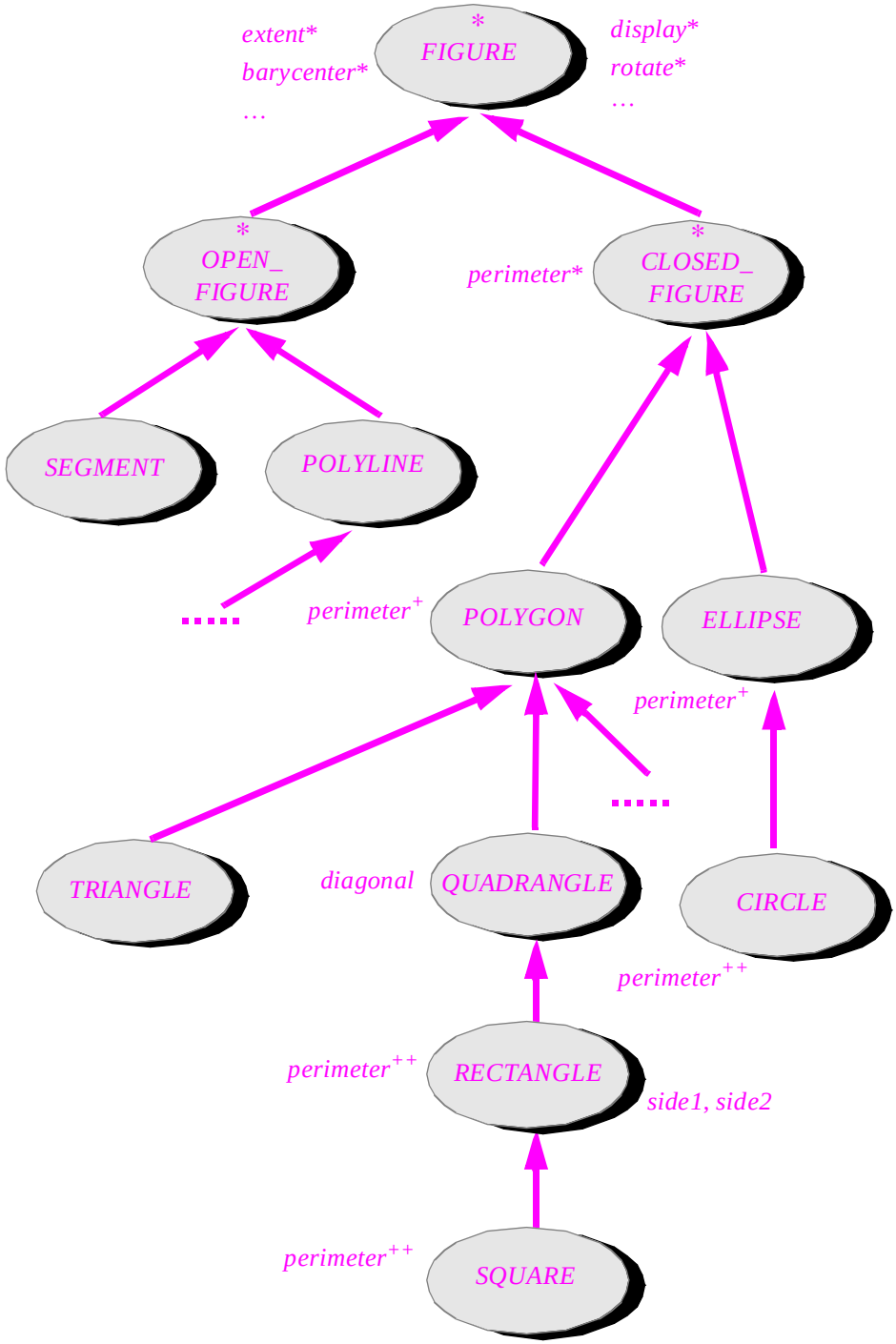


Figure type hierarchy

Assume, with the inheritance structure shown in the figure, the following declarations using short but mnemonic entity names:

p: *POLYGON*; *r*: *RECTANGLE*; *t*: *TRIANGLE*

Then the following assignments are valid:

p := *r*

p := *t*

These instructions assign to an entity denoting a polygon the value of an entity denoting a rectangle in the first case, a triangle in the second.

Such assignments, in which the type of the source (the right-hand side) is different from the type of the target (the left-hand side), are called **polymorphic assignments**. An entity such as *p* which appears in some polymorphic assignment is a **polymorphic entity**.

Before the introduction of inheritance, all our assignments were monomorphic (non-polymorphic): we could assign — in the various examples of earlier chapters — a point to a point, a book to a book, an account to an account. With polymorphism, we are starting to see more action on the attachment scene.

The polymorphic assignments taken as example are legitimate: the inheritance structure permits us to view an instance of *RECTANGLE* or *TRIANGLE* as an instance of *POLYGON*. We say that the type of the source **conforms to** the type of the target. In the reverse direction, as with *r* := *p*, the assignment would not be valid. This fundamental type rule will be discussed in more detail shortly.

Instead of an assignment, you may achieve polymorphism through argument passing, as with a call of the form *f* (*r*) or *f* (*t*) and a feature declaration of the form

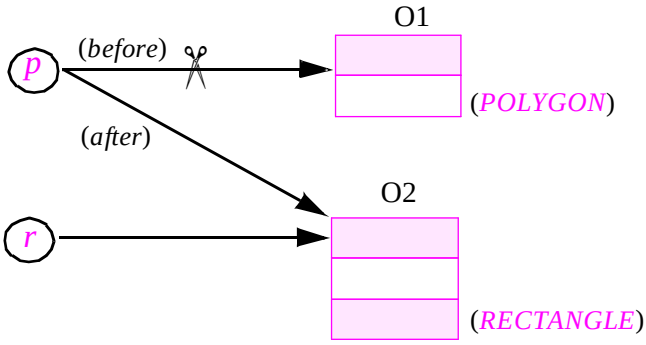
f (*p*: *POLYGON*) **is do** … **end**

As you will remember, assignment and argument passing have the same semantics, and are together called *attachment*; we can talk of *polymorphic* attachment when the source and target have different types.

What exactly happens during a polymorphic attachment?

All the entities appearing in the preceding cases of polymorphic attachment are of reference types: the possible values for *p*, *r* and *t* are not objects but references to objects. So the effect of an assignment such as *p* := *r* is simply to reattach a reference:

Polymorphic reference reattachment



So in spite of the name you should not imagine, when thinking of polymorphism, some run-time transmutation of objects. Once created, an object never changes its type. Only references do so by getting reattached to objects of different types. This also means that polymorphism does not carry any efficiency penalty; a reference reattachment — a very fast operation — costs the same regardless of the objects involved.

Polymorphic attachments will only be permitted for targets of a reference type — not for the other case, expanded types. Since a descendant class may introduce new attributes, the corresponding instances may have more fields; the last figure suggested this by showing the *RECTANGLE* object bigger than the *POLYGON* object. Such differences in object size do not cause any problem if all we are reattaching is a reference. But if instead of a reference *p* is of an expanded type (being for example declared as **expanded** *POLYGON*), then the value of *p* is directly an object, and any assignment to *p* would overwrite the contents of that object. No polymorphism is possible in that case.

See "COMPOSITE OBJECTS AND EXPANDED TYPES", 8.7, page 254.

Polymorphic data structures

Consider an array of polygons:

poly_arr: *ARRAY* [*POLYGON*]

When you assign a value *x* to an element of the array, as in

poly_arr.*put* (*x*, *some_index*)

(for some valid integer index value *some_index*), the specification of class *ARRAY* indicates that the assigned value's type must conform to the actual generic parameter:

This is extracted from class ARRAY as it appears on page 373.

```
class ARRAY [G] creation
    …
feature -- Element change
    put (v: G; i: INTEGER) is
            -- Assign v to the entry of index i
    …
end -- class ARRAY
```

Because *v*, the formal argument corresponding to *x*, is declared of type *G* in the class, and the actual generic parameter corresponding to *G* is *POLYGON* in the case of *poly_arr*, the type of *x* must conform to *POLYGON*. As we have seen, this does not require *x* to be of type *POLYGON*: any descendant of *POLYGON* is acceptable.

So assuming that the array has bounds 1 and 4, that we have declared some entities as

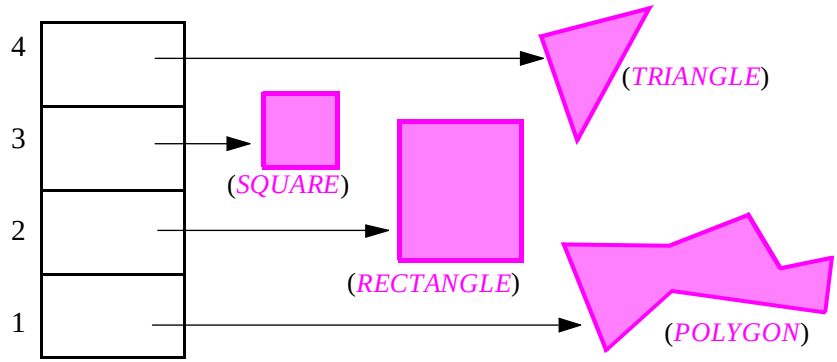
p: *POLYGON*; *r*: *RECTANGLE*; *s*: *SQUARE*; *t*: *TRIANGLE*

and created the corresponding objects, we may execute

```
poly_arr.put (p, 1)
poly_arr.put (r, 2)
poly_arr.put (s, 3)
poly_arr.put (t, 4)
```

yielding an array of references to objects of different types:

A polymorphic array



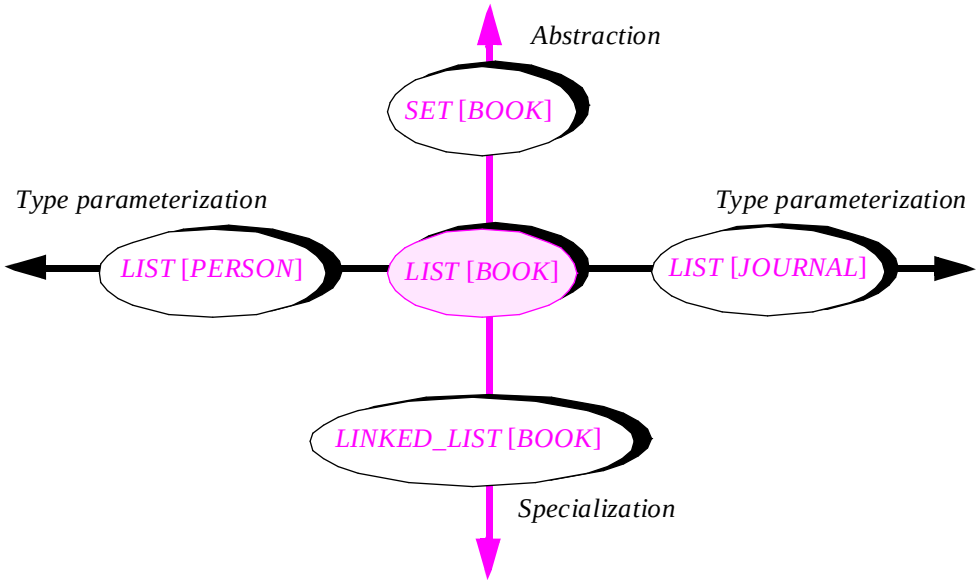
The graphical objects have been represented by the corresponding geometrical shapes rather than the usual multi-field object diagrams.

Such a data structure, containing objects of different types (all of them descendants of a common type), are called **polymorphic data structures**. We will encounter many examples in later discussions. The use of arrays is just one possibility; any other container structure, such as a list or stack, can be polymorphic in the same way.

The introduction of polymorphic data structures achieves the aim, stated at the beginning of chapter 10, of combining genericity and inheritance for maximum flexibility and safety. It is worth recalling the figure that illustrated the idea:

Dimensions of generalization

(See page 317.)



Types that were informally called *SET_OF_BOOKS* and the like on the earlier figure have been replaced with generically derived types, such as *SET [BOOK]*.

This combination of genericity and inheritance is powerful. It enables you to describe object structures that are as general as you like, but no more. For example:

- *LIST* [*RECTANGLE*]: may contain squares, but not triangles.
- *LIST* [*POLYGON*]: may contain squares, rectangles, triangles, but not circles.
- *LIST* [*FIGURE*]: may contain instances of any of the classes in the *FIGURE* hierarchy, but not books or bank accounts.
- *LIST* [*ANY*]: may contain objects of arbitrary types.

The last case uses class *ANY*, which by convention is an ancestor to all classes.

We will study ANY in "Universal classes", page 580.

By choosing as actual generic parameter a class at a varying place in the hierarchy, you can set the limits of what your container will accept.

14.3 TYPING FOR INHERITANCE

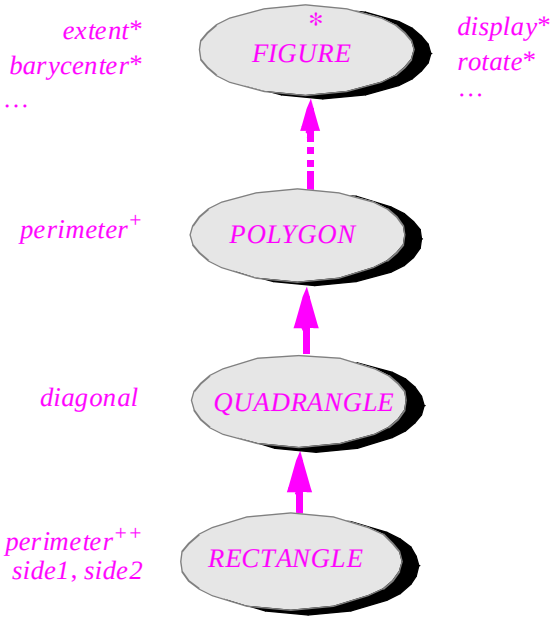
That the remarkable flexibility provided by inheritance does not come at the expense of reliability follows from the use of a *statically typed* approach, in which we guarantee at compile time that no incorrect run-time type combination can occur.

Type consistency

Inheritance is consistent with the type system. The basic rules are easy to explain on the above example. Assume the following declarations:

p: *POLYGON*
r: *RECTANGLE*

referring to the earlier inheritance hierarchy, of which the relevant extract is this:



Then the following are valid:

- *p*.*perimeter*: no problem, since *perimeter* is defined for polygons.
- *p*.*vertices*, *p*.*translate* (…), *p*.*rotate* (…) with valid arguments.
- *r*.*diagonal*, *r*.*side1*, *r*.*side2*: the three features considered are declared at the *RECTANGLE* or *QUADRANGLE* level.
- *r*.*vertices*, *r*.*translate* (…), *r*.*rotate* (…): the features considered are declared at the *POLYGON* level or above, and so are applicable to rectangles, which inherit all polygon features.
- *r*.*perimeter*: same case as the previous one. The version of the function to be called here is the redefinition given in *RECTANGLE*, not the original in *POLYGON*.

The following feature calls, however, are illegal since the features considered are not available at the polygon level:

p.*side1*

p.*side2*

p.*diagonal*

These cases all result from the first fundamental typing rule:

Feature Call rule

In a feature call *x*.*f*, where the type of *x* is based on a class *C*, feature *f* must be defined in one of the ancestors of *C*.

Recall that the ancestors of *C* include *C* itself. The phrasing "where the type of *x* is based on a class *C* " is a reminder that a type may involve more than just a class name if the class is generic: *LINKED_LIST* [*INTEGER*] is a class type "based on" the class name *LINKED_LIST*; the generic parameters play no part in this rule.

Like all other validity rules reviewed in this book, the Feature Call rule is static; this means that it can be checked on the sole basis of a system's text, rather than through run-time controls. The compiler (which typically is the tool performing such checking) will reject classes containing invalid feature calls. If we succeed in defining a set of tight-proof type rules, there will be no risk, once a system has been compiled, that its execution will ever apply a feature to an object that is not equipped to handle it.

Chapter 17 discusses typing.

Static typing is one of object technology's main resources for achieving the goal of software reliability, introduced in the first chapter of this book.

It has already been mentioned that not all approaches to object-oriented software construction are statically typed; the best-known representative of *dynamically typed* languages is Smalltalk, which has no static Feature Call rule but will let an execution terminate abnormally in the case of a "message not understood" run-time error. The chapter on typing will compare the various approaches further.

Limits to polymorphism

Unrestrained polymorphism would be incompatible with a static notion of type. Inheritance governs which polymorphic attachments are permissible.

The polymorphic attachments used as examples, such as *p := r* and *p := t*, all had as source type a descendant of the target's class. We say that the source type *conforms* to the target class; for example *SQUARE* conforms to *RECTANGLE* and to *POLYGON* but not to *TRIANGLE*. This notion has already been used informally but we need a precise definition:

Definition: conformance

A type *U* conforms to a type *T* only if the base class of *U* is a descendant of the base class of *T*; also, for generically derived types, every actual parameter of *U* must (recursively) conform to the corresponding formal parameter in *T*.

Why is the notion of descendant not sufficient? The reason is again that since we encountered genericity we have had to make a technical distinction between types and classes. Every type has a *base class*, which in the absence of genericity is the type itself (for example *POLYGON* is its own base class), but for a generically derived type is the class from which the type is built; for example the base class of *LIST [POLYGON]* is *LIST*. The second part of the definition indicates that *B [Y]* will conform to *A [X]* if *B* is a descendant of *A* and *Y* a descendant of *X*.

See "Types and classes", page 325.

Note that, as every class is a descendant of itself, so does every type conform to itself.

With this generalization of the notion of descendant we get the second fundamental typing rule:

Type Conformance rule

An attachment of target *x* and source *y* (that is to say, an assignment *x := y*, or the use of *y* as an actual argument to a routine call where the corresponding formal argument is *x*) is only valid if the type of *y* conforms to the type of *x*.

The Type Conformance rule expresses that you can assign from the more specific to the more general, but not conversely. So *p := r* is valid but *r := p* is invalid.

The rule may be illustrated like this. Assume I am absent-minded enough to write just "Animal" in the order form I send to the Mail-A-Pet company. Then, whether I receive a dog, a ladybug or a killer whale, I have no right to complain. (The hypothesis is that classes *DOG* etc. are all descendants of *ANIMAL*.) If, on the other hand, I specifically request a dog, and the mailman brings me one morning a box with a label that reads *ANIMAL*, or perhaps *MAMMAL* (an intermediate ancestor), I am entitled to return it to the sender — even if from the box come unmistakable sounds of yelping and barking. Since my order was not fulfilled as specified, I shall owe nothing to Mail-A-Pet.

Instances

The original discussion was "The mold and the instance", page 167.

With the introduction of polymorphism we need a more specific terminology to talk about instances. Informally, the instances of a class are the run-time objects built according to the definition of a class. But now we must also consider the objects built from the definition of its proper descendants. Hence the more precise definition:

Definition: direct instance, instance

A direct instance of a class *C* is an object produced according to the exact definition of *C*, through a creation instruction !! *x*… where the target *x* is of type *C* (or, recursively, by cloning a direct instance of *C*).

An instance of *C* is a direct instance of a descendant of *C*.

The last part of this definition implies, since the descendants of a class include the class itself, that a direct instance of *C* is also an instance of *C*.

So the execution of

```
p1, p2: POLYGON; r: RECTANGLE
…
!! p1 …; !! r …; p2 := r
```

will create two instances of *POLYGON* but only one direct instance (the one attached to *p1*). The other object, to which the extract attaches both *p2* and *r*, is a direct instance of *RECTANGLE* — and so an instance of both *POLYGON* and *RECTANGLE*.

Although the notions of instance and direct instance are defined above for a class, they immediately extend to any type (with a base class and possible generic parameters).

Polymorphism means that an entity of a certain type may become attached not only to direct instances of that type, but to arbitrary instances. We may indeed consider that the role of the type conformance rule is to ensure the following property:

Static-dynamic type consistency

An entity declared of a type *T* may at run time only become attached to instances of *T*.

Static type, dynamic type

The name of the last property suggests the concepts of "static type" and "dynamic type". The type used to declare an entity is the *static type* of the corresponding reference. If, at run time, the reference gets attached to an object of a certain type, this type becomes the *dynamic type* of the reference.

So with the declaration *p*: *POLYGON*, the static type of the reference that *p* denotes is *POLYGON*; after the execution of !! *p*, the dynamic type of that reference is also *POLYGON*; after the assignment *p* := *r*, with *r* of type *RECTANGLE* and non-void, the dynamic type is *RECTANGLE*.

The Type Conformance rule states that the dynamic type must always conform to the static type.

To avoid any confusion remember that we are dealing with three levels: an *entity* is an identifier in the class text; at run time its value is a *reference* (except in the expanded case); the reference may get attached to an *object*. Then:

See "States of a reference", page 240.

- An object only has a dynamic type, the type with which it has been created. That type will never change during the object's lifetime.
- At any time during execution, a reference has a dynamic type, the type of the object to which it is currently attached (or the special type *NONE* if the reference is void). The dynamic type may change as a result of reattachment operations.
- Only an entity has both a static type and dynamic types. Its static type is the type with which it was declared: *T* if the declaration was *x*: *T*. Its dynamic type at some execution-time instant is the type of its reference value, meaning the type of the attached object.

  In the expanded case there is no reference; the value of *x* is an object of type *T*, and *x* has *T* as both its static type and as its only possible dynamic type.

NONE will be seen in "The bottom of the pit", page 582.

Are the restrictions justified?

The two typing rules may sometimes seem too restrictive. For example, the second instruction in both of the following sequences will be statically rejected:

R1 • *p*:= *r*; *r* := *p*

R2 • *p* := *r*; *x* := *p*.*diagonal*

In R1, we refuse to assign a polygon to a rectangle entity even though that polygon happens at run time to be a rectangle (like refusing to accept a dog because it comes in a box marked "animal"). In R2, we decide that *diagonal* is not applicable to *p* even though at run time it would in fact be — as it were by accident.

But closer examination of these examples confirms that the rules are justified. If you attach a reference to an object, better avoid later problems by making sure that they are of compatible types. And if you want to apply a rectangle operation, why not declare the target as a rectangle?

In practice, cases of the form R1 and R2 are unlikely. Assignments such as *p* := *r* will normally occur as part of some control structure that depends on run-time conditions, such as user input. A more realistic polymorphic scheme may look like this:

```
!! r.make (…); …
screen.display_icons                -- Display icons representing various polygons
screen.wait_for_mouse_click         -- Wait for the user to click the mouse button
x := screen.mouse_position          -- Find out at what position
                                    -- the mouse was clicked
chosen_icon := screen.icon_where_is (x)  -- Find out what icon appears at the
                                         -- mouse's position
```

```
if chosen_icon = rectangle_icon then
        p := r
elseif …
        p := "Some other type of polygon" …
°
end
… Uses of p, for example p.display, p.rotate, …
```

On the last line, *p* can denote arbitrary polygons, so you should only apply general *POLYGON* features. Clearly, operations valid for rectangles only, such as *diagonal*, should be applied to *r* only (for example in the first clause of the **if**). Where *p* as such is going to be used, in the instructions following the **if** instruction, only operations defined for all variants of polygons are applicable to it.

In another typical case, *p* could just be a formal routine argument:

some_routine (*p*: *POLYGON*) **is**…

and you execute a call *some_routine* (*r*), valid as per the Type Conformance rule; but when you write the routine you do not know about this call. In fact a call *some_routine* (*t*) for *t* or type *TRIANGLE*, or any other descendant of *POLYGON* for that matter, would be equally valid, so all you can assume is that *p* represents some kind of polygon — *any* kind of polygon. It is quite appropriate, then, that you should be restricted to applying *POLYGON* features to *p*.

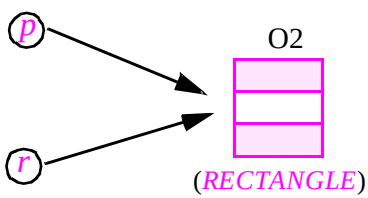
It is in this kind of situation — where you cannot predict the exact type of the attached object — that polymorphic entities such as *p* are useful.

Can ignorance be bliss?

It is worthwhile reinforcing the last few points a bit since the concepts now being introduced will be so important in the rest of our discussion. (There will be nothing really new in this short section, but it should help you understand the basic concepts better, preparing you for the more advanced ones which follow.)

If you are still uneasy at the impossibility of writing *p.diagonal* even after a call *p* :=*r* — case R2 — you are not alone; this is a shock to many people when they start grappling with these concepts. We know that *p* is a rectangle because of the assignment, so why may we not access its diagonal? For one thing, that would be useless. After the polymorphic assignment, as shown in the following extract from an earlier figure, the same *RECTANGLE* object now has two names, a polygon name *p* and a rectangle name *r*:

After a polymorphic attachment



In such a case, since you do know that the object O2 is a rectangle and have access to it through its rectangle name *r*, why would you write a diagonal access operation in the form *p.diagonal*? This is uninteresting since you can just write it as *r.diagonal*; using the object's official rectangle name removes any doubt as to the validity of applying a rectangle operation. Using the polygon name *p*, which could just as well denote a triangle object, brings nothing and introduces uncertainty.

Polymorphism, in fact, *loses* information: when as a result of the assignment *p := r* you are able to refer to the rectangle object O2 under its polygon name *p*, you have lost something precious: the ability to use rectangle-specific features. What then is the purpose? In this case, there is none. The only interesting application, as noted, arises when you do not know for sure what kind of polygon *p* is, as a result of a conditional instruction **if** *some_condition* **then** *p:= r* **else** *p := something_else …*, or because *p* is a formal routine argument and you do not know what the actual argument will be. But then in such cases it would be incorrect and dangerous to apply to *p* anything else than *POLYGON* features.

To continue with the animal theme, imagine that someone asks "do you have a pet?" and you answer "yes, a cat!". This is similar to a polymorphic assignment, making a single object known through two names of different types: "*my_pet*" and "*my_cat*" now denote the same animal. But they do not serve the same purpose; the first has less information than the second. You can use either name if you call the post-sales division of Mail-A-Pet, Absentee Owner Department ("*I am going on holiday*; *what's your price for keeping my_pet* [or: *my_cat*] *for two weeks*"); but if you phone their Destructive Control Department to ask "*Can I bring my_pet for a de-clawing Tuesday?*", you probably will not get an appointment until the employee has made you confirm that you really mean *my_cat*.

When you want to force a type

In some special cases there may be a need to try an assignment going against the grain of inheritance, and accept that the result is not guaranteed to yield an object. This does not normally occur, when you are properly applying the object-oriented method, with objects that are internal to a certain software element. But you might for example receive over the network an object advertized to be of a certain type; since you have no control over the origin of the object, static type declarations will guarantee nothing, and you must *test* the type before accepting it.

When we receive that box marked "Animal" rather than the expected "Dog", we might be tempted to open the "Animal" box anyway and take our chances, knowing that if its content is not the expected dog we will have forfeited our right to return the package, and depending on what comes out of it we may not even live to tell the story.

Such cases require a new mechanism, **assignment attempt**, which will enable us to write instructions of the form *r ?= p* (where *?=* is the symbol for assignment attempt, versus := for assignment), meaning "do the assignment if the object type is the expected one for *r*, otherwise make *r* void". But we are not equipped yet to understand how this instruction fits in the proper use of the object-oriented method, so we will have to return to it in a subsequent chapter. (Until then, you did not read about it here.)

See "ASSIGNMENT ATTEMPT", 16.5, page 591.

Polymorphic creation

The introduction of inheritance and polymorphism suggests a small extension to the mechanism for creating objects, allowing direct creation of objects of a descendant type.

See "The creation instruction", page 232, and "CREATION PROCEDURES", 8.4, page 236.

The basic creation instruction, as you will recall, is of one of the forms

```
!! x
!! x.make (…)
```

where the second form both assumes and requires that the base class of *x*'s type *T* contain a **creation** clause listing *make* as one of the creation procedures. (A creation procedure may of course have any name; *make* is the recommended default.) The effect of the instruction is to create a new object of type *T*, initialize it to the default values, and attach it to *x*. In addition, the second form will apply *make*, with the arguments given, to the just created and initialized object.

Assume that *T* has a proper descendant *U*. We may want to use *x* polymorphically and, in some cases, make it denote a newly created direct instance of *U* rather than *T*. A possible solution uses a local entity of type *U*:

```
some_routine (…) is
        local
                u_temp: U
        do
                …; !! u_temp.make (…); x := u_temp; …
        end
```

This works but is cumbersome, especially in a multi-choice context where we may want to attach *x* to an instance of one of several possible descendant types. The local entities, *u_temp* above, play only a temporary part; their declarations and assignments clutter up the software text. Hence the need for a variant of the creation instruction:

```
! U ! x
! U ! x.make (…)
```

The effect is the same as with the !! forms, except that the created object is a direct instance of *U* rather than *T*. The constraint on using this variant is obvious: type *U* must conform to type *T* and, in the second form, *make* must be defined as a creation procedure in the base class of *U*; if that class indeed has one or more creation procedures, only the second form is valid. Note that whether *T*'s own base class has creation procedures is irrelevant here; all that counts is what *U* requires.

A typical use involves creation of an instance of one of several possible types:

```
f: FIGURE
…
"Display a set of figure icons"
if chosen_icon = rectangle_icon then
        ! RECTANGLE ! f
else if chosen_icon = circle_icon then
        ! CIRCLE ! f
else
        …
end
```

This new form of creation instruction suggests introducing the notion of **creation type** of a creation instruction, denoting the type of the object that will be created:

- For the implicit-type form !! *x* …, the creation type is the type of *x*.
- For the explicit-type form ! *U* ! *x* …, the creation type is *U*.

14.4 DYNAMIC BINDING

Dynamic binding will complement redefinition, polymorphism and static typing to make up the basic tetralogy of inheritance.

Using the right variant

Operations defined for all polygons need not be *implemented* identically for all variants. For example, *perimeter* has different versions for general polygons and for rectangles; let us call them $perimeter_{POL}$ and $perimeter_{RECT}$. Class *SQUARE* will also have its own variant (yielding four times the side length). You may imagine further variants for other special kinds of polygon. This immediately raises a fundamental question: what happens when a routine with more than one version is applied to a polymorphic entity?

In a fragment such as

```
!! p.make (…); x := p.perimeter
```

it is clear that $perimeter_{POL}$ will be applied. It is just as clear that in

```
!! r.make (…); x := r.perimeter
```

$perimeter_{RECT}$ will be applied. But what if the polymorphic entity *p*, statically declared as a polygon, dynamically refers to a rectangle? Assume you have executed

```
!! r.make (…)
p := r
x := p.perimeter
```

The rule known as **dynamic binding** implies that **the dynamic form of the object** determines which version of the operation to apply. Here it will be $perimeter_{RECT}$.

As noted, of course, the more interesting case arises when we cannot deduce from a mere reading of the software text what exact dynamic type *p* will have at run time, as in

```
                -- Compute perimeter of figure built according to user choice
p: POLYGON
…
if chosen_icon = rectangle_icon then
        ! RECTANGLE ! p.make (…)
elseif chosen_icon = triangle_icon then
        ! TRIANGLE ! p.make (…)
elseif
        …
end
…
x := p.perimeter
```

or after a conditional polymorphic assignment **if** … **then** *p* := *r* **elseif**… **then** *p* := *t*…; or if *p* is an element of a polymorphic array of polygons; or simply if *p* is a formal argument, declared of type *POLYGON*, of the enclosing routine — to which callers can pass actual arguments of any conforming type.

Then depending on what happens in any particular execution, the dynamic type of *p* will be *RECTANGLE*, or *TRIANGLE*, and so on. You have no way to know which of these cases will hold. But thanks to dynamic binding you do not *need* to know: whatever *p* happens to be, the call will execute the proper variant of *perimeter*.

This ability of operations to adapt automatically to the objects to which they are applied is one of the most important properties of object-oriented systems, directly addressing some of the principal quality issues discussed at the beginning of this book. We will examine its consequences in detail later in this chapter.

Dynamic binding also gives the full story about the information-loss aspects of polymorphism discussed earlier. Now we really understand why it is not absurd to lose information about an object: after an assignment *p* := *q*, or a call *some_routine* (*q*) where *p* is the formal argument, we have lost the type information specific to *q* but we can rest assured that if we apply an operation *p*.*polygon_feature* where *polygon_feature* has a special version applicable to *q*, that version will be the one selected.

> It is all right to send your pets to an Absentee Owner Department that caters to all kinds — *provided* you know that when meal time comes your cat will get cat food and your dog will get dog food.

Redefinition and assertions

If a client of *POLYGON* calls *p*.*perimeter*, it expects to get the value of *p*'s perimeter, as defined by the specification of function *perimeter* in the definition of the class. But now, because of dynamic binding, the client may well be calling another routine, redefined in some descendant. In *RECTANGLE*, the redefinition, while improving efficiency, preserves the result; but what prevents you from redefining *perimeter* to compute, say, the area?

This is contrary to the spirit of redefinition. Redefinition should change the implementation of a routine, not its semantics. Fortunately we have a way to constrain the semantics of a routine — assertions. The basic rule for controlling the power of redefinition and dynamic binding is simple: the precondition and postcondition of a routine will apply (informally speaking) to any redefinition; and, as we have already seen, the class invariant automatically carries over to all the descendants.

The exact rules will be given in chapter 16. But you should already note that redefinition is not arbitrary: only semantics-preserving redefinitions are permitted. It is up to the routine writer to express the semantics precisely enough to express his intent, while leaving enough freedom to future reimplementers.

On the implementation of dynamic binding

One might fear that dynamic binding could be a costly mechanism, requiring a run-time search of the inheritance graph and hence an overhead that grows with the depth of that graph and becomes unacceptable with multiple inheritance (studied in the next chapter).

Fortunately this is not the case with a properly designed (and statically typed) O-O language. This issue will be discussed in more detail at the end of this chapter, but we can already reassure ourselves that efficiency consequences of dynamic binding should not be a concern for developers working with a decent environment.

14.5 DEFERRED FEATURES AND CLASSES

Polymorphism and dynamic binding mean that we can rely on abstractions as we design our software, and rest assured that execution will choose the proper implementations. But so far everything was fully implemented.

We do not always need everything to be fully implemented. Abstract software elements, partially implemented or not implemented at all, help us for many tasks: analyzing the problem and designing the architecture (in which case we may keep them in the final product to remind ourselves of the analysis and design); capturing commonalities between implementations; describing the intermediate nodes in a classification.

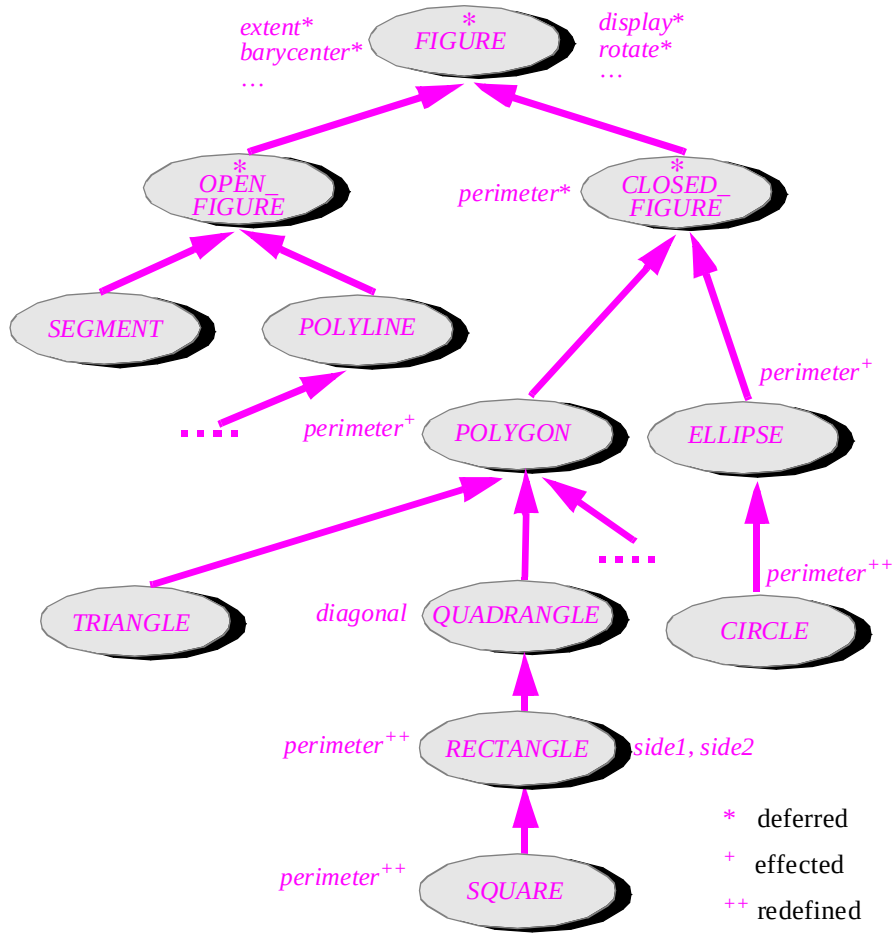
Deferred features and classes provide the needed abstraction mechanism.

Moving arbitrary figures

To understand the need for deferred routines and classes, consider again the *FIGURE* hierarchy, reproduced for convenience on the facing page.

The most general notion is that of *FIGURE*. Relying on the mechanisms of polymorphism and dynamic binding, you may want to apply the general scheme described earlier, as in:

The FIGURE hierarchy again




```
transform (f: FIGURE) is
        -- Apply a specific transformation to f.
    do
        f.rotate (…)
        f.translate (…)
    end
```

with appropriate values for the missing arguments. Then all the following calls are valid:

```
transform (r)                    -- with r: RECTANGLE
transform (c)                    -- with c: CIRCLE
transform (figarray.item (i))    -- with figarray: ARRAY [POLYGON]
```

In other words, you want to apply *rotate* and *translate* to a figure *f*, and let the underlying dynamic binding mechanism pick the appropriate version (different for classes *RECTANGLE* and *CIRCLE*) depending on the actual form of *f*, known only at run time.

This should work, and is a typical example of the elegant style made possible by polymorphism and dynamic binding, applying the Single Choice principle. You should simply have to redefine *rotate* and *translate* for the various classes involved.

But there is nothing to redefine! *FIGURE* is a very general notion, covering all kinds of two-dimensional figure. You have no way of writing a general-purpose version of *rotate* and *translate* without more information on the figures involved.

So here is a situation where routine *transform* would execute correctly thanks to dynamic binding, but is statically illegal since *rotate* and *translate* are not valid features of *FIGURE*. Type checking will catch *f.rotate* and *f.translate* as invalid operations.

You could, of course, introduce at the *FIGURE* level a *rotate* procedure which would do nothing. But this is a dangerous road to follow; *rotate* (*center, angle*) has a well-defined intuitive semantics, and "do nothing" is not a proper implementation of it.

Deferring a feature

What we need is a way to specify *rotate* and *translate* at the *FIGURE* level, while making it incumbent on descendants to provide actual implementations. This is achieved by declaring the features as "deferred". We replace the whole instruction part of the body (**do** *Instructions*) by the keyword **deferred**. Class *FIGURE* will declare:

```
rotate (center: POINT; angle: REAL) is
            -- Rotate by angle around center.
        deferred
        end
```

and similarly for *translate*. This means that the feature is *known* in the class where this declaration appears, but *implemented* only in proper descendants. Then a call such as catch *f.rotate* in procedure *transform* becomes valid.

With such a declaration, *rotate* is said to be a deferred feature. A non-deferred feature — one which has an implementation, such as all the features that we had encountered up to this one — is said to be **effective**.

Effecting a feature

In some proper descendants of *FIGURE* you will want to replace the deferred version by an effective one. For example:

```
class POLYGON inherit
        CLOSED_FIGURE
feature
        rotate (center: POINT; angle: REAL) is
                    -- Rotate by angle around center.
                do
                    … Instructions to rotate all vertices (see page 461) …
                end
        …
end -- class POLYGON
```

Note that *POLYGON* inherits the features of *FIGURE* not directly but through *CLOSED_FIGURE*; procedure *rotate* remains deferred in *CLOSED_FIGURE*.

This process of providing an effective version of a feature that is deferred in a parent is called **effecting**. (The term takes some getting used to, but is consistent: to effect a feature is to make it effective.)

A class that effects one or more inherited features does not need to list them in its **redefine** subclause, since there was no true definition (in the sense of an implementation) in the first place. It simply provides an effective declaration of the features, which must be type-compatible with the original, as in the *rotate* example.

Effecting is of course close to redefinition, and apart from the listing in the **redefine** subclause will be governed by the same rules. Hence the need for a common term:

Definition: redeclaration

To redeclare a feature is to redefine or effect it.

The examples used to introduce redefinition and effecting illustrate the difference between these two forms of redeclaration:

- When we go from *POLYGON* to *RECTANGLE*, we already had an implementation of *perimeter* in the parent; we want to offer a new implementation in *RECTANGLE*. This is a redefinition. Note that the feature gets redefined again in *SQUARE*.
- When we go from *FIGURE* to *POLYGON*, we had no implementation of *rotate* in the parent; we want to offer an implementation in *POLYGON*. This is an effecting. Proper descendants of *POLYGON* may of course redefine the effected version.

There may be a need to change some properties of an inherited deferred feature, while leaving it deferred. These properties may not include the feature's implementation (since it has none), but they may include the signature of the feature — the type of its arguments and result — and its assertions; the precise constraints will be reviewed in the next chapter. In contrast with a redeclaration from deferred to effective, such a redeclaration from deferred to deferred is considered to be a redefinition and requires the **redefine** clause. Here is a summary of the four possible cases of redeclaration:

Redeclaring from → ***to ↓***	**Deferred**	**Effective**
Deferred	Redefinition	Undefinition
Effective	Effecting	Redefinition

"Conflicts under sharing: undefinition and join", page 551.

This shows one case that we have not seen yet: *undefinition*, or redeclaration from effective to deferred — forgetting one's original implementation to start a new life.

Deferred classes

A feature, as we have seen, is either deferred or effective. This distinction extends to classes:

Definition: deferred, effective class

A class is deferred if it has a deferred feature. A class is effective if it is not deferred.

So for a class to be effective, all of its features must be effective. One or more deferred features make the class deferred. In the latter case you must mark the class:

Deferred class declaration rule

The declaration of a deferred class must use the juxtaposed keywords **deferred class** (rather than just **class** for an effective class).

So *FIGURE* will be declared (ignoring the **indexing** clause) as:

```
deferred class FIGURE feature
        rotate (…) is
                … Deferred feature declaration as shown earlier …
        … Other feature declarations …
end -- class FIGURE
```

Conversely, if a class is marked as **deferred** it must have at least one deferred feature. But a class may be deferred even if it does not declare any deferred feature of its own: it might have a deferred parent, from which it inherits a deferred feature that it does not effect. In our example, the class *OPEN_FIGURE* most likely does not effect *display*, *rotate* and other deferred features that it inherits from *FIGURE*, since the notion of open figure is still not concrete enough to support default implementations of these operations. So the class is deferred, and will be declared as

```
deferred class OPEN_FIGURE inherit
        FIGURE
…
```

even if it does not itself introduce any deferred feature.

A descendant of a deferred class is an effective class if it provides effective definitions for all features still deferred in its parents, and does not introduce any deferred feature of its own. Effective classes such as *POLYGON* and *ELLIPSE* must provide implementations of *display*, *rotate* and any other routines that they inherit deferred.

For convenience we will say that a type is deferred if its base class is deferred. So *FIGURE*, viewed as a type, is deferred; and if the generic class *LIST* is deferred — as it should be if it represents general lists regardless of the implementation — the type *LIST* [*INTEGER*] is deferred. Only the base class counts here: *C* [*X*] is effective if class *C* is effective and deferred if *C* if is deferred, regardless of the status of *X*.

Graphical conventions

The graphical symbols that have illustrated inheritance figures can now be fully explained. An asterisk marks a deferred feature or class:

*FIGURE**
*display**
*perimeter** -- At the level of *OPEN_FIGURE* in the illustration of page 483

A plus sign means “effective” and marks the effecting of a feature:

perimeter^{+} -- At the level of *POLYGON* in the illustration of page 483

You may mark a class with a plus sign $^{+}$ to indicate that it is effective. This is only used for special emphasis; an unmarked class is by default understood as effective, like a class declared as just **class** *C* …, without the **deferred** keyword, in the textual notation.

You may also attach a single plus sign to a feature, to indicate that it is being effected. For example *perimeter* appears, deferred and hence in the form *perimeter**, as early as class *CLOSED_FIGURE*, since every closed figure has a perimeter; then at the level of *POLYGON* the feature is effected to indicate the polygon algorithm for computing a perimeter, and so appears next to *POLYGON* as *perimeter*$^{+}$.

Finally, two plus signs (informally suggesting double effecting) mark redefinition:

perimeter^{++} -- At the level of *RECTANGLE* and *SQUARE* in the figure of page 483

What to do with deferred classes

The presence of deferred elements in a system prompts the question “what happens if we apply *rotate* to an object of type *FIGURE*?”; more generally, if we apply a deferred routine to a direct instance of a deferred class. The answer is draconian: there is no such thing as an object of type *FIGURE* — no such thing as a direct instance of a deferred class.

Deferred Class No-Instantiation rule

The creation type of a creation instruction may not be deferred

Recall that the creation type of a creation instruction is the type of *x* in the form !! *x*, and is *U* in the explicit-type form ! *U* ! *x*. A type is deferred if its base class is.

So the creation instruction !! *f*… is invalid, and will be rejected by the compiler, if the type of *f* is one of *FIGURE*, *OPEN_FIGURE*, *CLOSED_FIGURE*, all deferred. This rule removes any danger of causing erroneous feature calls.

Note, however, that even though *f*’s type is deferred you can still use *f* as target in the type-explicit form of the creation instruction, as in ! *RECTANGLE* ! *f*, as long as the creation type, here *RECTANGLE*, is one of the effective descendants of *FIGURE*. We saw how to use this technique in a multi-branch instruction to create a *FIGURE* object which, depending on the context, will be a direct instance of *RECTANGLE*, or of *CIRCLE*, etc.

At first the rule may appear to limit the usefulness of deferred classes to little more than a syntactic device to fool the static type system. This would be true but for polymorphism and dynamic binding. You cannot create an **object** of type *FIGURE*, but you can declare a polymorphic **entity** of that type, and use it without knowing the type (necessarily based on an effective class) of the attached object in a particular execution:

See also exercise E14.5, page 518.

```
f: FIGURE
…
f := "Some expression of an effective type, such as CIRCLE or POLYGON"
…
f.rotate (some_point, some_angle)
f.display
…
```

f could also be a formal argument, as in

```
some_routine (f: FIGURE) is …
```

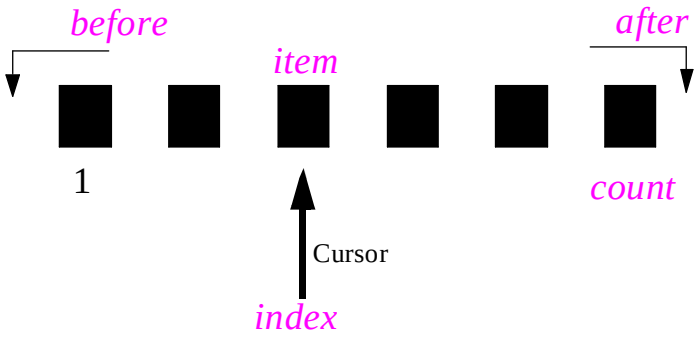
Such examples are the combination and culmination of the O-O method's unique abstraction facilities: classes, information hiding, Single Choice, inheritance, polymorphism, dynamic binding, deferred classes (and, as seen next, assertions). You manipulate objects without knowing their exact types, specifying only the minimum information necessary to ensure the availability of the operations that you require (here, that these objects are figures, so that they can be rotated and displayed). Having secured the type checker's stamp of approval, certifying that these operations are consistent with your declarations, you rely on a benevolent power — dynamic binding — to apply the correct version of each operation, without having to find out what that version will be.

Specifying the semantics of deferred features and classes

Although a deferred feature has no implementation, and a deferred class has either no implementation or a partial implementation only, you will often need to express their abstract semantic properties. You can use assertions for that purpose.

Like any other class, a deferred class can have a class invariant; and a deferred feature can have a precondition, a postcondition or both.

Consider the example of sequential lists, described independently of any particular implementation. As with many other such structures, it is convenient to associate with each list a cursor, indicating a currently active position:



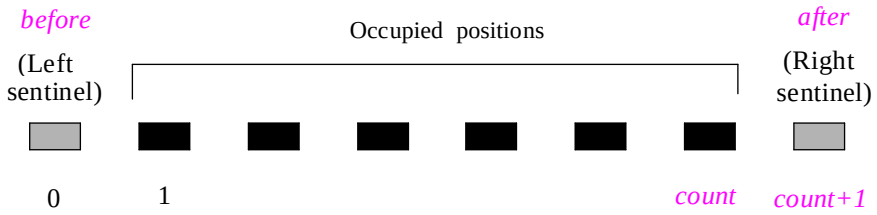
List with cursor

The class is deferred:

```
indexing
        description: "Sequentially traversable lists"
deferred class
        LIST [G]
feature -- Access
        count: INTEGER is
                        -- Number of items
                deferred
                end
        index: INTEGER is
                        -- Cursor position
                deferred
                end
        item: G is
                        -- Item at cursor position
                deferred
                end
feature -- Status report
        after: BOOLEAN is
                        --Is cursor past last item?
                deferred
                end
        before: BOOLEAN is
                        --Is cursor before first item?
                deferred
                end
feature -- Cursor movement
        forth is
                        --Advance cursor by one position.
                require
                        not after
                deferred
                ensure
                        index = old index + 1
                end
        … Other features …
invariant
        non_negative_count: count >= 0
        offleft_by_at_most_one: index >= 0
        offright_by_at_most_one: index <= count + 1
        after_definition: after = (index = count + 1)
        before  definition: before = (index = 0)
end -- class LIST
```

The invariant expresses the relations between the various queries. The first two clauses state that the cursor may only get off the set of items by one position left or right:

Cursor positions



The last two clauses of the invariant could also be expressed as postconditions: **ensure** *Result* = (*index* = *count* + *1*) in *after* and **ensure** *Result* = (*index* = *0*) in *before*. This choice always arises for a property involving argumentless queries only. In such a case I prefer to use an invariant clause, treating the property as applying globally to the class, rather than attaching it to any particular feature.

The assertions of *forth* express precisely what this procedure must do: advance the cursor by one position. Since we want to maintain the cursor within the range of list elements, plus two "sentinel" positions as shown on the last figure, application of *forth* requires **not** *after*; the result, as stated by the postcondition, is to increase *index* by one.

Here is another example, our old friend the stack. Our library will need a general *STACK* [*G*] class, which we now know will be deferred since it should cover all possible implementations; proper descendants such as *FIXED_STACK* and *LINKED_STACK* will describe specific implementations. One of the deferred procedures of *STACK* is *put*:

```
put (x: G) is
            -- Add x on top.
        require
            not full
        deferred
        ensure
            not_empty: not empty
            pushed_is_top: item = x
            one_more: count = old count + 1
        end
```

The boolean functions *empty* and *full* (also deferred at the *STACK* level) express whether the stack is empty, and whether its representation is full.

Only with assertions do deferred classes attain their full power. As noted (although the details will wait until two chapters from now), preconditions and postconditions apply to all redeclarations of a routine. This is especially significant in the deferred case: these assertions, if present, will set the limits for all permissible effectings. So the above specification constrains all variants of *put* in descendants of *STACK*.

Thanks to these assertion techniques you can make deferred classes informative and semantics-rich, even though they do not prescribe any implementation.

"THE ROLE OF DEFERRED CLASSES", 14.8, page 500.

At the end of this chapter we will come back to deferred classes and explore further their many roles in the object-oriented process of analysis, design and implementation.

14.6 REDECLARATION TECHNIQUES

The possibility of redeclaring a feature — redefining or effecting it — provides us with a flexible, incremental development style. Two techniques add to its power:

- The ability to redeclare a function into an attribute.
- A simple notation for referring to the original version in the body of a redefinition.

Redeclaring a function into an attribute

Redeclaration techniques provide an advanced application of one of the central principles of modularity that led us to the object-oriented method: uniform access.

See "Uniform Access", page 55.

As you will recall, the Uniform Access principle stated (originally in less technical terms, but we can afford to be precise now) that there should not be any fundamental difference, from a client's perspective, between an attribute and an argumentless function. In both cases the feature is a query; all that differs is its internal representation.

The first example was a class describing bank accounts, where the *balance* feature can be implemented as a function, which adds all the deposits and subtracts all the withdrawals, or as an attribute, updated whenever necessary to reflect the current balance. To the client, this makes no difference except possibly for performance.

With inheritance, we can go further, and allow a class that inherits a routine to redefine it as an attribute.

Our old example is directly applicable. Assume an original *ACCOUNT1* class:

```
class ACCOUNT1 feature
        balance: INTEGER is
                        -- Current balance
                do
                        Result := list_of_deposits.total – list_of_withdrawals.total
                end
        …
end -- class ACCOUNT1
```

Then a descendant can choose the second implementation of our original example, redefining *balance* as an attribute:

```
class ACCOUNT2 inherit
        ACCOUNT1
                        redefine balance end
feature
        balance: INTEGER
                        -- Current balance
        …
end -- class ACCOUNT2
```

ACCOUNT2 will likely have to redefine certain procedures, such as *withdraw* and *deposit*, so that on top of their other duties they update *balance*, maintaining invariant the property *balance = list_of_deposits.total – list_of_withdrawals.total*.

In this example the redeclaration is a redefinition. An effecting can also turn a deferred feature into an attribute. For example a deferred *LIST* class may have a feature

```
count: INTEGER is
        -- Number of inserted items
    deferred
    end
```

Then an array implementation may effect this feature as an attribute:

```
count: INTEGER
```

If we are asked to apply the classification that divides features into attributes and routines, we will by convention consider a deferred feature as a routine — even though, for a deferred feature with a result and no argument, the very notion of deferment means that we have not yet chosen between routine and attribute implementations. The phrase "deferred feature" is suitably vague and hence preferable to "deferred routine".

Combined with polymorphism and dynamic binding, such redeclarations of routines into attributes carry the Uniform Access principle to its extreme. Not only can we implement a client's request of the form *a.service* through either storage or computation, without requiring the client to be aware of our choice (the basic Uniform Access idea): we now have a situation where the same call could, in successive executions of the request during a single session, trigger a field access in some cases and a routine call in some others. This could for example happen with successive executions of the same *a.balance* call, if in the meantime *a* is polymorphically reattached to different objects.

Not the other way around

You might expect to be able to redefine an attribute into an argumentless function. But no. Assignment, an operation applicable to attributes, makes no sense for functions. Assume *x* is an attribute of a class *C*, and a routine of *C* contains the instruction

```
a := some_expression
```

Were a descendant of *C* to redefine *a*, then the routine — assuming it is not also redefined — would become inapplicable, since one cannot assign to a function.

The lack of symmetry (redeclaration permitted from function to attribute but not conversely) is unfortunate but inevitable, and not a real impediment in practice. It makes the use of an attribute a final, non-reversible implementation choice, whereas using a function still leaves room for later storage-based (rather than computation-based) implementations.

Using the original version in a redefinition

Consider a class that redefines a routine inherited from a parent. A common scheme for the redefinition is to perform what the original version did, preceded or followed by some other specific actions.

For example, a class *BUTTON* inheriting from *WINDOW* might redefine procedure *display* to indicate that to display a button is to display it as a window, then draw the border:

```
class BUTTON inherit
        WINDOW
                redefine display end
feature -- Output
        display is
                        -- Display as a button.
                do
                        "Display as a normal window"; -- See below
                        draw_border
                end
        … Other features …
end -- class BUTTON
```

where *draw_border* is a procedure of the new class. What we need to "Display as a normal window" is a call to the original, pre-redefinition version of *display*, known technically as the **precursor** of *draw_border*.

This case is common enough to justify a specific notation. The construct

```
Precursor
```

may be used in lieu of a feature name, but only in the body of a redefined routine. A call to this feature, with arguments if required, is a call to the parent's version of the routine (the precursor).

So in the last example the "Display as a normal window" part may be written as just

```
Precursor
```

meaning: call the version of this feature in class *WINDOW*. This would be illegal in any context other than the redefinition of a routine inherited from *WINDOW*, where *WINDOW* is a direct parent. *Precursor* is a reserved entity name, such as *Result* or *Current*, and like them is written in italics with an upper-case first letter.

In this example the redefined routine is a procedure, and so a call to the *Precursor* construct is an instruction. The call would be an expression in the redefinition of a function:

```
some_query (n: INTEGER): INTEGER is
        -- Value returned by parent version if positive, otherwise zero
    do
        Result := (Precursor (n)).max (0)
    end
```

In cases of multiple inheritance studied in the next chapter, a routine may have several precursors (enabling you to join several inherited routines into one). Then you will need to remove the ambiguity by specifying the parent, as in {{*WINDOW*}} *Precursor*.

"Keeping the original version of a redefined feature", page 555.

Note that the use of the *Precursor* construct does not make the precursor feature a feature of the class; only the redefined version is. (For one thing, the precursor version might fail to maintain the new invariant.) The only effect of the construct is to facilitate the task of the redefiner if the new job includes the old.

For any more complicated case, and in particular if you want to use both the precursor and the redefined version as features of the class, you will rely on a technique based on repeated inheritance, which actually *duplicates* a parent feature, yielding two full-fledged features in the heir. This will be part of the discussion of repeated inheritance.

14.7 THE MEANING OF INHERITANCE

We have now seen the basic techniques of inheritance. More remains to be studied, in particular how to deal with multiple inheritance, and the details of what happens to assertions in the context of inheritance (the notion of subcontracting).

But first we must reflect on the fundamental concepts and understand what they mean in the quest for software quality and an effective software development process.

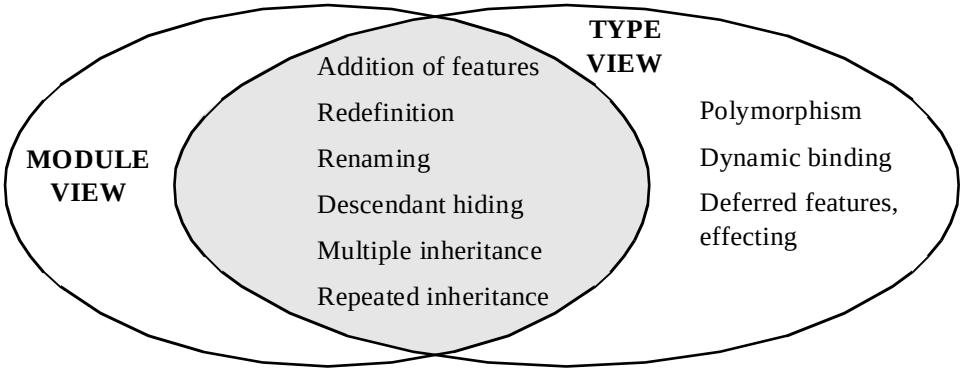
The dual perspective

Nowhere perhaps does the dual role of classes as modules and types, defined when we first encountered the notion of class, appear more clearly than in the study of inheritance. In the module view, an heir describes an extension of the parent module; in the type view, it describes a subtype of the parent type.

Although some aspects of inheritance belong more to the type view, most are useful for both views, as suggested by the following approximate classification (which refers to

a few facilities yet to be studied: renaming, descendant hiding, multiple and repeated inheritance). No aspect seems to belong exclusively to the module view.

Inheritance mechanisms and their role



See "ONE MECHANISM, OR MORE?", 24.6, page 833.

The two views reinforce each other, giving inheritance its power and flexibility. The power can in fact be intimidating, prompting proposals to separate the mechanism into two: a pure module extension facility, and a subtyping mechanism. But when we probe further (in the chapter on the methodology of inheritance) we will find that such a separation would have many disadvantages, and bring no recognizable benefit. Inheritance is a unifying principle; like many of the great unifying ideas of science, it brings together phenomena that had hitherto been treated as distinct.

The module view

From the module viewpoint, inheritance is particularly effective as a reusability technique.

A module is a set of services offered to the outside world. Without inheritance, every new module must itself define all the services it offers. Of course, the *implementations* of these services may rely on services provided by other modules: this is the purpose of the client relation. But there is no way to define a new module as simply adding new services to previously defined modules.

Inheritance gives that possibility. If *B* inherits from *A*, all the services (features) of *A* are automatically available in *B*, without any need to define them further. *B* is free to add new features for its own specific purposes. An extra degree of flexibility is provided by redefinition, which allows *B* to take its pick of the implementations offered by *A*, keeping some as they are while overriding others by locally more appropriate versions.

This leads to a style of software development which, instead of trying to solve every new problem from scratch, encourages building on previous accomplishments and extending their results. The spirit is one of both economy — why redo what has already been done? — and humility, in line with Newton's famous remark that he could reach so high only because he stood on the shoulders of giants.

"The Open-Closed principle", page 57.

The full benefit of this approach is best understood in terms of the **Open-Closed principle** introduced in an earlier chapter. (It may be worthwhile to reread the

corresponding section now in light of the concepts just introduced.) The principle stated that a good module structure should be both closed and open:

- Closed, because clients need the module's services to proceed with their own development, and once they have settled on a version of the module should not be affected by the introduction of new services they do not need.
- Open, because there is no guarantee that we will include right from the start every service potentially useful to some client.

This double requirement looks like a dilemma, and classical module structures offer no clue. But inheritance solves it. A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as a parent, adding new features and redeclaring inherited features; in this process there is no need to change the original or to disturb its clients. This property is fundamental in applying inheritance to the construction of reusable, extendible software.

If the idea were driven to the extreme, every class would add just one feature to those of its parents! This, of course, is not recommended. The decision to close a class should not be taken lightly; it should be based on a conscious judgment that the class as it stands already provides a coherent set of services — a coherent data abstraction — to potential clients.

See "Single-routine classes", page 728.

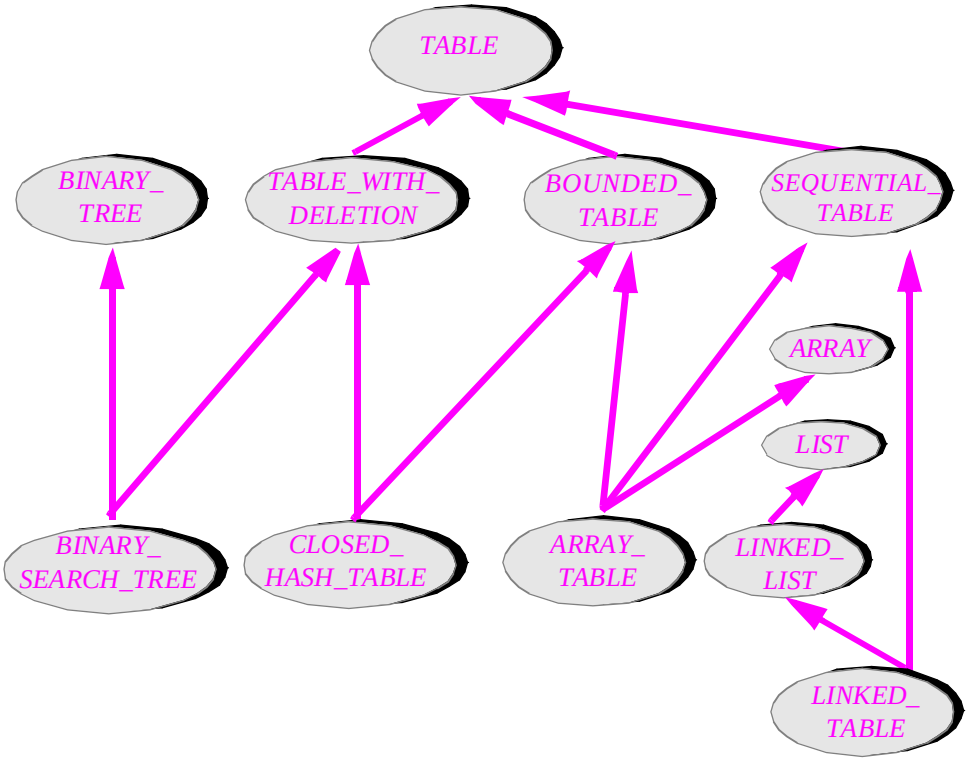
Also remember that the Open-Closed principle does not cover late hacking of inadequate services. If bad judgment resulted in a poor feature specification we cannot update the class without affecting its clients. Thanks to redefinition, however, the Open-Closed principle remains applicable if the change is compatible with the advertized specification.

Among one of the toughest issues in designing reusable module structures was the necessity to take advantage of commonalities that may exist between groups of related data abstractions — all hash tables, all sequential tables etc. By using class structures connected by inheritance, we can benefit from the logical relationships that exist between these implementations. The diagram below is a rough and partial sketch of a possible

"Factoring Out Common Behaviors", page 85.

structure for a table management library. The scheme naturally uses multiple inheritance, discussed in more detail in the next chapter.

Draft structure for a table library



This inheritance diagram is only a draft although it shows inheritance links typical of such a structure. For a systematic inheritance-based classification of tables and other containers, see [M 1994a].

With this view we can express the reusability requirement quite concretely: the idea is to move the definition of every feature **as far up** in the diagram as possible, so that it may be shared by the greatest possible number of descendant classes. Think of the process as the *reusability game*, played on boards that represent inheritance hierarchies such as the one on the last figure, with tokens that represent features. He who moves the most features the highest, as a result of discovering higher-level abstractions, and along the way merges the most tokens, as a result of discovering commonalities, wins.

The type view

From the type perspective, inheritance addresses both reusability and extendibility, in particular what an earlier discussion called continuity. The key is dynamic binding.

A type is a set of objects characterized (as we know from the theory of abstract data types) by certain operations. *INTEGER* describes a set of numbers with arithmetic operations; *POLYGON*, a set of objects with operations *vertices*, *perimeter* and others.

For types, inheritance represents the *is* relation, also known as *is-a*, as in "every dog is a mammal", "every mammal is an animal". Similarly, every rectangle is a polygon.

What does this relation mean?

- If we consider the values in each type, the relation is simply set inclusion: dogs make up a subset of the set of animals; similarly, instances of *RECTANGLE* make up a subset of the instances of *POLYGON*. (This comes from the definition of "instance" earlier in this chapter; note that a direct instance of *RECTANGLE* is not a direct instance of *POLYGON*).

"Instances", page 475.

- If we consider the operations applicable to each type, saying that every *B* is an *A* means that every operation applicable to instances of *A* is also applicable to instances of *B*. (With redefinition, however, *B* may provide its own implementation, which for instances of *B* overrides the implementation given in *A*.)

Using this relation, you can describe *is-a* networks representing many possible type variants, such as all the variants of *FIGURE*. Each new version of a routine such as *rotate* and *display* is defined in the class that describes the corresponding type variant. In the table example, each class in the graph will provide its own implementation of *search*, *insert*, *delete*, except of course when the parent's version is still appropriate.

A caveat about the use of "*is*" and "*is-a*". Beginners — but, I hope, no one who has read so far with even a modicum of attention — sometimes misuse inheritance to model the instance-to-mold relation, as with a class *SAN_FRANCISCO* inheriting from *CITY*. This is most likely a mistake: *CITY* is a class, which may have an instance representing San Francisco. To avoid such mistakes, it suffices to remember that the term *is-a* does not stand for "*x* is an *A*" (as in "*San_francisco* is a *CITY*"), a relation between an instance and a category, but for "every *B* is an *A*" (as in "Every *CITY* is a *GEOGRAPHICAL_UNIT*"), a relation between two categories — two classes in software terms. Some authors prefer to call this relation "*is-a-kind-of*" or, like [Gore 1996], "*can act as a*". This is partly a matter of taste (and partly a matter of substance, to be discussed in the chapter on inheritance methodology); once we have learned to avoid the trivial mistake, we can continue to use the well-accepted "*is*" or "*is-a*" terminology, never forgetting that it describes a relation between categories.

Inheritance and decentralization

With dynamic binding we can produce the **decentralized software architectures** necessary to achieve the goals of reusability and extendibility. Compare the O-O approach

— self-contained classes each providing its set of operation variants — with classical approaches. In Pascal or Ada, you may use a record type with variants

See "Single Choice", page 61.

```
type FIGURE =
        record
                "Common fields if any"
        case figtype: (polygon, rectangle, triangle, circle, …) of
                polygon: (vertices: LIST_OF_POINTS; count: INTEGER);
                rectangle: (side1, side2: REAL; …);
                …
        end
```

to define the various forms of figures. But this means that every routine that does something to figures (*rotate* and the like) must discriminate between possibilities:

```
case f.figure_type of
        polygon: …
        circle: …
        …
end
```

Routines *search* and others in the table case would use the same structure. The trouble is that all these routines possess far too much **knowledge** about the overall system: each must know exactly what types of figure are allowed in the system. Any addition of a new type, or change in an existing one, will affect every routine.

Ne sutor ultra crepidam, the shoemaker should not look beyond the sandal, is a software design principle: a rotation routine has no business knowing the exhaustive list of figure types. It should be content enough with the information necessary to do its job: rotating certain kinds of figure.

This distribution of knowledge among too many routines is a major source of inflexibility in classical approaches to software design. Much of the difficulty of modifying software may be traced to this problem. It also explains in part why software projects are so difficult to keep under control, as apparently small changes have far-reaching consequences, forcing developers to reopen modules that were thought to have been closed for good.

Object-oriented techniques deal with the problem head-on. A change in a particular implementation of an operation will only affect the class to which the implementation applies. Addition of a new type variant will in many cases leave the others completely unaffected. Decentralization is the key: classes manage their own implementations and do not meddle in each other's affairs. Applied to humans, this would sound like Voltaire's *Cultivez votre jardin*, tend your own garden. Applied to modules, it is an essential requirement for obtaining decentralized structures that will yield gracefully to requests for extension, modification, combination and reuse.

Representation independence

Dynamic binding also addresses one of the principal reusability issues: representation independence — the ability to request an operation with more than one variant, without having to know which variant will be applied. The discussion of this notion in an earlier chapter used the example of a call

"Representation Independence", page 84.

present := *has* (*x*, *t*)

which should use the appropriate search algorithm depending on the run-time form of *t*. With dynamic binding, we have exactly that: if *t* is declared as a table, but may be instantiated as any of binary search tree, closed hash table etc. (assuming all needed classes are available), then the call

present := *t*.*has* (*x*)

will find, at run time, the appropriate version of *has*. Dynamic binding achieves what the earlier discussion showed to be impossible with overloading and genericity: a client may request an operation, and let the underlying language system automatically find the appropriate implementation.

So the combination of classes, inheritance, redefinition, polymorphism and dynamic binding provides a remarkable set of answers to the questions raised at the beginning of this book: requirements for reusability; criteria, principles and rules of modularity.

The extension-specialization paradox

Inheritance is sometimes viewed as extension and sometimes as specialization. Although these two interpretations appear contradictory, there is truth in both — but not from the same perspective.

It all depends, again, on whether you look at a class as a type or a module. In the first case, inheritance, or *is*, is clearly specialization; "dog" is a more specialized notion than "animal", and "rectangle" than "polygon". This corresponds, as noted, to subset inclusion: if *B* is heir to *A*, the set of run-time objects represented by *B* is a subset of the corresponding set for *A*.

But from the module perspective, where a class is viewed as a provider of services, *B* implements the services (features) of *A* plus its own. *Fewer* objects often allows *more* features, since it implies a higher information value; going from arbitrary animals to dogs we can add the specific property of barking, and from arbitrary polygons to rectangles we can add the feature *diagonal*. So with respect to features implemented the subsetting goes the other way: the features applicable to instances of *A* are a subset of those for instances of *B*.

> Features *implemented* rather than services *offered* (to clients) because of the way information hiding combines with inheritance: as we will see, *B* may hide from its clients some of the features exported by *A* to its own.

Inheritance, then, is specialization from the type viewpoint and extension from the module viewpoint. This is the extension-specialization paradox: more features to apply, hence fewer objects to apply them to.

The extension-specialization paradox is one of the reasons for avoiding the term "subclass", which suggests "subset". Another, already noted, is the literature's sometimes confusing use of "subclass" to indicate direct as well as indirect inheritance. No such problem arises for the precisely defined terms *heir*, *descendant* and *proper descendant* and their counterparts *parent*, *ancestor* and *proper ancestor*.

14.8 THE ROLE OF DEFERRED CLASSES

Among the inheritance-related mechanisms addressing the problems of software construction presented at the beginning of this book, deferred classes are prominent.

Back to abstract data types

Loaded with assertions, deferred classes come close to representing abstract data types. A deferred class covering the notion of stack provides an excellent example. Procedure *put* has already been shown; here is a possible version for the full class.

```
indexing
        description:
                "Stacks (Last-in, First-Out dispenser structures), independently of %
                %any representation choice"
deferred class
        STACK [G]
feature -- Access
        count: INTEGER is
                        -- Number of elements inserted.
                deferred
                end
        item: G is
                        -- Last element pushed.
                require
                        not_empty: not empty
                deferred
                end
feature -- Status report
        empty: BOOLEAN is
                        -- Is stack empty?
                do
                        Result := (count = 0)
                end
        full: BOOLEAN is
                        -- Is stack full?
                deferred
                end
```

```
feature -- Element change
    put (x: G) is
            -- Push x onto top.
        require
            not full
        deferred
        ensure
            not_empty: not empty
            pushed_is_top: item = x
            one_more: count = old count + 1
        end
    remove is
            -- Pop top element.
        require
            not empty
        deferred
        ensure
            not_full: not full
            one_less: count = old count — 1
        end
    change_top (x: T) is
            -- Replace top element by x
        require
            not_empty: not empty
        do
            remove; put (x)
        ensure
            not_empty: not empty
            new_top: item = x
            same_number_of_items: count = old count
        end
    wipe_out is
            -- Remove all elements.
        deferred
        ensure
            no_more_elements: empty
        end
invariant
    non_negative_count: count >= 0
    empty_count: empty = (count = 0)
end
```

The class shows how you can implement effective routines in terms of deferred ones: for example, *change_top* has been implemented as a *remove* followed by a *put*. (This implementation may be inefficient in some representations, for example with arrays, but effective descendants of *STACK* may redefine the routine.)

Full specification page 139; also "FROM ABSTRACT DATA TYPES TO CLASSES", 6.5, page 142.

If you compare class *STACK* with the abstract data type specification given in the chapter on ADTs, you will find the similarities striking. Note in particular how the ADT functions map to features of the class, and the PRECONDITIONS paragraph to routine preconditions. Axioms are reflected in routine postconditions and in the class invariant.

See exercise E6.8, page 162: "more stack operations".

The addition of operations *change_top*, *count* and *wipe_out* is not an important difference since they could be specified as part of the abstract data type. Also minor is the absence of an explicit equivalent of the abstract data type function *new*, since creation instructions (which may rely on creation procedures introduced by effective descendants) will take care of object creation. There remain three significant differences.

See exercise E6.9, page 162: "bounded stacks".

The first is the introduction of a function *full*, accounting for implementations that will only accept a limited number of successive insertions, for example array implementations. This is typical of constraints that are irrelevant at the specification level but necessary in the design of practical systems. Note, however, that this is not an intrinsic difference between abstract data types and deferred classes, since we may adapt the ADT specification to cover the notion of bounded stack. Also, no generality is lost since some implementations (linked, for example) may have a version of *full* that always returns false.

"The imperative and the applicative", page 352.

The second difference, mentioned in the discussion of Design by Contract, is that an ADT specification is purely applicative (functional): it only includes functions, without side effects. A deferred class is imperative (procedural) in spite of its abstractness; *put*, for example, is specified as a procedure that will modify a stack, not as a function that takes a stack and returns a new stack.

Finally, as also noted in the earlier discussion, the assertion mechanism is not expressive enough for some ADT axioms. Of the four stack axioms

For any *x*: *G*, *s*: *STACK* [*G*]

A1 • *item* (*put* (*s*, *x*)) = *x*

A2 • *remove* (*put* (*s*, *x*)) = *s*

A3 • *empty* (*new*)

A4 • **not** *empty* (*put* (*s*, *x*))

"The expressive power of assertions", page 400, and subsequent section.

all but A2 have a direct equivalent in the assertions. (For A3 we assume that descendants' creation procedures will state **ensure** *empty*.) An earlier discussion explained the reasons for this limitation, and hinted at possible ways — formal specification languages, IFL — to remove it.

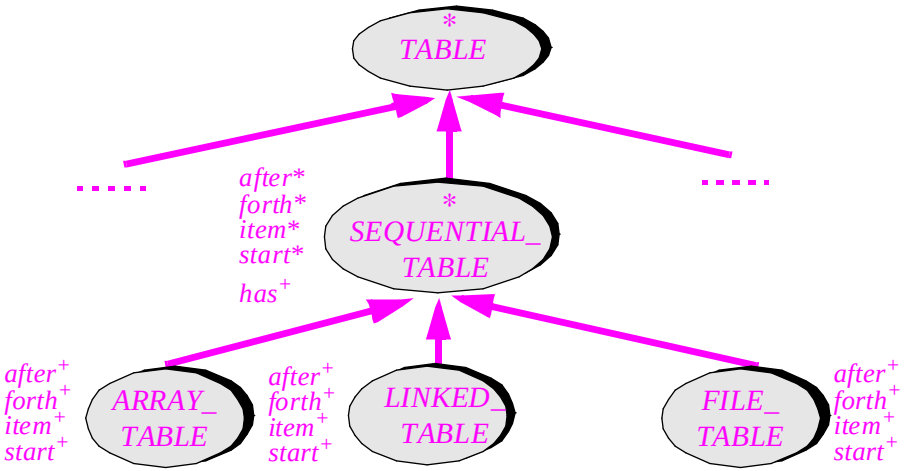
Deferred classes as partial implementations: the notion of behavior class

Not all deferred classes are as close as *STACK* to an abstract data type. In-between a fully abstract class like *STACK*, where all the fundamental features are deferred, and an effective class such as *FIXED_STACK*, describing just one implementation of an abstract data type, there is room for all degrees of partial ADT implementations or, said differently, groups of possible implementations.

The review of table implementation variants, which helped us understand the role of partial commonality in our study of reusability issues, provides a typical example. The original figure showing the relations between the variants can now be redrawn as an inheritance diagram:

See "Factoring Out Common Behaviors", page 85; figure on page 86.

Variants of the notion of table



The most general class, *TABLE*, is fully or almost fully deferred, since at that level we may specify a few features but not provide any substantial implementation. Among the variants is *SEQUENTIAL_TABLE*, representing tables in which elements are inserted sequentially. Examples of sequential tables include array, linked list and sequential file implementations. The corresponding classes, in the lowest part of the figure, are effective.

Classes such as *SEQUENTIAL_TABLE* are particularly interesting. The class is still deferred, but its status is intermediate between full deferment, as with *TABLE*, and full effecting, as with *ARRAY_TABLE*. It has enough information to allow implementing some specific algorithms; for example we can implement sequential search fully:

```
has (x: G): BOOLEAN is
            -- Does x appear in table?
        do
            from start until after or else equal (item, x) loop
                forth
            end
            Result := not after
        end
```

See the table on page 88 (which uses found in lieu of item).

This function is effective, although it relies for its algorithm on deferred features. The features *start* (bring the cursor to the first position), *forth* (advance the cursor by one position), *item* (value of element at cursor position), *after* (is the cursor after the last element?) are deferred in *SEQUENTIAL_TABLE*; each of the heirs of this class shown in the figure effects them in a different way, corresponding to its choice of implementation. These various effectings were given in the discussion of reusability. *ARRAY_TABLE*, for example, can represent the cursor as an integer *i*, so that the procedure *start* is implemented as *i := 1*, *item* as *t @ i* and so on.

"Specifying the semantics of deferred features and classes", page 488.

Note how important it is to include the precondition and postcondition of *forth*, as well as the invariant of the enclosing class, to make sure that all future effectings observe the same basic specification. These assertions appeared earlier in this chapter (in a slightly different context, for a class *LIST*, but directly applicable here).

This discussion shows the correspondence between classes and abstract data types in its full extent:

- A fully deferred class such as *TABLE* corresponds to an ADT.
- A fully effective class such as *ARRAY_TABLE* corresponds to an implementation of an ADT.
- A partially deferred class such as *SEQUENTIAL_TABLE* corresponds to a family of related implementations (or, equivalently, a partial implementation) of an ADT.

A class such as *SEQUENTIAL_TABLE*, which captures a behavior common to several ADT variants, may be called a **behavior class**. Behavior classes provide some of the fundamental design patterns of object-oriented software construction.

Don't call us, we'll call you

"Factoring Out Common Behaviors", page 85.

SEQUENTIAL_TABLE is representative of how object technology, through the notion of behavior class, answers the last one among the major reusability issues still open from the discussion in chapter 4: *Factoring out common behaviors*.

Particularly interesting is the possibility for an effective routine of a behavior class to rely on deferred routines for its implementation, as illustrated by *has*. This is how you may use partially deferred classes to capture common behaviors in a set of variants. The deferred class only describes what is common; variations are left to descendants.

Several of the design examples of later chapters rely on this technique, which plays a central role in the application of object-oriented techniques to building reusable software. It is particularly useful for domain-specific libraries and has been applied in many different contexts. A typical example, described in [M 1994a], is the design of the Lex and Parse libraries, a general-purpose solution to the problem of analyzing languages. Parse, in particular, defines a general parsing scheme, which will process any input text whose structure conforms to a certain grammar (for a programming language, some data format etc.). The higher-level behavior classes contain a few deferred features such as *post_action* describing semantic actions to be executed just after a certain construct has been parsed. To define your own semantic processing, you just effect these features.

This scheme is broadly applicable. Business applications, in particular, often follow standard patterns — process all the day's invoices, perform appropriate validation on a payment request, enter new customers … — with individual components that may vary.

In such cases we may provide a set of behavior classes with a mix of effective features to describe the known part and deferred features to describe the variable elements. Typically, as in the preceding example, the effective features call the deferred ones. Then descendants can provide the effectings that satisfy their needs.

> Not all the variable elements need to be deferred. If a default implementation is available, include it in the ancestor as an effective feature, which descendants may still redefine; this facilitates the work on the descendants, since they only need to provide new versions for features that depart from the default. (Recall that to become effective, that is, directly usable, a class must effect *all* its parents' deferred features.) Apply this technique only if a sound default exists; if not, as with *display* for *FIGURE*, the feature should be deferred.

This technique is part of a general approach that we may dub *don't call us, we'll call you*: rather than an application system that calls out reusable primitives, a general-purpose scheme lets application developers "plant" their own variants at strategic locations.

The idea is not entirely new. IBM's ancient and venerable database management system, IMS, already relied on something of the sort. In more recent software, a common structure for graphics systems (such as X for Unix) has an "event loop" which at each iteration calls specific functions provided by each application developer. This is known as a **callback** scheme.

What the O-O method offers, thanks to behavior classes, is systematic, safe support for this technique, through classes, inheritance, type checking, deferred classes and features, as well as assertions that enable the developer of the fixed part to specify what properties the variable replacements must always satisfy.

Programs with holes

With the techniques just discussed we are at the heart of the object-oriented method's contribution to reusability: offering not just frozen components (such as found in subroutine libraries), but flexible solutions that provide the basic schemes and can be adapted to suit the needs of many diverse applications.

One of the central themes of the discussion of reusability was the need to combine this goal with adaptability — to get out of the *reuse or redo* dilemma. This is exactly the effect of the scheme just described, for which we can coin the name "programs with holes". Unlike a subroutine library, where all is fixed except for the values of the actual arguments that you can pass, programs with holes, using classes patterned after the *SEQUENTIAL_TABLE* model, have room for user-contributed parts.

These observations help to put in perspective the "Lego block" image often used to discuss reusability. In a Lego set, the components are fixed; the child's creativity goes towards arranging them into an interesting structure. This exists in software, but sounds more like the traditional idea of subroutines. Often, software development needs to do exactly the reverse: keep the structure, but change the components. In fact the components

may not be there at all yet; in their place you find placeholders (deferred features), useful only when you plug in your own variants.

> In analogies with child games, we can go back to a younger age and think of those playboards where toddlers have to match shapes of blocks with shapes of holes — to realize that the square block goes into the square hole and the round block into the round hole.

You can also picture a partially deferred behavior class (or a set of such classes, called a "library" or a "framework") as having a few electrical outlets — the deferred features — into which the application developer will plug compatible devices. The metaphor nicely suggests the indispensable safeguards: the assertions, which express the requirements on acceptable pluggable devices, in the same way that an outlet's specification would prescribe a range of acceptable voltages, currents and other electrical parameters.

Deferred classes for analysis and global design

Deferred classes are also a key tool for using the method not just for implementation but also at the earliest and highest levels of system building — analysis and global design. The aim is to produce a system specification and architecture; for design, we also need an abstract description of each module, without implementation details.

The advice commonly given is to use separate notations: some analysis "method" (a term that in many cases just covers a graphical notation) and a PDL (Program Design Language, again often graphical). But this approach has many drawbacks:

"Seamless development", page 931.

- By introducing a gap between the successive steps, it poses a grave threat to software quality. The necessity of translating from one formalism to another may bring in errors and endangers the integrity of the system. Object technology, instead, offers the promise of a seamless, continuous software process.
- The multi-tiered approach is particularly detrimental to maintenance and evolution. It is very hard to guarantee that design and implementation will remain consistent throughout the system's evolution.
- Finally, most existing approaches to analysis and design offer no support for the formal specification of functional properties of modules independently of their implementation, in the form of assertions or a similar technique.

The last comment gives rise to the **paradox of levels**: precise notations such as the language of this book are sometimes dismissed as "low-level" or "implementation-oriented" because they externally look like programming languages, whereas thanks to assertions and such abstraction mechanisms as deferred classes they are actually *higher-level* than most of the common analysis and design approaches. Many people take a while to realize this, so early have they been taught the myth that high-level must mean vague; that to be abstract one has to be imprecise.

The use of deferred classes for analysis and design allows us to be both abstract and precise, and to keep the same language throughout the software process. We avoid conceptual gaps ("impedance mismatches"); the transition from high-level module descriptions to implementations can now proceed smoothly, within one formalism. But even unimplemented operations of design modules, now represented by deferred routines, may be characterized quite precisely by preconditions, postconditions and invariants.

The notation which we have by now almost finished developing covers analysis and design as well as implementation. The same concepts and constructs are applied at all stages; only the level of abstraction and detail differs.

14.9 DISCUSSION

This chapter has introduced the basic concepts of inheritance. Let us now assess the merits of some of the conventions introduced. Further comments on the inheritance mechanism (in particular on multiple inheritance) appear in the next chapter.

Explicit redefinition

The role of the **redefine** subclause is to enhance readability and reliability. Compilers do not really need it: since a class may have at most one feature of a given name, a feature declared in a class with the same name as an ancestor's feature can only be a redefinition of that feature — or a mistake.

The possibility of a mistake should not be taken lightly, as a programmer may be inheriting from a class without being aware of all the features declared in its ancestors. To avoid this dangerous case, any redefinition must be explicitly requested. This is the aim of the **redefine** subclause, which is also helpful to a reader of the class.

Accessing the precursor of a routine

You will have noted the rule on the *Precursor* (…) construct: it may only appear in the redefined version of a routine.

This ensures that the construct serves its purpose: enabling a redefinition to rely on the original implementation. The explicit naming of the parent avoids any ambiguity (in particular with multiple inheritance). Allowing arbitrary routines to access arbitrary ancestor features could make a class text very hard to understand, all the time forcing the reader to go the text of many other classes.

Dynamic binding and efficiency

One might fear that dynamic binding, for all its power, would lead to unacceptable run-time overhead. The danger exists, but careful language design and good implementation techniques avert it.

The problem is that dynamic binding requires some more work to be done at run time. Compare the usual routine call of traditional programming languages (Pascal, Ada, C…)

[1]

f (*x*, *a*, *b*, *c*…)

with the object-oriented form

[2]

x.*f* (*a*, *b*, *c*…)

"The Single Target principle", page 184.

The difference between the two was explained, in the introduction to the notion of class, as a consequence of the module-type identification. But now we know that more than style is at stake: there is also a difference of semantics. In [1], it is known statically — at compile time, or at worst at link time, if you use a linker to combine separately compiled modules — what exact feature the name *f* denotes. With dynamic binding, however, no such information is available statically for *f* in [2]: the feature to be selected depends on the type of the object to which *x* will be attached during a particular execution. What that type will be cannot, in the general case at least, be predicted from the text of the software; this is the source of the flexibility of the mechanism, touted earlier.

Let us think for a moment of a naïve implementation. We keep at run time a copy of the class hierarchy. Each object contains information about its type — a node in that hierarchy. To interpret *x.f*, the run-time environment looks at that node to see if the corresponding class has a feature *f*. If so, great, we have found what we need. If not, we look at the node's parent, and repeat the operation. We may have to go all the way to the topmost class (or the topmost classes in the case of multiple inheritance).

> In a typed language we have the guarantee that somewhere along the way we will find a suitable feature; in an untyped language such as Smalltalk we may fail to do so, and have to terminate the execution with a "message not understood" diagnostic.

This scheme is still applied, with various optimizations, in many implementations of non-statically typed languages. It implies a considerable performance penalty. Worse, that penalty is not predictable, and it *grows with the depth of the inheritance structure*, as the algorithm may have to go back all the way to the root of the inheritance hierarchy. This means introducing a direct conflict between reusability and efficiency, since working harder at reusability often leads to introducing more levels of inheritance. Imagine the plight of the poor designer who, whenever tempted to add an inheritance link, must assess whether it is really worth the resulting performance hit. No software developer should be forced into such choices.

This approach is one of the primary sources of inefficiency in Smalltalk environments. It also explains why Smalltalk does not (in common commercial implementations at least) support multiple inheritance, since the penalty in this case would be enormous, due to the need to traverse an entire graph, not just a linear chain.

Fortunately, the use of static typing avoids such unpleasantness. With the proper type system and compiling algorithms, there is no need ever to traverse the inheritance structure at run time. Because in a statically typed O-O language the possible types for *x* are not arbitrary but confined to descendants of *x*'s original type, the compiler can prepare the work of the run-time system by building arrayed data structures that contain all the needed type information. With these data structures, the overhead of dynamic binding becomes very small: *an index computation and an array access*. Not only is this penalty small; even more importantly, it is **constant** (more precisely, bounded by a constant), so that there is no need to worry about any reusability-efficiency tradeoff as discussed above. Whether the deepest inheritance structure in your system is 2 or 20, whether you have 100 classes or 10,000, the maximum overhead is exactly the same. This is true for both single and multiple inheritance.

The discovery, in 1985, of this property — that even in the presence of multiple inheritance it was possible to implement a dynamically-bound feature call in constant time — was the key impetus for the project that, among other things, yielded both the first and the present editions of this book: to build a modern software development environment, starting from the ideas brilliantly introduced by Simula 67 and extending them to multiple inheritance (prolonged experience with Simula having shown that the limitation to single inheritance was unacceptable, as explained in the next chapter), reconciling them with modern principles of software engineering, and combining them with the most directly useful results of formal approaches to software specification, construction and verification. The design of an efficient, constant-time dynamic binding mechanism, which may at first sight appear to be somewhat peripheral in this set of goals, was in reality an indispensable enabler.

These observations will be surprising to anyone who has been introduced to object technology through the lens of O-O analysis and design presentations that treat implementation and efficiency as mundane issues to be addressed after one has solved everything else. In the reality of industrial software development — the reality of engineering tradeoffs — efficiency is one of the key factors that must be considered at every step. (As noted in an earlier chapter, if you dismiss efficiency, efficiency will dismiss you.) Object technology is much more than constant-time dynamic binding; but without constant-time dynamic binding there can be no successful object technology.

Estimating the overhead

With the techniques described so far, it is possible to give rough figures on the overhead of dynamic binding. The following figures are drawn from ISE's experience, using dynamic binding (that is to say, disabling the static binding optimization explained next).

For a procedure that does nothing — a procedure declared as *p1* **is do end** — the penalty for dynamic binding over static binding (that is to say, over the equivalent procedure in C) is about 30%.

This is of course an upper bound, since real-life procedures do something. The price for dynamic binding is the same for any routine call regardless of what it does; so the more a routine does, the smaller the relative penalty. If instead of *p1* we use a procedure that performs some arbitrary but typical operations, as in

```
p2 (a, b, c: INTEGER) is
        local
                x, y
        do
                x := a; y := b + c +1; x := x * y; p2
                if x > y then x := x + 1 else x := x — 1 end
        end
```

then the overhead goes down to about 15%. For a routine that does anything more significant (for example by executing a loop), it can become very small.

Static binding as an optimization

In some cases you need the utmost in efficiency, and even the small overhead just discussed may be undesirable. Then you will notice that the overhead is not always justified. A call *x.f (a, b, c…)* need not be dynamically bound when either:

S1 • *f* is not redeclared anywhere in the system (it has only one declaration).

S2 • *x* is not polymorphic, that is to say is not the target of any attachment whose source has a different type.

In any such case — detectable by a good compiler — the code generated for *x.f (a, b, c…)* can be identical to what a compiler for C, Pascal, Ada or Fortran would generate for *f (x, a, b, c…)*. No overhead of any kind is necessary.

ISE's compiler, part of the environment described in the last chapter of this book, currently applies optimization S1; the addition of S2 is planned. (S2 analysis is in fact a consequence of the type analysis mechanisms described in the chapter on typing.)

Although S1 is interesting in itself, its direct benefit is limited by the relatively low cost of dynamic binding given in the preceding statistics. The real payoff is indirect, since S1 enables a third optimization:

S3 • Apply **automatic routine inlining** when appropriate

Routine inlining means expanding the body of a routine within the text of its caller, eliminating the need for any actual call. For example, with a routine

```
set_a (x: SOME_TYPE) is
            -- Make x the new value of attribute a.
        do
            a := x
        end
```

the compiler may generate, for the call *s.set_a (some_value)*, the same code that a Pascal compiler would generate for the assignment *s.a := some_value* (not permitted by our notation, of course, since it violates information hiding). In this case there is no overhead at all, since the generated code does not use a routine call.

Inline expansion has traditionally been viewed as an optimization that **programmers** should specify. Ada includes the provision for an **inline** pragma (directive to the compiler); C and C++ offer similar mechanisms. But this approach suffers from inherent limitations. Although for a small, stationary program a competent developer can have a good idea of what should be inlined, this ceases to be true for large, evolutionary developments. In that case, a compiler with a decent inlining algorithm will beat the programmers' guesses 100% of the time.

For any call to which automatic static binding (S1) is applicable, an O-O compiler can (as in the case of ISE's) determine whether automatic routine inlining (S3) is worthwhile, based on an analysis of the space-time tradeoffs. This is one of the most

dramatic optimizations — one of the reasons why it is possible to match the efficiency of hand-crafted C or Fortran code and sometimes, especially on large systems, exceed it.

To the efficiency advantage, which grows with the size and complexity of the software, the automatic approach to inlining adds the advantage of safety and flexibility. As you will have noted, inlining is semantically correct only for a routine that can be statically bound, as in cases S1 and S2. It is not only common but also consistent with the method, in particular the Open-Closed principle, to see a developer, midway through the development of a large system, add a redefinition of a feature which until then had only one implementation. If that routine has been inlined manually, the result is erroneous semantics (since dynamic binding is now required, and inlining of course means static binding). Developers should concentrate on building correct software, not performing optimizations that are tedious, error-prone when done manually, and automatable.

> There are some other correctness requirements for inlining; in particular, it is only applicable to non-recursive calls. When correct, inlining should only be applied when the space-time tradeoff makes sense: the inlined routine should be small, and should be called from only one place or a small number of places.

A final note on efficiency. Published statistics for object-oriented languages show that somewhere between 30% and 60% of calls truly need dynamic binding, depending on how extensively the developers use the method's specific facilities. (In ISE's software the proportion is indeed around 60%.) With the optimizations just described, you will only pay the price of dynamic binding for calls that need it. For the remaining dynamic calls, the overhead is not only small and constant-bounded, it is *logically necessary*; in most cases, achieving the equivalent effect without O-O mechanisms would have required the use of conditional instructions (**if** … **then** … or **case** … **of** …), which can be more costly than the simple array-indirection mechanism outlined above. So it is not surprising that O-O software, processed by a good compiler, can compete with hand-produced C code.

A button by any other name: when static binding is wrong

By now the reader will have understood a key consequence of the principles of inheritance presented in this chapter:

Dynamic Binding principle

Static binding is semantically incorrect unless its effect is identical to that of dynamic binding.

In the call $x.r$, if x is declared of type A but ends up at run time attached to an object of type B, and you have redefined r in B, calling the original version (say r_A) is not a choice; it is a bug!

No doubt you had a reason for redefining r. The reason may have been optimization, as with *perimeter* for *RECTANGLE*; but it may have been that the original version r_A was simply incorrect for B. Consider the example, sketched earlier, of a class *BUTTON* that inherits from a class *WINDOW* in a window system, because buttons are a special kind of

window; the class redefines procedure *display* because displaying a button is a little different from displaying an ordinary window (for example you must display the border). Then if *w* is of type *WINDOW* but dynamically attached, through polymorphism, to an object of type *BUTTON*, the call *w.display* **must** execute the button version! Using $display_{WINDOW}$ would result in garbled display on the screen.

As another example, assume a video game with a data structure *LIST* [*AIRCRAFT*] — a polymorphic data structure, as we have learned to use them — and a loop that executes *item.land* on each element of the list. Each aircraft type may have a different version of *land*, the landing procedure. Executing the default version is not an option but a mistake. (We may of course imagine real flight control software rather than just a game.)

We should not let the flexibility of the inheritance-based type system — specifically, the type conformance rule — fool us here: the ability to declare an entity at a level of abstraction (*WINDOW*, *AIRCRAFT*) higher than the actual type of the attached object during one particular execution (*BUTTON* or *BOEING_747_400*) is only a facility for the *engineering* of the software, at system writing time. During program *execution* the only thing that matters is the objects to which we apply features; entities — names in the text of the software — have long been forgotten. A button by any other name is still a button; whether the software called it a button, or for generality treated it as a window, does not change its nature and properties.

Mathematical analysis supports and explains this reasoning. From the chapter on assertions you may remember the correctness condition for a routine:

From the definition of class correctness on page 371.

$$\{pre_r\ (x_r)\ \textbf{and}\ INV\}\ \ Body_r\ \ \{post_r\ (x_r)\ \textbf{and}\ INV\}$$

which we can simplify for the benefit of this discussion (keeping the part relative to the class invariant only, ignoring the arguments, and using as subscript the name *A* of the enclosing class) as

[A-CORRECT]

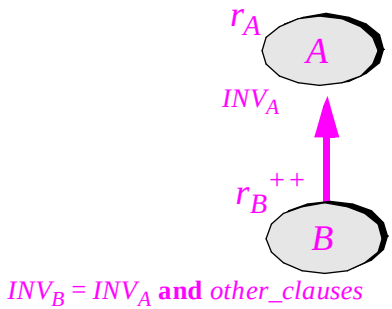
$$\{INV_A\}\ \ r_A\ \ \{INV_A\}$$

meaning in plain English: any execution of routine *r* from class *A* will preserve the invariant of class *A*. Now assume that we redefine *r* in a proper descendant *B*. The corresponding property will hold if the new class is correct:

[B-CORRECT]

$$\{INV_B\}\ \ r_B\ \ \{INV_B\}$$

As you will recall, invariants accumulate as we go down an inheritance structure: so INV_B implies INV_A, but usually not the other way around.



r_A *preserves the invariant of* A…

… *and* r_B *preserves the invariant of* B…

… *but* r_A *has no particular reason to preserve the invariant of* B!

A parent version may fail to satisfy the new invariant

Remember for example how *RECTANGLE* added its own clauses to the invariant of *POLYGON*. Another example, studied in the presentation of invariants, is a class *ACCOUNT1* with features *withdrawals_list* and *deposits_list*; then, perhaps for efficiency reasons, a proper descendant *ACCOUNT2* adds an attribute *balance* to store an account's current balance at all time, with the new invariant clause given in the earlier discussion:

On the ACCOUNT example *see "CLASS INVARIANTS", 11.8, page 364.*

consistent_balance: *deposits_list*.*total* – *withdrawals_list*.*total* = *current_balance*

As a result, we may have to redefine some of the routines of *ACCOUNT1*; for example a procedure *deposit* that merely used to add a list element to *deposits_list* must now update *balance* as well. Otherwise the class is simply wrong. This is similar to *WINDOW*'s version of the *display* procedure not being correct for an instance of *BUTTON*.

Now assume static binding applied to an object of type B, accessible through an entity of type A. Because the corresponding routine version, r_A, will usually not preserve the needed invariant — as with $deposit_{ACCOUNT1}$ for an object of type *ACCOUNT2*, or $display_{WINDOW}$ for an object of type *BUTTON* — the result will be to produce an inconsistent object, such as an *ACCOUNT2* object with an incorrect *balance* field, or a *BUTTON* object improperly displayed on the screen.

Such a result — an object that does not satisfy the invariant of its generating class, that is to say, the fundamental and universal constraints on all objects of its kind — is one of the worst events that could occur during the execution of a software system. If such a situation can arise, we can no longer hope to predict what execution will do.

To summarize: **static binding is either an optimization or a bug**. If it has the same semantics as dynamic binding (as in cases S1 and S2), it is an optimization, which compilers may perform. If it has a different semantics, it is a bug.

Cases S1 and S2 appeared were defined on page 510.

The C++ approach to binding

Given its widespread use and its influence on other languages, it is necessary to explain how the C++ language addresses some of the issues discussed here.

The C++ convention is surprising. By default, binding is static. To be dynamically bound, a routine (function or method in C++ terms) must be specially declared as **virtual**.

Two decisions are involved here:

C1 • Making the programmer responsible for selecting static or dynamic binding.

C2 • Using static binding as the default.

Both are damaging to object-oriented software development, but there is a difference of degree: C1 is arguable; C2 is hard to defend.

"PROGRAMMER-CONTROLLED DEALLOCATION", 9.4, page 294

Compared to the approach of this book, C1 results from a different appreciation of which tasks should be handled by humans (software developers), and which by computers (more precisely, compilers). This is the same debate that we encountered with automatic memory management. The C++ approach, in the C tradition, is to give the programmer full control over the details of what happens at run time, be it object deallocation or routine call. The spirit of object technology instead suggests relying on compilers for tasks that are tedious and error-prone, if algorithms are available to handle them. On a large scale and in the long run, compilers always do a better job.

Developers are responsible for the efficiency of their software, of course, but they should direct their efforts to the area where they can make a real difference: the choice of proper software structures and algorithms. Language designers and compilers writers are responsible for the rest.

Hence the disagreement on decision C1: C++ considers that static binding, as well as inlining, should be specified by developers; the O-O approach developed in this book, that it is the responsibility of the compiler, which will optimize calls behind the scenes. Static binding is an optimization, not a semantic choice.

C1 has another negative consequence on the application of the method. Whenever you declare a routine you must specify a binding policy: virtual or not, that is to say dynamic or static. This policy runs against the Open-Closed principle since it forces you to guess from the start what will be redefinable and what will not. This is not how inheritance works in practice: you may have to redefine a feature in a distant descendant, without having ever foreseen the need for such a redefinition in the original. With the C++ approach, if the original designer did not have enough foresight, you need to go back to the ancestor class to change the declaration to **virtual**. (This assumes that you can modify its source text. If it is not available, or you are not entitled to change it, tough luck.)

Because of all this, decision C1 — requiring programmers to specify a binding policy — impedes the effectiveness of the object-oriented method.

C2 — the use of static binding as the default in the absence of a special "virtual" marker — is worse. Here it is hard to find justifications for the language design. Static binding, as we have seen, is always the wrong choice when its semantics differs from that of dynamic binding. There can be not reason for choosing it as the default.

Making programmers rather than compilers responsible for optimization when things are safe (that is to say, asking them to request static binding explicitly when they

think it is appropriate) is one thing; forcing them to write something special to *get the correct semantics* is quite another. When the concern for efficiency, misplaced or not, starts to prevail over the basic requirement of correctness, something is wrong.

Even in a language that makes the programmer responsible for choosing a binding policy (decision C1), the default should be the reverse: instead of requiring dynamically bound functions to be declared as **virtual**, the language should by default use dynamic binding and allow programmers to mark as **static**, or some such keyword, those features for which they want to request the optimization — trusting them, in the C-C++ tradition, to ascertain that it is valid.

The difference is particularly important for beginners, who naturally tend to stick with the default. Even with less intimidating a language than C++, no one can be expected to master all the details of inheritance right away; the responsible policy is to guarantee the correct semantics for novices (and more generally for developers starting a new project, who will "want to make it right before making it faster"), then provide an optimization facility for people who need it and understand the issues.

Given the software industry's widespread concern for "upward compatibility", getting the C++ committee to change the language's binding policy, especially C2, will be hard, but it is worth trying in light of the dangers of the current conventions.

> The C++ approach has regrettably influenced other languages; for example the dynamic binding policy of Borland's Delphi language, continuing earlier Pascal extensions, is essentially that of C++. Note, however, that Java, a recent derivative of C++, has adopted dynamic binding as its policy.

These observations call for some practical advice. What can the developer do in C++ or a language that follows its policy? The best suggestion — for developers who do not have the option of switching to better tools, or waiting for the language to change — is to declare **all** functions as virtual, hence allowing for arbitrary redeclarations in the spirit of object-oriented software development. (Some C++ compilers unfortunately put a limit on the number of virtuals in a system, but one may hope that such limitations will go away.)

The paradox of this advice is that it takes you back to a situation in which all calls are implemented through dynamic binding and require a bit of extra execution time. In other words, language conventions (C1 and C2) that are promoted as enhancing efficiency end up, at least if one follows correctness-enhancing rules, working against performance!

Not surprisingly, C++ experts have come to advise against becoming "too much" object-oriented. Walter Bright, author of a best-selling C++ compiler, writes

> *It's generally accepted that the more C++* [*mechanisms*] *you use in a class, the slower your code will be. Fortunately, you can do a few things to tip the scales in your favor. First, don't use virtual functions* [i.e. dynamic binding], *virtual base classes* [deferred classes], *destructors, and the like, unless you need them.* […] *Another source of bloat is multiple inheritance* […] *For a complex class hierarchy with only one or two virtual functions, consider removing the virtual aspect, and maybe do the equivalent with a test and branch.*

[Bright 1995].

"Modular decomposability", page 40.

In other words: avoid using object-oriented techniques. (The same text also advocates "*grouping all the initialization code*" to favor locality of reference — an invitation to violate elementary principles of modular design which, as we have seen, suggest that each class be responsible for taking care of its own initialization needs.)

This chapter has suggested a different approach: let the O-O software developer rely on the guarantee that the semantics of calls will always be the correct one — dynamic binding. Then use a compiler sophisticated enough do generate statically bound or inlined code for those calls that have been determined, on the basis of rigorous algorithmic analysis, not to require a d ynamically bound implementation.

14.10 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- With inheritance, you can define new classes by extension, specialization and combination of previously defined ones.
- A class inheriting from another is said to be its heir; the original is the parent. Taken to an arbitrary number of levels (including zero), these relations yield the notion of descendant and ancestor.
- Inheritance is a key technique for both reusability and extendibility.
- Fruitful use of inheritance requires redefinition (the possibility for a class to override the implementation of some of its proper ancestors' features), polymorphism (the ability for a reference to become associated at run time with instances of different classes), dynamic binding (the dynamic selection of the appropriate variant of a redefined feature), type consistency (the requirement that an entity be only attached to instances of descendant types).
- From the module perspective, an heir extends the services of its parents. This particularly serves reusability.
- From the type perspective, the relation between an heir and a parent of the original class is the *is* relation. This serves both reusability and extendibility.
- You may redefine an argumentless function into an attribute, but not the other way around.
- Inheritance techniques, especially dynamic binding, permit highly decentralized software architectures where every variant of an operation is declared within the module that describes the corresponding data structure variant.
- With a typed language it is possible to achieve dynamic binding at low run-time cost. Associated optimizations, in particular compiler-applied static binding and automatic in-line expansion, help O-O software execution match or surpass the efficiency of traditional approaches.
- Deferred classes contain one or more deferred (non-implemented) features. They describe partial implementations of abstract data types.

- The ability of effective routines to call deferred ones provides a technique for reconciling reusability with extendibility, through "behavior classes".
- Deferred classes are a principal tool in the use of object-oriented methods at the analysis and design stages.
- Assertions are applicable to deferred features, allowing deferred classes to be precisely specified.
- When the semantics is different, dynamic binding is always the right choice; static binding is incorrect. When they have the same abstract effect, using static binding as the implementation is an optimization technique, best left to the compiler to detect and apply safely, together with inlining when applicable.

14.11 BIBLIOGRAPHICAL NOTES

The concepts of (single) inheritance and dynamic binding were introduced by Simula 67, on which references may be found in chapter 35. Deferred routines are also a Simula invention, under a different name (virtual procedures) and different conventions.

The *is-a* relation is studied, more with a view towards artificial intelligence applications, in [Brachman 1983].

A formal study of inheritance and its semantics is given in [Cardelli 1984].

The double-plus graphical convention to mark redefinition comes from Nerson's and Waldén's Business Object Notation for analysis and design; references in chapter 27.

Some elements of the discussion of the role of deferred features come from [M 1996].

The *Precursor* construct (similar to the Smalltalk **super** construct, but with the important difference that its use is limited to routine redefinitions) is the result of unpublished work with Roger Browne, James McKim, Kim Waldén and Steve Tynor.

EXERCISES

E14.1 Polygons and rectangles

Complete the versions of *POLYGON* and *RECTANGLE* sketched at the beginning of this chapter. Include the appropriate creation procedures.

E14.2 How few vertices for a polygon?

The invariant of class *POLYGON* requires every polygon to have at least three vertices; note that function *perimeter* would not work for an empty polygon. Update the definition of the class so that it will cover the degenerate case of polygons with fewer than three vertices.

E14.3 Geometrical objects with two coordinates

Write a class *TWO_COORD* describing objects that are characterized by two real coordinates, having among its heirs classes *POINT*, *COMPLEX* and *VECTOR*. Be careful to attach each feature to its proper level in the hierarchy.

E14.4 Inheritance without classes

This chapter has presented two views of inheritance: as a module, an heir class offers the services of its parent plus some; as a type, it embodies the *is-a* relation (every instance of the heir is also an instance of each of the parents). The "packages" of modular but not object-oriented languages such as Ada or Modula-2 are modules but not types; inheritance in its first interpretation might still be applicable to them. Discuss how such a form of inheritance could be introduced in a modular language. Be sure to consider the Open-Closed principle in your discussion.

E14.5 Non-creatable classes

"Rules on creation procedures", page 238.

It is not permitted to create an instance of a deferred class. In an earlier chapter we saw another way to make a class non-creatable: include an empty creation clause. Are the two mechanisms equivalent? Can you see cases for using one rather than the other? (**Hint**: a deferred class must have at least one deferred feature.)

E14.6 Deferred classes and rapid prototyping

Deferred classes may not be instantiated. It was argued, on the other hand, that a first version of a class design might leave all the features deferred. It may be tempting to attempt the "execution" of such a design: in software development, one sometimes wishes, early in the game, to execute incomplete implementations, so as to get an early hands-on experience of some aspects of the system even though other aspects have not been finalized. Discuss the pros and cons of having a "prototype" option in the compiler, which would allow instantiating a deferred class and executing a deferred feature (amounting to a null operation). Discuss the details of such an option.

E14.7 Table searching library (term project)

Based on the discussion of tables in this chapter and the chapter on reusability, design a library of table classes covering various categories of table representations, such as hash tables, sequential tables, tree tables etc.

E14.8 Kinds of deferred feature

Can an attribute be deferred?

E14.9 Complex numbers

(This exercise assumes that you have read up to at least chapter 23.) An example in the discussion of module interfaces uses complex numbers with two possible representations, changes in representations being carried out behind the scenes. Study whether it is possible to obtain equivalent results through inheritance, by writing a class *COMPLEX* and its heirs *CARTESIAN_COMPLEX* and *POLAR_COMPLEX*.

"Legitimate side effects: an example", page 759.